Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo empírico sobre performance do módulo NSGA-III na ferramenta DCT

João Gabriel L. Neves

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo empírico sobre performance do módulo NSGA-III na ferramenta DCT

João Gabriel L. Neves

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 28 de maio de 2021

Dedicatória

Eu dedico esse trabalho a minha família, meus professores e meus colegas do curso que sempre me apoiaram na minha jornada de graduação.

Agradecimentos

Gostaria de agradecer o apoio do Professor Bonifácio na orientação desse trabalho, e a ajuda de meus colegas: Ana Tarcheti, Luís Amaral e Marcos César que me deram suporte na compreensão da ferramenta trabalhada.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Manter sistemas de software complexos é uma tarefa desafiadora e que consome bastante tempo. É necessário para tal que desenvolvedores tenham uma boa compreensão de como um sistema é decomposto e como suas features foram implementadas. Trabalhos anteriores demonstraram que ferramentas de clusterização de software (SMCs) facilitam consideravelmente os desenvolvedores no processo de dar manutenção em código. No entanto apesar da literatura sobre o assunto ter evoluído bastante nos últimos 20 anos, ainda há poucas ferramentas disponíveis publicamente, e mesmo essas ferramentas não escalam bem para cenários grandes, principalmente quando optimizando multi-objetivos. Uma dessas ferramentas publicas que apresenta uma abordagem multi-objetiva é a Ferramenta de Clusterização Draco(DCT), apresenta soluções viáveis para sistemas de software composto até mesmo de mais de 1000 módulos. esse estudo empírico visa introduzir ao DCT uma abordagem many-objetiva para o problema e verificar como ela se compara com a abordagem multi-objetiva previamente implementada do DCT.

Palavras-chave: Optimização Many-objetiva, Ferramentas de Clusterização de Software, Algoritmos Genéticos

Abstract

Maintaining complex software systems is a challenging and time-consuming task. It is necessary for this that developers have a good understanding of how a system is decomposed and how its features were implemented. Previous work has shown that software clustering tools (SMCs) considerably ease developers in the process of maintaining code. However, despite the fact that the literature on the subject has evolved considerably in the last 20 years, there are still few tools available publicly, and even these tools do not scale well for large scenarios, especially when optimizing multi-objectives. One of these public tools that presents a multi-objective approach is the Draco Clustering Tool (DCT), it presents viable solutions for software systems composed even of more than 1000 modules. this empirical study aims to introduce the DCT to a many-objective approach to the problem and to see how it compares with the previously implemented multi-objective approach of the DCT.

Keywords: Many-Objective Optimization, Software Module Clustering, Genetic Algorithms

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Conforme há o aumento de complexidade no software nos tempos atuais, há também um aumento de demanda de ferramentas automatizadas que dão suporte na escalabilidade e manutenção desses sistemas[1]. Nesse contexto uma contribuição importante para esse problema, foi a introdução de Ferramentas de Clusterização de Software(SMC) por [2]. A proposta das SMCs era de revelar a estrutura de um software agrupando seus módulos em clusters, de tal forma que seu algoritmo foi baseado no princípio de baixo acoplamento e alta coesão.

Estudos como o de [3] demonstram que o uso de SMCs ajuda superar complicações relacionadas a área de manutenção de código(documentação errada ou insuficiente). Nesse estudo vários engenheiros de software foram entrevistados e revelaram que na frequentemente a documentação apresenta algum problema que a impede de ser utilizada, seja por estar ultrapassada, por ser mal escrita ou não ter conteúdo relevante para o entendimento do sistema. Nesse contexto, SMCs se demonstraram eficientes como uma forma de compensar essas dificuldades encontradas no trabalho de manutenção de software.

Vários SMCs proporcionam soluções mono-objetivas para a questão de clusterização de software, isso é, eles buscam apenas uma solução de clusterização válida. Em contraposto a essas formas de soluções, o paradigma multi-objetivo visa encontrar múltiplas soluções, que buscam maximizar múltiplos objetivos, válidas para o problema, cabendo ao desenvolvedor escolher a que mais o ajuda. Um SMCs que utiliza desse paradigma multi-objetivo é a Ferramenta de Clusterização Draco (DCT), uma ferramenta pública que faz uso de um algoritmo genético multi-objetivo chamado de NSGA-II para atacar esse problema. No trabalho de [4] a ferramenta teve sua performance comparada com outras ferramentas multi-objetivas e mono-objetivas obtendo resultados satisfatórios.

No entanto estudos como o de [5] apontam limitações de abordagens clássicas multi-objetivas, apontando que este tipo de abordagem apresenta dificuldades em encontrar melhores soluções para um numero grande objetivos, no caso de [5] é descrito como maior

que três. Como resposta a esse problema foi-se desenvolvido uma variação do paradigma multi-objetivo, chamado de many-objetivo, que visa lidar com problemas de otimização com um numero alto de objetivos. Nesse contexto surge uma nova versão NSGA-II, chamada de NSGA-III que aborda esse novo paradigma many-objetivo.

Tendo em vista que o estudo de [4] em sua configuração do DCT otimiza cinco objetivos, nesse trabalho é introduzido um novo módulo ao DCT que aplica o algoritmo NSGA-III e o experimento realizado no trabalho de [4] é replicado ara verificar se há uma melhora na performance nos resultados obtidos no estudo para o novo módulo, principalmente em relação aos resultados obtidos pelo módulo NSGA-II do DCT.

Capítulo 2

Fundamentação Teórica

Nesta seção são apresentados os conhecimentos técnicos na área de clusterização de software e algoritmos genéticos que embasam este estudo.

2.1 Clusterização de software

O processo de clusterização de módulos de um sistema ajuda a diminuir drasticamente o custo de manutenção de um sistema. Isso se mostra ainda mais relevante quando se trata de sistemas enormes, compostos de uma grande quantidade de módulos com várias dependências entre eles, neste caso arrumar algum problema no sistema pode se demonstrar bem custoso, fazendo o desenvolvedor procurar em uma alta gama de módulos e suas dependências para achar o lugar exato em que a falha se encontra. Com os módulos do sistema propriamente particionados e categorizados de acordo com suas dependências ou semânticas, encontrar e corrigir o módulo em que o erro se encontra se torna uma atividade menos custosa.

Várias maneiras de se encarar a clusterização de um sistema foram propostas com o passar do tempo. Dentre elas uma bastante utilizada é encarar o problema como um Grafo de Dependências de Módulos(MDG)[6], neste grafo os vértices são encaradas como os módulos do sistema e uma aresta entre dois módulos é encarado como uma dependência existente entre eles, seja essa dependência uma importação, uma chamada de método ou o uso de uma variável. Tomando um MDG como entrada o problema de modularização de software pode ser formalmente definido da seguinte forma como proposto por [7] da seguinte forma: Tendo $G = (V, E)$ como sendo um MDG, aonde V é um conjunto de módulos de um sistema e $E = \{(u, v)/u, v \in V\}$ é o conjunto de dependências entre esses módulos, dessa forma o problema de clusterização consiste em achar uma partição de G em k clusters $C_1, C_2, ..., C_k$ que dividem V em $V_1, V_2, ..., V_k$ componentes tendo $\cup_{i=1}^{k} V_i = V$; $V_i \cap V_j = \varnothing$ e $V_i \neq \varnothing$ *para* $1 \leq i \leq k$, cada vértice em V_i esta associado

ao cluster C_i para $1 \leq i \leq k$. Cada aresta do MDG pode conter um peso associado que demonstra a relevância da dependência entre os módulos que ela conecta, no caso da falta desse peso é considerado seu valor como sendo 1. Com o MDG conseguimos modelar o problema de clusterização como sendo um *problema de partição de grafo*, que é NP-Hard como mostrado por [8]. A figura 2.1 apresenta um exemplo de um MDG clusterizado.

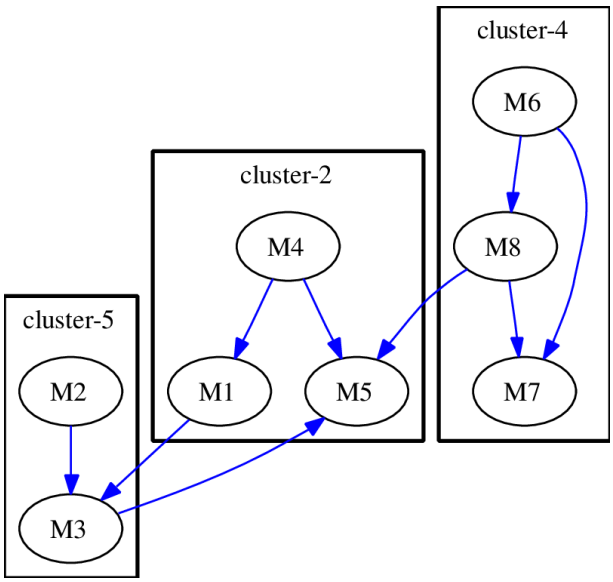


Figura 2.1: Exemplo de um MDG extraido de [9]

Tendo em mente o MDG como forma de resolver o problema queremos agora avaliar os possíveis clusters de acordo com as dependências de seus módulos, de maneira geral queremos uma partição que contenha um alto grau de coesão entre módulos de um mesmo cluster e baixo acoplamento entre módulos de clusters diferentes. Uma métrica utilizada por muitos autores em seus estudos no tópico ([2], [10], [11], [7]) é métrica de *Qualidade de Modularização*(MQ), ela representa o trade-off entre inter-conectividade e a intra-conectividade em um MDG. Intra-conectividade mede a quantidade de conexões entre módulos de um mesmo cluster, um alto índice de intra-conectividade indica uma alta coesão presente partição do sistema. Inter-conectividade, por outro lado, mede a quantidade de conexões entre módulos de clusters diferentes, um alto índice de inter-conectividade indica um alto nível de acoplamento do sistema. Dado um MDG particionado em k clusters a equação de calculo do MQ é demonstrada abaixo:

$$MQ = \sum_{i=1}^{k} CF_i$$
$$CF_i = \begin{cases} 0, & \text{if } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \sum_{j=1, j\neq i}^{k} \varepsilon_{i,j} + \varepsilon_{j,i}}, & \text{otherwise} \end{cases} \tag{2.1}$$

Nessa equação para cada cluster C_i, $1 \leq i \leq k$, é calculado o seu *fator de cluster*(CF_i)[12] e é feito um somatório com seus resultados, μ_i é a soma de todos os pesos das arestas internas do cluster C_i e $\varepsilon_{i,j}$ e $\varepsilon_{j,i}$ são as somas dos pesos das arestas que conectam os clusters C_i e C_j em cada direção respectivamente.

Apesar do amplo uso da métrica MQ, alguns estudos como [10] e [13] demonstram que levar somente em consideração o paradigma de busca de alta coesão e pouco acoplamento não garantem uma boa clusterização de módulos de um sistema. [10] aplicou esse paradigma e comparou os seus resultados com a clusterização de desenvolvedores experientes e com a clusterização recomendada pelos próprios desenvolvedores do sistema testado, e chegou a conclusão de que a clusterização gerada por sua heurística se mostrou menos eficaz do que as outras duas .[13] utiliza aliado a questão das dependências dos módulos do sistema a questão semântica existente entre esses módulos, ele apresenta uma clusterização que ajudam de maneira mais prática os desenvolvedores dos sistemas testados.

2.2 Algoritmos Genéticos

Algoritmos Genéticos(AGs) são uma tipo de heurística bem popular usada sobretudo para resolver problemas de optimização. O termo foi proposto pela primeira vez no livro *Adaptation in Natural and Artificial Systems* de 1975 por [14] e trata de uma simulação de conceitos da biologia evolutiva, como mutação, hereditariedade, seleção natural e recombinação, em uma população de representações abstratas de soluções de um problema, de forma a encontrar a solução mais otimizada para esse problema.

Para entendermos o funcionamento de um algoritmo genético é necessário entender as definições de componentes importantes para um algoritmo genético, esses são, de acordo com o livro *Introduction to Evolutionary Computing* escrito por [15]:

1. The labels consists of sequential numbers.

2. The numbers starts at 1 with every call to the enumerate environment.

Esses componentes serão definidos a seguir.

2.2.1 Representação

Representação diz respeito à forma como nós escolhemos abstrair soluções reais de forma que essas possam ser entendidas e analisadas pelo algoritmo. [15] definem como sendo **fenótipo** a representação das possíveis soluções do problema em seu contexto original, e como **genótipo** a representação codificada dessas de forma a serem usadas pelo algoritmo.

Por exemplo na figura 2.1 os exemplos de como o MDG foi clusterizado se enquadram como **fenótipos**, já um exemplo de **genótipo** seria representar os exemplos como sendo uma codificação binária onde o bit i indica se o módulo Ni pertence ao cluster 1(caso 0) ou cluster 2(caso 1), dessa forma o primeiro MDG clusterizado da figura seria representado por 001011 e o segundo por 000111.

2.2.2 Função de Avaliação (Função Fitness)

A função de avaliação, mais referenciada como *função fitness* representa os requerimentos pelo qual a população deveria se adaptar, isto é, ela é usada para verificar quais dos elementos da nossa população possuem a(s) característica(s) desejada(s). No exemplo da figura 2.1 a função *fitness* poderia usar a métrica MQ por exemplo para determinar qual das duas soluções é melhor avaliada, ela também poderia avaliar o valor semântico de cada módulo para selecionar a melhor solução ou poderia ainda fazer um apoderamento sobre estes dois aspectos.

2.2.3 População

A população é o conjunto de **genótipos** que representam possíveis soluções para o problema. Dessa forma a população acaba sendo a unidade evolutiva do algoritmo de forma que os seus indivíduos são estáticos e não mudam e sim a população que sofre modificações com o decorrer da execução do algoritmo. Na maioria das aplicações de GA o tamanho da população é constante, cabendo aos mecanismos de seleção (seja o de seleção de sobreviventes seja o de seleção de pais) selecionar os indivíduos melhor avaliados pela função *fitness* para comporem a próxima geração de indivíduos e os piores avaliados serem substituídos por novos indivíduos gerados.

2.2.4 mecanismo de seleção de pais

Mecanismos de seleção de pais em um GA tem o papel de selecionar os indivíduos mais bem avaliados pela função *fitness* para servirem como "pais"de uma nova geração. Um indivíduo é considerado um "pai"de uma interação de um GA caso ele seja usado nesta interação para sofrer variações que gerem novos indivíduos para compor a população. Essa seleção é tipicamente probabilística com indivíduos com uma melhor avaliação tendo maior probabilidade de servir como pais de uma nova geração e indivíduos com uma pior avaliação tendo uma menor probabilidade disso.

2.2.5 Operadores de Variação(Mutação e Recombinação)

Operadores de variação são os que permitem o GA a gerar indivíduos novos a partir de antigos. Eles se apresentam em GAs principalmente de duas formas, como mutação (que possui aridade unitária) e como recombinação (que possui aridade binária).

Mutação é uma operação que se dá em somente um **genótipo**, e que resulta em um descendente mutado em relação ao **genótipo** pai. A operação causa uma mudança aleatória e sem enviesamento no indivíduo pai. Normalmente em GAs é utilizada mais como uma maneira de auxiliar a recombinação, sendo aplicada aos genótipos produtos de uma recombinação, isso ajuda a manter a diversidade e introdução de novas combinações de características dentro da população do GA. Um exemplo de mutação pode ser visto na figura 2.2 onde decentes de uma recombinação tem bits, escolhidos aleatoriamente, onde seu valor é trocado.

Recombinação, também chamada de *crossover*, é uma operação de variação binária que envolve misturar as informações de dois *genótipos* pais para gerar um ou dois *genótipos* descendentes. Assim como a mutação a recombinação acontece de maneira aleatória, as partes de cada pai a serem combinadas são escolhidas de maneira randômica. O princípio é simples, dado dois indivíduos com características desejáveis só que diferentes, podemos gerar um descendente que combine as características dos dois pais. Na figura 2.2 temos um exemplo de recombinação, aonde, de maneira aleatória, foi escolhido o bit 8 para combinar as partes dos dois **genótipos** pais, a cadeia binária do primeiro pai antes do bit 8 é concatenada com com a cadeia de bits do segundo pai a partir do bit 8 para gerar um descendente, e o inverso é feito para gerar o outro descendente.

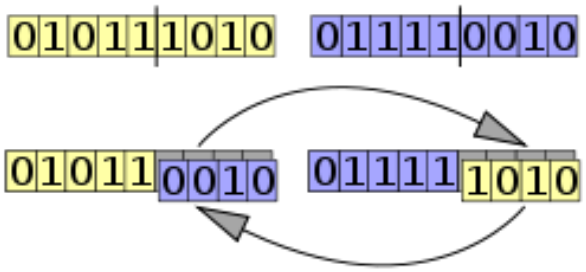


Figura 2.2: Exemplo de uma operação de recombinação sendo realizada extraído de [16]

É importante também notar que essas operações dependem da maneira como as soluções são representadas, nos exemplos dados elas são representadas como uma cadeia binária, no entanto esses exemplos não serviriam caso elas fossem representadas como uma árvore por exemplo, outros operadores de mutação e recombinação seriam necessários.

2.2.6 Mecanismo de Seleção de Sobreviventes

Como mencionado anteriormente, a quantidade de indivíduos em uma população de um GA geralmente é constante, isso significa que é necessário fazer uma seleção para decidir quais indivíduos formarão uma nova população. Mecanismos de seleção de sobreviventes são o último passo executado em um ciclo de um GA, logo depois de ser formada uma nova geração de indivíduos o mecanismo de seleção de sobreviventes é chamado para decidir quais dos **genótipos**, sejam pais ou descendentes, que vão compor a população do próximo ciclo. Essa escolha normalmente é baseada na qualidade de cada indivíduo de acordo com o resultado de sua função *fitness*, a idade de um indivíduo também pode ser levada em consideração. Ao contrário de mecanismos de seleção de pais que são probabilísticos ou até mesmo randômicos, A seleção de sobreviventes é na maioria das vezes determinística, ranqueando os indivíduos da população em relação ao seu valor *fitness* e selecionando os indivíduos melhor ranqueados que possuem as características mais desejáveis.

A figura 2.3 exemplifica esse processo de seleção do NSGA-II, nela é possível perceber que os indivíduos da população anterior $\mathbf{P}_t$ e os indivíduos frutos do processo de recombinação e mutação $\mathbf{Q}_t$ são ranqueados em relação ao seus valores *fitness* entres os grupos $\mathbf{F}_1$, $\mathbf{F}_2$, $\mathbf{F}_3$, ..., $\mathbf{F}_n$, dos quais só os melhores ranqueados são selecionados para compor a próxima geração $\mathbf{P}_{t+1}$.

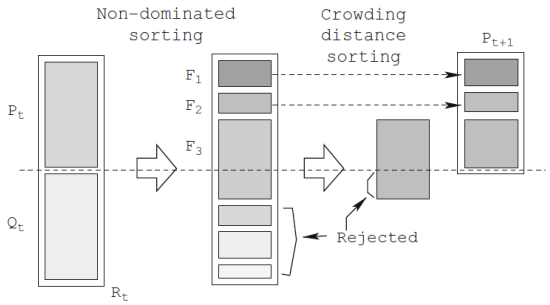


Figura 2.3: Fluxo de funcionamento do NSGA-II extraido de [17].

2.2.7 Inicialização e Condição de Parada

Outros conceitos também importantes de uma GA a se considerar são de como formar uma população inicial para o algoritmo e determinar uma condição de parada de sua execução.

A inicialização da população de um GA é geralmente feita de maneira simples, com os indivíduos da primeira população sendo gerados de maneira randômica. A princípio, uma heurística específica para o problema pode ser usada para gerar uma primeira população

com uma valor de *fitness* alto, no entanto isso nem sempre vale o valor computacional gasto.

Para a condição de parada há duas situações a serem analisadas. A primeira, no caso da existência de um valor *fitness* mínimo ϵ aceita, podemos parar o GA quando um indivíduo com um valor *fitness* maior ou igual a ϵ é encontrado. No entanto GAs tem natureza aleatória e portanto portanto não há garantia que um indivíduo com valor *fitness* maior ou igual a ϵ ser encontrado, dessa forma é necessário considerar uma outra situação que force o algoritmo a parar. As seguintes opções são normalmente usadas para esse tipo de situação:

1. O tempo máximo permitido de CPU é ultrapassado.
2. O número de vezes da chamada da função *fitness* chegar a um limite.
3. Os valores *fitiness* encontrados alcançam um limite por um período de tempo.
4. A diversidade da população chega a um valor mínimo.

A figura 2.4 representa o fluxo de funcionamento do NSGA-II, nela fica exemplificado os componentes de um GA discutidos acima.

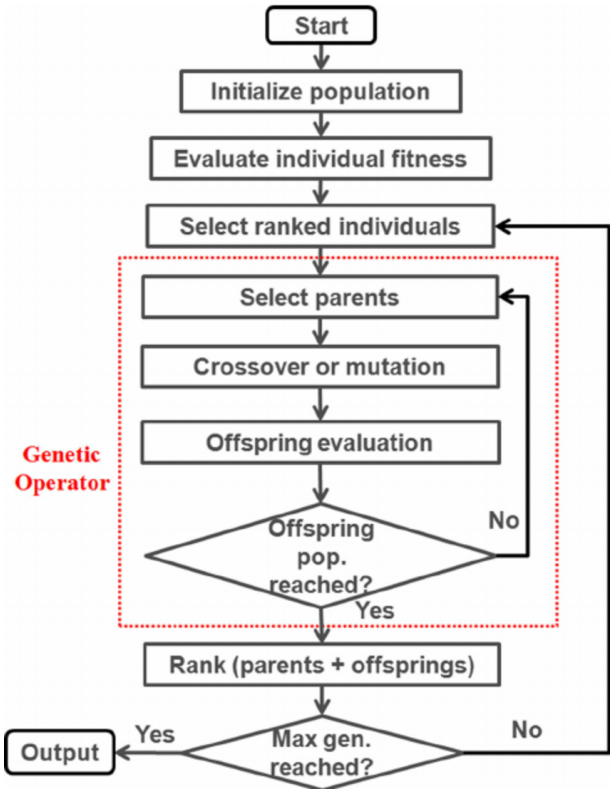


Figura 2.4: Fluxo de funcionamento do NSGA-II extraído de [18].

2.3 Optimização Mono-objetiva, Multi-objetiva

Quando se trata de otimização de soluções de um problema existem dois principais paradigmas que podemos adotar em relação ao número de características(objetivos) que queremos avaliar dessas soluções, são eles *mono-objetividade* e *multi-objetividade*.

2.3.1 Optimização Mono-Objetiva

Dizemos que uma optimização é *mono-objetiva* quando ela busca maximizar somente um aspecto das possíveis soluções de um problema. Por exemplo, como demonstrado por [2], a ferramenta Bunch busca maximizar somente a métrica MQ ao buscar soluções de clusterização para um sistema. Assim optimizações *mono-objetivas* apresentam como resultado somente uma melhor solução para um problema.

2.3.2 Optimização Multi-Objetiva

Uma optimização *multi-objetiva* leva em consideração tentar maximizar dois ou mais objetivos das soluções de um problema. O estudo feito por [13] exemplifica o uso de uma otimização multi-objetiva, onde o autor busca uma clusterização levando em conta as dependências do sistema, mas também levando em conta a semântica existente entre os módulos desse sistema. É importante apontar que nas otimização multi-objetivas os objetivos que queremos maximizar muitas vezes entram em contradição, o trabalho de [13] demonstra que muitas vezes para maximizar a questão da semântica de um cluster, compromissos tiveram que ser feitos a maximização da questão de dependências deste cluster, por isso esse tipo de otimização apresenta como resultado várias possíveis soluções. A escolha de quais soluções são as melhores para o problema é feita com o uso do conceito de dominância de Pareto, nele uma solução v tem dominância sobre outra u, caso nenhum objetivo de v for menor que esse objetivo em u, isso é for menos ótimo segundo nossos critérios, e a solução v tiver pelo um objetivo maior que esse objetivo em u. Assim encontrar o conjunto de melhores soluções pode ser entendido como encontrar soluções que não são dominadas por nenhuma outra, em um gráfico essas soluções formam o que é conhecido como Fronteira de Pareto que são os pontos do gráfico que formam um linha superior a todos os outros, isso é exemplificado no gráfico da figura 2.5.

2.4 NSGA-III

NSGA-III é um algoritmo genético, evolução do NSGA-II, que busca encontrar clusterização adequadas a sistemas, usando um paradigma de otimização chamado *many-objetivo*

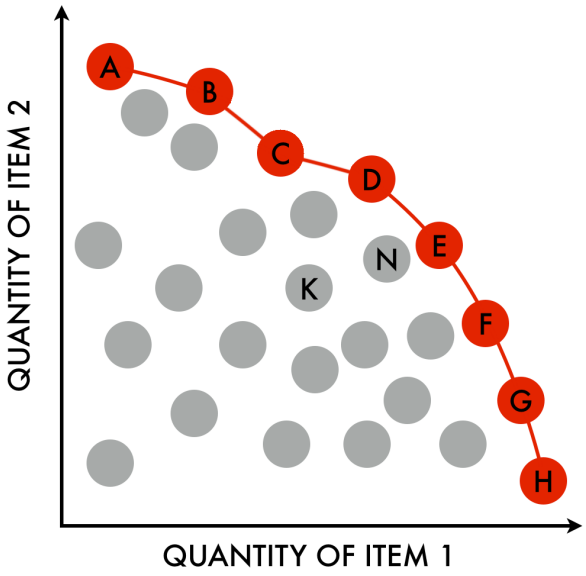


Figura 2.5: Exemplo da Fronteira de Pareto extraída de [19].

derivado do paradigma *multi-objetivo*. O trabalho [20] aborda o NSGA-III e clusterização no contexto do Java, com classes do Java sendo encaradas como módulos e *packages* como clusters. Segundo o trabalho ao contrário dos algoritmos de clusterização discutidos acima que buscam encontrar a melhor decomposição do sistema em termos de clusters, o NSGA-III se diferencia dessa abordagem ao tentar buscar melhorar a clusterização já existente no sistema, ele procura também fazer isso com o mínimo de mudanças possíveis já que desenvolvedores na prática tendem a preferir soluções que melhoram a estrutura do projeto com o mínimo de mudanças possíveis. O trabalho também ressalta que as operações propostas pelo algoritmo são mais profundas do que simplesmente alocar uma classe a um novo *package* ou dividir um *package* já existente, podendo também realizar operações de transferência de métodos de uma classe a outra que se encontre em um *package* distinto para melhorar a clusterização do projeto. O uso de histórico de desenvolvimento também é levado em consideração.

2.4.1 Abordagem Many-Objetiva

Como mencionado anteriormente o NSGA-III usa o paradigma de otimização conhecido por optimização *many-objetiva*[5], esse paradigma é derivado do paradigma *multi-objetivo* e se diferencia por demonstrar um número alto de objetivos a serem otimizados, por definição tem um número de objetivos M tal que $M > 3$.

A razão do porque o NSGA-II, que utiliza o paradigma *multi-objetivo* clássico, não conseguir lidar bem com mais de três objetivos se dá pelo fato que encontrar soluções que convergem a fronteira de Pareto para seus objetivos se torna uma tarefa difícil, o

algoritmo procura enfatizar os seus melhores de indivíduos que não são dominados por nenhum outro indivíduo, de maneira a convergir-los a fronteira de Pareto, quando esses objetivos passam a ser maior que 3 quase todos as soluções demarcadas para o *rank* 1 por não serem dominadas por nenhuma outra, com uma pouca variedade de *ranks* o NSGA-II não consegue manter a precisão de busca para encontrar uma convergência adequada a fronteira[21].

Várias abordagens foram propostas para lidar com o problema mencionado acima para a optimização *many-objetiva*, Estas são:

1. **abordagem de redução de objetivos** busca reduzir os objetivos do problema para somente os objetivos conflitantes entre [22].

2. **incorporação da *decision maker's preferences*** escolhe uma partição de soluções equivalentes a fronteira de Pareto, Região de Interesse(ROI), que melhor se encaixam as preferências do desenvolvedor,*decision maker(DM)*[23].

3. **nova preferência ordenando relações** faz a análise da população e do *rank* de suas soluções em relação a diferentes objetivos de maneira isolada. Quarta proposta é a **técnica de decomposição** que consiste em decompor o problema em vários subproblemas que podem ser resolvidos simultaneamente usando-se a capacidade de busca paralela de GAs[24].

4. **técnica de decomposição** consiste em decompor o problema em vários subproblemas que podem ser resolvidos simultaneamente usando-se a capacidade de busca paralela de GAs[25].

5. **o uso de uma busca predefinida de múltiplos alvos** envolve guiar a população durante o processo de otimização baseado em múltiplos alvos predefinidos(pontos de referência, direções de referência) no espaço de objetivos[26].

Capítulo 3

Implementação

Neste capítulo é apresentada a implementação do algoritmo nsga-iii na ferramenta Clustering Tool (DCT), para isso primeiro é apresentado uma breve explicação do que é a ferramenta e seu funcionamento para o algoritmo NSGA-II, e depois é discutido a implementação do NSGA-III e suas semelhanças e diferenças com o nsga-II previamente apresentado.

3.1 Draco Clustering Tool (DCT)

O trabalho de [4] apresenta o Draco como uma ferramenta com interface de linha de comando (CLI) implementada na linguagem de programação Go, que visa através de um um MDG de entrada do programa computar um grafo com os módulos da entrada propriamente clusterizados em um grafo no formato DOT. Ele faz isso através do uso de algoritmos genéticos discutidos no capítulo 2. Seu algoritmo default é o GA *multi-objetivo* NSGA-II, mas devido ao uso de interfaces da linguagem Go o GA a ser aplicado na operação pode ser facilmente substituído, havendo assim hoje na ferramenta a opção de se utilizar também o NSGA que é *mono-objetivo*. Um exemplo de uma invocação típica do DCT seria:

$ clustering < software.mdg > software.dot

De acordo com [4], a linguagem de Go foi escolhida para a implementação do DCT pelo fato de programas em Go serem compilados com uma compilação AOT (Ahead-Of-Time), que permite que os programas compilados sejam executados com mais eficiência e faz o uso da interface CLI ser mais conveniente. Além disso, Cada um dos componentes de um GA, discutidos no capítulo 2, é definido como uma interface Go, oque faz com que a substituição por outras implementações de GAs seja possível. Dessa forma a ferramenta consegue abordar os seguintes princípios:

- Uma interface de fácil uso. Apesar de uma interface de usuário gráfica potencialmente ser mais intuitiva, ela faz a automatização de experimentos ser mais difícil;
- Uso mínimo de memória. Usuários do DCT podem querer rodar a ferramenta em paralelo, portanto o consumo de memória deve ser mínimo;
- Eficiência em tempo de execução. Similarmente, o tempo gasto rodando um experimento deve ser mínimo;
- Extensível. Para experimentar com múltiplos cenários deve ser possível substituir porções do algoritmo de clusterização ou ajustar o valor de seus parâmetros;
- Formatos standards. Para fazer comparações entre o DCT com outras ferramentas mais fácil, DCT deve adotar formatos de arquivo bem conhecidos, tanto para a entrada (MDG) como para a saída(DOT);

As figuras 3.1 e 3.2 demonstram um diagrama UML dos módulos da ferramenta Draco.

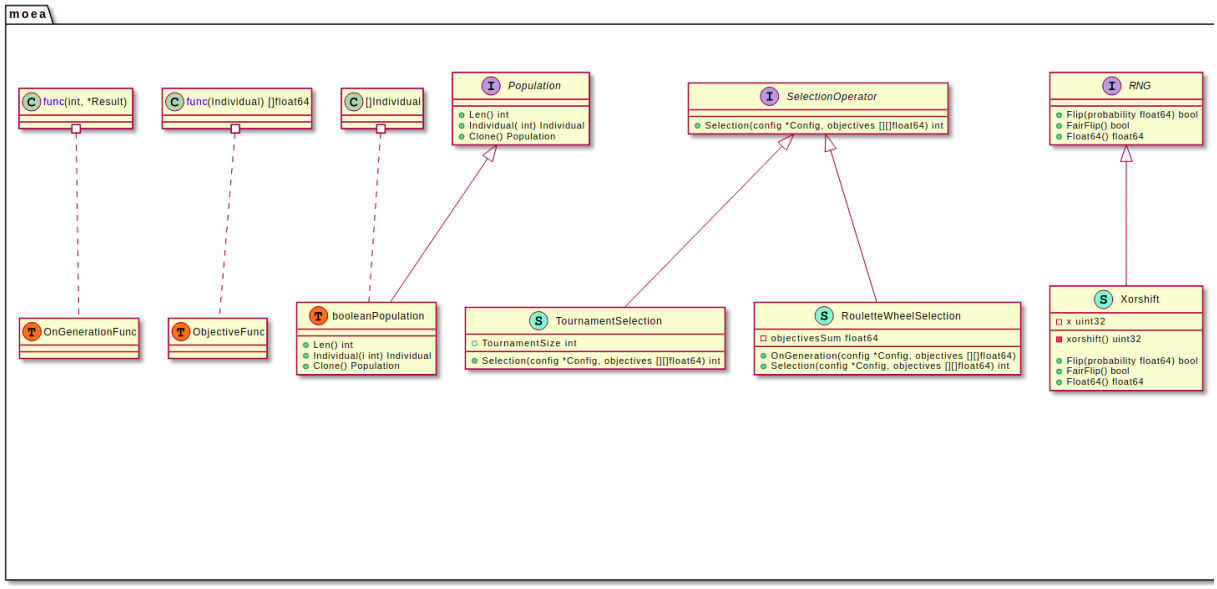


Figura 3.1: Diagrama UML dos módulos da ferramenta Draco

Como mencionado anteriormente o DCT utiliza uma implementação do NSGA-II como GA default em sua execução. Nesse algoritmo, como discutido no capítulo 2, cada indivíduo do GA possui seu vetor de valores fitness e é utilizado a fronteira de Pareto para decidir a dominância entre os indivíduos comparados. Dessa forma o Draco representa cada indivíduo como um mapeamento de um módulo do MDG para o cluster que ele pertence, tecnicamente um indivíduo é representado como um array onde cada posição corresponde a um módulo e cada valor corresponde a um cluster. A figura 3.3 do trabalho de [4], exemplifica essa representação:

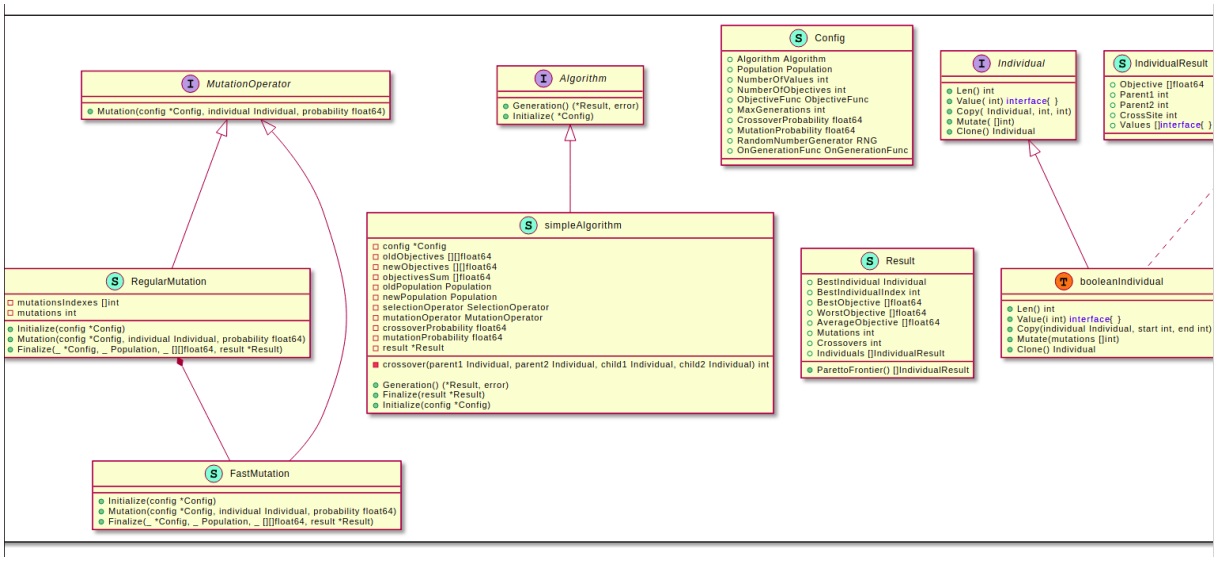


Figura 3.2: Diagrama UML dos módulos da ferramenta Draco

Nela conseguimos perceber que os módulos m_1, m_3 e m_4 pertencem ao cluster C_0, e que os módulos m_2 e f_1 pertencem ao cluster C_1, o indivíduo representado pelo array na figura reflete isso, com as posições 0, 2, 3 do array, que representam os módulos m_1, m_3 e m_4 respectivamente, possuem valor 0 e as posições 1 e 4, que representam os módulos m_2 e f_1 respectivamente, possuem valor 1.

É importante notar também que o DCT, ao contrário de outras implementação multi-objetivas que utilizam um array de inteiros para representar seus indivíduos, utiliza um array de binários o que contribui para a economia no uso de memória da ferramenta. O número máximo de clusters que um MDG pode apresentar é $\frac{|V|}{2}$, com essa representação de indivíduos cada elemento do array ocupa $[\log_2 \frac{|V|-1}{2}]$ bits da string binária, onde V é a quantidade de vértices do MDG. O trabalho de [4] cita como exemplo que no caso de um MDG com 10000 vértices um elemento do array vai ocupar apenas 13 bits na memória ao invés de 64 bits que é oque ocuparia caso representasse esses elementos como inteiros.

O módulo NSGA-II do DCT também implementa mecanismos de recombinação de indivíduos com a intenção de manter a diversidade entre os indivíduos da população, ele utiliza a operação de de *crossover*, discutida no capítulo 2, para tal.

Por fim, o módulo é configurado para otimizar cinco objetivos:

- Maximizar a medida MQ;
- Maximizar intra-conectividade;
- Minimizar inter-conectividade;
- Maximizar o número de clusters;

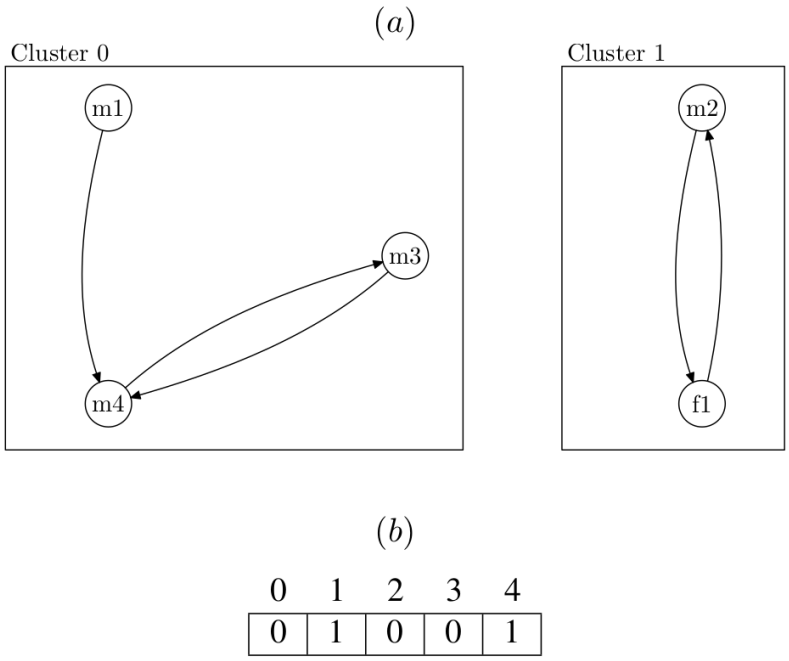


Figura 3.3: Exemplo da representação de um indivíduo no DCT extraída de [4]

- Minimizar a diferença entre a quantidade de entidades de código fonte máxima e mínima em um cluster.

A figura 3.4 abaixo demonstra um diagrama UML da implementação do NSGA-II na ferramenta Draco.

3.2 Implementação do NSGA-III

Em relação a implementação do algoritmo NSGA-III, os trabalhos de [26], [20] e [5] a descrevem como sendo bastante similar a implementação do NSGA-II, diferenciando apenas em relação ao operador de seleção de uma nova população.

No NSGA-II essa seleção se dá de tal maneira: considere o ciclo da geração n do algoritmo, suponha uma população P_n de tamanho N e uma população Q_n de descendentes criados a partir de P_n também de tamanho N. O algoritmo quer selecionar os melhores N membros da população $R_n = P_n \bigcup Q_n$(de tamanho 2N). Para atingir isso usamos os conceitos de dominância e da fronteira de Pareto para ranquear os membros da população R_n (membros do ranque F_1, F_2, F_3, e assim vai). Então cada ranque é selecionado do menor para o maior para formar a população P_{n+1} até que essa população atinja o tamanho N. Na maioria dos casos o último ranque a ser adicionado F_n, para não ultrapassar a cotação máxima de N membros da nova população, é integrado a P_{n+1} parcialmente. Nesse caso o algoritmo seleciona os membros do ranque F_n de forma que a diversidade

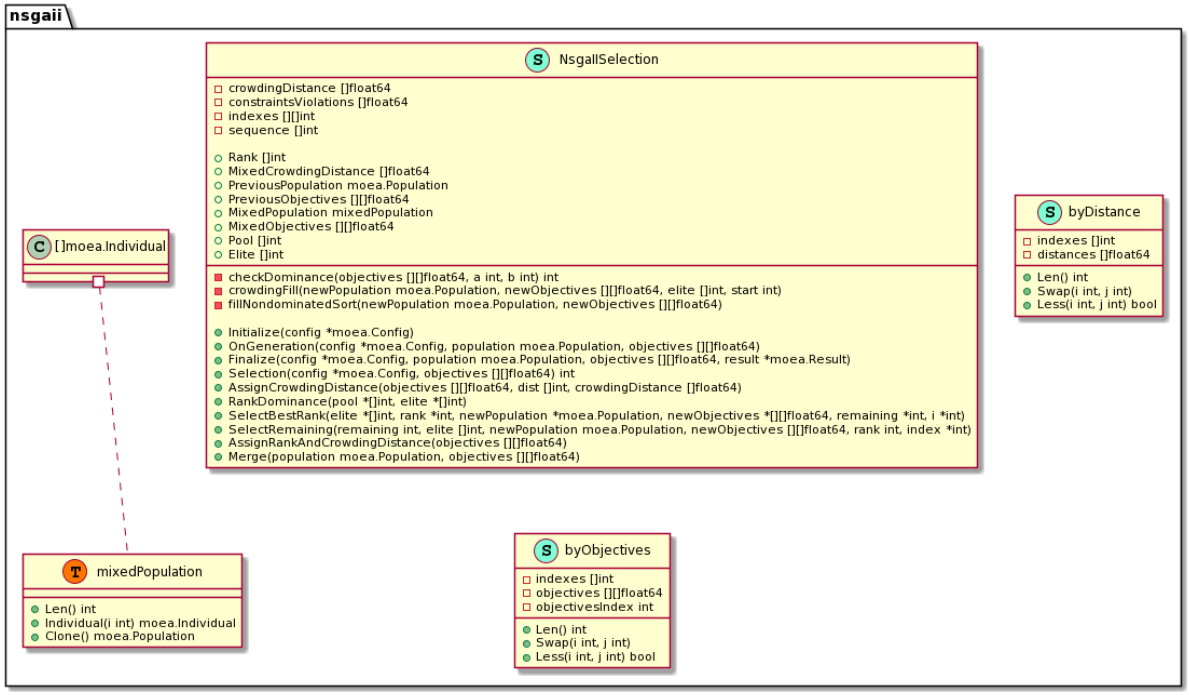


Figura 3.4: Diagrama UML do módulo NSGA-II no Draco

de P_{n+1} seja maximizada. Isso é obtido através de um método computacionalmente eficiente, no entanto aproximado, de preservação de nicho que computa a distância de multidão(*crowding distance*) como o somatório das distâncias dos objetivos normalizadas entre duas soluções vizinhas. Depois disso, os membros que apresentarem uma maior distância de multidão são selecionados.

Diferente do método de seleção do NSGA-II apresentado acima, no NSGA-III a manutenção de diversidade dentro de uma população é feita com o uso e atualização de pontos de referência bem espaçados em um hiperplano. [26] descreve esse processo como a execução de 5 passos, que serão descritas abaixo além de serem apresentados seus pseudo códigos:

3.2.1 Classificação da População em Ranques de Dominação

Os procedimentos discutidos acima sobre identificar a fronte de Pareto para ranquear os membros da população e selecionar os membros dos melhores ranques para compor a nova população também é realizado para o NSGA-III. Todos os membros do ranque 1 ao ranque n são incluídos em S_n. Se $|S_n| = N$ então acabam-se os procedimentos e a próxima geração começa com a população $P_{n+1} = S_n$. Para os casos em que $|S_n| > N$ então os membros dos ranques de F_1 a F_{n-1} são selecionados ($P_{n+1} = \bigcup_{i=1}^{n-1} F_i$). Os $K = |S_n|$ - $|P_n + 1|$ membros restantes serão selecionados da população de ranque F_n através

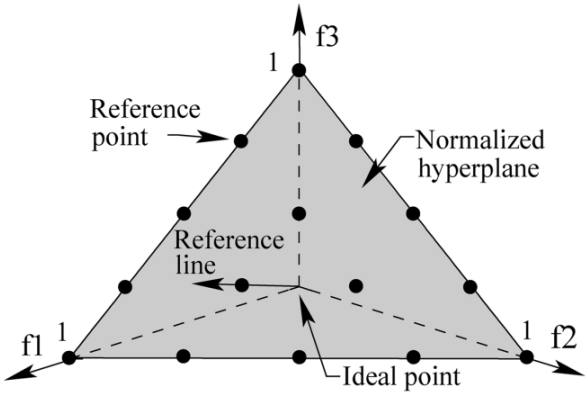


Figura 3.5: Exemplo dos pontos de referência sobre um hiper-plano extraídos de [26]

dos próximos métodos. Este procedimento descrito pode ser notado nas linhas 1 a 11 do Algoritmo 1 da figura 3.6.

3.2.2 Determinação dos pontos de referência em um hiperplano

Como discutido anteriormente, o NSGA-III usa um conjunto de pontos de referência em um hiper-plano para garantir diversidade nas soluções obtidas. Esses pontos de referência podem ser predefinidos de uma maneira estrutural ou ser definidos pelo próprio usuário. Para esse trabalho foi usado uma predefinição estrutural com a abordagem sistemática de [27] que coloca pontos em um hiper-plano normalizado igualmente inclinado em todos os eixos de seus objetivos e tem uma interceptação de um em cada eixo. Se um valor p para as divisões é considerado assim como o valor M de número de objetivos, o número de pontos de referência H é dado por:

$$H = \binom{M + p - 1}{p} \tag{3.1}$$

No algoritmo procuramos encontrar membros do ranque restante próximos desses pontos de referência. Como os pontos criados estão bem espaçados no hiper-plano normalizado, as soluções encontradas provavelmente estarão bem distribuídas na fronteira ótima de Pareto. O algoritmo proposto tem alta probabilidade de encontrar soluções próximas da fronteira ótima de Pareto correspondentes aos pontos de referência oferecidos na figura 3.6 esse procedimento pode ser observado na linha 14.

Por exemplo, em um problema com três objetivos (M = 3), os pontos de referência são criados em um triângulo com seus vértices nos pontos (1, 0, 0), (0, 1, 0) e (0, 0, 1). Se o número de divisões é decidido como 4 (p = 4) para cada eixo de objetivo H = $\binom{3+4-1}{4}$

Algorithm 1 Generation t of NSGA-III procedure

Input: H structured reference points Z^s or supplied aspiration points Z^a, parent population P_t

Output: P_{t+1}

$S_t = \emptyset$, $i = 1$
Q_t = Recombination+Mutation(P_t)
$R_t = P_t \cup Q_t$
$(F_1, F_2, \dots)$ = Non-dominated-sort(R_t)
repeat
  $S_t = S_t \cup F_i$ and $i = i + 1$
until $|S_t| \geq N$
Last front to be included: $F_l = F_i$
if $|S_t| = N$ **then**
  $P_{t+1} = S_t$, break
else
  $P_{t+1} = \cup_{j=1}^{l-1} F_j$
  Points to be chosen from F_l: $K = N - |P_{t+1}|$
  Normalize objectives and create reference set Z^r: `Normalize`($\mathbf{f}^n, S_t, Z^r, Z^s, Z^a$)
  Associate each member $\mathbf{s}$ of S_t with a reference point: $[\pi(\mathbf{s}), d(\mathbf{s})]$ =`Associate`(S_t, Z^r) % $\pi(\mathbf{s})$: closest reference point, d: distance between $\mathbf{s}$ and $\pi(\mathbf{s})$
  Compute niche count of reference point $j \in Z^r$: $\rho_j = \sum_{\mathbf{s} \in S_t / F_l} ((\pi(\mathbf{s}) = j)\ ?\ 1\ :\ 0)$
  Choose K members one at a time from F_l to construct P_{t+1}: `Niching`($K, \rho_j, \pi, d, Z^r, F_l, P_{t+1}$)
end if

Figura 3.6: Algoritmo de seleção do NSGA-III extraído de [26]

= 15 pontos de referência serão criados para ocupar o hiperplano. Esse exemplo pode ser observado na imagem 3.5.

3.2.3 Normalização Adaptativa dos Membros da População

Primeiro é preciso encontrar o valor ideal entre os membros de S_n, para isso identifica-se o valor mínimo (${Z_i}^{min}$) para cada objetivo 1,2,...,M e assim gerar o ponto ideal Z = (${Z_1}^{min}$, ${Z_2}^{min}$,...,${Z_M}^{min}$). Com isso, é traduzido os valores de cada objetivo dos membros de S_n através da subtração de cada objetivo f_i por ${Z_i}^{min}$ para cada membro. Esse procedimento pode ser denotado pelo seguinte método $f_i'(x) = f_i(x) - {Z_i}^{min}$. Dessa forma, o ponto extremo (z^i, max) para cada eixo de um objetivo i é identificado a solução $x \in S_n$ que

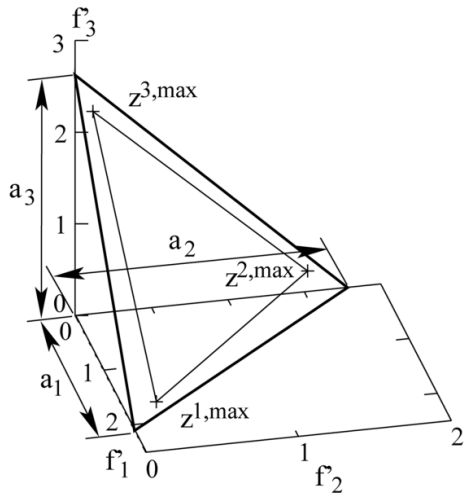


Figura 3.7: Procedimento para computar intercessões e elas formando o hiper-plano para pontos extremos para problemas de três objetivos extraída de [26].

faz a correspondente função escalar de realização (formada com f_i'(x) e um vetor de peso perto da eixo do objetivo i) mínima.

Esses M vetores extremos são usados para constituir um hiper-plano de M dimensões. A interceptação a_i do eixo do objetivo i e o hiperplano linear podem ser então computados como mostra a figura 3.7. Os objetivos podem então ser normalizados como:

$$f_i{}^n(x) = \frac{f_i{}'(x)}{a_i} \tag{3.2}$$

No caso dos pontos de referência estruturados calculados através com o método de [27] já estão presentes nesse hiper-plano normalizado. Como o procedimento de normalização e a criação do hiperplano é feito em cada geração usando pontos extremos encontrados no início da simulação, o procedimento NSGA-III proposto mantém adaptativamente uma diversidade no espaço ocupado pelos membros de S_n em todas as gerações. Isso possibilita o NSGA-III a resolver problemas com uma fronteira de Pareto cuja os valores de seus objetivos possam ser diferentemente escalados. O pseudocódigo desse procedimento pode ser observado na figura 3.8.

3.2.4 Operação de Associação

Após o processo de normalização dos membros de S_n no espaço de objetivo, é preciso associar cada membro da população com um ponto de referência. Para esse propósito, é definido uma linha de referência correspondendo a cada ponto de referência no hiper-plano

Algorithm 2 Normalize ($\mathbf{f}^n$, S_t, Z^r, Z^s/Z^a) procedure

Input: S_t, Z^s (structured points) or Z^a (supplied points)
Output: $\mathbf{f}^n$, Z^r (reference points on normalized hyper-plane)

1: **for** $j = 1$ **to** M **do**
2: Compute ideal point: $z_j^{\min} = \min_{\mathbf{s} \in S_t} f_j(\mathbf{s})$
3: Translate objectives: $f_j'(\mathbf{s}) = f_j(\mathbf{s}) - z_j^{\min} \quad \forall \mathbf{s} \in S_t$
4: Compute extreme points ($\mathbf{z}^{j,\max}$, $j = 1, \ldots, M$) of S_t
5: **end for**
6: Compute intercepts a_j for $j = 1, \ldots, M$
7: Normalize objectives ($\mathbf{f}^n$) using Equation 4
8: **if** Z^a is given **then**
9: Map each (aspiration) point on normalized hyper-plane using Equation 4 and save the points in the set Z^r
10: **else**
11: $Z^r = Z^s$
12: **end if**

Figura 3.8: Algoritmo de normalização de objetivos do NSGA-III extraído de [26]

juntando o ponto de referência com a origem. Então, é calculado a distância perpendicular de cada membro de S_n para cada um dos pontos de referência. O ponto de referência cuja linha de referência é a mais próxima de um membro da população no espaço de normalização é considerado ser associado a esse membro da população. Esse procedimento é ilustrado na imagem 3.9. O pseudocódigo desse procedimento é observado na imagem 3.10.

3.2.5 Operação de Preservação de Nicho

É importante notar que um ponto de referência pode ser associado a mais de um membro da população ou mesmo associado a nenhum. É contado o número de membros da população de $P_{n+1} = S_n/F_n$ que são associados com cada ponto de referência. A contagem de nicho para cada ponto de referência j é denotado como p_j. A operação de preservação de nicho é definida como seguinte. Primeiro, é identificado o conjunto de pontos de referência $J_{min} = \{ j : \text{argmin}_j p_j \}$ que tem o valor de p_j mínimo. No caso de múltiplos pontos de referência com valor mínimo de p_j um ponto (j' $\in J_{min}$) é escolhido aleatoriamente.

Se $p_j = 0$, podem haver dois cenários possíveis para j'. No primeiro, existem um ou mais membros no fronte F_n que são associados ao ponto j', nesse caso o membro que tem a menor distância perpendicular da linha de referência é adicionado a P_{n+1} e a contagem p_j do ponto j' é incrementado em 1. No segundo, o fronte F_n não tem nenhum membro

Algorithm 3 `Associate`(S_t, Z^r) procedure

Input: Z^r, S_t
Output: $\pi(\mathbf{s} \in S_t)$, $d(\mathbf{s} \in S_t)$

for each reference point $\mathbf{z} \in Z^r$ **do**
  Compute reference line $\mathbf{w} = \mathbf{z}$
end for
for each $\mathbf{s} \in S_t$ **do**
  for each $\mathbf{w} \in Z^r$ **do**
    Compute $d^{\perp}(\mathbf{s}, \mathbf{w}) = \| \left(\mathbf{s} - \mathbf{w}^T\mathbf{s}\mathbf{w}/\|\mathbf{w}\|^2\right) \|$
  end for
  Assign $\pi(\mathbf{s}) = \mathbf{w} : \text{argmin}_{\mathbf{w} \in Z^r} d^{\perp}(\mathbf{s}, \mathbf{w})$
  Assign $d(\mathbf{s}) = d^{\perp}(\mathbf{s}, \pi(\mathbf{s}))$
end for

Figura 3.9: Algoritmo de associação de objetivos do NSGA-III extraído de [26]

associado a j', nesse caso o ponto de referência j' de futuras considerações da geração atual.

No caso de $p_j > 1$, um membro aleatoriamente escolhido, se existir, de do fronte F_n que está associado ao ponto de referência j' é adicionado a P_{n+1} e p_j é incrementado em 1. Depois que as contagem de nicho são atualizadas, o procedimento é repetido K vezes até que as vagas restantes para a população P_{n+1} sejam preenchidas. Figura 3.11 apresenta o pseudocódigo desta operação.

3.3 Implementação do modulo NSGA-III na ferramenta Draco

Como discutido acima a ferramenta Draco foi feita para ser facilmente extensível, de tal maneira que trocar o GA que realiza a operação de clusterização é feito de maneira simples. Dessa forma, para a adição de um módulo NSGA-III ao projeto foi necessário apenas a criação de uma struct Go que implementa o método de seleção de indivíduos adotado pelo NSGA-III e o passar como parâmetro para ferramenta como o algoritmo a ser utilizado para realizar a clusterização.

Devido a alta semelhança, discutida acima, de funcionamento entre o NSGA-II, já implementado na ferramenta, e o NSGA-III, se diferenciando apenas na seleção dos indivíduos do último ranque a integrar a nova população, para a implementação do algoritmo fez-se uso de um recurso Go conhecido Embutir(*Embedding*) na struct do NSGA-III, foi embutido a struct Go do NSGA-II dentro da struct do NSGA-III oque permiti o módulo NSGA-III reutilizar varias porções de código do NSGA-II que são iguais em sua implemen-

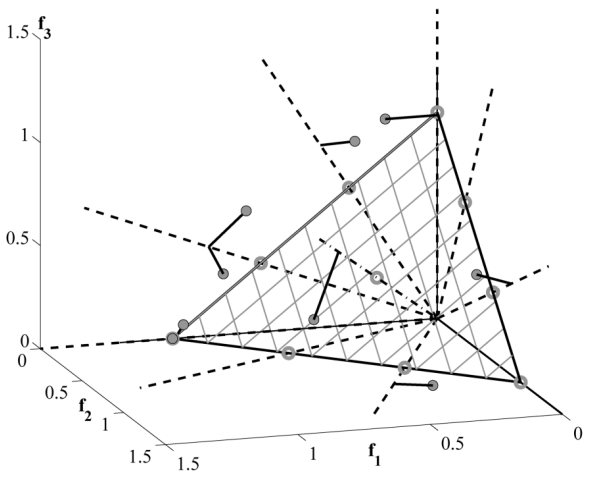


Figura 3.10: Associação dos membros da população com pontos de referencia ilustrada extraída de [26].

tação. Assim, a implementação do GA pode se resumir em implementar os mecanismos de seleção(Determinação dos pontos de referência em um hiper-plano, Normalização Adaptativa dos Membros da População, Operação de Associação e Operação de Preservação de Nicho), propostos por [26], discutidos a acima.

O diagrama da figura 3.12 demonstra a implementação do NSGA-III no Draco. Nela é importante ressaltar como o módulo NSGA-III utiliza o módulo NSGA-II como um a interface Go e, além dos métodos do NSGA-III discutidos acima, faz implementações apenas de métodos do NSGA-II que possuem algum tipo de diferença com NSGA-III.

Algorithm 3 `Associate`(S_t, Z^r) procedure

Input: Z^r, S_t
Output: $\pi(\mathbf{s} \in S_t)$, $d(\mathbf{s} \in S_t)$

for each reference point $\mathbf{z} \in Z^r$ **do**
    Compute reference line $\mathbf{w} = \mathbf{z}$
end for
for each $\mathbf{s} \in S_t$ **do**
    for each $\mathbf{w} \in Z^r$ **do**
        Compute $d^{\perp}(\mathbf{s}, \mathbf{w}) = \| \left(\mathbf{s} - \mathbf{w}^T \mathbf{s}\mathbf{w}/\|\mathbf{w}\|^2\right) \|$
    end for
    Assign $\pi(\mathbf{s}) = \mathbf{w} : \text{argmin}_{\mathbf{w} \in Z^r} d^{\perp}(\mathbf{s}, \mathbf{w})$
    Assign $d(\mathbf{s}) = d^{\perp}(\mathbf{s}, \pi(\mathbf{s}))$
end for

Figura 3.11: Algoritmo de preservação de nicho do NSGA-III extraído de [26]

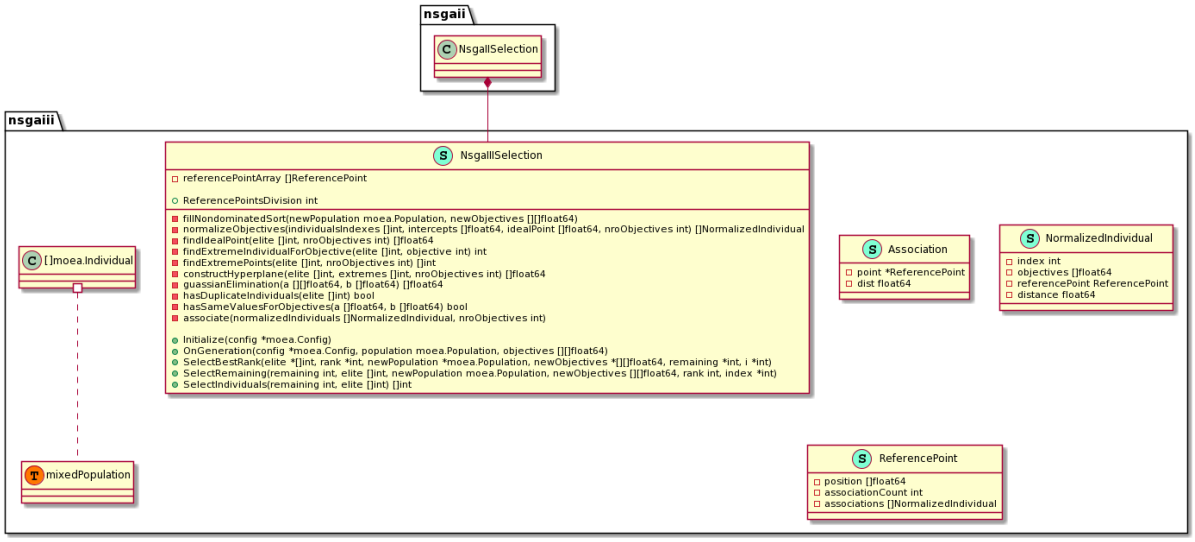


Figura 3.12: Diagrama UML do modulo NSGA-III no Draco

Capítulo 4

Resultados Experimentais

Neste capítulo é realizado o experimento para comparação de resultados das implementações dos algoritmos NSGA-II e NSGA-III na ferramenta Draco. Para isso, primeiro é descrito o cenário do experimento, depois são demonstrados os resultados obtidos.

4.1 Descrição do Cenário do Experimento

Esta avaliação empírica visa replicar o experimento realizado no trabalho [4] para o módulo NSGA-III, adicionado ao DCT, em comparação com o módulo NSGA-II, já existente na ferramenta. Esse estudo visa responder às seguintes questões:

1 Como a complexidade dos sistemas afeta a performance do módulo NSGA-III?

2 Como a performance do módulo NSGA-III se compara com a performance do módulo NSGA-II?

Responder a primeira pergunta permite entender se o módulo NSGA-III pode ser usado para clusterizar sistemas de software em situações reais. A segunda permite entender a performance do módulo NSGA-III em comparação ao módulo NSGA-II.

Para respondê-las foram utilizados as seguintes métricas: TS é o tempo, em segundos, decorrido da execução para clusterizar cada sistema estudado; MMC é o consumo máximo de memória, em KB, para clusterizar cada sistema estudado; e MQ é a métrica utilizada para estimar a qualidade de modularização dos clusters.

Para o NSGA-III foi configurado o número de divisões dos pontos de referência p discutido na sessão 3.2 com o valor 3.

Foi usada a ferramenta de *time* do Linux para calcular as duas primeiras métricas. Para a métrica MQ, ela foi recuperada dos resultados retornados dos dois módulos, recuperando o maior valor de MQ da população gerada para cada módulo. Foi utilizado um dataset

com 17 MDGs, 5 repetições de execução para cada um, sendo calculado o valor médio obtido para cada métrica. Os MDGs são todos sistemas open source que cobrem diferentes domínios de tamanho, indo de sistemas de tamanho pequeno até tamanho médio, em termo de linhas de código, esses podem ser vistos na tabela 4.1. Também foi firmado um limite de 48h para tempo máximo de execução.

Por fim o experimento foi excitado em um Intel(R) Xeon(R) E- 2124 CPU @ 3.30GHz com 32 GB de RAM, rodando em uma distribuição Ubuntu do Linux (18.04.4 LTS).

Sistema	Módulos	Deps.	KLOC	Commits
React Native Framework	190	1006	48	7842
Storm distributed realtime system	388	3249	213	7451
Bigbluebutton web conf. system	497	3661	82	13420
Minecraft Forge	501	3403	72	5498
CAS - Enterprise Single Sign On	513	1718	87	6268
Atmosphere Event Driven Framework	658	3523	41	5748
Druid analytics data store	668	2648	297	7452
Liquibase database source control	716	3981	77	5360
Kill Bill Billing Payment Platform	767	5422	139	5361
Actor Messaging Platform	768	7452	157	8772
The ownCloud Android App	833	3389	36	5329
Hibernate Object-Relational Mapping	836	2935	628	7302
jOOQ SQL generator	851	4118	133	5022
LanguageTool Style/Grammar Checker	871	1931	75	19121
Bazel build system	965	3813	375	7258
H2O-3 - Machine Learning Platform	1586	27725	143	19336
Jitsi communicator	2557	6742	326	12420

Tabela 4.1: Sistemas usados no estudo empírico

4.2 Avaliação dos Resultados

Nessa sessão os resultados do experimento empírico discutido na sessão anterior o serão expostos e serão fornecidas respostas para as perguntas introduzidas previamente.

4.2.1 Como a complexidade dos sistemas afeta a performance do módulo NSGA-III?

Nos resultados percebemos que o módulo NSGA-III do DCT encontramos para um MDG de tamanho pequeno com 190 módulos e 48 KLOC(*React Native Framework*) foi encontrado uma solução em média em 00:02:05, para um MDG de tamanho médio com 767 módulos e 139 KLOC(*Kill Bill Billing Payment Platform*) foi encontrado uma solução

em média em 00:24:44, e para um MDG de tamanho grande com 2557 módulos e e 326 KLOC(*Jitsi communicator*) foi encontrado uma solução em média em 09:25:20. Apesar disso configurar um crescimento exponencial, é possível argumentar que ainda pode ser utilizado em prática, principalmente para sistemas de tamanho pequeno ou médio. Para sistemas de tamanho grande, o módulo NSGA-III do DCT pode encontrar resultados em intervalos de algumas horas até intervalos de alguns dias(para sistemas muito grandes). Então como resposta a primeira pergunta temos:

- O estudo empírico sugere que é possível predizer o tempo necessário para o módulo NSGA-III do DCT encontrar uma solução através de uma função exponencial no número de módulos do sistema.
- No cenário mais longo do experimento, o módulo NSGA-III encontrou um resultado em 09:25:20. É possível afirmar que isso ainda é um tempo razoável para uma tarefa de clusterização multi-objetiva em um sistema com mais de 2500 módulos.

4.2.2 Como a performance do módulo NSGA-III se compara com a performance do módulo NSGA-II?

Às gráficos da figura 4.1 e as *boxplots* da figura 4.2 mostram as performances do módulos NSGA-III e NSGA-II do DCT, considerando tempo de execução(TS), consumo de memória(MMC), e qualidade de modularização. Pode-se observar que para o experimento proposto o módulo NSGA-II apresenta em geral resultados melhores em relação ao NSGA-III. Em relação ao consumo de memória pode-se observar que o NSGA-III teve um consumo de memória maior que o NSGA-II, enquanto ele obteve um consumo médio de memória de 1000MB o do NSGA-II foi de 800MB. Em relação aos valors obtidos do MQ o NSGA-II também demonstrou melhor performance, obtendo em média 92,77 como valor da métrica em comparação a 87.54 do NSGA-III. Por fim, tempo de execução foi a única em que o NSGA-III obteve uma performance melhor que o dois, mesmo que pequena a diferença, obtendo um tempo médio de execução de 3444 segundos em comparação ao NSGA-II que obteve 3526 segundos. Dessa forma a segunda pergunta pode ser respondida da seguinte maneira:

- Para o caso do experimento executado, o NSGA-II ainda apresenta uma performance melhor do que o NSGA-III. Principalmente no quesito de consumo de memória e valor obtido da métrica MQ.
- Apesar disso o NSGA-III teve uma performance equiparável com o NSGA-II, não tendo sido obtidos resultados com uma diferença muito significante entre os dois casos.

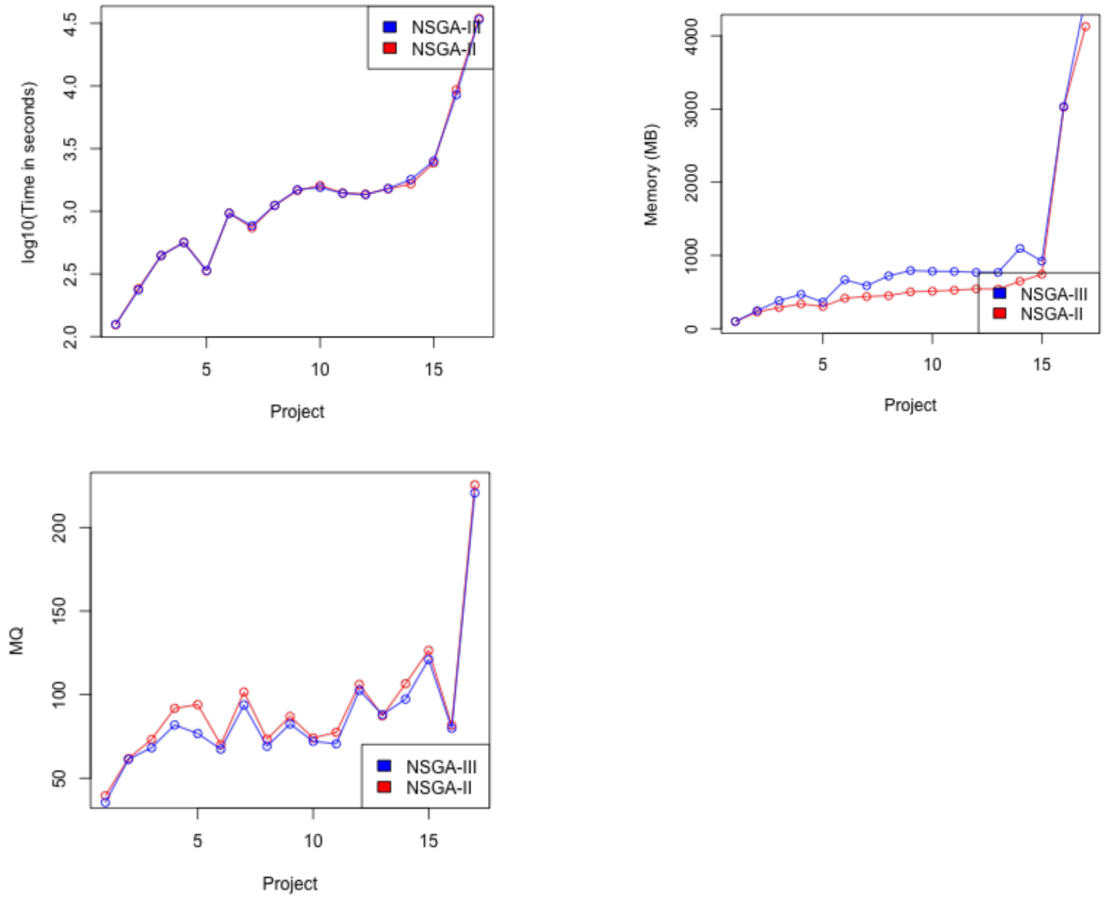
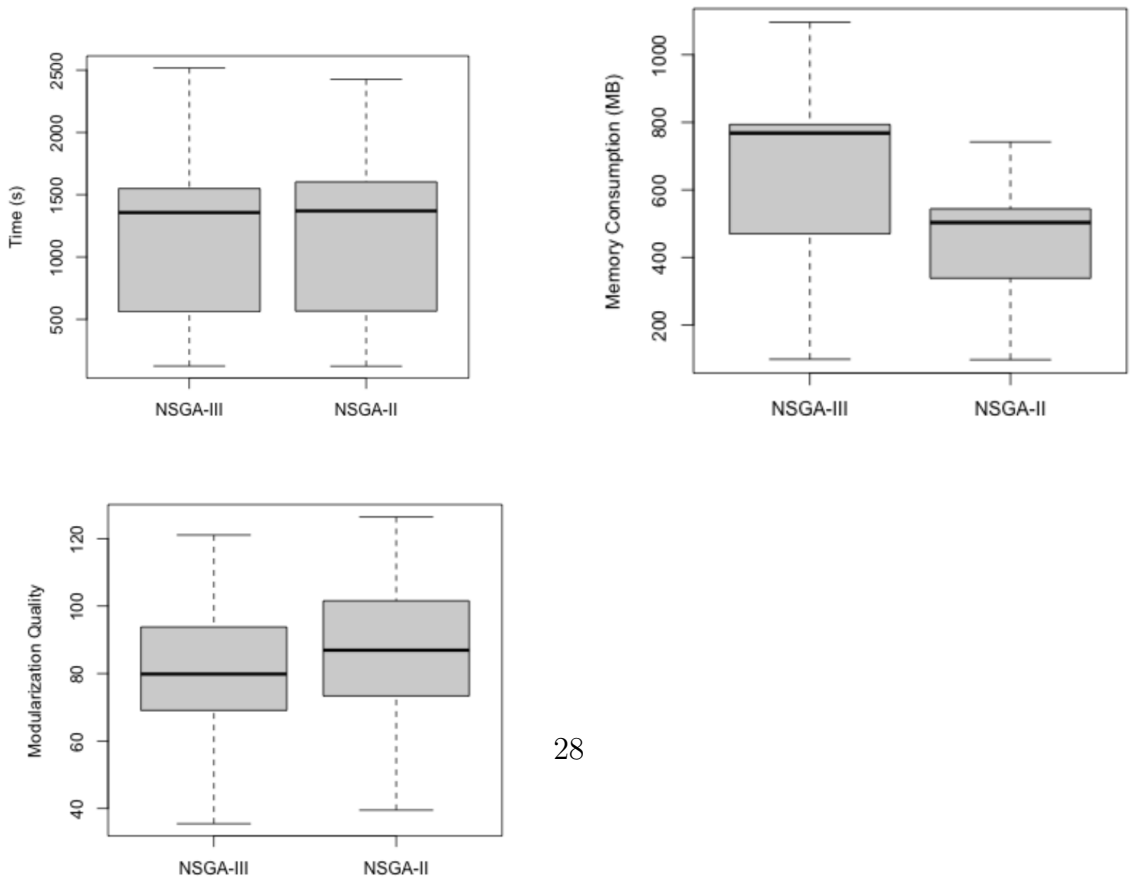

log10(Time in seconds)
Project
NSGA-III
NSGA-II
Memory (MB)
Project
NSGA-III
NSGA-II
MQ
Project
NSGA-III
NSGA-II

Figura 4.1


Time (s)
NSGA-III
NSGA-II
Memory Consumption (MB)
NSGA-III
NSGA-II
Modularization Quality
NSGA-III
NSGA-II

Capítulo 5

Conclusão

Este trabalho apresentou a implementação de um novo módulo a ferramenta DCT, o NSGA-III, para resolver o problema de clusterização de software e a submeteu aos mesmos experimentos realizados no trabalho de [4] para comparar sua performance com ao do módulo NSGA-II já implementado na ferramenta. Era esperado devido ao fato do DCT ser configurado para otimizar 5 objetivos, que o módulo NSGA-III obtivesse melhores resultados, principalmente em relação a métrica MQ, mas o trabalho conclui que apesar da performance do NSGA-III no experimento ser comparável ao NSGA-II, ela ainda é inferior. Isso pode se dar pelo fato dos objetivos otimizados apresentarem uma correlação alta entre si, sobretudo os objetivos de maximizar a metrica MQ, minimizar inter-conectividade e maximizar intra-conectividade. Para trabalhos futuros seria interessante realizar experimentos que pudessem demonstrar a performance do NSGA-III em comparação ao NSGA-II sobre um número maior de objetivos que de preferencia apresentassem menos correlação entre si, também poderia ser interessantes experimentações em relação ao parâmetro de número de divisões entre os pontos de referência que neste trabalho por recomendações de outros trabalhos[20] foi dado o valor 3, para trabalhos futuros outras configurações desse valor poderiam ser experimentadas.

Bibliografia

[1] Surender Singh Dahiya, Jitender Kumar Chhabra e Shakti Kumar. "Use of genetic algorithm for software maintainability metrics' conditioning". Em: *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*. 2007, pp. 87–92. DOI: 10.1109/ADCOM.2007.69.

[2] B. S. Mitchell e S. Mancoridis. "On the automatic modularization of software systems using the Bunch tool". Em: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 193–208.

[3] T.C. Lethbridge, J. Singer e A. Forward. "How software engineers use documentation: the state of the practice". Em: *IEEE Software* 20.6 (2003), pp. 35–39. DOI: 10.1109/MS.2003.1241364.

[4] A. P. M. Tarchetti et al. "DCT: An Scalable Multi-Objective Module Clustering Tool". Em: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2020, pp. 171–176. DOI: 10.1109/SCAM51674.2020.00024.

[5] K. Deb e H. Jain. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints". Em: *IEEE Transactions on Evolutionary Computation* 18.4 (2014), pp. 577–601.

[6] S. Mancoridis et al. "Using automatic clustering to produce high-level system organizations of source code". Em: (1998), pp. 45–52.

[7] Christian Artigues et al. "Mixed-Integer Linear Programming Formulations". Em: *Handbook on Project Management and Scheduling Vol.1*. Ed. por Christoph Schwindt e Jürgen Zimmermann. Cham: Springer International Publishing, 2015, pp. 17–41. ISBN: 978-3-319-05443-8. DOI: 10.1007/978-3-319-05443-8_2. URL: https://doi.org/10.1007/978-3-319-05443-8_2.

[8] David S. Johnson Michael R. Garey. Em: (1979).

[9] Mark Harman. *The Module Dependency Graph (MDG) after Clustering by the Two-Archive Algorithm*. URL: https://www.researchgate.net/figure/The-Module-Dependency-Graph-MDG-after-Clustering-by-the-Two-Archive-Algorithm_fig1_224111742. (accessed: 05.12.2021).

[10] Márcio O.Barros C.Monçores Adriana C.F.Alvim. "Large Neighborhood Search applied to the Software Module Clustering problem". Em: (2018), pp. 92–111.

[11] Jeno Lovesum Bright Gee Varghese R Kumudha Raimond. "A novel approach for automatic remodularization of software systems using extended ant colony optimization algorithm". Em: 41 (2019), pp. 107–120.

[12] B.S. Mitchell. "A Heuristic Search Approach to Solving the Software Clustering Problem". Em: (2002).

[13] Nicolas A. e Timothy C. Lethbridge. "Experiments with clustering as a software remodularization method". Em: (1999).

[14] J.H. Holland. Em: ().

[15] J.E. Smith A.E. Eiben. Em: ().

[16] *Recombinação(computação$_e$volutiva)*. URL: https://pt.wikipedia.org/wiki/Recombina%C3%A7%C3%A3o_(computa%C3%A7%C3%A3o_evolutiva). (accessed: 05.12.2021).

[17] Paul Calle. *NSGA-II explained!* URL: http://oklahomaanalytics.com/data-science-techniques/nsga-ii-explained. (accessed: 05.12.2021).

[18] Yuqing Chang. *NSGA-II algorithm flowchart*. URL: https://www.researchgate.net/figure/NSGA-II-algorithm-flowchart_fig2_283129353. (accessed: 05.12.2021).

[19] *Eficiência de Pareto*. URL: https://pt.wikipedia.org/wiki/Efici%C3%AAncia_de_Pareto. (accessed: 05.12.2021).

[20] Mohamed Wiem Mkaouer et al. "Many-objective software remodularization using NSGA-III". Em: *ACM Transactions on Software Engineering and Methodology* 24 (jan. de 2014). DOI: 10.1145/2729974.

[21] X. Yao V. Khare e K. Deb. "Performance scaling of multi-objective evolutionary algorithms". Em: *Proceedings of the 2nd International Conference on Evolutionary Multi-Criterion Optimization* 4 (2003), pp. 376–390.

[22] D. K. Saxena et al. "Objective Reduction in Many-Objective Optimization: Linear and Nonlinear Algorithms". Em: *IEEE Transactions on Evolutionary Computation* 17.1 (2013), pp. 77–99.

[23] Laila Ben Said et al. “Bioprotective Culture: A New Generation of Food Additives for the Preservation of Food Quality and Safety”. Em: *Industrial Biotechnology* 15 (jun. de 2019), pp. 138–147. DOI: 10.1089/ind.2019.29175.lbs.

[24] L. Ben Said, S. Bechikh e K. Ghedira. “The r-Dominance: A New Dominance Relation for Interactive Evolutionary Multicriteria Decision Making”. Em: *IEEE Transactions on Evolutionary Computation* 14.5 (2010), pp. 801–818.

[25] Q. Zhang e H. Li. “MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition”. Em: *IEEE Transactions on Evolutionary Computation* 11.6 (2007), pp. 712–731.

[26] Himanshu Jain e Kalyanmoy Deb. “An Improved Adaptive Approach for Elitist Nondominated Sorting Genetic Algorithm for Many-Objective Optimization”. Em: (2013). Ed. por Robin C. Purshouse et al., pp. 307–321.

[27] Indraneel Das e J. E. Dennis. “Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems”. Em: *SIAM J. on Optimization* 8.3 (mar. de 1998), pp. 631–657. ISSN: 1052-6234. DOI: 10.1137/S1052623496307510. URL: https://doi.org/10.1137/S1052623496307510.