RODRIGO GALVAN

UTILIZANDO ALGORITMOS DE BUSCA META-HEURÍSTICA PARA ESTABELECER SEQUÊNCIAS DE TESTE DE INTEGRAÇÃO PARA PROGRAMAS ORIENTADOS A ASPECTOS

Dissertação apresentada como requisito à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Silvia Regina Vergilio

CURITIBA

2010

RODRIGO GALVAN

UTILIZANDO ALGORITMOS DE BUSCA META-HEURÍSTICA PARA ESTABELECER SEQUÊNCIAS DE TESTE DE INTEGRAÇÃO PARA PROGRAMAS ORIENTADOS A ASPECTOS

Dissertação apresentada como requisito à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Silvia Regina Vergilio

CURITIBA

2010

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O uso de novos paradigmas, tais como os de orientação a objetos e a aspectos, trouxe novos desafios para o teste de software, atividade considerada fundamental na maioria das organizações de software. Estes desafios têm sido tema de pesquisa em diferentes trabalhos da literatura. No teste de integração de programas orientados a aspectos, similarmente ao teste de programas orientados a objetos, um problema que se destaca é o de estabelecer uma ordem de integração e teste para classes e aspectos que implique em um custo mínimo de criação de stubs. Algumas estratégias para este problema foram investigadas considerando um grafo específico para o contexto de programas orientados a aspectos que representa as relações de dependência entre classes e aspectos. Quando existem ciclos de dependência em tais grafos, o problema não é trivial e muitas estratégias levam à geração de soluções sub-ótimas. Outra desvantagem é que soluções baseadas em grafos geralmente não permitem o uso de diferentes medidas e fatores que podem influenciar no processo de criação de stubs tais como número de atributos, número de métodos, etc. Para reduzir estas limitações, este trabalho explora uma nova estratégia baseada em algoritmos de busca meta-heurísticos. Diferentes algoritmos de busca são investigados e comparados, além de diferentes funções objetivo que permitem lidar com objetivos múltiplos baseadas em agregação de funções e conceitos de dominância de Pareto.Os resultados obtidos nos experimentos mostram que os algoritmos introduzidos apresentam soluções com menor custo que as estratégias tradicionais.

ABSTRACT

Software test is considered a fundamental and challenging activity in most software organizations. The use of the object and aspect oriented paradigms to develop software has brought new test challenges, which are research subject of different works in the literature. In the integration test of aspect-oriented programs, similarly to the object-oriented software, a challenging problem is to establish an order to integrate and test classes and aspects that implies a minimal stubbing cost. Some strategies to this problem were investigated considering a specific graph to the aspect oriented context that represents dependency relations among classes and aspects. When dependency cycles exist in such graphs, the problem is not trivial and many strategies generate sub-optimal solutions. Other disadvantage is that graph based solutions, in general, do not allow the use of different measures and factors that influence the stubbing process. To overcome these limitations, this work explores a new strategy based on search based algorithms. Different search based algorithms are investigated and compared. In addition to this, different fitness functions are studied to allow a multi-objective approach based on aggregation of functions, and on Pareto dominance concepts. Evaluation results show that the introduced algorithms are capable to generate orders with lower costs than traditional strategies.

CAPÍTULO 1

INTRODUÇÃO

1.1 Contexto

À medida que os sistemas começaram a ficar maiores e mais complexos, e as empresas passaram a demandar cada vez mais o desenvolvimento de novos softwares, a comunidade científica passou a se preocupar também com atividades de garantia de qualidade. Assim a atividade de teste passou a ser essencial no desenvolvimento de sistemas. O objetivo principal da atividade de teste é revelar defeitos, sendo um objetivo secundário verificar se o sistema atende aos requisitos especificados [30]. Os dados que são construídos durante a atividade de teste são boas medidas que garantem confiabilidade e fornecem certa indicação da qualidade do sistema como um todo [30].

Esse aumento de complexidade dos sistemas fez com que novos paradigmas de programação surgissem. Entre eles está a Orientação a Objetos e mais recentemente a Orientação a Aspectos. Ambos paradigmas apresentam algumas especificidades para realização do teste, entretanto, a fase de teste de integração é uma das que mais exigem cuidado por parte do testador nos dois paradigmas.

O teste de integração de um sistema Orientado a Objeto ou Orientado a Aspecto é extremamente importante pois explora as interfaces entre os módulos procurando identificar defeitos de comunicação e garantir assim o funcionamento adequado do conjunto, além de analisar se módulos implementados separadamente executam suas tarefas da maneira esperada quando colocados juntos. Isso permite que a implementação dos componentes seja realizada em paralelo não causando nenhum dano ao produto final.

No teste de integração, as classes são integradas geralmente duas a duas. O fato de uma classe necessitar que outra esteja disponível para ela poder ser testada é chamado de relação de dependência. Indicando que as classes precisam umas das outras.

Assim, para o teste de uma classe dependente A é necessário que a classe B da qual A

depende esteja disponível. Essa dependência possui uma direção e é baseada em um ou mais relacionamentos. Quando uma das classes não está disponível, o teste de integração demanda a criação de um *stub* [12] que é uma unidade que substitui uma classe usada pela classe em teste. Ela simula o comportamento de uma unidade chamada pela classe que se está sendo testada e deve utilizar o menor custo computacional possível ou o mínimo de manipulação de dados possível. Essa situação está relacionada ao problema de estabelecer uma sequencia de teste de integração para as classes, (*CITO - Class Integration and Test Order*) [1] que se preocupa em determinar a ordem em que as classes devem ser integradas e testadas que corresponda a um custo para a construção de *stubs* que seja o mínimo possível.

Para resolver esse problema, abordagens utilizando grafos direcionados foram propostas [24, 36, 38]. Quando não existem ciclos de dependência, o problema CITO pode ser resolvido por uma simples inversão topológica das classes baseadas nas suas dependências. Entretanto quando ciclos estão presentes a solução é mais difícil. Mas um estudo recente [26] mostra que esses ciclos de dependência são extremamentes comuns em sistemas reais.

A maioria dos trabalhos baseados em grafos existentes visa a identificar os ciclos fortemente conexos (CFC) recursivamente, e em cada CFC remover a dependência que maximiza o número de ciclos quebrados [7], ou seja, o número de *stubs* a serem criados. Entretanto, o custo relacionado à construção de um *stub* pode depender de diversos fatores, como por exemplo, o número de atributos de uma classe e o número de chamadas de métodos, e não pode ser completamente medido ou estimado [7]. Muitas vezes é extremamente difícil, ou até impossível, a adaptação da maioria das soluções baseadas em grafos para utilizarem os fatores citados [6].

Para reduzir estas limitações o trabalho proposto por Briand et al [6] utiliza Algoritmos Genéticos e apresenta resultados melhores quando comparados a outras abordagens baseadas em grafo. Entretanto este trabalho utiliza uma função de *fitness* que agrega diferentes objetivos para estimar se uma determinada solução é boa ou não. Mais recentemente, Cabral et al. [9] propuseram um tratamento multi-objetivo para o problema

CITO baseado nos conceitos de dominância de Pareto [28]. O trabalho apresenta resultados promissores e permite tratar o problema CITO.

No contexto de programas Orientados a Aspectos, o teste de integração também apresenta a mesma importância e os desafios mencionados. Analogamente ao problema CITO, um problema semelhante existe que é estabelecer uma sequência de integração para classes e aspectos que será chamado neste trabalho de CAITO *(Class and Aspects Integration Test Order)*.

Alguns autores têm dedicado atenção ao tema, como por exemplo Ceccato et al [10] que utilizam uma estratégia em que as classes são testadas primeiro sem a integração dos aspectos. Após isso, os aspectos são integrados e testados com as classes, e ao final as classes são testadas na presença dos aspectos. Ré et al [31] propõem uma extensão do grafo utilizado em Orientação a Objetos para contemplar as relações presentes em sistemas Orientados a Aspectos e realizam estudos de diferentes estratégias baseadas em grafos [32].

Porém no contexto de programas Orientados a Aspectos os algoritmos de busca meta-heurística ainda não foram explorados. Como a maioria das estratégias baseadas em grafos apresentam limitações, como por exemplo, a geração de soluções sub-ótimas por não analisarem as consequências futuras da quebra de uma dependência quando se está resolvendo o problema, abordagens de busca meta-heurística se apresentam como boa alternativa para tratamento do problema CAITO. Os bons resultados obtidos no contexto de teste de software Orientado Objetos servem como motivação para explorar estes algoritmos no contexto de teste de software Orientado a Aspectos.

1.2 Objetivos

Dado o contexto acima o objetivo do trabalho é explorar o uso de algoritmos meta-heurísticos no contexto do teste de integração, para estabelecer sequências de teste de classes e aspectos usando a relação do custo de construção de *stubs*. A utilização desses algoritmos permitirá que não só o número de *stubs* seja considerado mas também outros objetivos, fatores associados a complexidade dos *stubs* a serem construídos tais

como número de métodos e atributos. Diferentes algoritmos de busca meta-heurística são investigados e comparados além de diferentes funções objetivo que permitem lidar com objetivos múltiplos baseados em agregação de funções e conceitos de dominância de Pareto [28]. Os resultados obtidos nos experimentos mostram que estes algoritmos apresentam melhores soluções que as estratégias baseadas em grafos.

1.3 Organização

Este trabalho está organizado em quatro capítulos da seguinte forma. O Capítulo 2 apresenta as definições relacionadas aos algoritmos de busca meta-heurística utilizados, descrevendo seus funcionamentos e pseudo-códigos. O Capítulo 3 faz uma introdução ao problema CITO e CAITO, e apresenta, os trabalhos relacionados. O Capítulo 4 apresenta a proposta desse trabalho com a adaptação dos algoritmos para o problema CAITO e análise dos resultados obtidos. No Capítulo 5 são apresentadas as conclusões e propostas de trabalho futuro. O trabalho apresenta um apêndice (Apêndice A) no qual se encontram os grafos e matrizes de dependência correspondentes aos relacionamentos existentes entre classes e aspectos dos sistemas utilizados no experimento descrito no Capítulo 4.

CAPÍTULO 2

ALGORITMOS META-HEURÍSTICOS

Os algoritmos heurísticos têm se destacado como abordagens promissoras para solução de diversos problemas do mundo real.

Praticamente todos os problemas NP-completos ou NP-difíceis têm uma versão de solução heurística que usa um algoritmo deste tipo [20]. Métodos ditos meta-heurísticos são métodos que podem lidar com qualquer problema de otimização, não estando atrelados a um problema específico necessariamente. Uma meta-heurística é um conjunto de conceitos que podem ser utilizados para definir métodos heurísticos aplicáveis a um extenso conjunto de problemas. As meta-herísticas podem ser divididas em [5]:

1. Meta-heurísticas de busca por entornos;
2. Meta-heurísticas de relaxação;
3. Meta-heurísticas construtivas;
4. Meta-heurísticas evolutivas;

As meta-heurísticas de busca por entornos percorrem o espaço de busca levando em conta, fundamentalmente, a *vizinhança* da solução em mãos, definida como o conjunto de soluções que podem ser obtidas a partir da aplicação de algum operador à solução atual, por exemplo, a Busca Tabu [17] como será visto adiante. As meta-heurísticas de relaxação simplificam o problema e utilizam a solução encontrada como guia para o problema original, por exemplo, Relaxação Lagrangeana [20] que remove algumas restrições de um problema de programação linear, atribui um peso (multiplicador de Lagrange) a cada uma delas e altera a função objetivo para penalizar as soluções que seriam inviáveis no problema original.

As meta-heurísticas construtivas definem de forma meticulosa o valor de cada componente da solução, por exemplo, o *GRASP (Greedy Randomized Adaptive Search Proce-*

dure) [20] que cada iteração é composta por uma fase construtiva e uma fase de busca por entornos. Em cada passo da fase construtiva selecionam-se os componentes que causam melhor efeito se adicionados à solução atual. Acrescenta-se um desses elementos (selecionado aleatoriamente) à solução. Finalmente as meta-heurísticas evolutivas que lidam com uma população de soluções, que evolui, principalmente, através da interação entre seus elementos, por exemplo, Algoritmos Genéticos [21] como será visto adiante.

Para a resolução do problema CAITO diferentes tipos de algoritmos meta-heurísticos foram utilizadas, implementados e comparados. São elas: NSGA-II [11], PACO [9] e MOTS [17], elas são descritas em detalhes neste capítulo. Outra questão observada é que o problema CAITO é multi-objetivo, alguns conceitos sobre otimização multi-objetivo são primeiramente introduzidos.

2.1 Problemas Multi-objetivo

Técnicas tradicionais de otimização exigem uma avaliação global única, porém, muitas vezes, os problemas no mundo real possuem restrições e variáveis conflitantes, possuindo múltiplos objetivos a serem atingidos. Esses problemas são conhecidos como:

1. Problemas de Otimização Multi-Objetivo (MOO),
2. Problemas de tomada de decisão multi-critério (MCDM)

Muitas técnicas foram criadas e aplicadas para tentar lidar com esse tipo de situação. Diz-se que um problema é multi-objetivo quando o mesmo depende de mais de um critério de avaliação (função objetivo) para determinar a qualidade de uma potencial solução para o problema [35]. Por exemplo, o problema de se minimizar o custo e maximizar a qualidade de um produto pode ser dito multi-objetivo. Ao se tentar melhorar apenas um dos objetivos na maioria das vezes não se obtém um resultado satisfatório, ou seja, como no exemplo fornecido, ao se minimizar ao máximo o custo de um produto muitas vezes acaba-se criando um produto sem qualidade, e vice-versa, a otimização de um único objetivo pode fazer com que os outros objetivos acabem não sendo satisfeitos.

Uma maneira de se tratar o problema com muitos objetivos é a agregação desses objetivos em uma única função objetivo. Essa abordagem, define uma função que consiste na soma ponderada dos objetivos. Essa abordagem é simples de implementar e pode ser usada em técnicas tradicionais de otimização, porém, na prática apresenta um fraco desempenho pois exige aplicação de conhecimentos específicos do problema para a especificação ótima dos pesos da soma ponderada dos objetivos além de necessitar que as avaliações sejam normalizadas para a soma de valores comparáveis entre si.

Assim, para um problema multi-objetivo a relação de dominância de Pareto ($\preceq$) [35] aparece como uma abordagem mais interessante pois leva em consideração os diversos objetivos separadamente. Essa relação serve para comparar as soluções de um determinado problema da seguinte maneira. Uma solução x^1 e dita dominar uma solução x^2 se ambas as condições a seguir forem satisfeitas:

1. A solução x^1 não é pior que a solução x^2 em nenhum dos objetivos, ou seja, $fm(x^1) \leq fm(x^2)$ para todo $m = 1, ..., M$.

2. A solução x^1 e estritamente melhor que a solução x^2 em pelo menos um objetivo, ou seja, $fm(x^1) < fm(x^2)$ para algum $m \in 1, ..., M$.

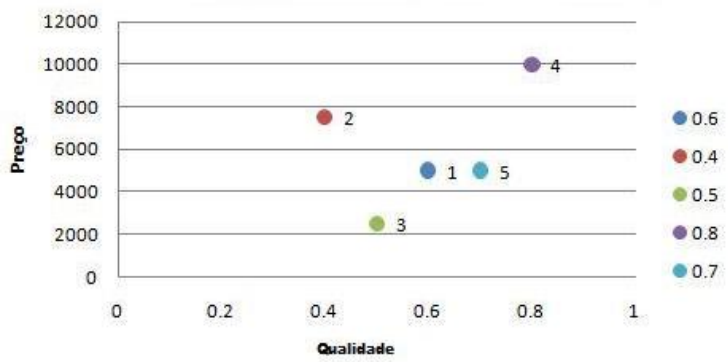

Figura 2.1: Relação Preço X Qualidade

Assim, para o exemplo da Figura 2.1 que representa a relação de custo e qualidade de um determinado produto, temos para soluções conflitantes que a alternativa 4 possui maior custo e maior qualidade; a alternativa 3 possui menor custo; a alternativa 5 é melhor que 1 e 2 ($5 \preceq 1 \preceq 2$) e a alternativa 3 é melhor que 2 ($3 \preceq 2$).

O conjunto de soluções não dominadas formam o *conjunto Pareto-ótimo* dado por $PO = \{3, 4, 5\}$ para o exemplo acima. A Fronteira de Pareto [28] é formada pelos vetores das funções objetivo $F(x)$ tal que x pertence ao *conjunto Pareto-ótimo*. A Figura 2.2 mostra a fronteira formada com os dados da Figura 2.1.

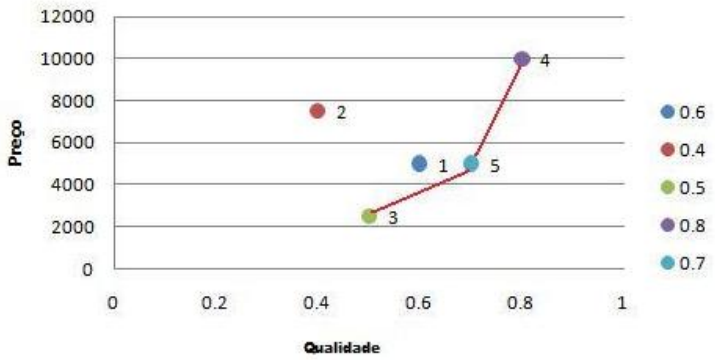

Figura 2.2: Fronteira de Pareto

2.2 Algoritmos Genéticos

Algoritmos Genéticos fazem parte de um ramo da ciência da computação, denominado Computação Evolutiva (CE) [27], que utiliza os princípios dos mecanismos evolutivos encontrados na natureza, esses diretamente relacionados com a teoria da evolução de Darwin que afirma que a vida na Terra é o resultado de um processo de seleção feito pelo meio ambiente em que somente os mais adaptados possuirão chances de sobreviver e reproduzir-se.

Os Algoritmos Genéticos começaram a ser desenvolvidos no início dos anos 50, quando biólogos e cientistas dessa área começaram a utilizar computadores e algoritmos em simulações de sistemas biológicos. Porém, os Algoritmos Genéticos da maneira como são conhecidos hoje, ou seja, como algoritmos que realizam uma busca multi direcional em um espaço de soluções representada por indivíduos de uma população que nada mais são que possíveis soluções para um determinado problema, foram definidos seguindo a idéia de John Holland no final dos anos 60 e início dos anos 70 [21]. Como apresentado em [27] basicamente a estrutura de um algoritmo da classe CE pode ser definida como o Algoritmo 1 [27]:

Algoritmo 1 Algoritmo CE

```
1: t := 0
2: inicia população P(t)
3: calcula fitness P(t)
4: t = t + 1
5: se critério de parada alcançado então
6:     vá para passo 12
7: fim se
8: selecione P(t) de P(t-1)
9: crossover P(t)
10: mutação P(t)
11: vá para passo 3
12: retorne melhor e termine
```

O algoritmo mantém uma população de indivíduos $P(t) = \{x_1^t, ..., x_n^t\}$ por iteração t. Cada indivíduo representa uma potencial solução para o domínio do problema corrente. Cada solução x_i^t é avaliada para nos dar uma medida de seu *fitness* (valor de adaptação do indivíduo). Então, uma nova população (iteração $t+1$) é criada selecionando os indivíduos com as melhores medidas de *fitness* (selecionar $P(t)$ de $P(t-1)$). Alguns indivíduos da nova população sofrem modificações (crossover P(t) e mutação P(t)), essas modificações são realizadas através de operadores genéticos, como por exemplo, a mutação. Essa operação é unária e cria novos indivíduos a partir de pequenas modificações no indivíduo que sofre a mutação. Outro operador de uma ordem mais poderosa de modificação é o operador *crossover* (ou cruzamento). Cada indivíduo criado por este operador é resultado da combinação de diversos indivíduos da população.

Após algumas iterações o algoritmo tende a convergir para soluções melhores muito próximas da solução ótima.

Cada técnica pertencente a CE pode utilizar diferentes estruturas de dados para representar os indivíduos de suas populações, o que obriga que os operadores genéticos utilizados em cada técnica também apresentem diferenças. A ordem em que a seleção e o processo de diversificação (alterar $P(t)$) são executados também pode ser diferente [40] de acordo com a técnica utilizada. Por exemplo, quando se utilizam estratégias de evolução, a população e seus indivíduos são primeiramente modificados para depois sofrer a seleção. Mesmo quando consideram-se as soluções em uma técnica em particular ainda existem

modificações e ajustes de acordo com os requisitos e necessidades do problema. Isso é observado no método de seleção dos indivíduos escolhidos para reprodução ou alteração genética que podem ser feitos por uma seleção proporcional na qual a probabilidade de seleção de um indivíduo é proporcional ao valor de *fitness* do indivíduo; ou métodos de seleção por ranqueamento no qual todos os indivíduos da solução são ordenados do melhor para o pior indivíduo e a probabilidade de seleção é fixada por todo o processo de evolução. Finalmente, o método de seleção por torneio no qual um determinado número de indivíduos competem entre si para serem selecionados para a próxima geração, sendo essa disputa realizada tantas vezes quanto for o tamanho da população.

As estruturas de dados utilizadas em problemas específicos juntamente com o conjunto de operadores genéticos escolhidos representam os componentes essenciais de qualquer Algoritmo evolutivo. Essa dupla é o que permite a distinção entre os diversos paradigmas dos métodos desses algoritmos.

2.3 Non-dominated Sorting Genetic Algorithm - NSGA-II

Com o passar do tempo diversos algoritmos multi-objetivos foram implementados e pesquisados pelo meio científico. O *Non-dominated Sorting Genetic Algorithm (NSGA)* [34] foi um dos primeiros algoritmos multi-objetivo evolutivo. Em casos de problemas com populações grandes o algoritmo muitas vezes se tornava computacionalmente caro ou inviável. Esse fato levou ao desenvolvimento de uma extensão dessa solução chamada de NSGA-II [11]. A principal mudança foi a inclusão de elitismo. Elitismo é o procedimento em que os melhores pontos encontrados são preservados na população. Na prática isto resulta numa busca mais agressiva e geralmente bastante efetiva. No entanto existe o perigo de uma convergência prematura para mínimos locais.

O pseudo-código do algoritmo NSGA-II pode ser visto no Algoritmo 2. Inicialmente o algoritmo gera uma população aleatória P_0, com tamanho N, ou seja, $|P_0| = N$. Utilizando os operadores genéticos de mutação (p_m), cruzamento (p_c) e seleção por torneio (S) a primeira população filha é gerada Q_0 ($|Q_0| = N$). As populações formadas por P_0 e Q_0 são unidas gerando assim uma nova população R_0 de tamanho $2N$ ($|R_0| = 2N$).

Nas iterações que se seguem, dadas no algoritmo por $n = 1, 2, 3, \ldots, max\_iterations$ o algoritmo passa a trabalhar com a população R_n que é ordenada utilizando o conceito de dominância, ou seja, é ordenada por soluções não dominadas. Inicialmente todas as soluções são colocadas em um conjunto *front* F_0. Depois disso os pontos de F_0 são descartados do conjunto de soluções global e um novo conjunto de soluções não dominadas formam então um novo conjunto *front* F_1 e assim por diante. A Figura 2.3 mostra o esquema de seleção dos indivíduos no NSGA-II.

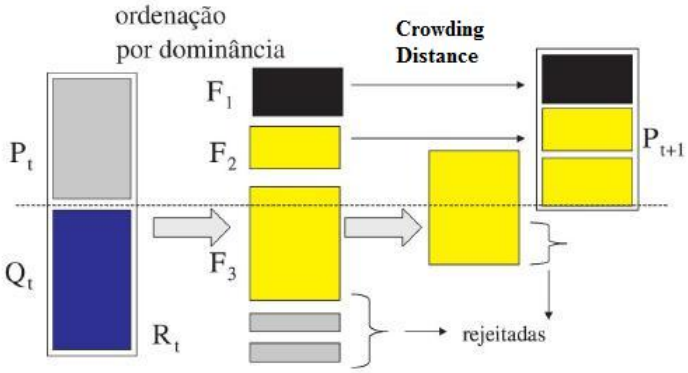

Figura 2.3: Esquema de seleção dos indivíduos no NSGA-II extraído de [11]

Para formar uma nova população, a população P_{n+1} é gerada com as soluções dispersas de acordo com um valor de *Crowding Distance*. Isso é feito através da função `CalculateCrowdingDistances` [11] aplicada a cada ponto em F_j.

Essa função representa uma estimativa do perímetro formado pelo cubóide cujos vértices são os seus vizinhos mais próximos, ou seja, para uma solução i temos a sua *Crowding Distance* dada por d_i conforme visto no Algoritmo 3, e ilustrado na Figura 2.4.

Assim, depois da seleção dos N pontos pertencentes a P_{n+1}, essa população é então ordenada usando o operador $\geq_n$ que é definido pela relação de dominância e *Crowding Distance*. Finalmente, os operadores genéticos são aplicados em P_{n+1} para formar uma nova população Q_{n+1} e as iterações continuam.

Algoritmo 2 Algoritmo do NSGA-II

1: $P_0 = Q_0 = \emptyset$ {Initialize populations P_0 and Q_0}
2: $n = 0$ {Initialize number of generations}
3: Tournament Selection
4: Crossover
5: Mutation
6: Generate population Q_0
7: **enquanto** $n < max\_iterations$ **faça**
8: $R_n = P_n \cup Q_n$
9: Sort R_n by non-dominance
10: $P_{n+1} = \emptyset$
11: **enquanto** $|P_{n+1}| \leq N$ **faça**
12: `CalculateCrowdingDistances` for F_j
13: $P_{n+1} = P_{n+1} \cup F_j$
14: **fim enquanto**
15: Sort P_{n+1} using the $\geq_n$ operator
16: $P_{n+1} = P_{n+1}[0 : (N-1)]$ {Choose N first elements of P_{n+1}}
17: Crossover
18: Mutation
19: Generate population Q_{n+1}
20: $n++$
21: **fim enquanto**

Algoritmo 3 Medida de Crowding-distance

1: **função** CalculateCrowdingDistance(F_j, I)
2: F_j : conjunto de soluções na fronteira i
3: I = número de soluções em F_j
4: Para cada solução em F_j , $d_i = 0$
5: Para cada função objetivo $m = 1, 2, .., M$
6: Ordena decrescentemente as soluções por f_m na lista I^m
7: Para cada solução extrema(min e max) em cada um dos M objetivos
8: $d_{I_1^m} = d_{I_1^l} = \infty$
9: **para** $i < -2...l - 1$ **faça**
10:

$$d_{I_i^m} = d_{I_i^m} + \frac{f_m^{I_{i+1}^m} - f_m^{I_{i-1}^m}}{f_m^{max} - f_m^{min}}$$

11: **fim para**

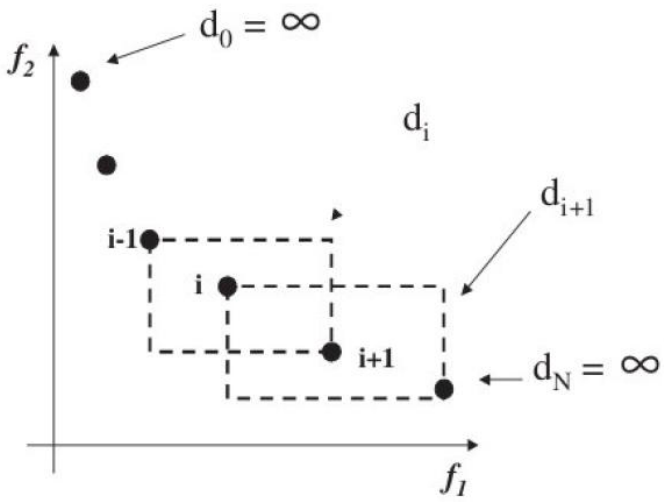

Figura 2.4: Medida de Crowding-distance

2.4 PACO - Pareto Ant Colony

O algoritmo Ant Colony Optimization (ACO) foi inicialmente apresentado por [14]. Esse algoritmo é inspirado no comportamento real na natureza de colônias de formigas. Principalmente na inteligência coletiva que as formigas utilizam para procurar comida e retornar ao ninho.

Uma das principais idéias é a comunicação indireta realizada entre os indivíduos de uma colônia de formigas, ou seja as formigas. Essa comunicação é realizada através de substâncias químicas, chamadas de feromônio, sendo que a medida que uma formiga percorre um trajeto em busca de comida, elas depositam essa substância utilizando como guia até a comida e volta para ao ninho, e muitas vezes, as formigas acabam encontrando o menor caminho de ida e volta baseadas nessas informações. O caminho artificial de feromônio, usado no ACO, é uma espécie de distribuição numérica, a qual pode ser modificada pelas formigas conforme a experiência por elas acumulada enquanto resolvem um problema específico.

Considerando um problema de otimização combinatorial, uma formiga constrói iterativamente uma solução, ou seja, a formiga "procura pelo menor caminho até a comida". A construção desse caminho, solução, dá-se a cada passo através de uma distribuição probabilística que seria o correspondente aos caminhos de feromônio deixados por formigas

reais.

Após a solução estar completa, ou seja, caminhos até a comida estarem prontos, as trilhas de ferômonio são atualizadas de acordo com a qualidade da melhor solução construída, ou seja, a experiência das formigas passa a ser utilizada na construção de novas soluções. Elas passam a partilhar uma estrutura em comum chamada de matriz de ferômonio compartilhada.

Para problemas multi-objetivo, que são de fundamental importância para este trabalho, o algoritmo Pareto Ant Colony (PACO), originalmente proposto para a resolução do problema Multiobjective Portfolio Selection [13] , utiliza essa estrutura do ACO um pouco modificada.

O PACO funciona utilizando K matrizes de feromônio, nas quais K é o número de objetivos existente no problema, e usa uma função de agregação heurística calculada a partir desses K objetivos. A regra de transição usada para a escolha de um operador de mutação j é dada pela Equação 2.1, onde k é um contador.

$$j = \begin{cases} \text{argmax}_{j \in U} \left[\sum_{k=1}^{K} p_k \cdot \tau_{ij}^k \right]^{\alpha} \cdot \eta_{ij}^{\beta} & \text{if } q \leq q_0 \\ p\left(j\right) & \text{caso contrário} \end{cases} \tag{2.1}$$

Onde $p(j)$ é dada pela probabilidade definida pela Equação 2.2.

$$p(j) = \begin{cases} \dfrac{\left[\sum_{k=1}^{K} p_k \cdot \tau_{ij}^k \right]^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in U} \left[\sum_{k=1}^{K} p_k \cdot \tau_{ij}^k \right]^{\alpha}} & \text{if } j \in U \\ p\left(j\right) = 0 & \text{caso contrário} \end{cases} \tag{2.2}$$

As matrizes de feromônio são representadas por τ^k. Essa regra é uma extensão para múltiplas matrizes de feromônio da regra usada pelo ACO [14] conforme pode ser visto em [29]. Nessa fórmula o valor U representa o conjunto dos índices da matriz de feromônio.

Os parâmetros α e β determinam a influência relativa do feromônio e a informação heurística respectivamente. Para o objetivo k, a informação heurística relacionada é dada por η_{ij}^k, $p^k \in [0, 1]$, são os pesos que são distribuídos uniformemente tais que $\sum_{k=1}^{k} = 1$.

Em cada iteração do algoritmo, a toda formiga é atribuído um vetor de pesos. O

parâmetro q é um número aleatório pertencente ao intervalo [0, 1] e $q_0 \in [0, 1]$ é o parâmetro que define as propriedades de intensificação e diversificação do algoritmo.

O valor do feromônio local é atualizado sempre que uma formiga segue de um ponto i para outro ponto j na trilha, e utiliza a seguinte regra: $\tau_{ij}^k = (1 - \rho) \cdot \tau_{ij}^k + \rho \cdot \tau_0$, na qual ρ é a classificação de evaporação do feromônio e τ_0 é o valor inicial desse feromônio.

Já o feromônio global é atualizado quando cada formiga da população já construiu seu caminho, ou seja, a solução, e já utilizou a regra $\tau_{ij}^k = (1 - \rho) \cdot \tau_{ij}^k + \rho \cdot \Delta\tau_{ij}^k$, na qual $\Delta\tau_{ij}^k$ recebe o valor: 15 se o componente (i, j) pertence à melhor ou segunda melhor solução (caminho); 10 se (i, j) pertence apenas à melhor solução (caminho); 5 se (i, j) pertence apenas à segunda melhor solução (caminho); e 0 nos demais casos.

Algoritmo 4 Algoritmo do PACO

```
1: Initialize pheromone (F1, F2, t0)
2: enquanto nriter ≤ max_iterations faça
3:     para todo ant in Ants faça
4:         p1 = rand (0, 1)
5:         p2 = 1 − p1
6:         q = rand (0, 1)
7:         q0 = rand (0, 1)
8:         TakeInitialCandidates = ()
9:         s = GenerateInitialPath ()
10:        BuildPath s = (s, q, q0, p1, p2, F1, F2)
11:        LocalPheromoneUpdate (s, F1, F2)
12:        s = LocalSearch ()
13:        s' = LocalSearch (s)
14:        b = BestIteration ()
15:        b' = SecondBestIteration ()
16:    fim para
17:    GlobalPheromoneUpdate (b, b', F1, F2)
18:    ParetoSetUpdate (P, s, s')
19:    nriter+ = 1
20: fim enquanto
```

O pseudo-código do PACO é apresentado no Algoritmo 4. Uma função inicial é executada inicializando as matrizes de feromônio $F1$ e $F2$ de cada objetivo para t_0. Depois disso, o algoritmo executa um laço no qual a cada iteração, todas as formigas constroem um conjunto de soluções s e atualizam o seu feromônio através da função (`LocalPheronomeUpdate`).

Depois, buscas locais são realizadas para se alcançar uma maior exploração do espaço de busca. Para cada objetivo, essa busca é executada em cada formiga (`LocalSearch`).

Finalmente, todas as formigas são avaliadas e a atualização global do feromônio é realizada (`GlobalPheromoneUpdate`) baseada nas melhores soluções da iteração. As melhores soluções encontradas até aquele momento (Aproximação da Fronteira de Pareto) são armazenadas e atualizadas (`ParetoSetUpdate`). No final da iteração do laço principal as soluções armazenadas nessa aproximação da Fronteira de Pareto, são as soluções obtidas pelo algoritmo.

2.5 MOTS - Multi-Objective Tabu Search

A busca Tabu foi uma abordagem inicialmente proposta por Glover [17], tendo como principal objetivo, permitir o uso do algoritmo Hill climbing, que é um método de busca (local) que utiliza um procedimento de melhora iterativa (iterative improvement) e ao mesmo tempo evitar locais ótimos, ou seja, aumentar a diversificação da busca. A principal característica da busca Tabu é que o algoritmo sempre busca por outros locais no espaço de busca sempre que um local ótimo é encontrado, isso é feito através de uma lista de movimentos proibidos (lista Tabu) que não permite determinados movimentos na busca de soluções no espaço de busca. Essa lista impossibilita que soluções já previamente visitadas sejam acessadas novamente através de sistema de memória que mantém a história recente dessa busca.

Para este trabalho, é importante o entendimento do algoritmo Multi-objetivo de Busca Tabu. Ele foi proposto em [4] e foi chamado de algoritmo MTabu (Multiobjective optimization using Tabu search), esse algoritmo percorre sobre a vizinhança das soluções utilizando o conceito de Pareto. Para seu funcionamento as seguintes listas são necessárias:

- *candidateList* - Lista com todos os vizinhos não dominados da solução atual.
- *paretoList* - Lista que contém todas as soluções não dominadas já exploradas.
- *tabuList* - Lista que mantém todas as soluções já exploradas.

O algorimto MTabu, é apresentado pelo Algoritmo 5. Ele começa com a geração de uma solução aleatória (`SeedSolution`). Essa solução, *currentSolution*, é expandida e a

Algoritmo 5 Algoritmo MTabu.

```
1: Initialize numberseed, candidateList, paretoList, tabuList;
2: enquanto numberseed < maxseed faça
3:     counter = 0;
4:     currentSolution = SeedSolution(numberseed);
5:     enquanto counter < max_iterations faça
6:         para i = 1 to neighborhoodSize faça
7:             candidate = neighbor(currentSolution);
8:             se (not contains(tabuList, candidate)) então
9:                 Evaluate(candidate);
10:                Update(candidateList, candidate);
11:            fim se
12:        fim para
13:        Update(tabuList, currentSolution);
14:        Update(paretoList, currentSolution);
15:        currentSolution = randomlySelectedCandidate(candidateList);
16:        counter =+ ;
17:    fim enquanto
18:    numberseed =+ ;
19: fim enquanto
```

vizinhança é mantida na lista de candidatos denominada *candidateList*. Então as listas de pareto, *paretoList*, e tabu, *tabuList*, são atualizadas juntamente com a *currentSolution*. Esses passos são realizados em uma iteração do laço principal do algoritmo. Esse laço é repetido tantas vezes quanto for o valor definido de iterações do algoritmo, *max_iterations*. Um laço externo controla o número de diferentes soluções aleatórias geradas (*maxseed*) a serem exploradas.

2.6 Comparações entre algoritmos

Para comparar algoritmos multi-objetivo podem-se utilizar alguns indicadores já existentes na literatura, tais como, *Dominance Rank* e *Indicadores de qualidade* (*Hypervolume*, *Epsilon* e *R*) [23].

O *Dominance Rank* funciona da seguinte forma. Para se comparar a qualidade dos conjuntos de aproximações gerados por q algoritmos multi-objetivos, com $q \geq 2$, cada algoritmo $i \in 1, ..., q$ com $r_i \geq 1$ número de execuções realizadas que geram $A_1^1, A_2^1, ..., A_{r1}^1, ...A_1^q, ..., A_{r_q}^q$ conjuntos de aproximações.

O conjunto C, formado por todos os conjuntos de aproximações, apresenta conjuntos que dominam outros conjuntos em C. Assim, pode-se obter medidas para cada conjunto

de tal forma que se pode comparar a relação de dominância existente entre cada conjunto. Cada conjunto de aproximações está associado a um valor de *ranking*, que pode ser definido de diversas formas, como o total de conjuntos pelo qual um determinado conjunto é dominado [15] ou então realizando uma ordenação de não-dominância no conjunto C [18]. Mas uma melhor definição desse *ranking* é dada pela Equação (2.3) na qual $\lhd$ representa a relação de não-dominância:

$$rank(C_i) = 1 + |C_j \in C : C_j \lhd C_i| \tag{2.3}$$

.

Como citado anteriormente, existem também os indicadores de qualidade, formalmente definidos com um I que é um mapeamento de um conjunto de aproximações Ω para um conjunto de números reais dados pela Equação (2.4).

$$I : \Omega \longmapsto \Re \tag{2.4}$$

Supostamente a ordem que I estabelece em Ω representa a qualidade dos conjuntos. Assim, dado um par de conjuntos de aproximações, A e B, a diferença entre os seus respectivos indicadores $I(A)$ e $I(B)$ deve ser a diferença da qualidade desses dois conjuntos. Dos indicadores de qualidade, três deles, são considerados mais importantes para esse trabalho, todos gerando informações de qualidade de forma diferente. Medindo o total de soluções do espaço de busca que foi coberto pela aproximação desses conjuntos.

O indicador de qualidade *Hypervolume* I_H proposto por Zitzler et al [40] mede o *Hypervolume* da porção do espaço de busca que está mais fracamente dominada por um conjunto de aproximações A; e deverá ser maximizado. Para que se possa medir essa quantidade de espaço, o espaço de busca deve ser delimitado, caso contrário, pelo menos um ponto de referência que seja dominado pelos outros pontos deve ser usado como ponto de delimitação. Considerando a diferença de *Hypervolume* de um determinado conjunto R, definido como I_H^- e dado um conjunto de aproximação A, o indicador pode ser definido de acordo com a Equação (2.5):

$$I_H^-(A) = I_H(R) - I_H(A) \tag{2.5}$$

Quanto menor o valor do indicador melhor será a qualidade do conjunto. O indicador I_H tem uma propriedade desejável: Dizemos que $A \lhd B$; quando $I_H(A) > I_H(B)$ desde que o ponto de delimitação seja estritamente dominado por todos os pontos pertencentes a A e B. Portanto, de $I_H(A) > I_H(B)$ podemos presumir que A não pode ser melhor que B. Sendo, o indicador de *Hypervolume* o único capaz de detectar que um conjunto A não é melhor que B para todos os pares $B \lhd A$.

O indicador de qualidade *Epsilon* foi proposto por Zitzler et al [40] e está composto por uma versão multiplicativa e outra aditiva. A versão multiplicativa do indicador é dada por $I_\epsilon(A, B)$, que fornece o fator mínimo ϵ pelo qual cada ponto pertencente a B pode ser multiplicado, de tal forma que o resultado do conjunto de aproximações seja fracamente dominado por A, conforme definido pela Equação (2.6):

$$I_\epsilon(A, B) = inf_{\epsilon \in \Re} \forall z^2 \in B \exists z^1 \in A : z^1 \preceq_\epsilon z^2 \tag{2.6}$$

Sendo a relação, $\preceq_\epsilon$, definida pela Equação (2.7):

$$z^1 \preceq_\epsilon z^2 \Longleftrightarrow \forall i \in 1..n : z_i^1 \leq \epsilon . z_i^2 \tag{2.7}$$

para um problema de minimização, e assumindo que todos os pontos são positivos em todos os objetivos. Baseado nisso, o indicador I_ϵ^1 pode ser definido pela Equação (2.8):

$$I_\epsilon^1 = I_\epsilon(A, R) \tag{2.8}$$

onde R é um conjunto de pontos de referência. O indicador aditivo pode ser criado de forma análoga, apenas alterando-se a definição de $\preceq_{\epsilon+}$ de acordo com a Equação (2.9):

$$z^1 \preceq_{\epsilon+} z^2 \Longleftrightarrow \forall i \in 1..n : z_i^1 \leq \epsilon + z_i^2 \tag{2.9}$$

Ambos os indicadores são minimizados, assim, um valor menor que 1 (I_ϵ^1) e 0 ($I_{\epsilon+}^1$)

implica que A domina o conjunto R. Para os indicadores *Epsilon* vale dizer que sempre que $A \lhd B$, então $I^1_\epsilon(A) \leq I^1_\epsilon(B)$ respectivamente $I^1_{\epsilon+}(A) \leq I^1_{\epsilon+}(B)$. Assim, de $I^1_\epsilon(A) > I^1_\epsilon(B)$ e respectivamente $I^1_{\epsilon+}(A) > I^1_{\epsilon+}(B)$, pode-se deduzir que A não é melhor que B. Mas é importante observar que os indicadores *Epsilon* trabalham de uma forma diferente do *Hypervolume*. Em alguns casos podem-se ter diferentes ordens de preferência de um par de conjuntos de aproximação, ficando claro, que neste caso, são conjuntos que não podem ser comparados.

Finalmente um útil indicador de qualidade é o indicador R proposto por Hansen et al [19], é usado para comparar os conjuntos de aproximações baseado em um conjunto de funções de utilidades. Uma função de utilidade u é definida como um mapeamento do conjunto Z de vetores n-dimensionais para o conjunto de números reais conforme a Equação (2.10):

$$u : Z \longmapsto \Re \tag{2.10}$$

A tomada de decisão, se um conjunto é melhor que outro, é dada de acordo com uma função de utilidade parametrizada u_λ e um conjunto Λ de parâmetros. u_λ deve representar a soma dos pesos dos valores objetivos, no qual $\lambda = (\lambda_1....\lambda_n) \in \Lambda$. A partir disso pode-se definir os indicadores de acordo com a Equação (2.11):

$$I_{R2}(A, B) = \frac{\sum \lambda \in \Lambda u^*(\lambda; A) - u^*(\lambda; B)}{\Lambda} \tag{2.11}$$

$$I_{R3}(A, B) = \frac{\sum \lambda \in \Lambda [u^*(\lambda; B) - u^*(\lambda; A)]/u^*(\lambda; B)}{\Lambda} \tag{2.12}$$

onde u^* é o maior valor alcançado pela função de utilidade u_λ com o vetor de pesos λ em um conjunto de aproximação A, por exemplo, $u^*(\lambda, A) = max_{z \in A} u_\lambda(z)$. Da mesma forma como acontece com os indicadores *Epsilon*, os indicadores R unários são definidos baseados nas versões binárias trocando-se B por um conjunto arbitrário. A escolha da função u_λ pode ser feita de várias maneiras, sendo uma possibilidade usar uma função linear baseada em pesos como definido pela fórmula (2.13):

$$u_\lambda(z) = -\sum_{j \in 1...n} \lambda_j |z_j * - z_j| \tag{2.13}$$

onde z^* é um ponto ideal, se conhecido, ou então qualquer ponto que domina fracamente todos os pontos no conjunto. Os indicadores R garantem que, em um caso de minimização, o valor do indicador para um conjunto A de aproximações é menor ou igual ao valor do indicador associado ao conjunto B, $A \lhd B$. De $I^1_{R2}(A) > I^1_{R2}(B)$ e respectivamente $I^1_{R3}(A) > I^1_{R3}(B)$ pode inferir que A não seja melhor que B.

2.7 Considerações finais

Este capítulo apresentou algoritmos multi-objetivos utilizando abordagens meta-heurísticas como PACO, MOTS e NSGA-II. Cada abordagem utiliza maneiras diferentes para a geração e evolução das populações. Cada uma segue uma idéia e são extensões de soluções que já deram certo para problemas com características diferentes. Além disso, vale ressaltar a dificuldade ao se comparar os resultados e fronteiras geradas pelos algoritmos. Os indicadores citados no trabalham permitem uma avaliação mais precisa e consequentemente nos levam a conclusões mais corretas relativas a diferença e qualidade dos resultados gerados pelas abordagens.

CAPÍTULO 3

TESTE DE INTEGRAÇÃO DE CLASSES E ASPECTOS

A principal tarefa da atividade de teste é revelar erros, e isto normalmente acontece quando os testes são realizados de uma maneira sistemática [30]. Além de revelar erros o teste apresenta como benefícios secundários mostrar que o sistema atende aos requisitos especificados para o mesmo, ou seja, o teste mostra que o software faz o que deve fazer. De acordo com Pressman [30] os dados que foram gerados durante as atividades de teste são boas medidas que garantem confiabilidade e fornecem certa indicação da qualidade do sistema como um todo. De acordo com Jorgensen [22] pode se dizer que a principal tarefa do teste de software é determinar um conjunto de casos de teste para o software ser testado. Um caso de teste é dado por uma entrada para o programa (dados de teste) e uma saída esperada para sua execução.

Os resultados do teste são comparados às saídas esperadas avaliando-se se a aplicação comporta-se conforme o esperado.

Os valores definidos pelos resultados esperados são gerados a partir de um oráculo, ou seja, algo que conhece a saída. Esse oráculo geralmente é o desenvolvedor ou alguém que é conhecedor do domínio do problema e tem idéia dos valores prováveis de saída para determinados tipos de teste. O conjunto de casos de teste definido para uma dada aplicação é denominado de suíte de teste. Segundo [33] :

- Testar é o processo de executar um programa com a intenção de encontrar um erro.
- Um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.
- Um teste bem sucedido é aquele que revela um erro ainda não descoberto.
- Testes não podem assegurar a ausência de erros, somente mostram a presença de erros no software.

A atividade de teste deve compreender quatro fases [30]:

1. Teste de Unidade: O objetivo é explorar a menor unidade funcional (sub-rotina ou procedimento) do programa, portanto se concentra na implementação do código fonte.

2. Teste de Integração: O objetivo é explorar as interfaces entre os módulos, quando se procura identificar defeitos de comunicação, visando garantir o funcionamento adequado do conjunto.

3. Teste de Validação: O objetivo é identificar defeitos em funcionalidades e características de desempenho que não estejam conforme a especificação.

4. Teste de Sistema: O objetivo é identificar a presença de defeitos de funções e desempenho global, observando-se o seu comportamento de forma mais ampla, incluindo-se, por exemplo: pessoas, banco de dados, hardware, etc.

A atividade de teste não garante a ausência de erros em um sistema, pois a única maneira de garantir que um sistema é livre de erros seria através de um teste exaustivo, inviável devido a restrições de tamanho, tempo e custo. Por isso, algumas técnicas foram criadas na escolha e geração de casos de teste. Essas técnicas levam em consideração diferentes características dos programas para a gerar os dados de teste. Elas selecionam subconjuntos dos dados do domínio do problema de acordo com algumas características dos mesmos.

Estas técnicas são ditas complementares, ou seja, a utilização delas é mais viável em determinados estágios do desenvolvimento e também revelam problemas em diferentes pontos do sistema. A técnica funcional, tem como base a funcionalidade do software; a estrutural tem como base a estrutura interna de uma implementação específica [30] e, a técnica baseada em defeitos, deriva os requisitos de teste a partir de erros típicos e comuns introduzidos durante o processo de desenvolvimento do software sendo testado.

3.1 Teste de Software Orientado a Objetos

À medida em que novos paradigmas de programação começaram a surgir, surgiu também a necessidade das técnicas de teste de software se adaptarem aos diferentes modos de programação. Entre esses paradigmas existe a Orientação a Objetos. Nesse paradigma a ênfase é dada a um elemento único chamado objeto, que possui atributos e métodos que permite a comunicação com outros objetos através de mensagens, diferente da programação estruturada onde a ênfase está na construção dos procedimentos e a comunicação entre os dados é feita através da passagem de parâmetros. A relação entre as fases de teste de programas procedimentais e Orientados a Objeto pode ser vista na Tabela 3.1 [39], percebe-se que há duas abordagens, na primeira, considera-se o método como menor unidade, ou seja, o teste de integração contempla o teste entre os métodos da classe e das subclasses; e teste de métodos de classes diferentes. Na segunda considera-se a classe como menor unidade, ou seja, o teste de integração é o teste entre as diferentes classes. Entretanto independentemente da abordagem adotada, percebe-se que existe uma etapa na qual é necessária a integração das classes do software.

Tabela 3.1: Relação entre as fases de teste de programas procedimentais e Orientados a Objeto

Menor Unidade: Método		
Fase	Teste Procedimental	Teste Orientado a Objetos
Unidade	Intraprocedimental	Intra-método
Integração	Intraprocedimental	Inter-método, Intra-classe e Inter-classe
Sistema	Toda a aplicação	Toda a aplicação
Menor unidade: Classe		
Fase	Teste Procedimental	Teste Orientado a Objetos
Unidade	Intraprocedimental	Intra-método, Inter-classe e Intra-classe
Integração	Intraprocedimental	Inter-classe
Sistema	Toda a aplicação	Toda a aplicação

3.2 O Problema CITO

A etapa de teste de integração das classes é um dos pontos mais críticos no teste de software Orientados a Objetos. Isto se deve ao fato de que as classes não necessitam

seguir uma ordem para serem criadas. Ou seja, classes podem ser criadas em momentos diferentes e integradas depois.

Para realizar isto é necessário resolver-se o problema de dependências entre as classes. Esse problema está associado ao fato de que uma dada classe cliente necessita que outra classe dita servidora esteja disponível antes que ela possa ser executada. Se a classe servidora não estiver disponível a solução é construir estruturas chamadas *drivers* e/ou *stubs* no momento dos testes.

Um *driver*, de acordo com Delamaro et al [12] é uma unidade que coordena o teste de uma classe, sendo responsável por ler os dados de teste fornecidos pelo testador, repassar esses dados na forma de parâmetros para a classe, coletar os resultados relevantes produzidos pela classe, e apresentá-los para o testador.

Um *stub* é uma unidade que substitui, na hora do teste, uma unidade usada pela classe em questão, ela simula o comportamento de uma unidade chamada pela classe que está sendo testada utilizando o menor custo computacional possível ou o mínimo de manipulação de dados possível.

A geração de *stubs* pode ser um processo custoso dependendo da complexidade da classe a ser testada. A ordem em que as classes são testadas, influi diretamente na criação de *stubs*, pois as dependências existentes em sistemas complexos gera os chamados ciclos de dependência. Ou seja, em um sistema formado pelas classes A, B, C, D, E e F, por exemplo, pode-se ter uma situação em que a classe A utiliza a classe D, que utiliza a classe E, que utiliza a classe A, criando assim o que se chama de ciclo de dependência. A Figura 3.1 ilustra esse ciclo. A solução ao se realizar o teste de integração, quando existem ciclos no sistema, é a quebra dos mesmos, ilustrado na Figura 3.2. Para isso se realizar é necessária a geração de um *stub* relativo àquela quebra. A ordem em que as classes são testadas influi nas quebras desses ciclos e consequentemente na geração de *stubs*. Um estudo recente [26] mostra que a presença de ciclos complexos entre as classes são muito comuns em sistemas desenvolvidos em Java e por isto o problema de encontrar a sequência de classes que possui o menor custo na geração de *stubs* (denominado (*CITO - Class Integration and Test Order*) [1] tem sido tema de alguns trabalhos na literatura.

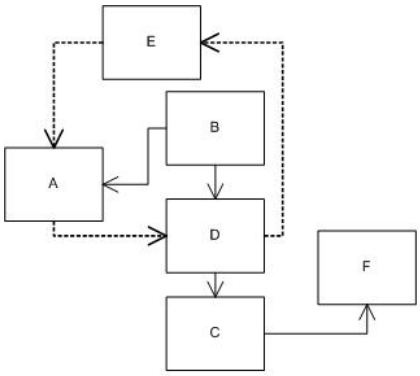

Figura 3.1: Exemplo do ciclo existente entre classes

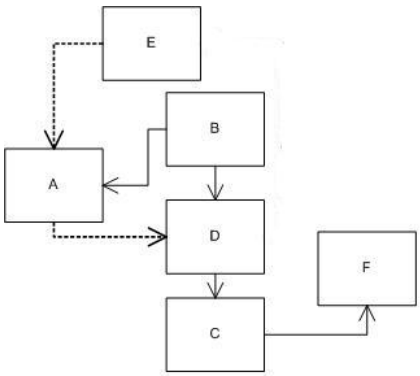

Figura 3.2: Exemplo de quebra de um ciclo existente entre classes

A maioria das soluções propostas para resolver o problema CITO geralmente utiliza abordagens baseadas em grafos. Isso se deve ao fato de que a representação na forma de grafos direcionados das relações entre as classes é simples de ser criada. Esses grafos direcionados são chamados de (*Object Relation Diagrams - ORDs*) [25]. Em um ORD os vértices representam as classes e as arestas representam os relacionamentos entre elas.

As relações de dependência entre as classes podem ser diversas. No contexto Orientado a Objetos, as dependências presentes nos sistemas mais importantes para esse trabalho são:

1. *Herança*: é o mecanismo pelo qual uma classe (sub-classe) pode estender outra classe (super-classe), aproveitando seus comportamentos (métodos) e variáveis possíveis (atributos). Um exemplo de herança: Brasileiro é super-classe de Paranaense. Ou seja, um Paranaense é um brasileiro.

2. *Agregação*: é o mecanismo pelo qual uma das classes do relacionamento é uma

parte, ou está contida em outra classe. Um exemplo de agregação: A marinha contém navios.;

3. *Associação*: é o mecanismo pelo qual um objeto utiliza os recursos de outro. Pode ser uma associação simples. Por exemplo: Uma pessoa usa um carro.

Essas relações são representadas no ORD da seguinte maneira:

1. I: representa o relacionamento de herança;
2. Ag: representa o relacionamento de agregação;
3. As: representa o relacionamento de associação;

A Figura 3.3 mostra um exemplo de um ORD (à direita) baseado em diagrama de classes (à esquerda).

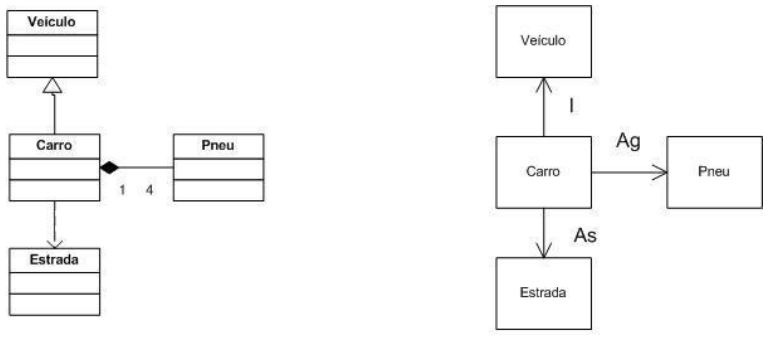

Figura 3.3: Exemplo de ORD em OO criado a partir de classes

3.2.1 Abordagens com Grafos para o CITO

A maioria das soluções para o problema CITO calcula o custo da geração de *stubs* através do número de dependências a serem quebradas no ORD que representa o sistema, para que assim o grafo torne-se acíclico de dependência das classes, e possa então, ser ordenado, topologicamente obtendo-se a ordem menos custosa de teste de integração das classes. Sistemas simples, que não possuem ciclos, podem ter seus testes realizados através da inversão topológica das classes. Entretanto, outras soluções são necessárias na presença de ciclos.

Kung et al [24] propuseram uma abordagem que consiste em identificar os Componentes Fortemente Conexos (CFC), ou seja, os subgrafos do grafo que são fortemente conexos (um grafo é dito fortemente conexo se existe um caminho de u até v e de v até u para qualquer par distinto de vértices u e v) e remover as associações até que não exista mais nenhum ciclo. Quando existe mais de uma associação candidata a ser removida para quebrar o ciclo, uma escolha aleatória é realizada entre as associações candidatas.

Tai e Daniels [36] propuseram uma abordagem na qual atribuem a cada classe dois possíveis níveis de classe, respectivamente chamados de *"major"* e *"minor"*. Os números de níveis *"major"* são atribuídos para as classes baseados apenas nas suas dependências de herança e agregação. Diferentemente dessas classes, os números de níveis *"minor"* são atribuídos baseados apenas nas relações de associação.

Componentes Fortemente Conexos são identificados no nível *"major"* e cada aresta do componente fortemente conexo recebe um peso baseado nas relações que "entram" (eferentes) e nas relações que "saem" (aferentes) das classes relacionadas à associação. Esse peso é calculado através da soma do número de associações quem "entram" no vértice de origem com o número de associações que "saem" do vértice de destino. Por exemplo, na Figura 3.4 que ilustra essas relações de entradas e saídas, pode-se dizer que três associações "entram" na classe origem A e duas associações "saem" da classe destino B, dessa maneira, o peso da associação entre a classe A e a classe B é 5 .

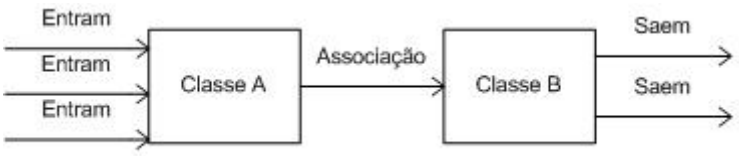

Figura 3.4: Relações que entram e relações que saem

Arestas com os maiores pesos são escolhidas para serem removidas e assim quebrar os ciclos pois estas estão supostamente relacionadas a mais ciclos. De acordo com Briand et al [8] esse tipo de pressuposto não é sempre verdade. Existem casos nos quais as associações entre classes não necessariamente estão envolvidas com ciclos, e esta solução acaba não sendo a melhor em relação à quantidade de *stubs* gerados para o teste.

No trabalho apresentado por Le Traon et al [38] é apresentada uma maneira diferente de lidar com os ciclos presentes no ORD. Os Componentes Fortemente Conexos são identificados através do algoritmo de Tarjan. O algoritmo é adaptado fazendo com que cada vértice seja classificado de acordo com algumas definições propostas em [38]. A classificação mais importante para esse trabalho é chamada de *frond dependency*. Os vértices classificados com o tipo *frond dependency* são usados na decisão de quebra de ciclos. Essa classificação *frond dependency* é definida como a associação que sai a partir de um vértice para um dito "antecessor", ou seja, uma classe que dele depende, direta ou indiretamente [8].

As quebras dos ciclos nessa abordagem dá-se através das remoções, nos Componentes Fortemente Conexos, das dependências de maior peso que "entram" na classe. O peso é definido como a soma das *frond dependencies* que "entram" e "saem" de uma determinada classe dentro do Componente Fortemente Conexo em questão. Para cada Conponente Fortemente Conexo não trivial, ou seja, componentes com mais de um vértice, esse processo é executado recursivamente. Essa abordagem é não-determinística, pois diferentes conjuntos de arestas que representam as relações entre as classes, podem ser nomeados como *frond dependency* dependendo do vértice que inicia o processo e quando dois ou mais vértices têm o mesmo peso, a escolha se dá aleatoriamente.

Um abordagem baseada em grafos que combina o trabalho de Le Traon et al e Tai e Daniels foi proposta por Briand et al [8]. Nessa abordagem, assim como em [38], o algoritmo de Tarjan também é usado recursivamente para identificar os Componentes Fortemente Conexos. Já a definição dos pesos das relações é parecida com as definidas em [36]. O valor é calculado através do número estimado de ciclos nos quais uma relação está envolvida em um Componente Fortemente Conexo.

Cada vez que o algoritmo de Tarjan, recursivamente, encontra um Componente Fortemente Conexo não trivial, a relação de dependência que apresenta o maior peso é escolhida para ser quebrada. Esse peso é calculado através de uma versão modificada do cálculo proposto por Tai e Daniels, sendo a multiplicação entre o número das associações que "entram" no vértice de origem com o número de associações que "saem" do vértice de

destino.

Essa abordagem não possui o não-determinismo, como na abordagem de Le Traon et al, pois não usa classificações como *frond dependency* e quando existe a possibilidade de escolhas alternativas de quebra de arestas com o mesmo peso, a escolha irá gerar o mesmo número de *stubs* pois a abordagem lida com a remoção de apenas uma relação de associação, enquanto a abordagem proposta por Le Traon et al, lida com a remoção de toda associação que "entra" da classe escolhida. A grande vantagem dessa abordagem é não quebrar arestas de herança e agregação e calcular os pesos de maneira mais precisa.

Os trabalhos mencionados acima apresentam algumas limitações [7]. A maioria das soluções baseadas em grafos, conforme visto, consistem em identificar os Componentes Fortemente Conexos recursivamente e em cada um remover a associação de dependência que maximiza o número de ciclos quebrados baseados nos ciclos identificados. A maioria tenta otimizar a decisão sem determinar as consequências que isso terá nos resultados finais e a solução encontrada é muitas vezes sub-ótima.

Outra desvantagem como a apontada por Briand et al [6] é que o custo relacionado com a construção de um *stub* pode depender de diversos fatores e não pode ser completamente medido ou estimado [7], por exemplo, o número de atributos de uma classe e o número de chamadas de métodos entre outros. Muitas vezes é extremamente difícil, ou até impossível, a adaptação da maioria das soluções baseadas em grafos para considerarem os fatores citados acima.

Uma tentativa de se considerar mais informações no processo de minimização de custo na criação de *stubs* é encontrada no trabalho de Abdurazik e Offutt [1]. Os pesos, nesse caso, são derivados de uma análise de nove tipos de acoplamento, e são atribuídos às arestas e vértices que representam, respectivamente, dependências e classes. As medidas de acoplamento usam o número de parâmetros, números de tipos de valores de retorno, número de atributos e métodos. O peso de cada vértice está diretamente relacionado ao custo estimado para removê-lo.

Caso a classe seja usada por diversas classes, então todo, ou parte do mesmo *stub* dessa classe pode ser compartilhado entre todas as classes que o usam, isso reduz consi-

deravelmente o custo gerado pela criação de *stubs*. O peso de um vértice é no mínimo maior que o peso máximo de todas as arestas que acessam o vértice, e não maior que a soma dos pesos de todas as arestas que acessam o vértice. A utilização dessa abordagem em alguns estudos de caso mostraram que os resultados são relativamente bons quando comparados as abordagens descritas anteriormente.

Quando o problema CITO é analisado, percebe-se que o espaço de busca associado é formado por todas as possíveis ordens de classes para um determinado sistema. Assim, a solução para o problema é uma sequência de classes que minimiza o custo de geração de *stubs*.

O custo dessa geração é obtido baseado em medidas de acoplamento entre classes por exemplo, sendo assim, pode-se dizer que o problema é naturalmente um problema de otimização multi-objetivo com restrições. Isso naturalmente levou a pesquisas para resolver esse problema através de Algoritmos Genéticos. Conforme visto no Capítulo 2 esses algoritmos são técnicas específicas de otimização baseadas em um conjunto de heurísticas e são boas estratégias para resolução de problemas com diversas restrições como é o caso do problema CITO. Na próxima seção são discutidas abordagens usando algoritmos genéticos para o problema CITO.

3.2.2 Abordagens utilizando Algoritmos Genéticos

Quando o problema *CITO* é analisado percebe-se que ele é basicamente um problema de otimização multi-objetivo com restrições, o que torna a utilização de Algoritmos Genéticos uma excelente escolha para a resolução do mesmo. Os objetivos são baseados em medidas do acoplamento entre as classes cliente e servidora e na minimização desse acoplamento enquanto quebram-se dependências na remoção de ciclos do grafo que representa o problema.

Briand et al [6] propõem algumas definições sobre a estrutura da solução de um Algoritmo Genético para o problema CITO. O primeiro ponto é a representação do cromossomo, ou seja, dos indivíduos que representam soluções para o problema. O cromossomo é uma sequência de rótulos que representam uma determinada classe. Por exemplo, para

um conjunto de classes (A, B, C, D) um cromossomo possível pode ser representado por (D, B, A, C).

A implementação dos operadores que agirão sobre os cromossomos é baseada nessa estrutura. Tipicamente são eles: *mutação* e *crossover*. A mutação é implementada selecionando duas classes e trocando-as de posição, por exemplo, um cromossomo (**A**, B, **C**, D) quando mutado vira (**C**, B, **A**, D). A operação de crossover é um pouco mais complexa, exigindo assim um maior cuidado. São diversas as maneiras de se realizar essa operação.

Em seu trabalho, Briand et al selecionam aleatoriamente genes do primeiro cromossomo pai. As suas posições são localizadas no segundo cromossomo, e os genes que restaram são copiados para o primeiro cromossomo na mesma ordem em que aparecem no segundo. Por exemplo, um cromossomo (A, B, C, D) e (D, C, B, A). Suponha que B e D tenham sido selecionados aleatoriamente do primeiro cromossomo, gera-se um novo cromossomo filho em dois passos. Primeiro removem-se os genes B e D do cromossomo obtendo-se assim (A , - , C , -), depois, os espaços vazios do cromossomo são preenchidos com B e D na ordem que aparecem no segundo cromossomo, obtendo-se assim o cromossomo (A, D, C, B).

Outros parâmetros influem diretamente no algoritmo genético, entre eles o tamanho da população e a probabilidade de ocorrência de mutação ou crossover. O tamanho da população utilizada por Briand et al foi o valor típico utilizado na maioria dos problemas desse tipo, ou seja, uma população inicial de 100 indivíduos. A probabilidade de ocorrência de mutação utilizada foi de 0.15 e para crossover foi 0.5, ou seja o peso maior para utilização de um operador é dado ao operador de crossover pois esse geralmente cria indivíduos melhores que os outros operadores (aumenta a diversividade), outros valores foram testados, porém nenhum deles trouxe uma melhora significativa.

Finalmente, Briand et al definiram as funções que determinam as medidas de complexidade da geração dos *stubs* para teste baseado nos valores de acoplamento a partir das dependências, associações, agregações simples e uso entre as classes. As duas medidas de acoplamento propostas por Briand et al são:

1. A : O número de atributos localmente declarados na classe quando o atributo aparece

na lista de argumentos de algum método da classe, ou como tipo de retorno, atributo no método, ou como parâmetro local. A complexidade medida é o numero máximo de atributos com os quais o *stub* gerado precisará lidar caso a dependência seja quebrada.

2. M : O número de métodos, inclusive construtores, localmente declarados na classe que é invocada pelos métodos da classe. A complexidade medida é o número de métodos que é necessário simular no *stub* caso a dependência seja quebrada.

Baseados nessas duas medidas Briand et al definiram a função a ser minimizada pelo algoritmo genético. Essa função é a média geométrica de todas as medidas de complexidade, assim, transformando o problema multi-objetivo em um problema mono-objetivo. Mas para que não haja uma perda grande ao se transformar o problema em mono-objetivo é realizada uma normalização matemática. Ou seja, para a medida $Cplx()$, que representa a complexidade existente no relacionamento entre duas classes baseado no seu nível de acoplamento, tem-se o seu correspondente $\overline{Cplx()}$, que é $Cplx()$ normalizada, para facilitar os cálculos obtendo-se valores mais fáceis de manipular matematicamente.

Essa complexidade é definida por uma função $Cplx(i, j)$ em que uma matriz na qual as linhas e colunas representam classes e a relação de dependência entre elas, ou seja, a classe i depende da classe j. A partir disso obtém-se a seguinte Equação (3.1):

$$\overline{Cplx(i,j)} = Cplx(i,j)/(Cplx_{max} - Cplx_{min}) \tag{3.1}$$

Onde $Cplx_{max} = Max\{Cplx(i,j), i,j = 1,2,...\}$ e $Cplx_{min} = \{Cplx(i,j), i,j = 1,2,...\}$

E finalmente a função $SCplx(i, j)$ calcula a complexidade da geração de um *stub* que contempla uma dependência entre um par de classes (i, j) é definida como a média geométrica dos pesos da normalização definida pela Equação (3.2):

$$SCplx(i,j) = (W_A.\overline{A}(i,j)^2 + W_M.\overline{M}(i,j)^2)^{1/2} \tag{3.2}$$

onde W_A e W_M são pesos que somados são igual a 1. Sendo assim para uma de-

terminada ordem de teste o, e um conjunto de dependências d identificadas, o valor da função $OCplx(o)$, que é complexidade para a ordem o, pode ser definida de acordo com a Equação (3.3):

$$OCplx(o) = \sum SCplx(k)_{k=1}^{d} \tag{3.3}$$

Sendo o valor $OCplx$ que o Algoritmo Genético tenta minimizar nessa abordagem para resolver o problema.

Outro ponto importante que se deve colocar, é que na solução proposta por Briand et al, outra matriz de restrição é utilizada para impedir que dependências de herança ou composição sejam quebradas. Ou seja, as sequências são otimizadas porém respeitando uma série de restrições, como no caso, as citadas acima.

Essa abordagem foi aplicada usando quatro funções de *fitness* em alguns sistemas e obteve resultados satisfatórios. Essas funções foram estipuladas da seguinte forma:

1. Apenas o número de dependências quebradas é usado como a função de custo: $SCplx(i,j)¸0,1$. Dependendo de quando i depende de j ou não.

2. Acoplamento de atributo é usado com função de custo: $SCplx(i,j) = A(i,j)$.

3. Acoplamento de método é usado com função de custo: $SCplx(i,j) = M(i,j)$.

4. A média geométrica ponderada do acoplamento de atributo e método ($OCplx$) é usada na definição de $SCplx$.

Dessa última maneira calcular o *fitness*, percebe-se que ela utiliza uma única função de *fitness* que exige uma média entre os fatores que afetam a complexidade do *stub*. Um procedimento para ajustar os pesos foi proposto. Entretanto, ajustar os pesos para essa média não é um procedimento trivial, principalmente para grandes sistemas. Por isso um tratamento diferente para o problema multi-objetivo pode resolver essa questão e aumentar a aplicabilidade da solução com algoritmos evolutivos. Esse é o objetivo dos trabalhos descritos na próxima seção.

3.2.3 Usando Dominância de Pareto para o problema CITO

Uma abordagem baseada nos conceitos de dominância de Pareto para lidar com o problema CITO foi proposta por Cabral el al. [9]. Neste trabalho foi implementado um algoritmo PACO com a mesma estrutura apresentada no Capítulo 2, apenas diferindo que em cada caminho completo construído por cada formiga a matriz de feromônio é atualizada localmente e uma busca local é utilizada como uma heurística de refinamento para encontrar ótimos locais no espaço de soluções.

Além do PACO outros algoritmos evolutivos também foram implementados o NSGA-II [11] e MOTS [17]. O Algoritmo NSGA-II utilizou a ferramenta *Open BEAGLE* [16], que é um *framework* que facilita a programação de algoritmos genéticos para os mais variados fins. Foi implementada uma solução seguindo a idéia de Briand et al. Uma ordenação de classe representa um indivíduo, e consequentemente, as classes são os genes do cromossomo. Para a matriz de restrições uma matriz de valores binários é implementada, sendo que o valor igual a 1 significa que há restrição e 0 que não há. Duas outras matrizes, uma matriz de métodos e outra de atributos, respectivamente, representando a dependência entre as classes no que se refere a métodos e atributos também foram utilizados nos quais uma dependência associada entre classes é representada por valores inteiros.

Em [6] também são utilizados dois métodos de normalização, que nada mais são que dois métodos que garantem que indivíduos iguais não sejam gerados e fiquem na população e que nenhuma sequência que não respeite a matriz de dependência seja gerada. O cálculo do *fitness* para métodos e atributos é único baseado na solução de Briand et al [8]. E o algoritmo segue o mesmo pseudo-código explicado no Capítulo 2. Os parâmetros do algoritmo foram definidos com probabilidades de 0,6 para cruzamento e 0,1 para mutação, população inicial com 200 indivíduos e 100 gerações.

O algoritmo MOTS [17] segue a mesma idéia descrita no Capítulo 2 de percorrer a estrutura de vizinhança utilizando o conceito de Pareto e utilizar a Lista de Candidatos, Lista Pareto e Lista Tabu de apoio.

Os resultados dos algoritmos foram comparados com os do Algoritmo Genético obtidos por Briand et al [7] e apresentaram soluções sempre melhores, não dominadas, com um

menor custo com relação ao número de métodos e atributos.

3.3 Teste de Software Orientado a Aspectos

A Orientação a Objetos facilitou a reutilização de código dentre outros fatores, permitindo um melhor tratamento de complexidades inerentes ao software.

Mesmo assim, a Orientação a Objetos não permite uma clara separação de certos interesses (*concerns*). Um interesse é uma parte do problema que se quer tratar como uma unidade conceitual única na solução do software [2]. O trabalho de qualquer engenheiro é gerenciar a complexidade durante as diferentes fases do desenvolvimento de sistemas. Isso deve ocorrer em especial na engenharia de software.

Muitas pesquisas na área tratam de como gerenciar a complexidade de sistemas de software, oferecendo melhores métodos e técnicas de estruturação. A separação de *interesses* [2] está no centro da engenharia de software, em geral, como um princípio fundamental que tenta resolver as limitações da cognição humana ao lidar com a complexidade do software. Em Orientação a Aspectos esses *interesses* são chamados de interesses transversais (ou *crosscutting concerns*), pois sua implementação não fica localizada em unidades isoladas.

A Programação Orientada a Aspectos surgiu como possível solução para esse tipo de problema pois oferece estrutura para utilização de interesses transversais (*crosscutting concerns*) e permite que esse código seja encapsulado e modularizado. A idéia principal é que a Programação Orientada a Aspectos seja utilizada com os paradigmas já existentes, como por exemplo, com o paradigma de Orientação a Objetos.

A possibilidade de implementação de módulos isolados (aspectos) que têm capacidade de afetar outros módulos de maneira transversal é o principal ponto que vem tornando este paradigma muito estudado nos últimos anos. Agora, quando se fala de teste de software, da mesma maneira que sistemas Orientação a Objetos possuem certas peculiaridades, sistemas Orientados a Aspectos também apresentam características que merecem serem analisadas quando testes são realizados.

Para entender essas características, e alguns pontos que aparecerão neste trabalho, é

necessária a compreensão de algumas definições sobre técnicas de Programação Orientada a Aspectos.

Um dos mecanismos importantes em Programação Orientada a Aspectos são os pontos de junção (*join points*) que permitem a inserção de um comportamento adicional no sistema em questão [3]. Um conjunto de pontos de junção (*pointcut*) identifica diversos pontos de junção em um sistema.

Os *crosscutting concerns* que utilizam esse conjunto são implementados por uma construção similar a de um método, chamado de adendo (*advice*), que executa antes, depois ou, às vezes, no lugar do ponto de junção identificado. De uma maneira geral, pode-se dizer que os aspectos servem para dois propósitos: auxiliar atividades de desenvolvimento, ou implementar interesses que estarão na aplicação final.

Para realizar o teste de programas Orientados a Aspectos, aproveitou-se muito do conhecimento existente no teste de programas Orientados a Objeto com a extensão de técnicas e critérios de teste existentes nesse contexto. As fases da atividade de teste em Programação Orientada a Aspectos são muito semelhantes as fases de teste de Orientação a Objetos. Porém, vale ressaltar aqui alguns pontos como visto em [12]:

1. **Teste de unidade:** Cada método e adendo são testados isoladamente (intra-método e intra-adendo).

2. **Teste de módulo:** Teste das unidades dependentes que interagem por meio de chamadas de métodos ou uso de adendos. Pode-se dividir essa fase em subfases conforme abaixo:

   Inter-método: Testa cada método público juntamente com outros métodos acessados de forma direta ou indireta.

   Adendo-método: Testa cada adendo juntamente com outros métodos acessados de forma direta ou indireta.

   Método-adendo: Testa cada método público juntamente com os adendos que o afetam direta ou indiretamente. Não se considera aqui a integração dos métodos afetados com os outros métodos chamados por ele ou pelos adendos.

Inter-adendo: Testa cada adendo juntamente com outros adendos que o afetam direta ou indiretamente.

Inter-método-adendo: Testa cada método público juntamente com os adendos que o afetam direta ou indiretamente e com métodos chamados direta ou indiretamente. Esse teste inclui os quatro testes de integração descritos acima.

Intra-classe: Testa interações entre métodos públicos de uma classe quando chamado em diferentes sequências. A idéia é a mesma que foi explicada em teste de software Orientação a Objetos.

Inter-classe: Testa interações entre classes diferentes, considerando ou não a interação dos aspectos. A idéia é a mesma que foi explicada para intra-classe porém considerando a interação entre as diferentes classes.

3.4 O Problema CAITO

O paradigma da Orientação a Aspectos surgiu a partir da programação Orientada a Objetos e as técnicas de teste de software Orientado a Objetos foram estendidas para este novo contexto. Isto fez com que alguns problemas do teste de software Orientado a Objetos fossem herdados no teste de software Orientado a Aspectos, tais como o problema CITO. No contexto de Orientação a Aspectos, além das classes é necessário que as sequências contenham também aspectos e novas relações passam a existir. Esse novo problema será chamado neste trabalho de *CAITO - Class and Aspects Integration Test Order*.

Para resolver este problema, Ré et al [31] apresentam uma adaptação do grafo ORD que pode ser vista na Figura 3.5. Assim, é definido um ORD estendido o ORD que contempla relações de dependências entre aspectos e entre classes e aspectos. As relações a seguir passam a fazer parte do ORD.

1. *It*: representa as declarações de dependências inter-tipo geradas mudando a hierarquia de herança. Por exemplo, um aspecto Aa que declara que A extends B;

2. *C*: representa uma associação de crosscutting, ou seja, quando um aspecto possui um pointcut relacionado a outra classe.

3. *As*: representa a dependência de associação gerada pela necessidade de alguns adendos definidos em pontos de junção precisarem informações sobre os pontos de junção.

4. *U*: representa, as dependências de usos, entre classes; entre pontos de junção e adendos; e entre pontos de junção apenas.

Dessas relações é importante ressaltar que em seu trabalho Ré et al [31], não permitem a quebra das relações de agregação e herança; e inter-tipos pois essas indicam acoplamento de dados, fluxo de controle e de código entre as classes e aspectos existentes no sistema.

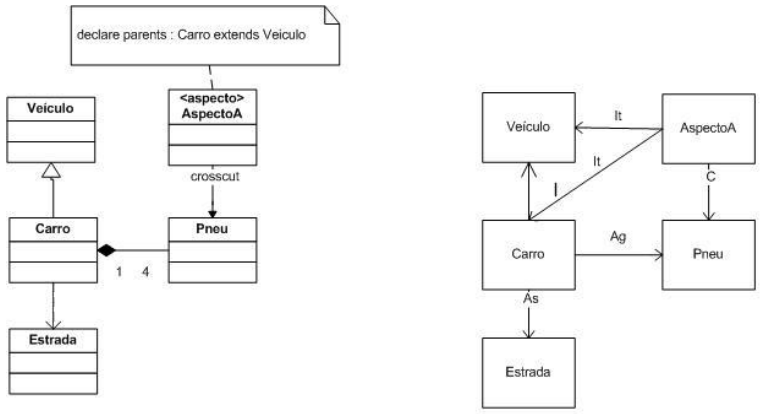

Figura 3.5: Exemplo de ORD em OA criado a partir de classes

Em outro trabalho, Ré et al [32] aplicaram algumas técnicas baseadas em grafos, utilizando as adaptações vistas anteriormente, ao sistema Telecom [32] e os resultados obtidos foram satisfatórios no intuito de diminuir o custo da geração de *stubs*, ou seja, resolver o problema CAITO. Para isso quatro estratégias foram aplicadas e comparadas ao final. As estratégias foram as seguintes:

1. *Combined*: É a mesma estratégia proposta pelo algoritmo de Briand et al [6] mas aplicada a um ORD estendido.

2. *Incremental+*: Consiste no teste de integração das classes primeiramente utilizando o algoritmo proposto por Briand et al [6] e depois integrando os aspectos na ordem proposta pelo algoritmo de otimização aplicado apenas nos aspectos.

3. *Reverse*: Consiste na sequência inversa encontrada pela estratégia *Combined*

4. *Random*: Consiste na geração aleatória das sequências.

No estudo de caso apresentado por Ré et al [32] a estratégia combined apresentou melhores resultados que a as outras estratégias quando levado em consideração o número de métodos e atributos como valores de custo de geração de *stubs*.

3.5 Considerações finais

Os algoritmos baseados em busca meta-heurística no contexto de Orientação a Objetos já foram utilizados obtendo soluções que diminuíam consideravelmente o custo da geração de *stubs* no teste de integração. Porém devido a natureza do problema acredita-se que resultados tão bons ou melhores podem ser alcançados adaptando-se os algoritmos atuais com a inserção de abordagens multi-objetivo por exemplo. No contexto de Orientação a Aspectos, algoritmos de busca meta-heurística ainda não foram aplicados. Para isto são necessárias adaptações nas abordagens e estruturas para que as mesmas consigam utilizar as técnicas já propostas. Este é o objetivo do próximo capítulo.

CAPÍTULO 4

APLICAÇÃO DE ALGORITMOS DE BUSCA PARA O PROBLEMA CAITO

Quando se realiza o teste de integração no contexto de software Orientado a Aspectos deve-se considerar além das relações entre as classes, as novas relações entre classes e aspectos, para se definir uma ordem teste, pois isso influi no custo da geração dos *stubs*. Conforme visto no Capítulo 3, Abdurazik e Offutt [1] apresentaram nove tipos de acoplamento existentes em sistemas Orientados a Objeto, sendo dois deles, o número de atributos e métodos de uma classe. Esses foram utilizados com sucesso por Briand [6] como valores para definir uma ordem menos custosa na geração de *stubs* no teste de integração. Considerando as semelhanças dos paradigmas Orientado a Objetos e Orientado a Aspectos, e os bons resultados obtidos por Briand et al [6] em seu trabalho, iniciou-se uma investigação por algoritmos de busca que pudessem utilizar de uma melhor forma esses valores de acoplamento e a melhor forma de adaptar essas soluções no contexto Orientado a Aspectos.

Esse capítulo descreve soluções, considerando algoritmos de busca, para o problema CAITO comparando-as com as soluções obtidas pelas estratégias baseadas em grafos apresentadas em [32].

4.1 Implementação das Estratégias baseados em grafos

Para permitir a aplicação das estratégias baseadas em grafos o algoritmo de Tarjan foi implementado (Algoritmo 6) em Java. A entrada para o algoritmo é um grafo $G = (V, E)$, correspondente ao ORD estendido. As classes foram enumeradas, representando assim os vértices, e as relações de dependência são as arestas com pesos dos grafos.

Para identificar os Ciclos Fortemente Conexos, que nada mais são os ciclos nos grafos que fazem parte de um Componente Fortemente Conexo do grafo, as arestas de asso-

ciação nos Ciclos Fortemente Conexos são atribuídas com pesos correspondentes ao valor estimado de ciclos que podem existir para um determinado ponto, chamados de ciclos envolventes. Esse valor é calculado de acordo com o número de dependências que "entram", ou seja, número de classes ou aspectos que dependem da classe, e que "saem", ou seja, número de classes que a classe depende para existir. A aresta com maior peso é removida. O processo então é repetido até nenhum Ciclo Fortemente Conexo permanecer.

Algoritmo 6 Pseudo-códgo Tarjan

```
1: procedure CalculaTarjan(v)
2: index = 0
3: S = vazio
4: para todo n ∈ V faça
5:    se v.index não definido então
6:       tarjan(v)
7:    fim se
8: fim para

9: procedure tarjan(v)
10: v.index = index
11: v.lowlink = index
12: index = index + 1
13: S.push(v)
14: para todo (v, v') ∈ E faça
15:    se v'.index não definido então
16:       tarjan(v')
17:       v.lowlink = min(v.lowlink, v'.lowlink)
18:    senão se v' ∈ S então
19:       v.lowlink = min(v.lowlink, v'.index)
20:    fim se
21:    se v'.lowindex = v.index então
22:
23:       imprima SCC:
24:       repita
25:          v' = S.pop:
26:
27:          imprima v':
28:       enquanto v' = v
29:    fim se
30: fim para
```

No processo de remoção de uma aresta com maior peso, nas estratégias *combined* e *incremental+* propostas em [32], quando ocorre o empate entre duas arestas, uma delas é escolhida aleatoriamente. Essa escolha aleatória gera ao final, sequências diferentes, e

consequentemente com valores diferentes quanto ao custo de criação de um *stub*. Devido ao tamanho dos sistemas, foi possível aplicar as estratégias usando todas as escolhas possíveis em momentos de empate. Permitindo ao final da execução, ter-se o melhor resultado possível da estratégia aplicada.

Como já visto, a estratégia *Combined*, é aplicada utilizando o mesmo algoritmo proposto por Briand et al [6] em seu trabalho mas a um ORD estendido. A estratégia *Incremental+* consiste no teste de integração das classes primeiramente utilizando o algoritmo proposto em [6] e depois integrando os aspectos na ordem proposta pelo algoritmo de otimização aplicado apenas nos aspectos.

A estratégia *Reverse* é o inverso da sequência encontrada pela estratégia *Combined*. A estratégia *Random* gera aleatoriamente uma sequência através de um algoritmo que apenas cria sequências do tamanho dos sistemas utilizados.

4.2 Algoritmo Genético Usando uma Função de Agregação

Após a aplicação das estratégias nos dois sistemas escolhidos, uma abordagem utilizando Algoritmo Genético foi implementada e aplicada. Para isso, seguindo a idéia mais simplista de algoritmo genético, uma estrutura é utilizada baseada no Algoritmo 7.

Algoritmo 7 Exemplo de um Algoritmo AG Simples

```
1: t := 0
2: inicia população P(t)
3: calcula fitness P(t)
4: t = t + 1
5: se critério de parada alcançado então
6:     vá para passo 12
7: fim se
8: selecione P(t) de P(t-1)
9: crossover P(t)
10: mutação P(t)
11: vá para passo 3
12: retorne melhor e termine
```

O algoritmo foi implementado seguindo a idéia de criação de uma população, e a partir dela, selecionar (seleção por torneio) os melhores para reproduzirem (conforme Figura 4.3) e serem mutados (conforme a Figura 4.2), até se ter a criação do melhor indivíduo.

O critério de parada definido como parâmetro de entrada, sendo escolhido pelo número máximo de avaliações dos algoritmos ou calibrados de acordo com os resultados obtidos.

Para a aplicação do algoritmo, primeiramente é necessário o mapeamento do problema em estruturas que possam ser utilizadas por esse.

Tendo como base o seu ORD estendido, três matrizes são criadas, duas delas representam as relações de dependências entre as classes/aspectos sendo uma referente ao número de atributos e a outra ao número de métodos, que definem o custo dispendido na geração de um *stub* com a quebra daquela relação. Essas duas matrizes diferem das matrizes utilizadas em soluções Orientadas a Objeto pois contemplam as relações novas do ORD estendido, e finalmente uma matriz que representa a relação de precedência hierárquica das classes e aspectos, ou seja, a matriz que contém as relações entre classes ou aspectos que não podem ser quebradas em ciclos. Exemplos de matrizes de dependência são apresentados no Apêndice A. Essas matrizes são utilizadas pela função de cálculo de *fitness* e assim permitir a avaliação e continuidade na busca de soluções.

Os indivíduos do algoritmo são representados por sequências de classes e aspectos que definem uma possível ordem de integração. Para cada sequência existe um valor que representa o custo total de geração dos stubs. Um indivíduo é representado conforme a Figura 4.1.

Figura 4.1: Representação de um indivíduo

O processo de reprodução do algoritmo utiliza operadores para a geração de novas populações. Esse operadores são os operadores de mutação e crossover. Cada operador obedece as regras de restrições impostas pelas matrizes de restrições, ou seja, toda vez que um indivíduo é criado uma função nos algoritmos é chamada para validar se o indivíduo de acordo com essas regras. As regras são as mesmas utilizadas no trabalho de Ré et al [31], assim heranças, agregações e dependências inter-tipos não são quebradas. O operador de mutação altera a posição de uma classe/aspecto no cromossomo, desde que, nenhuma sequência ilegal seja introduzida. Por exemplo o cromossomo A da Figura 4.2 após sofrer

uma operação de mutação se torna o cromossomo *MUTANTE A* da Figura 4.2.

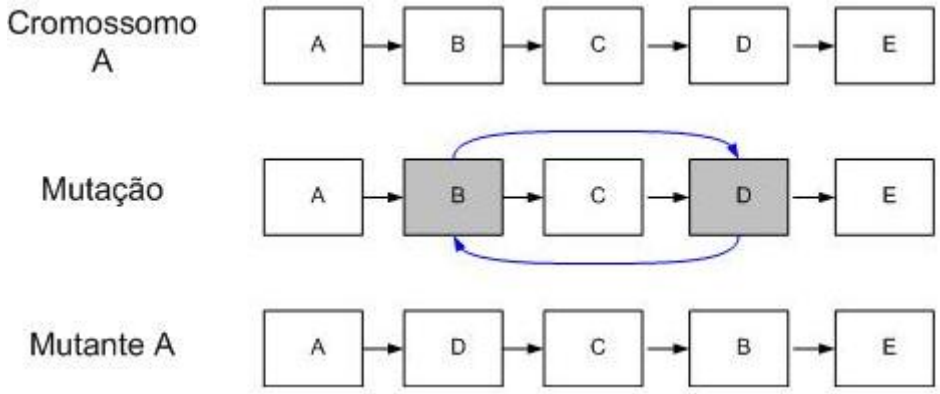

Figura 4.2: Representação de um indivíduo após sofrer mutação

O operador crossover faz com que dois cromossomos troquem *pedaços* das sequências garantindo que nenhuma sequência ilegal seja introduzida, ou seja, quando o operador de crossover é aplicado nos dois cromossomos A e B o cromossomo AB é gerado, formado pela primeira metade do cromossomo A e completado pela ordem da sequência do cromossomo B sem as classes que já estão na primeira parte do cromossomo conforme visto na Figura 4.3.

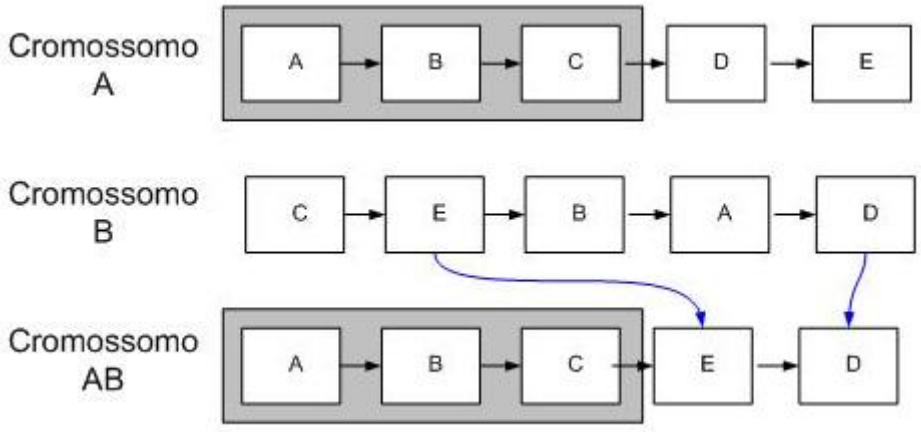

Figura 4.3: Representação de um indivíduo após sofrer crossover

Como função de avaliação (*fitness*), de uma sequência gerada, ou seja, o valor que determina a qualidade daquela solução, é utilizada uma função baseada na função utilizada nos experimentos de Briand et al [6] apresentadas na Equação (3.3) do Capítulo 3. As medidas consideradas para cálculo de *fitness* são o número de atributos e o número de

métodos do sistema que passam a ser contados da seguinte forma:

1. A : O número de atributos localmente declarados na classe ou aspecto quando o atributo aparece na lista de argumentos de algum método da classe ou pointcut/advice de uma aspecto, ou como tipo de retorno, atributo no método ou pointcut/advice, ou como parâmetro local. A complexidade medida é o numero máximo de atributos com os quais o *stub* gerado precisará lidar caso a dependência seja quebrada.

2. M : O número de métodos, inclusive construtores, localmente declarados na classe que é invocada pelos métodos da classe ou por pointcuts e advices nos aspectos. A complexidade medida é o número de métodos que é necessário simular no *stub* caso a dependência seja quebrada.

O método de seleção utilizado é o por torneio, ou seja, escolhem-se dois indivíduos aleatoriamente da população e o melhor é selecionado. Os pesos estipulados Wa e Wm da função de *fitness* são parâmetros do algoritmo de tal forma que $Wa + Wm = 1$. A função objetivo implementada é a mesma apresentada anteriormente seguindo a idéia da Equação 4.1. A maneira como é calculado esses valores segue a mesma proposta por Briand et al e pode ser vista no Capítulo 2.

$$OCplx(o) = \sum SCplx(k)_{k=1}^{d} \tag{4.1}$$

4.3 Algoritmos Multi-Objetivo Usando Dominância de Pareto

Como já explicado, o problema CAITO apresenta características de um problema multi-objetivo. Por isso a aplicação de algoritmos multi-objetivo pode trazer bons resultados. Foram implementados os algoritmos PACO, MOTS e NSGA-II descritos nas próximas seções. A população presente nesses algoritmos segue a mesma estrutura proposta no Algoritmo Genético visto na seção anterior. Cada indivíduo da população representa uma sequência de classes e aspectos que representam uma possível ordem de teste de integração.

4.3.1 PACO

O pseudo-código PACO pode ser visto no Algoritmo 8 retirado de [17]. A utilização do algoritmo no contexto Orientado a Aspectos é parecida a utilizada em Orientação a Objetos apenas utilizando as matrizes do problema no cálculo do *fitness* adaptadas ao contexto de Orientação a Aspectos.

Algoritmo 8 Exemplo PACO

1: $Inicicializaferomonio(F_1, F_2, T_0)$
2: **enquanto** $numiter \leq maxiter$ **faça**
3: **para** cada $Formiga$ **faça**
4: $p_1 = aleatorio(0, 1)$
5: $p_2 = 1 - p1$
6: $ini = GeraCandidatas()$
7: $s = GeraCaminhoInicial()$
8: $s = ConstroiCaminho(s, q, q_o, p_1, p_2, F_1, F_2)$
9: $AtualizaferomonioLocal(s, F_1, F_2)$
10: $s = BuscaLocal(s)$
11: $s^1 = BuscaLocal(s)$
12: $b = MelhorIteracao()$
13: $b^1 = SegundaMelhorIteracao()$
14: $AtualizaferomonioGlobal(b, b^1, F_1, F_2)$
15: $AtualizaPareto(P, s, s^1)$
16: **fim para**
17: $numiter = numiter + 1$
18: **fim enquanto**

O procedimento `GeraCandidatas` gera um conjunto inicial de classes e aspectos que não possui qualquer restrição quanto à precedência. A função `GeraCaminhoInicial` faz com que o conjunto gerado pela função anterior seja permutado aleatoriamente para gerar a parte inicial do vetor de classes, nessa função é verificado se a sequência criada é valida seguindo as dependências baseadas nas matrizes de restrições. O procedimento `Inicializaferomônio` inicializa duas matrizes de feromônios utilizando valor 1.0 para cada elemento da matriz. Os caminhos são construídos colocando-se cada formiga inicialmente na última classe ou aspecto em s resultante do procedimento `GeraCaminhoInicial` e a construção do restante do caminho é realizada utilizando o procedimento `ConstroiCaminho` que inicialmente gera uma lista de classes ou aspectos candidatos a serem incluídos no vetor pela formiga e que não violem a restrição de precedência. Desta lista um elemento

é selecionado de acordo com a Equação 4.2.

$$j = \begin{cases} \max_{i \in U} \left[\sum_{k=1}^{2} p_i^k \cdot \tau_{il}^k\right]^{\alpha} \cdot (\eta_{ij}^k)^{\beta} & \text{se } q \leq q_0 \\ p\,(i) & \text{caso contrário} \end{cases} \tag{4.2}$$

Onde $p(j)$ é dada pela probabilidade definida pela Equação 4.3.

$$p(i) = \frac{\sum_{k=1}^{2} p_i^k . (\tau_{ij}^k)^{\alpha} . (\eta_{ij}^k)^{\beta}}{\sum_{u \in U} (\sum_{k=1}^{2} p_i^k . (\tau_{il}^k)^{\alpha})} \tag{4.3}$$

Nessa fórmula o valor U representa o conjunto dos índices da matriz de feromônio, ou seja o número de classes e/ou aspectos presentes no sistema em teste. Para p1 e p2 têm-se valores aleatórios entre 0 ou 1 de modo que p1 + p2 = 1.0. Estes pesos representam a importância de cada objetivo k (k=1 métodos, k=2 atributos) na tomada de decisão da formiga h. A atualização local das matrizes de feromônio é realizada sempre após a construção do caminho completo por cada formiga conforme explicado no Capítulo 2. O valor do feromônio local é atualizado sempre que uma formiga segue de um ponto i para outro ponto j na trilha, e utiliza a seguinte regra: $\tau_{ij}^k = (1 - \rho) \cdot \tau_{ij}^k + \rho \cdot \tau_0$, na qual ρ é a classificação de evaporação do feromônio e τ_0 é o valor inicial desse feromônio. A cada atualização um contador é incrementado para se ter no final o número de avaliações realizadas. Já o feromônio global é atualizado quando cada formiga da população já construiu seu caminho, ou seja, a solução, e já utilizou a regra $\tau_{ij}^k = (1 - \rho) \cdot \tau_{ij}^k + \rho \cdot \Delta\tau_{ij}^k$, na qual $\Delta\tau_{ij}^k$ recebe o valor: 15 se (i, j) componente pertence a melhor ou segunda melhor solução (caminho); 10 se (i, j) pertence apenas a melhor solução (caminho); 5 se (i, j) pertence apenas a segunda melhor solução (caminho); e 0 nos demais casos.

A cada caminho completo construído por cada formiga a matriz de feromônio é atualizada localmente e a busca local é uma heurística de refinamento para encontrar ótimos locais no espaço de soluções. A busca local utilizada é o *Hill Climbing* que utiliza estrutura de vizinhanças geradas através de determinados operadores. A combinação de busca local com algoritmos de colônias de formigas tem melhorado o desempenho do algoritmo. [9]. O algoritmo utiliza o algoritmo de busca local para cada objetivo para melhorar as

soluções construídas pelas formigas em função de cada objetivo (método e atributo).

4.3.2 MOTS

A algoritmo MOTS também foi adaptado e aplicado para resolução do problema CAITO (Algoritmo 9) retirado de [17]. A utilização do algoritmo no contexto Orientado a Aspectos é parecida a utilizada em Orientação a Objetos apenas utilizando as matrizes do problema no cálculo do *fitness* adaptadas ao contexto de Orientação a Aspectos.

Algoritmo 9 Exemplo MOTS

```
1: enquanto numiter ≤ maxiter faça
2:   para x = 1; x ≤ TamanhoDaVizinhanca;x = x + 1 faça
3:     Vizinho = Operador(SolucaoGeradora)
4:     se contem(ListaTabu, Vizinho) então
5:       continue
6:     fim se
7:     custoA = FuncaoObjetivoA(Vizinho);
8:     custoB = FuncaoObjetivoB(Vizinho);
9:     Candidato = novoParPareto(Vizinho, custoA, custoB);
10:    se ListaCandidatos.tamanho() = 0 então
11:      ListaCandidatos.adiciona(Candidato);
12:    senão
13:      AtualizaPareto(ListaCandidatos, Candidato);
14:    fim se
15:  fim para
16:  CandidatoSorteado = Sorteio(ListaCandidatos.tamanho());
17:  SolucaoGeradora = ListaCandidatos.sorteio(CandidatoSorteado);
18:  ListaCandidatos.remove(CandidatoSorteado);
19:  se ListaTabu.tamanho() ≥ TamanhoMaximoTabu então
20:    ListaTabu.remove(1);
21:  fim se
22:  se ListaCandidatos.tamanho() = 0 então
23:    ListaPareto.adiciona(SolucaoGeradora);
24:  senão
25:    AtualizaPareto(ListaPareto, SolucaoGeradora);
26:  fim se
27:  ListaTabu.adiciona(SolucaoGeradora);
28:  numiter = numiter + 1;
29: fim enquanto
```

Conforme descrito no Capítulo 2, o MOTS segue basicamente a idéia de percorrer a estrutura de vizinhança utilizando o conceito de Pareto e utiliza: *ListaCandidatos*, *ListaPareto* e *ListaTabu*, como listas de apoio. Essas listas quando criadas, cada indivíduo

é validado para garantir que esse respeite as restrições impostas e mapeadas pelas matrizes de restrições. A lista de Candidatos recebe todos os vizinhos da solução inicial da iteração. No final da iteração, um desses vizinhos (solução geradora) é sorteado aleatoriamente para gerar a nova vizinhança. Cada vizinho gerado é adicionado nessa lista e nesse momento é chamada uma função que remove da lista as soluções que são dominadas pela solução que acabou de entrar. Se a nova solução vizinha não domina nenhuma solução na lista de candidatos então ela é removida. A Lista Pareto adiciona, ao final de cada iteração, a solução geradora (utilizada para gerar os vizinhos). Sempre que isso ocorre, a lista é atualizada comparando-se a solução que entrou com todas as outras. Saem as soluções dominadas e permanecem as não-dominadas.

A Lista Tabu, clássica das implementações Tabu, evita os ciclos na busca. Nessa implementação, um movimento que está na Lista Tabu é descartado e não entra na Lista de Candidatos. Ao final de cada iteração a solução geradora da vizinhança é adicionada nessa lista.

4.3.3 NSGA-II

O último algoritmo adaptado é o NSGA-II. O pseudo-código é apresentado pelo Algoritmo 10 retirado de [11]. Para implementação, a ferramenta *Open BEAGLE* [16], que é um *framework* que facilita a programação de Algoritmos Genéticos para os mais variados fins, foi utilizada. A utilização do algoritmo no contexto Orientado a Aspectos é a mesma que a utilizada em Orientação a Objetos. Os operadores de mutação e crossover seguem a mesma lógica aplicada no Algoritmo Genético criado para esse trabalho.

Foi implementada uma solução seguindo a idéia de Briand et al [6]. Uma ordenação de classes e aspectos representa um indivíduo, e consequentemente, as classes ou aspectos são os genes do cromossomo. Para a matriz de restrições uma matriz binária é implementada, sendo que o valor igual a 1 significa que há restrição e 0 que não há. São utilizadas duas outras matrizes, matriz de métodos e de atributos, respectivamente representando a dependência entre as classes e/ou aspectos no que se refere a métodos e atributos. Uma dependência associada entre as classes e/ou aspectos são representados por valores

Algoritmo 10 Exemplo Nsga-II

```
1: GeraPopulacaoInicial(P_0) , Q_0 = ∅
2: n = 0
3: selecao()
4: cruzamento()
5: mutacao()
6: GeraPopulacaoFilha(Q_0)
7: enquanto n ≤ maxiter faça
8:    R_n = P_n ⋃ Q_n
9:    OrdenaPorNãoDominância(R_n)
10:   Gera P_{n+1} = ∅
11:   enquanto |P_{n+1} + F_j| ≤ N faça
12:      Copia soluções de F_j em P_{n+1}
13:   fim enquanto
14:   Calcula as distâncias de multidão em F_j
15:   Ordena F_j conforme as distâncias d_j
16:   Copia as primeiras N − |P_{n+1}| soluções de F_j para P_{n+1}
17:   cruzamento()
18:   mutacao()
19:   GeraPopulacaoFilha(Q_{n+1})
20: fim enquanto
```

inteiros. Também são propostos dois métodos de normalização, que nada mais são que dois métodos que garantem que indivíduos iguais não sejam gerados e permaneçam na população e, que nenhuma sequência que não respeite a matriz de dependência seja gerada.

Inicialmente o algoritmo gera uma população aleatória P_0. Utilizando os operadores genéticos de mutação e crossover idênticos aos do Algoritmo Genético, e seleção por torneio a primeira população filha é gerada Q_0. As populações formadas por P_0 e Q_0 são unidas gerando assim uma nova população R_0.

Nas demais iterações que se seguem, dadas no algoritmo por $n = 1, 2, 3, \ldots, maxiter$ o algoritmo passa a trabalhar com a população R_n ordenada por não dominância, ou seja, por soluções não dominadas. Inicialmente todas as soluções são colocadas para um conjunto *front* F_0. Depois disso os pontos de F_0 são descartados do conjunto global de soluções e um novo conjunto de soluções não dominadas forma então um novo *front* F_1 e assim por diante.

Para formar uma nova população, a população P_{n+1} é gerada com as soluções dispersas de acordo com um valor de distância de multidão. Isso é feito através da função

`CalculateCrowdingDistances` [11] aplicada a cada ponto em F_j vista no Capítulo 2.

Assim, depois da seleção dos pontos pertencentes a P_{n+1}, essa população é então ordenada usando o operador $\geq_n$ que é definido pela relação de dominância e distância de multidão. Finalmente, os operadores genéticos são aplicados em P_{n+1} para formar uma nova população Q_{n+1} e as iterações continuam.

4.4 Experimentos

Esta seção descreve experimentos que avaliam a utilização dos algoritmos de busca descritos no contexto de programação Orientada a Aspectos comparando seus resultados com as estratégias tradicionais.

4.5 Sistemas Usados

A Orientação a Aspectos é um paradigma novo, isso torna escasso o número de sistemas *Open Source* baseados em Orientação a Aspectos disponibilizados pela comunidade científica.

O principal *pacote* disponível para comunidade utilizar no desenvolvimento Orientado a Aspectos é o pacote *AspectJ* [37]. Juntamente com esse pacote, um conjunto de sistemas implementados em Orientação a Aspectos, é disponibilizado para estudo. Esses sistemas são relativamente pequenos. Para aplicação dos algoritmos dois sistemas foram escolhidos:

1. *Telecom*: é uma implementação que simula a comunicação entre indivíduos através de mensagens, formada por 10 classes e 3 aspectos. A versão utilizada para esse trabalho é a presente no pacote AspectJ 1.6.0, diferente da versão adaptada com o uso de persistência utilizada em [31] para avaliação de estratégias tradicionais.

2. *Spacewar*: é uma implementação do clássico jogo de video game Spacewar. Esse sistema foi desenvolvido para ser um exemplo de game usando aspectos, formado por 14 classes e 4 aspectos.

Os ORDs estendidos para os sistemas *Telecom* e *Spacewar* foram criados através de

uma engenharia reversa das classes e aspectos para mapeamento das relações. Esses ORDs e matrizes gerados estão no Apêndice A.

4.6 Execução dos Algoritmos e Estratégias

Para o sistemas em questão foram aplicadas as quatro estratégias explicadas no Capítulo 3: *combined*, *incremental+*, *reverse* e *random*. Os algoritmos meta-heurísticos foram executados de acordo com os parâmetros da Tabela 4.1. Para esse trabalho os pesos do Algoritmo Genético foram calculados de acordo com o número de atributos e métodos respectivamente de cada classe. Para isso foram somados o número total de métodos e total de atributos e determinado as porcentagens relativas ao total da soma dos mesmos para cada aplicação. O valor encontrado ficou 50% para atributos e 50% para métodos em ambas as aplicações. Os demais parâmetros utilizados nos outros algoritmos foram definidos a partir dos resultados obtidos por aplicações dos mesmos algoritmos em sistemas Orientados a Objeto e alterados empiricamente [11] [17] [9]. Todas as abordagens baseadas em buscas meta-heurísticas foram executadas 30 vezes.

4.7 Resultados

Após as aplicações dos algoritmos as sequências resultantes foram obtidas e os seus respectivos valores de atributos e métodos. As Tabelas 4.2 e 4.3 ilustram os resultados obtidos respectivamente para os sistemas Spacewar e Telecom. As sequências de teste obtidas por todas as estratégias são apresentadas no Apêndice A. Por serem sistemas pequenos os algoritmos multi-objetivo, em todas as execuções, sempre conseguiram chegar às melhores aproximações, muitas vezes com sequências diferentes embora seu valor de *fitness* fosse o mesmo.

Tabela 4.1: Tabela de Parâmetros

Programa	Telecom	Spacewar
AG		
população	100	100
maxiter	300	300
crossover	0.6	0.6
mutação	0.1	0.1
Wa	0.5	0.5
Wm	0.5	0.5
PACO		
formigas	11	22
maxiter	300	300
$q0$	0.75	0.75
α	1	1
β	1	1
τ_{min}	0.0001	0.0001
τ_0	1	1
NSGA-II		
população	100	100
maxiter	300	300
crossover	0.6	0.6
mutação	0.1	0.1
MOTS		
maxseed	7	7
maxiter	300	300
Lista tabu	7	7
Vizinhança	15	15

4.8 Análise dos Resultados

Para facilitar a visualização, as estratégias *random* e *reverse* por terem obtido resultados muito inferiores às demais não foram utilizadas nas comparações. Ao final pode-se realizar uma comparação visual das estratégias tradicionais e do Algoritmo Genético. Pelo gráfico da Figura 4.4 percebe-se que para o sistema Spacewar, nenhuma das soluções das estratégias são melhores, no entanto, quando comparadas ao resultado obtido pelo Algoritmo Genético, percebe-se que as soluções das estratégias tradicionais são piores. Visualmente percebe-se que a estratégia incremental+ apresenta resultados melhores que a estratégia combined.

Já para o sistema Telecom, percebe-se pelo gráfico da Figura 4.5 que a solução obtida pela estratégia *combined* é melhor que a solução obtida pela estratégia *incremental+*, mas

Tabela 4.2: Resultados Spacewar

Abordagem	atributos	métodos
Combined	55	46
Incremental+	51	46
Random	61	46
Reverse	68	47
AG	49	44
PACO	47	44
	48	42
MOTS	48	42
	48	36
NSGA-II	47	36

Tabela 4.3: Resultados Telecom

Abordagem	atributos	métodos
Combined	14	9
Incremental+	15	9
Random	26	30
Reverse	29	35
AG	11	8
PACO	11	3
	7	4
MOTS	11	4
NSGA-II	11	4
	7	4

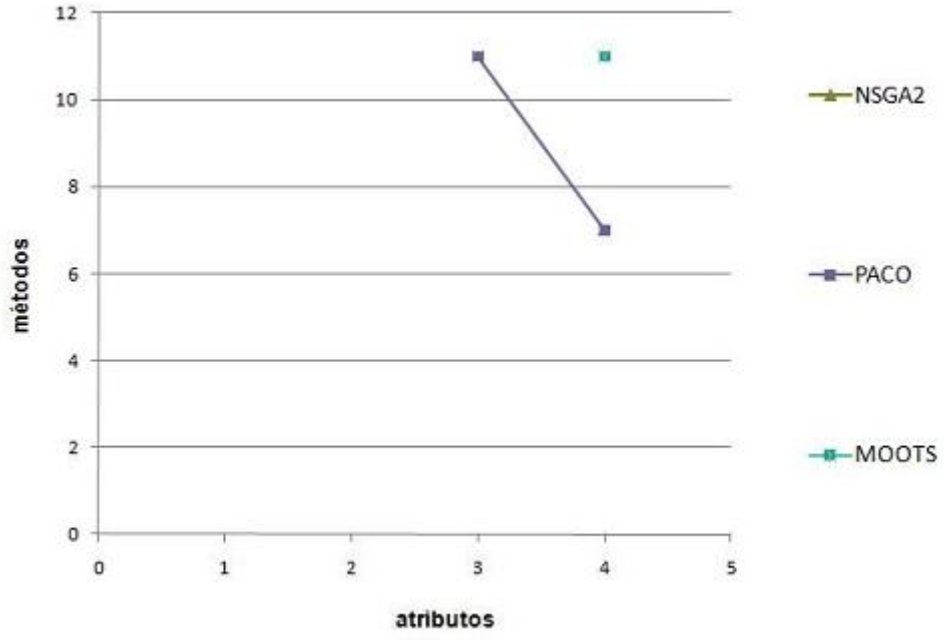

Figura 4.4: Resultados das estratégias tradicionais para o programa Spacewar

mesmo assim os resultados são piores que os obtidos pelo Algoritmo Genético.

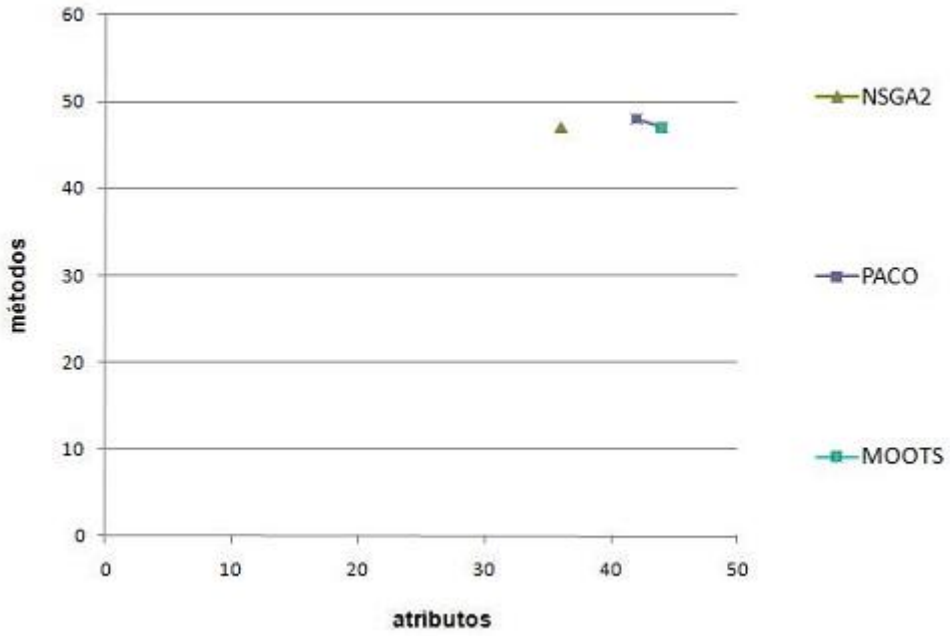

Figura 4.5: Resultados das estratégias tradicionais para o programa Telecom

Para os algoritmos multiobjetivos têm-se os gráficos das Figuras 4.6 e 4.7. Algumas fronteiras são formadas por diversas sequências que estão no mesmo ponto no espaço de soluções. Ou seja, fronteiras formadas apenas por um ponto.

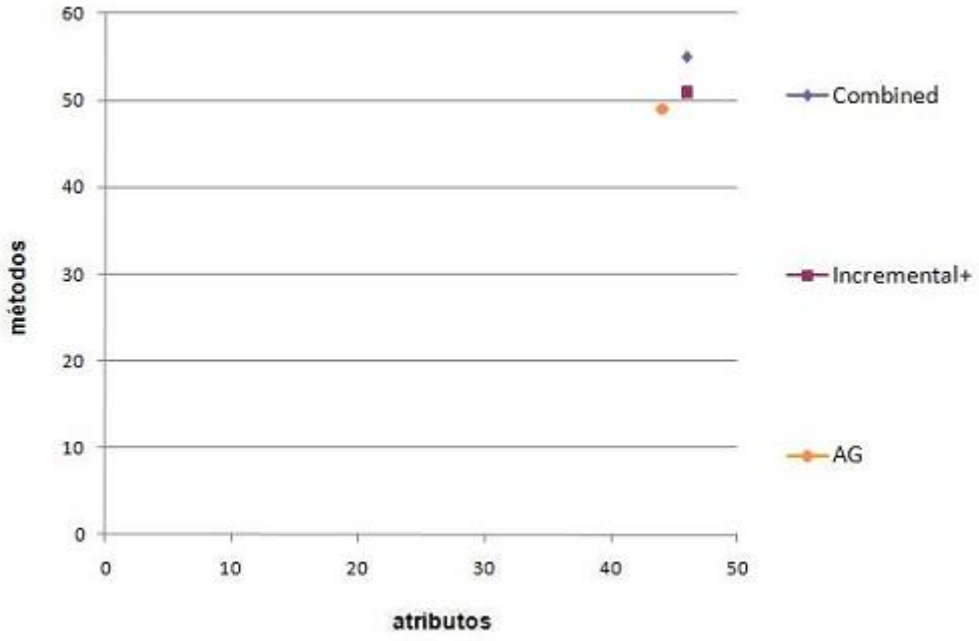

Figura 4.6: Resultados dos algoritmos meta-heurísticos para o programa Spacewar

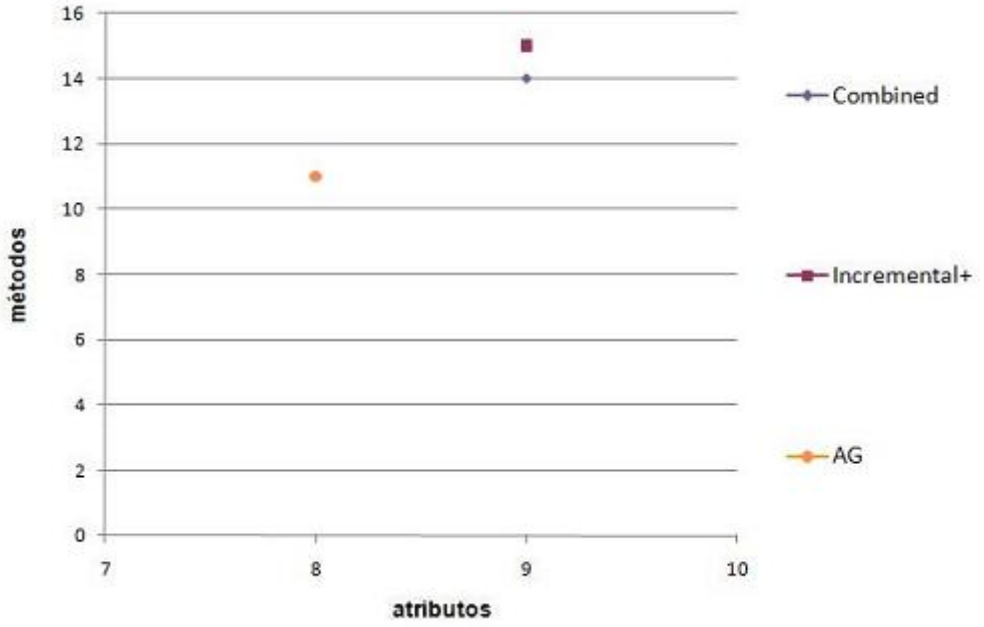

Figura 4.7: Resultados dos algoritmos meta-heurísticos para o programa Telecom

Para o Sistema Spacewar quando comparam-se os resultados entre os algoritmos multi-objetivos através dos gráficos percebe-se que os melhores resultados são do algoritmo NSGA-II e PACO, sendo que o algoritmo PACO apresentou uma maior diversidade no resultado da busca do espaço de soluções apresentando dois pontos em sua fronteira enquanto o algoritmo NSGA-II apenas um. O algoritmo MOTS apresentou a terceira melhor fronteira. Todos os algoritmos multi-objetivos apresentaram pontos melhores que as estratégias tradicionais e que o Algoritmo Genético Simples. Da mesma forma que o ocorrido com o sistema Spacewar, para o sistema Telecom percebe-se que todos os algoritmos NSGA-II, MOTS e PACO apresentaram melhores resultados do que os obtidos através das técnicas (Combined e Incremental+) e que o Algoritmo Genético Simples pois seus resultados sempre foram melhores em todos os objetivos como pode ser visto na Tabela 4.3. Quando comparam-se os resultados entre os algoritmos multi-objetivos através dos gráficos percebe-se que os melhores resultados são do algoritmo PACO. Os algoritmos NSGA-II e MOTS obtiveram a segunda e terceira melhores aproximações. Além disso percebe-se que os algoritmos multi-objetivos acabavam chegando em espaços ótimos da solução, porém esses espaços eram diferentes para cada abordagem.

4.8.1 Número de avaliações de fitness

Uma medida que permite uma melhor avaliação dos resultados obtidos é o número de avaliações realizadas pelos algoritmos aplicados. Toda vez que o algoritmo calcula o *fitness* de uma determinada sequência um contador é incrementado no algoritmo, assim, revelando ao final quantas vezes para cada solução, após 300 iterações, é necessário recalcular o valor de *fitness*. Ao final das iterações obtiveram-se os totais apresentados na Tabela 4.4 para os sistemas Spacewar e Telecom.

Tabela 4.4: Número de Avaliações de Fitness

Algoritmo	SpaceWar	Telecom
AG	90000	90000
PACO	2611340	1813011
MOTS	530006	380006
NSGA-II	19469	19423

Analisando-se os resultados obtidos por cada abordagem e o respectivo número de avaliações necessário para chegar a esses resultados, resolveu-se delimitar como critério de parada o menor número de avaliação dos algoritmos multi-objetivo utilizados, no caso, o algoritmo que obteve o menor número de avaliações foi o NSGA-II. Assim, o critério de parada dos algoritmos passou a ser a realização de 19469 avaliações para o sistema SpaceWar e 19432 para o sistema Telecom de uma sequência encontrada.

Os demais parâmetros foram mantidos conforme apresentados na Tabela 4.1, ignorando-se apenas o valor de *maxiter*. Obtiveram-se então os resultados apresentados nas Tabelas 4.5 e 4.6.

Tabela 4.5: Resultados Spacewar - Critério de Parada número de avaliações

Estratégia	atributos	métodos
PACO	47	44
	48	42
MOTS	48	42
	48	36
NSGA-II	47	36

Mesmo assim pelo fato de as aplicações serem pequenas, os algoritmos conseguiram chegar às mesmas soluções obtidas com o critério de parada definido número máximo

Tabela 4.6: Resultados Telecom - Critério de Parada número de avaliações

Estratégia	atributos	métodos
PACO	11	3
	7	4
MOTS	11	4
NSGA-II	11	4
	7	4

de iterações. Ao final, percebeu-se que a análise das soluções poderia ser feita sem a utilização dos indicadores, pois graficamente já era possível uma análise completa desses.

CAPÍTULO 5

CONCLUSÕES

No teste de integração de programas Orientado a Aspectos, da mesma forma que para sistemas Orientados a Objeto, é necessário estabelecer uma ordem de integração para o teste. Pensando nisso, como visto neste trabalho, algumas estratégias foram aplicadas considerando um tipo especial de grafo denominado ORD. Esse grafo representa as relações de dependência entre classes e aspectos. Abordagens baseadas em grafos foram propostas sendo assim um primeiro passo para a resolução do problema de custo de geração de *stubs*, aqui chamado de CAITO.

Diante dos recursos já utilizados e propostas existentes no meio científico, este trabalho investigou o uso de soluções baseados em algoritmos de busca, com o intuito de se obter uma maior diversificação das soluções disponíveis. Para isso os dois sistemas escolhidos, embora pequenos, foram suficientes para demonstrar que soluções baseadas em algoritmos meta-heurísticos conseguem encontrar melhores soluções para o problema em questão.

O trabalho propôs que o problema CAITO fosse tratado como multi-objetivo pois diferentes fatores podem estar relacionados ao custo de construção de *stubs*, tendo sido explorados duas medidas de acoplamento: número de métodos e número de atributos e dois tipos de função de avaliação: uma baseada numa função de agregação e outra baseada no conceito e dominância de Pareto, através dos algoritmos NSGA-II, PACO e MOTS.

Em um estudo de avaliação os algoritmos meta-heurísticos foram comparados com estratégias tradicionais. Os resultados obtidos por todos os algoritmos foram sempre superiores ao das estratégias tradicionais. O Sistema Spacewar apresentou os melhores resultados através dos algoritmos NSGA-II e PACO, a outra abordagem multi-objetiva, MOTS, apresentou o terceiro melhor resultado. Os resultados obtidos pelo Algoritmo Genético Simples foram inferiores as abordagens multi-objetivas porém melhores que as estratégias tradicionais (combined, incremental+, reverse e random). Entre as estratégias

tradicionais, a estratégia incremental+ apresentou resultados melhores que a estratégia combined para esse sistema.

O Sistema Telecom apresentou os melhores resultados através do algoritmo PACO, as outras duas abordagens multi-objetivas, NSGA-II e MOTS, apresentaram respectivamente, segunda e terceira melhores fronteiras. Os resultados obtidos pelo Algoritmo Genético Simples foram inferiores as abordagens multi-objetivas porém melhores que as estratégias tradicionais (combined, incremental+, reverse e random). Entre as estratégias tradicionais, a estratégia combined apresentou resultados melhores que a estratégia incremental+ para esse sistema.

Como trabalho futuro sugere-se a condução de experimentos com sistemas que contenham um maior número de classes e aspectos e relações de dependência permitindo assim resultados mais significativos.

Além disso, sugere-se a utilização de outros objetivos tais como as métricas de acoplamento propostos por Abdurazik e Offutt [1] e outras medidas relacionadas especificamente a Orientação a Aspectos tais como relacionamentos inter-tipos.

BIBLIOGRAFIA

[1] A. Abdurazik e J. Offutt. Coupling-based class integration and test order. *International Workshop on Automation of Software Test*, Shanghai, China, Maio de 2006. ACM.

[2] R. T. Alexander, J. M. Bieman, e A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Relatório técnico, Colorado State University, Fort Collins, CO, 2004.

[3] R.T. Alexander. The real costs of aspect-oriented programming. *IEEE Software*, 20(6), 2003.

[4] A. Baykasoglu, S. Owen, e N. Gindy. A Taboo Search Based Approach to Find the Pareto Optimal Set in Multiple Objective Optimisation. *Journal of Engineering Optimization*, páginas 731–748, 1999.

[5] C. Blum e A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[6] L. C. Briand, J. Feng, e Y. Labiche. *Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders.* Carleton University, Technical Report SCE-02-03, Outubro de 2002.

[7] L. C. Briand, J. Feng, e Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. *14th International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, Julho de 2002.

[8] L. C. Briand e Y. Labiche. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, Julho de 2003.

[9] R. V. Cabral, A. Pozo, e S. R. Vergilio. A Pareto ant colony algorithm applied to the class integration and test order problem. *International Conference on Testing Software and Systems 2010*.

[10] M. Ceccato, P. Tonella, e F. Ricca. Is AOP oce easier or harder to test than OOP code. *First Workshop on Testing Aspect-Oriented Program (WTAOP)*, Chicago, Illinois, 2005.

[11] K. Deb, S. Agrawal, A. Pratap, e T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization:NSGA-II. Relatório técnico, 2000.

[12] M. Delamaro, J. C. Maldonado, e M Jino. *Introdução ao Teste de Software*. Campus - Elsevier, 2007.

[13] K. Doerner, W. J. Gutjahr, R. F. Hartl, C. Strauss, e C. Stummer. Pareto ant colony optimization: A metaheuristic approach to multiobjective portfolio selection. *Annals of Operation Research*, (131):79–99, 2004.

[14] M. Dorigo, V. Maniezzo, e A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:29–41, 1996.

[15] C.M. Fonseca e P.J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. *Proceedings of the 5th International Conference on Genetic Algorithms*, páginas 416–423, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[16] C. Gagné e M. Parizeau. Open Beagle: a C++ framework for your favorite evolutionary algorithm. *SIGEVOlution*, 1(1):12–15, 2006.

[17] F. Glover. Tabu search, Part I. *ORSA Journal on Computing*, 1(3), Verão de 1989.

[18] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, Janeiro de 1989.

[19] M. P. Hansen e A. Jaszkiewicz. *Evaluating the Quality of Approximations to the Non-Dominated Set*. IMM Technical Report, 1998.

[20] M. Harman. The current state and future of search based software engineering. *Future of Software Engineering - FOSE*, páginas 342–357. IEEE Computer Society, Maio de 2007.

[21] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[22] P. C. Jorgensen. *Software Testing: A Craftsman's Approach, Third Edition*. AUERBACH, 3 edition, 2008.

[23] J. Knowles, L. Thiele, e E. Zitzler. A tutorial on the performance assessment of stochastic multiobjective optimizers. Relatório técnico, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, fevereiro de 2006. Revised version.

[24] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, e C. Chen. A test strategy for object-oriented programs. *19th International Computer Software and Applications Conference*. IEEE Computer Society, Agosto de 1995.

[25] D. C. Kung, J. Gao, P. Hsia, J. Lin, e Y. Toyoshima. Class firewal, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Program*, 8(2), 1995.

[26] H. Melton e E. Tempero. An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12:389–415, 2007.

[27] M. Michalewicz. Evolutionary computation techniques and their applications. *International Conference on Intelligent Processing System*, 1997.

[28] M. A. C. Pacheco. Otimização de problemas de múltiplos objetivos. Relatório técnico, ICA DEE/PUC-Rio, 2 de 2006.

[29] J. M. Pasia, R.F. Hart, e K. F. Doerner. Solving a bi-objective flowshop scheduling problem by pareto-ant colony optimization. *Lecture Notes in Computer Science*, páginas 294–305, Agosto de 2006.

[30] R.S. Pressman. *Software Engineering: A Practitioner Approach*. McGraw-Hill, 2006.

[31] R. Ré, O. A. L. Lemos, e P. C. Masiero. Minimizing stub creation during integration test of aspect-oriented programs. *3rd workshop on Testing aspect-oriented programs*, páginas 1–6, Vancouver, British Columbia, Canada, Março de 2007.

[32] R. Ré e P. C. Masiero. Integration testing of aspect-oriented programs: a characterization study to evaluate how to minimize the number of stubs. *Brazilian Symposium on Software Engineering*, páginas 411–426, João Pessoa, Brazil, Outubro de 2007. Brazilian Computer Society.

[33] S. Reid e Glenford J. Myers. The art of software testing. *Software Test*, 15(2):136–137, 2005.

[34] N. Srinivas e K Deb. Multi-objective function optimization using Non-Dominated Sorting Genetic Algorithms. *Evolutionary Computation*, 1995.

[35] J. E. Stiglitz. The invisible hand and modern welfare economics. Working Paper 3641, National Bureau of Economic Research, March de 1991.

[36] K.-C. Tai e F. J. Daniels. Test order for inter-class integration testing of object-oriented software. *21st International Computer Software and Applications Conference*, páginas 602–607. IEEE Computer Society, Agosto de 1997.

[37] The AspectJ Team. *AspectJ programming guide*. Available from: http://dev.eclipse.org/viewcvs/indextech.cgi/ checkout /aspectj-home/doc/progguide/index.html, acessado em 31.01.2005 de 2005.

[38] Y. L. Traon, T. Jéron, J.-M. Jézéquel, e P. Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, páginas 12–25, 2000.

[39] A.M.R Vincenzi, M. E. Delamaro, e A.L.S. Domingues. Teste orientado a objetos e de componentes. *In: Delamaro, M. E. Maldonado, J. C. and Jino, M. Introdução ao teste de software*. Ed. Campus-SBC, 2007.

[40] E. Zitzler, K. Deb, e L. Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 2000.

APÊNDICE A

ORDS E MATRIZES DE DEPENDÊNCIA UTILIZADOS

A.1 Spacewar ORD

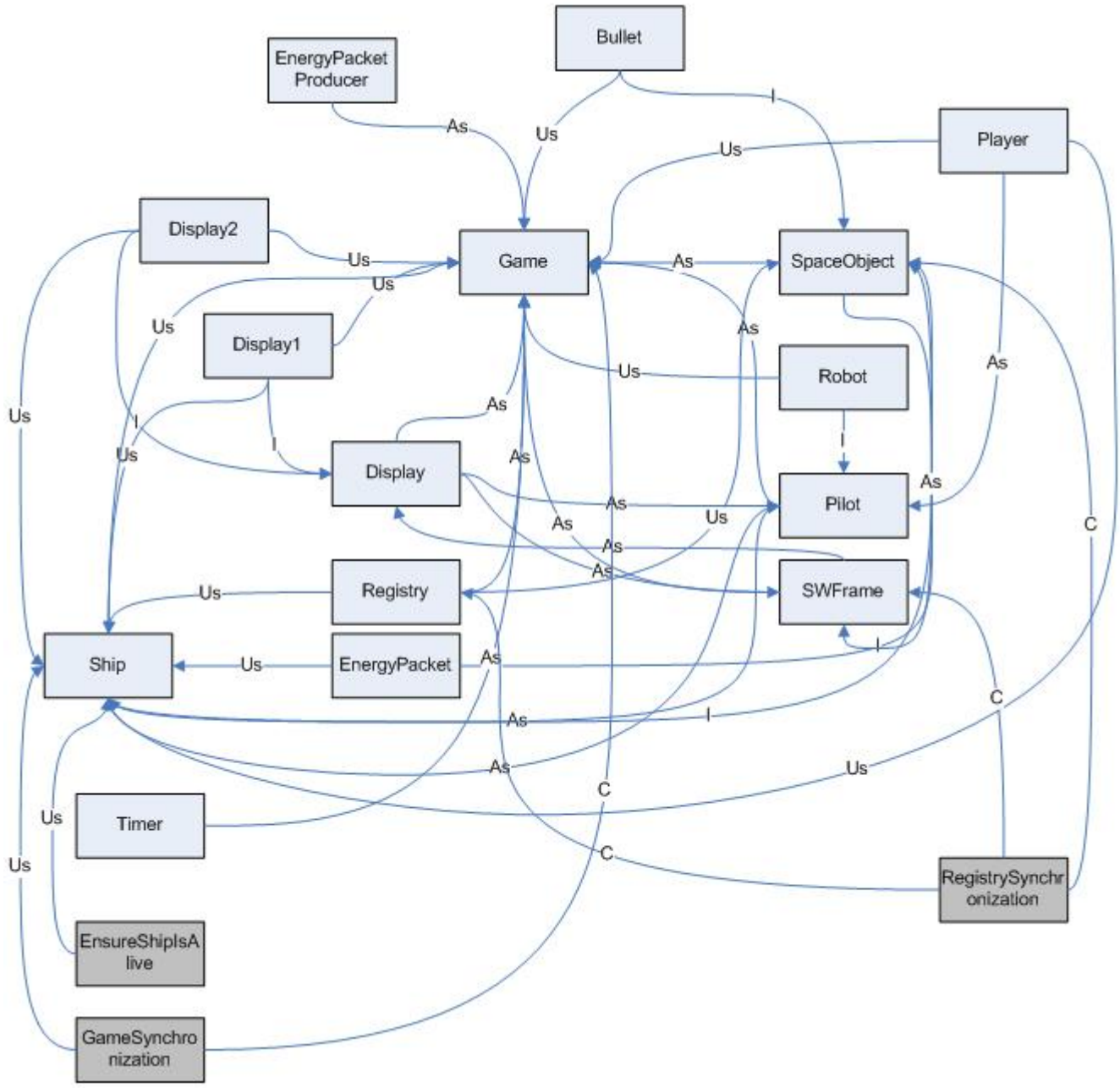

Figura A.1: Spacewar ORD

A.2 Tabela Nome de Classes/Aspectos Spacewar

Tabela A.1: Classes e Aspectos SPACEWAR

1	Bullet
2	Display
3	Display1
4	Display2
5	EnergyPacket
6	EnergyPacketProducer
7	EnsureShipIsAlive
8	Game
9	GameSynchronization
10	Pilot
11	Player
12	Registry
13	RegistrySynchronization
14	Robot
15	Ship
16	SpaceObject
17	SWFrame
18	Timer

A.3 Matriz de atributos - Spacewar

Tabela A.2: Matriz de Atributos SPACEWAR

-	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	3	0	7	0	0	0	0	0	0	2	0
3	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1	0	0
4	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1	0	0
5	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	2	0	0
6	0	0	0	0	1	0	0	2	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
8	2	0	0	0	0	1	0	0	0	6	1	3	0	1	5	5	0	2
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	0	2	0	0	0	0	0	0	3	0	0	0
11	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	3	0	0	0	0	0	0	3	7	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
14	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
15	1	0	0	0	2	0	0	1	0	3	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
17	0	3	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0

A.4 Matriz de métodos - Spacewar

Tabela A.3: Matriz de Métodos SPACEWAR

-	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	1	0
3	1	1	0	0	1	0	0	0	0	2	0	0	0	0	6	1	1	0
4	1	1	0	0	1	0	0	0	0	2	0	0	0	0	6	1	1	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0
8	0	0	0	0	0	1	0	0	0	2	1	4	0	2	5	0	0	2
9	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	0	0	5	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	1	0	0	0	0	16	0	0	0
15	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A.5 Matriz de dependências - Spacewar

Tabela A.4: Lista de dependências Spacewar

Bullet	8,16
Display	8,10,17
Display1	1,2,5,8,10,15,16,17
Display2	1,2,5,8,10,15,16,17
EnergyPacket	8,16
EnergyPacketProducer	5,8
EnsureShipIsAlive	-
Game	1,6,10,12,18
GameSynchronization	8
Pilot	-
Player	10
Registry	8,15,16
RegistrySynchronization	12,15,16
Robot	10,15
Ship	16
SpaceObject	8
SWFrame	2,8
Timer	8

A.6 Matriz de restrições - Spacewar

Tabela A.5: Lista de restrições Spacewar

Bullet	8,16
Display	10
Display1	1,2
Display2	1,2
EnergyPacket	8,16
EnergyPacketProducer	5,8
EnsureShipIsAlive	-
Game	-
GameSynchronization	-
Pilot	8
Player	-
Registry	8
RegistrySynchronization	-
Robot	15
Ship	5,10
SpaceObject	8
SWFrame	2,8
Timer	-

A.7 Sequências - Spacewar

Tabela A.6: Sequências Spacewar

Abordagem	sequências
Combined	[9, 8, 7, 11, 13, 18, 12, 10, 16, 2, 17, 5, 1, 4, 3, 6, 15, 14]
Incremental+	[11, 9, 8, 7, 13, 18, 12, 10, 16, 2, 17, 5, 1, 4, 3, 6, 15, 14]
Random	[15,18,11,13,12,7,19,17,10,5,14,3,1,2,4,6,8,16]
Reverse	[14,15,6,3,4,1,5,17,2,16,10,12,18,13,11,7,8,9]
AG	[7, 11, 9, 8, 13, 18, 12, 10, 16, 2, 17, 5, 1, 4, 3, 6, 15, 14]
PACO	[8, 11, 13, 12, 10, 16, 2, 17, 7, 5, 1, 4, 3, 9, 6, 15, 18, 14] [8, 11, 13, 12, 10, 16, 2, 17, 7, 5, 1, 4, 6, 3, 9, 15, 18, 14] [8, 11, 13, 12, 10, 16, 2, 7, 17, 5, 1, 4, 3, 6, 9, 15, 18, 14] [8, 11, 13, 12, 10, 16, 2, 17, 18, 7, 5, 1, 4, 3, 6, 9, 15, 14]
MOTS	[8, 11, 13, 12, 10, 16, 2, 17, 18, 5, 1, 4, 7, 6, 9, 15, 14, 3] [8, 11, 13, 12, 10, 16, 2, 17, 5, 18, 1, 4, 7, 6, 9, 15, 14, 3]
NSGA-II	[8, 11, 13, 12, 10, 16, 2, 17, 18, 5, 1, 7, 4, 6, 9, 15, 14, 3]

A.8 Telecom ORD

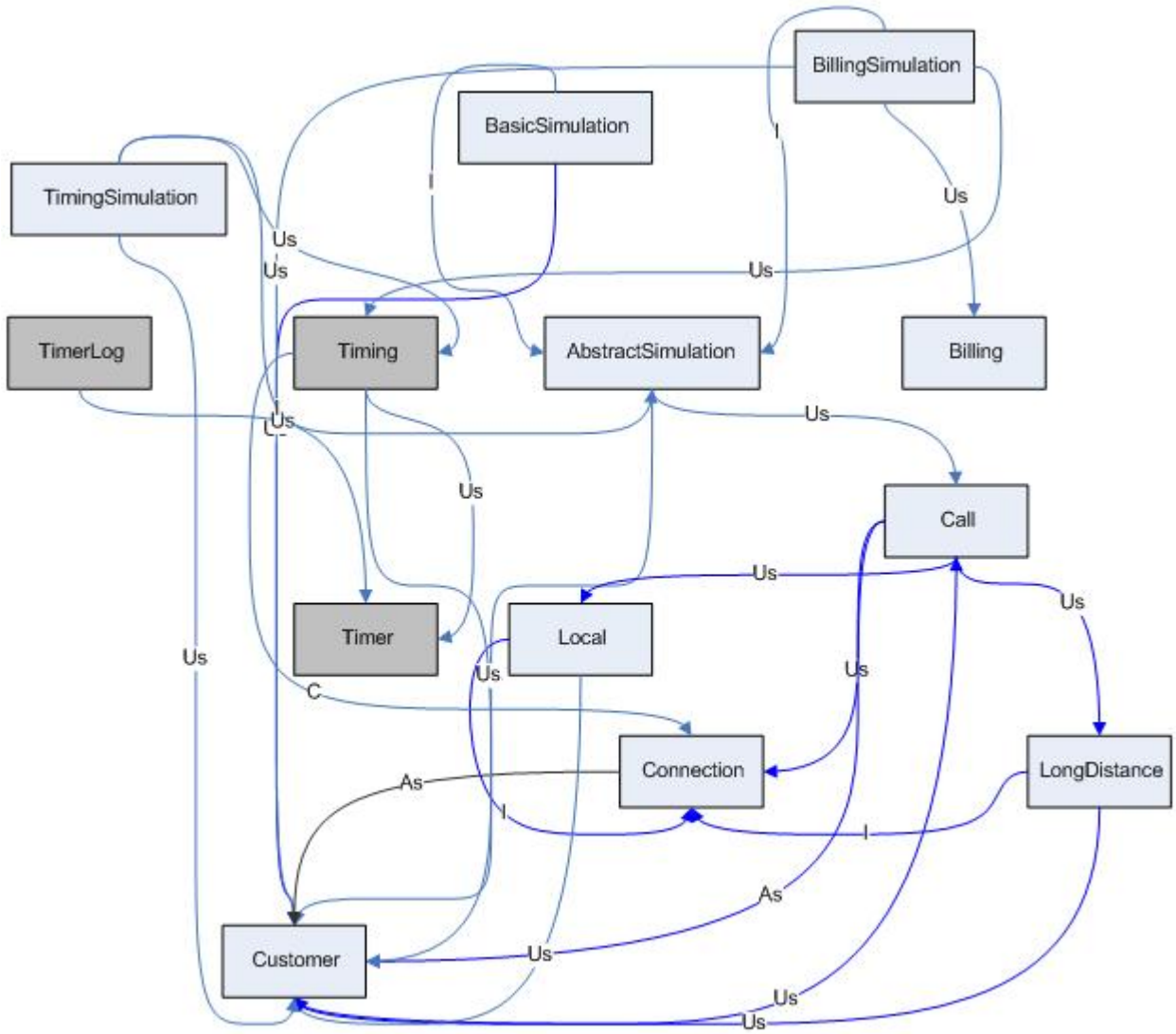

Figura A.2: Telecom ORD

A.9 Tabela Nome de Classes/Aspectos - Telecom

Tabela A.7: Classes e Aspectos TELECOM

1	AbstractSimulation
2	BasicSimulation
3	Billing
4	BillingSimulation
5	Call
6	Connection
7	Customer
8	Local
9	LongDistance
10	Timer
11	TimerLog
12	Timing
13	TimingSimulation

A.10 Matriz de atributos - Telecom

Tabela A.8: Matriz de Atributos TELECOM

-	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	0	2	0	4	0	0	0	0	0	0
2	0	0	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	1	0	0	0	0	1	0
5	0	0	0	0	0	6	6	0	0	0	0	0	0
6	0	0	0	0	0	0	5	0	0	0	0	0	0
7	0	0	0	0	7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	2	0	0	0	0	0	0
9	0	0	0	0	0	0	2	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	4	1	0	0	2	0	0	0
13	0	0	0	0	0	0	1	0	0	0	0	1	0

A.11 Matriz de métodos - Telecom

Tabela A.9: Matriz de Métodos TELECOM

-	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	0	0	0	9	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	2	0	0	0	0	0	0	0	0	2	0
5	0	0	0	0	0	3	1	1	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	4	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	4	0	0	0
12	0	0	0	0	0	5	1	0	0	2	0	0	0
13	1	0	0	0	0	0	0	0	0	0	0	2	0

A.12 Matriz de restrições - Telecom

AbstractSimulation	5,7,
BasicSimulation	1,7,
Billing	-
BillingSimulation	1,3,7,12,
Call	6,7
Connection	
Customer	
Local	6,7,
LongDistance	6,7,
Timer	-
TimerLog	10
Timing	6,7,10
TimingSimulation	1,7,12,

A.13 Matriz de Dependências - Telecom

AbstractSimulation	5,7,
BasicSimulation	1,7,
Billing	-
BillingSimulation	1,3,7,12,
Call	6,7,8,9,
Connection	7,
Customer	5,
Local	6,7,
LongDistance	6,7,
Timer	-
TimerLog	1
Timing	6,7,1
TimingSimulation	1,7,12,

A.14 Sequências - Telecom

Tabela A.10: Sequências Telecom

Abordagem	sequências
Combined	[6, 9, 10, 2, 3, 5, 7, 1, 12, 13, 8, 4, 11]
Incremental+	[6, 10, 8, 9, 11, 5, 7, 12, 3, 1, 13, 2, 4]]
Random	[11,3,8,1,13,5,9,12,6,2,4,7,1]
Reverse	[11,4,8,13,12,1,7,5,3,2,10,9,6]
AG	[6, 10, 9, 5, 3, 7, 1, 11, 2, 12, 8, 4, 13]
PACO	[7, 10, 6, 12, 9, 3, 8, 5, 11, 1, 4, 2, 13] [10, 6, 5, 7, 8, 12, 9, 3, 11, 1, 2, 13, 4]
MOTS	[7, 10, 6, 12, 9, 3, 8, 5, 11, 1, 4, 2, 13] [7, 10, 6, 8, 3, 12, 11, 5, 1, 9, 4, 13, 2] [7, 6, 8, 3, 10, 11, 2, 9, 12, 5, 1, 4, 13] [10, 7, 6, 3, 8, 9, 12, 13, 5, 1, 4, 11, 2]
NSGA-II	[7, 3, 6, 8, 9, 10, 5, 12, 4, 2, 11, 1, 13] [7, 6, 5, 3, 10, 8, 1, 12, 13, 11, 9, 4, 2]

RODRIGO GALVAN

UTILIZANDO ALGORITMOS DE BUSCA META-HEURÍSTICA PARA ESTABELECER SEQUÊNCIAS DE TESTE DE INTEGRAÇÃO PARA PROGRAMAS ORIENTADOS A ASPECTOS

Dissertação apresentada como requisito à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Silvia Regina Vergilio

CURITIBA

2010