UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

GENETIC PACKING: SOFTWARE PARA EMPACOTAMENTO TRIDIMENSIONAL HETEROGÊNEO EM CONTÊINERES

DANIEL PAMPLONA SOARES

BLUMENAU
2017

DANIEL PAMPLONA SOARES

GENETIC PACKING: SOFTWARE PARA EMPACOTAMENTO TRIDIMENSIONAL HETEROGÊNEO EM CONTÊINERES

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Daniel Theisges dos Santos, Mestre - Orientador

BLUMENAU
2017

GENETIC PACKING: SOFTWARE PARA EMPACOTAMENTO TRIDIMENSIONAL HETEROGÊNEO EM CONTÊINERES

Por

DANIEL PAMPLONA SOARES

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Daniel Theisges dos Santos, Mestre – Orientador, FURB
Membro:	Prof(a). Andreza Sartori, Doutora – FURB
Membro:	Prof(a). Roberto Heinzle, Doutor – FURB

Blumenau, 06 de Julho de 2017

Dedico este trabalho aos meus familiares, em especial aos que sempre me apoiaram no decorrer do curso.

AGRADECIMENTOS

Dedico esse trabalho aos meus pais Nelson Wendelino Pamplona Soares e Isoldina Soares, por sempre me apoiarem e me incentivarem ao decorrer do curso.

A minha namorada Bruna Rodrigues, por toda paciência e apoio.

Ao meu filho que está a caminho, Arthur Pamplona Soares, por ser meu principal motivador nos momentos difíceis.

Ao meu orientador Daniel Theisges dos Santos, por acreditar no desenvolvimento do trabalho, pelos conselhos, por confiar no meu conhecimento técnico, pelo tempo, empenho e ótimas orientações durante o desenvolvimento desse trabalho.

O insucesso é apenas uma oportunidade para recomeçar com mais inteligência.

Henry Ford

RESUMO

Este trabalho apresenta o desenvolvimento de um software para resolução do problema de empacotamento tridimensional heterogêneo. Por tratar-se de um problema difícil dentro da teoria da complexidade computacional, foi utilizado algoritmo genético, um ramo dos algoritmos evolucionários para resolução do problema. Seu principal objetivo é alocar um conjunto de caixas de variados tamanhos em um número mínimo de contêineres, considerando que cada contêiner deve estar o mais ocupado possível. A visualização do resultado do algoritmo foi desenvolvida em um ambiente 3D para Java, utilizando a biblioteca JOpenGL, que permite ao usuário andar pelo cenário e observar o resultado do algoritmo gerado. Com esta visualização, o usuário pode também observar alguns dados relativos a execução do algoritmo, como por exemplo o número de gerações necessárias para chegar no objetivo e o percentual de ocupação do contêiner. A partir dos resultados analisados, foi possível observar que algoritmo genético encontrou boas soluções para o problema e que a parametrização do software é sensível ao cenário proposto.

Palavras-chave: Logística. Empacotamento tridimensional. Algoritmos aproximados. Algoritmo genético.

ABSTRACT

This work presentes the development of a software for solving the Heterogeneous Three-dimensional Bin Packing Problem. As it is a difficult problem in the theory of computational complexity, a genetic algorithm was used to solve the problem. Its main purpose is to allocate a set of heterogeneous bins in a minimum number of containers, considering that each container should have the maximum occupation. To visualize the results of the algorithm a 3D environment for Java was developed using the JOpenGL library, allowing the user to walk through the scenario and observe the result of the generated algorithm. With this visualization, the user can also observe the data regarding algorithm execution, such as, the number of generations to achieve the objective and percentage occupancy of the container. From the results analyzed, it was possible to observe that the genetic algorithm found solutions for the problem and the parameterization of the software is sensitive to the proposed scenario.

Key-words: Logistics. Three-dimensional packing. Approximate algorithms. Genetic algorithms.

LISTA DE FIGURAS

Figura 1 – Exemplo de AG simples 20
Figura 2 – Exemplo de Indivíduo e População 21
Figura 3 – Exemplo de seleção por roleta 22
Figura 4 – Exemplo de Crossover Point 23
Figura 5 – Exemplo de Subtree Crossover 23
Figura 6 – Exemplo de solução 24
Figura 7 – Exemplo de mesclagem de espaços vazios 25
Figura 8 – Resultado dos testes 26
Figura 9 – Apresentação do EMS 27
Figura 10 – Exemplo de seleção de alocação 27
Figura 11 – Diagrama de caso de uso do software 30
Figura 12 – Diagrama de atividade da solução como um todo 31
Figura 13 – Diagrama de atividade da solução de AG 32
Figura 14 – Diagrama da solução de AG 33
Figura 15 – Diagrama da modelagem do contêiner e das caixas 34
Figura 16 – Diagrama da view 35
Figura 17 – Recorte do diagrama de atividade representando a população inicial 43
Figura 18 – Recorte do diagrama de atividade representando a seleção dos pais 44
Figura 19 – Recorte do diagrama de atividade representando a mutação e reprodução 45
Figura 20 – Recorte do diagrama de atividade representando ordenação 48
Figura 21 – Recorte do diagrama de atividade representando decisão 49
Figura 22 – Tela de configuração 52
Figura 23 – Tela de execução do algoritmo genético 53
Figura 24 – Tela 3D para visualização do resultado 53
Figura 25 – Médias de aptidão por população 55
Figura 26 – Médias de tempo por população 56
Figura 27 – Médias de aptidão por ponto de corte 57
Figura 28 – Médias de tempo por ponto de corte 57
Figura 29 – Médias de tempo por população 58
Figura 30 – Médias de aptidão por população 59
Figura 31 – Médias de aptidão por ponto de corte 60

Figura 32 – Médias de tempo por ponto de corte 60
Figura 33 – Médias de aptidão por população 61
Figura 34 – Médias de tempo por população 62
Figura 35 – Médias de tempo por ponto de corte 63
Figura 36 – Médias de aptidão por ponto de corte 63
Figura 37 – Médias de tempo por população 64
Figura 38 – Médias de contêineres utilizados por ponto de corte 65
Figura 39 – Médias de tempo por ponto de corte 65
Figura 40 – Médias de geração objetivo por população 66
Figura 41 – Médias de geração objetivo por ponto de corte 67
Figura 42 – Médias de tempo total por ponto de corte 68
Figura 43 – Primeiro contêiner gerado 69
Figura 44 – Segundo contêiner gerado 70
Figura 45 – Segundo contêiner gerado - rotacionado 71

LISTA DE QUADROS

Quadro 1 – Método para adição de caixa 36
Quadro 2 – Método para verificação de alocação de caixa 37
Quadro 3 – Pseudocódigo para separar trechos por largura no contêiner 38
Quadro 4 – Pseudocódigo para análise de trechos disponíveis da caixa 39
Quadro 5 – Pseudocódigo do método para análise de espaços disponíveis 40
Quadro 6 – Marcação dos pontos no contêiner 41
Quadro 7 – Pseudocódigo do looping principal do sistema 42
Quadro 8 – Pseudocódigo para população randômica 43
Quadro 9 – Pseudocódigo do método para nova geração 44
Quadro 10 – Classe de seleção 45
Quadro 11 – Pseudocódigo do método de mutação e reprodução 47
Quadro 12 – Método de ordenação da população 48
Quadro 13 – Pseudocódigo do método de decisão 49
Quadro 14 – Adição de um objeto no mundo gráfico 50
Quadro 15 – Métodos implementados para desenhar os objetos 50
Quadro 16 – Método responsável pela inicialização da renderização 51
Quadro 17 – Trabalhos correlatos 71

LISTA DE TABELAS

Tabela 1 – Médias de aptidão por população 54
Tabela 2 – Médias de aptidão por ponto de corte 56
Tabela 3 – Médias de aptidão por população 58
Tabela 4 – Médias de aptidão por ponto de corte 59
Tabela 5 – Médias de aptidão por população 61
Tabela 6 – Médias de aptidão por ponto de corte 62
Tabela 7 – Médias de número de contêineres e tempo por população 64
Tabela 8 – Médias de número de contêineres e tempo por ponto de corte 65
Tabela 9 – Médias de geração objetivo e tempo por população 66
Tabela 10 – Médias de número de contêineres e tempo por ponto de corte 67

LISTA DE ABREVIATURAS E SIGLAS

AG – Algoritmo Genético

BPP – Bin Packing Problem

DBLF – Deepest Bottom Left With Fill

EMS – Empty Maximal Spaces

HGA – Hybrid Genetic Algorithm

HGAI – Hybrid Genetic Algorithm Improved

RF – Requisito Funcional

RNF – Requisito Não-Funcional

TI – Tecnologia da Informação

UC – Caso de Uso

SUMÁRIO

1 INTRODUÇÃO 15
1.1 OBJETIVOS 15
1.2 ESTRUTURA 16
2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 LOGÍSTICA 17
2.1.1 Problema de empacotamento 17
2.1.2 Uso da TI na logística 18
2.2 ALGORITMOS GENÉTICOS 19
2.2.1 Indivíduo/População 20
2.2.2 Função objetivo 21
2.2.3 Seleção 21
2.2.4 Reprodução/Mutação 22
2.3 TRABALHOS CORRELATOS 24
2.3.1 A Biased Random Key Genetic Algorithm For 2D and 3D Bin Packing Problems 24
2.3.2 A Hybrid Genetic Algorithm For 3D Bin Packing Problems 25
2.3.3 A Genetic Algorithm For The Three-dimensional Bin Packing Problem With Heterogeneous Bins 26
3 DESENVOLVIMENTO DO SOFTWARE 29
3.1 REQUISITOS 29
3.2 ESPECIFICAÇÃO 29
3.2.1 Casos de uso 29
3.2.2 Diagrama de atividade 30
3.2.3 Diagrama de classes 33
3.3 IMPLEMENTAÇÃO 35
3.3.1 Técnicas e ferramentas utilizadas 35
3.3.2 Operacionalidade da implementação 51
3.4 ANÁLISE DOS RESULTADOS 54
3.4.1 Análise dos resultados – Classe 1 54
3.4.2 Análise dos resultados – Classe 2 57
3.4.3 Análise dos resultados – Classe 3 60
3.4.4 Análise dos resultados – Classe 4 63

3.4.5 Análise dos resultados – Classe 5 66
3.4.6 Comparativo entre o trabalho desenvolvido aos correlatos 71
4 CONCLUSÕES 73
4.1 EXTENSÕES 73
REFERÊNCIAS 75

1 INTRODUÇÃO

Em sua origem, a logística estava essencialmente ligada às operações militares, no deslocamento de munição, víveres, equipamentos e socorro médico para o campo de batalha (NOVAES, 2015). Novaes (2015) explica que ao longo do tempo, a logística foi empregada nas indústrias, quando as empresas passaram a perceber o valor do tempo para o seu negócio, como transportar seus produtos de fábrica para os depósitos, ou para lojas, ou ainda transportar matéria prima.

A logística possui várias vertentes, sendo o problema de empacotamento uma delas. Por definição, este problema consiste em empacotar, sem sobreposições, um conjunto de caixas tridimensionais em formato retangular (itens), no número mínimo de contêineres tridimensionais em formato retangular (GONÇALVES; RESENDE, 2013). O problema de empacotamento tem sido amplamente estudado, uma vez que têm muitos interesses práticos relevantes em aplicações industriais. Pode-se citar, por exemplo, o corte de espuma de borracha, produção de poltronas, contêineres, paletes de carga e design de embalagens (WANG; CHEN, 2010).

Segundo Wang e Chen (2010), este problema pode ser resolvido computacionalmente através dos Algoritmos Genéticos (AG), ramo dos algoritmos evolucionários também definidos como técnicas de busca baseada no processo biológico de evolução natural. O AG é indicado para a solução de problemas de otimização complexos, que envolvem um grande número de variáveis e, consequentemente, espaços de soluções de dimensões elevadas (MIRANDA, 2011).

Diante deste contexto, percebe-se o potencial no estudo de AG aplicados no problema de empacotamento tridimensional. Sendo assim, propõe-se o desenvolvimento de um software que utilize AG aplicado no problema de empacotamento, permitindo edição de variáveis de contexto do problema.

1.1 OBJETIVOS

O objetivo deste trabalho é resolver o problema de empacotamento tridimensional utilizando AG.

Os objetivos específicos são:

a) disponibilizar seleção de pacotes com tamanhos variáveis;
b) disponibilizar seleção de tamanho de contêiner;
c) realizar a alocação de pacotes dentro de contêineres considerando o maior preenchimento.

1.2 ESTRUTURA

Este trabalho está dividido em quatro capítulos. O primeiro capítulo apresenta a introdução do trabalho e os objetivos. O segundo capítulo apresenta a fundamentação teórica sobre logística e algoritmos genéticos. No terceiro capítulo é demonstrado o desenvolvimento do sistema com requisitos, especificação, implementação, resultados e operacionalidade da aplicação. Por fim, no quarto capítulo são relatadas as conclusões e também as possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados aspectos teóricos relacionados ao trabalho. A seção 2.1 expõe informações sobre o ramo da logística. A seção 2.2 descreve sobre algoritmos genéticos. Por fim, na seção 2.3 são apresentados os trabalhos correlatos ao trabalho que foi desenvolvido.

2.1 LOGÍSTICA

Logística nasceu nas operações militares, onde os generais precisavam ter uma equipe que providenciasse soluções importantes, tais como: deslocamento, munição, equipamentos para o campo de batalha. Estas atividades eram feitas pelos grupos logísticos, que trabalhavam sem serem notados, em silêncio, na retaguarda (NOVAES, 2015).

Ao longo dos anos, as indústrias passaram a ter a logística mais presente em suas atividades, de uma forma proativa. Segundo Moura (2006), a logística é uma área muito complexa. Um dos fatores que torna isso realidade é a necessidade de terem que tomar decisões e encontrar soluções eficientes de equilíbrio entre um elevado número de fatores divergentes. O gestor, por exemplo, apesar de dispor de diversos recursos altamente avançados, tiveram suas atividades com complexidades aumentadas, afinal a hipercompetitividade e os múltiplos fatores contribuem para isto (MOURA, 2006).

Dentre estes múltiplos fatores, as indústrias passaram a observar um fator importante na cadeia produtiva, o valor do tempo. Este conceito antes não era considerado como fator competitivo na produção de um produto e, atualmente, é um dos elementos mais críticos do processo logístico (NOVAES, 2015).

Tendo cada etapa do processo considerada como importantíssima em um ramo extremamente concorrente, o problema de empacotamento torna-se um fator decisivo no valor do tempo e desperdícios de viagem.

2.1.1 Problema de empacotamento

O problema de empacotamento (BPP - Bin Packing Problem) é contido dentro de um grupo de problemas NP-Difíceis e define-se por agrupar itens de variados tamanhos dentro de contêineres de tamanhos fixos (FALKENAUER, 1996). Existem diversos tipos de problemas de empacotamento, desde o unidimensional até o tridimensional. O problema de empacotamento unidimensional é o mais antigo de todos e consiste em empacotar n pedaços, onde cada pedaço é um número racional entre 0 ou 1, em um número mínimo de caixas considerando que a soma dos tamanhos dos pedaços deve ser menor ou igual a 1

(KARMAKAR; KARP, 1982). O problema de empacotamento bidimensional é bastante conhecido como Cutting Stock Problem (problema de corte), e é definido por agrupar um número máximo de peças retangulares em um número mínimo de planos (MARTELLO; VIGO, 1998). Já o problema de empacotamento tridimensional, é definido por empacotar um número máximo de caixas dentro de um número mínimo de contêineres (PARK et al, 1996).

De acordo com Wäscher, Haubner e Schuman (2007), o problema de empacotamento é classificado por: dimensão, tipo de tarefa, variedade dos objetos de entrada e variedade dos objetos de saída, onde:

a) dimensão: existem 4 classificações diferentes, sendo elas: unidimensional, bidimensional, tridimensional e n-dimensional onde n > 3;
b) tipo de tarefa: indica qual é a tarefa principal do algoritmo, podendo ser elas: entrada grande e uma saída menor ou uma entrada selecionada e uma saída ampla;
c) variedade dos objetos de entrada: indica qual a variedade dos objetos de entrada, podendo ser apenas um objeto, objetos idênticos ou objetos variados;
d) variedade de objetos de saída: indica qual a variedade de objetos de saída, podendo ser objetos diferentes de pouca variedade, muitos objetos de múltiplas variedades, muitos objetos de relativamente poucas variedades não congruentes e objetos totalmente congruentes.

Segundo Park et al. (1996) e Hartmann (2000), o problema de empacotamento pode ainda gerar subproblemas com complexidades ainda maiores, tais como:

a) agendamento de entrega, onde além de considerar o número máximo de empacotamento, deve-se considerar a ordem de saída das caixas;
b) caixas frágeis, que devem ficar sempre acima das demais caixas;
c) transporte através de prateleiras para determinados tipos de mercadoria;
d) entre outros.

Faz-se necessário o uso da tecnologia da informação (TI) para resolver problemas complexos deste tipo.

2.1.2 Uso da TI na logística

Segundo Moura (2006), com a globalização e o aumento no número de fornecedores disputando o mesmo nicho, o mercado inflou, forçando mudanças em múltiplos domínios, por exemplo, nos sistemas de transportes, circuitos de distribuição, embalagens, entre outros.

É importante lembrar que existem inúmeras operações que utilizam mais do que uma maneira de transporte. Por exemplo combinando transporte ferroviário com marítimo, e que, por conta disto, tornam-se potenciais utilizadores de plataformas logísticas (MOURA, 2006).

Como visto anteriormente, nas últimas décadas intensificou-se a utilização da TI em muitas áreas da logística, o que facilitou seu desempenho, principalmente nas ligações no interior das organizações e com os parceiros externos (MOURA, 2006). Segundo Moura (2006), foi a TI que facilitou a desintermediação, ou seja, reduziu ou até em alguns casos eliminou passos dentro de uma empresa ou organizações externas terceirizadas para fazerem determinado serviço.

2.2 ALGORITMOS GENÉTICOS

Segundo Linden (2008), Algoritmos Genéticos (AG) são um ramo de algoritmos evolucionários e podem ser definidos como técnicas de busca baseadas na metáfora do processo biológico de evolução natural. Em muitos problemas o caminho até o objetivo torna-se irrelevante, sendo o estado final do algoritmo o mais importante (RUSSEL; NORVIG, 2013). Para estes problemas é introduzido o conceito de busca local, que operam baseados em um único estado atual, onde em geral se movem apenas para os vizinhos desse estado (RUSSEL; NORVIG, 2013).

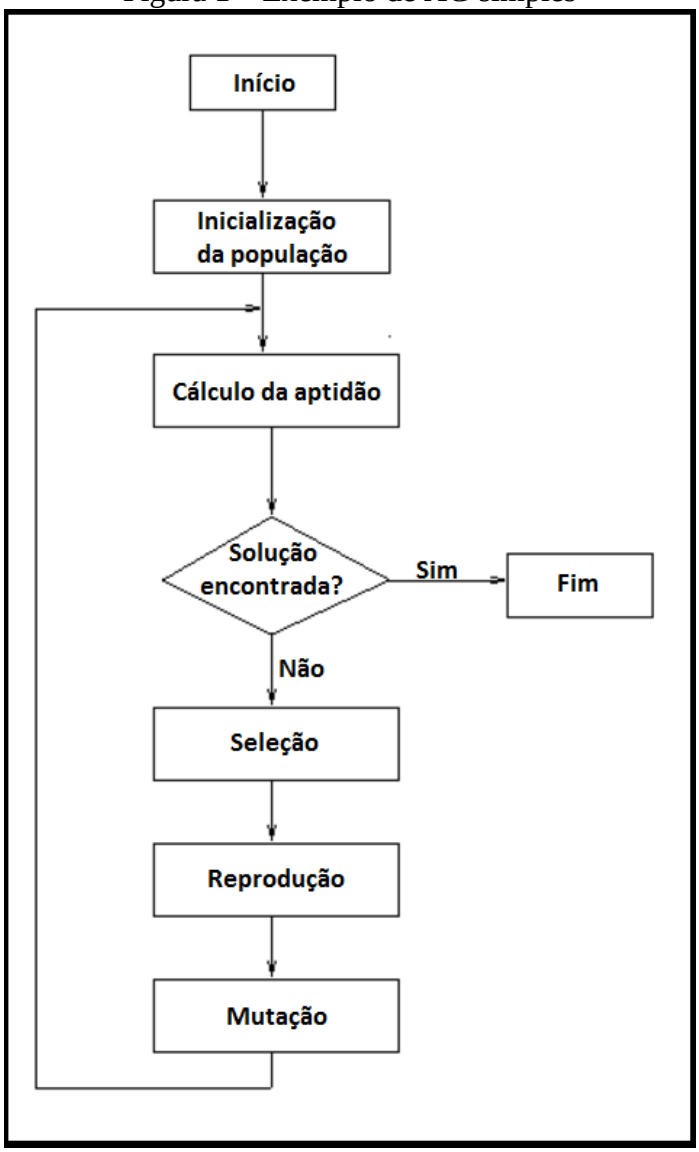
Além de encontrar objetivos, os algoritmos de busca local são indicados para problemas de otimização, onde o objetivo é encontrar o melhor estado de acordo com a função objetivo (RUSSEL; NORVIG, 2013). O AG utiliza técnicas heurísticas de forma a encontrar o máximo de uma função, ou seja, a solução o mais próximo possível da solução ótima, mesmo em casos em que a solução ótima não seja possível de ser alcançada, como por exemplo problemas de agendamento de horário, otimizações de metodologias, problemas de paletização e o próprio problema de empacotamento.

As técnicas heurísticas são regras gerais de influência utilizadas para julgamentos em decisões de incertezas (TONETTO et al., 2006). Linden (2008) afirma também que os AG utilizam tais técnicas para tomada de decisão que não visam necessariamente encontrar a solução ótima de um problema e, quando conseguem, nem sempre podem repetir o feito.

A figura 1 representa um fluxo de uma solução AG simples. Na imagem é possível verificar que o algoritmo inicia adicionando os indivíduos na população e, daí em diante, entra num *loop* para verificar se a solução foi encontrada. O *loop* funciona nos seguintes passos: verifica se a solução já foi encontrada (através do cálculo de aptidão); enquanto a solução não for encontrada o algoritmo executa a seleção, reprodução e mutação para então

executar o cálculo de aptidão e assim sucessivamente até encontrar a solução (MIRANDA, 2001).

Figura 1 – Exemplo de AG simples



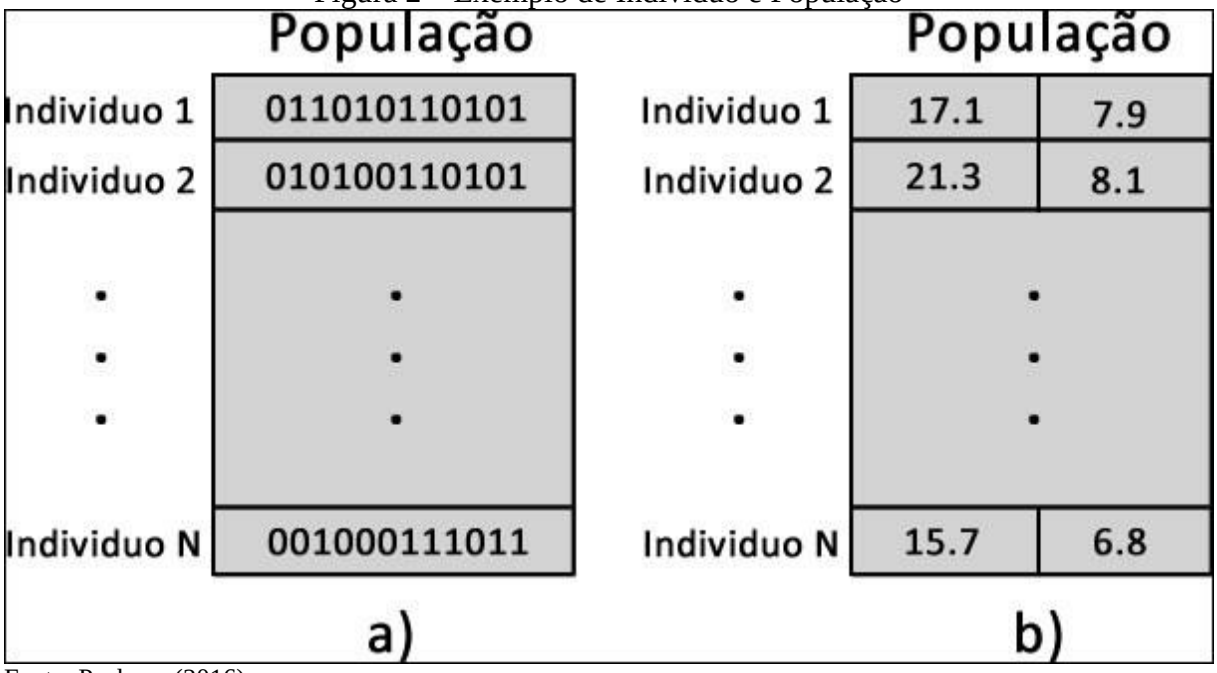
Fonte: Adaptado de Miranda (2011).

As etapas descritas na figura 1 serão apresentadas nas subseções seguintes. Segundo Poli et. al (2008), estas etapas são separadas em indivíduo, população, função objetivo, seleção, reprodução e mutação.

2.2.1 Indivíduo e População

A população é definida por um agrupamento de indivíduos. A figura 2 ilustra duas populações de diferentes indivíduos. É possível perceber que cada indivíduo possui seus próprios genes e uma população pode ser composta por `n` indivíduos. Segundo Russel e Norvig (2013) o indivíduo muito frequentemente é representado por uma cadeia sobre um alfabeto finito ou uma cadeia de valores 0 e 1.

Figura 2 – Exemplo de Indivíduo e População



Fonte: Pacheco (2016).

2.2.2 Função objetivo

Identifica a natureza do problema, como por exemplo, se deseja maximizar ou minimizar o resultado de um determinado algoritmo. A função objetivo tem por objetivo encontrar o melhor estado dentro de uma população (RUSSEL, NORVIG, 2013).

Russel e Norvig (2013) identificam um exemplo de função objetivo na própria natureza, denominada adaptação reprodutiva, onde a natureza otimiza-se do ponto de vista da evolução de Darwin. Esta função objetivo pode ser chamada também de decisão, função avaliadora, função de adequação entre outras. Um exemplo de função objetivo seria a minimização de materiais utilizados para determinado processo.

2.2.3 Seleção

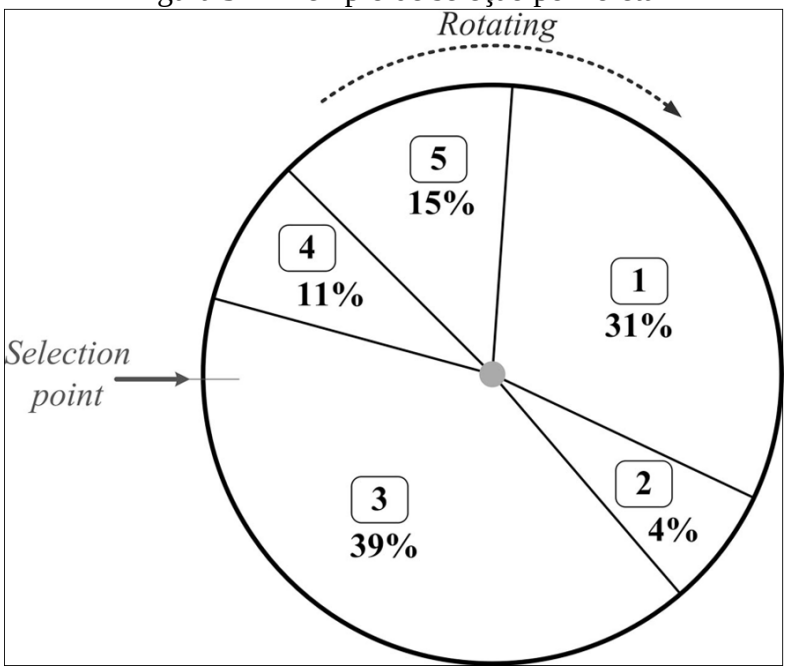
Seleciona amostragem de indivíduos para comporem um conjunto de pais para a próxima geração. Segundo Silva e Soma (2003), existem duas formas comuns de executar o método de seleção. A primeira delas é através do método de roleta viciada ou tendenciosa, que possibilita que os indivíduos melhor avaliados tenham mais probabilidade de escolha, através da seleção baseada em sua função de objetivo. Outro método comum é a seleção por critérios elitistas, onde os melhores indivíduos são sempre escolhidos, sem a necessidade da roleta para randomizar.

Poli et al (2008) defende que existem inúmeros algoritmos de seleção diferentes, sendo o mais comum o Tournament Selection, que escolhe um número de indivíduos

randomicamente da população, para então serem comparados entre si, onde o melhor deles é escolhido para ser o pai da próxima geração, desta maneira, não sendo necessário roleta ou classificação por resultado da função de objetivo.

A figura 3 representa graficamente a seleção por roleta. Os indivíduos são divididos 5 grupos onde os grupos que têm maior aptidão possuem mais chances de serem selecionados. Feito esta divisão, é girada a roleta aleatoriamente e onde o `selection point` indica será selecionado.

Figura 3 – Exemplo de seleção por roleta

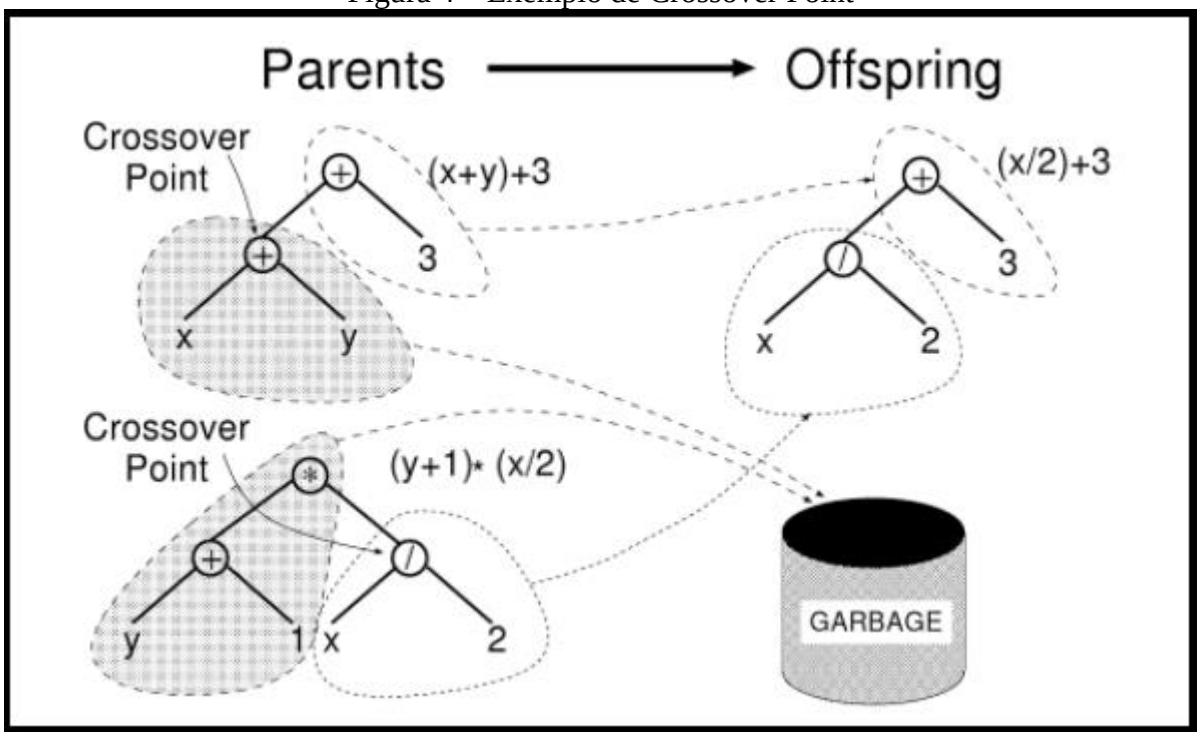


Fonte: Chen et. al (2011).

2.2.4 Reprodução e Mutação

A reprodução é responsável pela cópia de características da geração atual para servir de base para a próxima geração que será alterada através da mutação. A reprodução é conhecida também por recombinação de genes. Segundo Poli et al. (2008), torna-se interessante o uso do método comum, como o Subtree Crossover. A figura 4 detalha o algoritmo que espera uma entrada de duas árvores pais, então seleciona um ponto de cruzamento (Crossover Point), depois cria a prole (árvore filha) substituindo o nodo no ponto de cruzamento da árvore filha, fazendo com que a árvore filha fique recombinada com os genes das árvores pai.

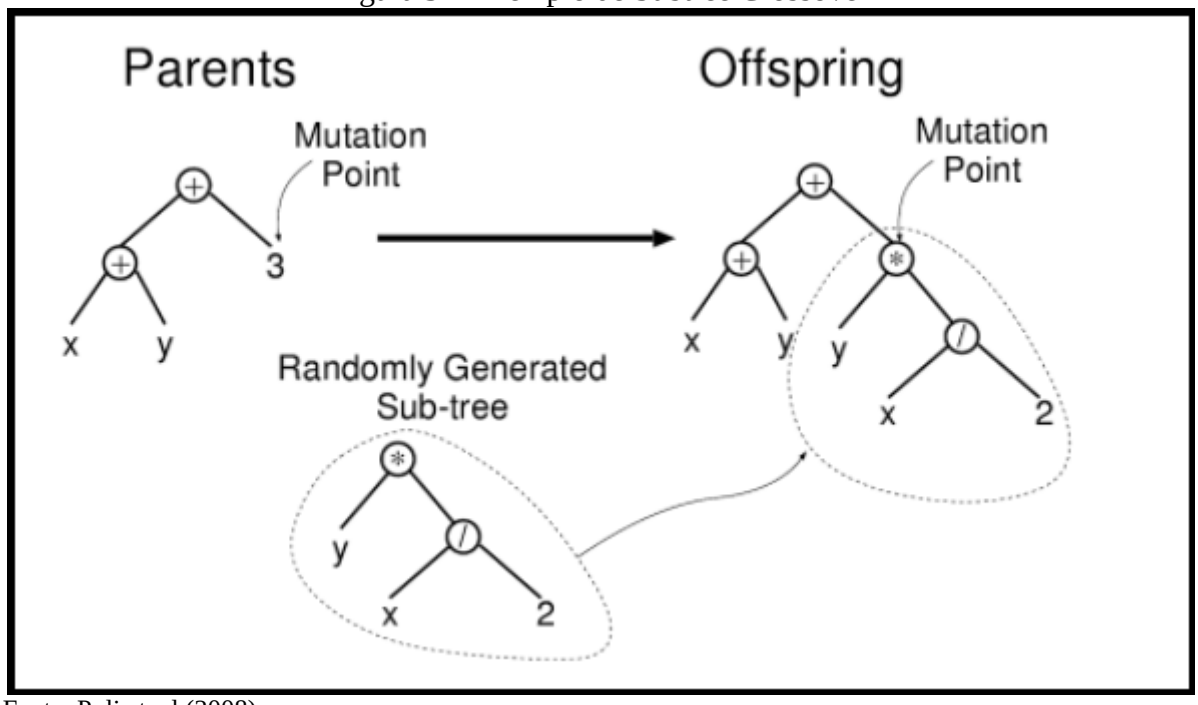
Figura 4 – Exemplo de Crossover Point



Fonte: Poli et. al (2008).

A mutação tem por objetivo alterar alguma característica importante para evolução da geração atual. Parecido com a recombinação de genes, para realizar a mutação, Poli et al. (2008) comenta que o método mais comum é o Subtree Mutation (figura 5). O algoritmo escolhe randomicamente um ponto de mutação (Mutation Point) e substitui na árvore filha, a partir do ponto de mutação, uma árvore filha randômica.

Figura 5 – Exemplo de Subtree Crossover



Fonte: Poli et. al (2008).

2.3 TRABALHOS CORRELATOS

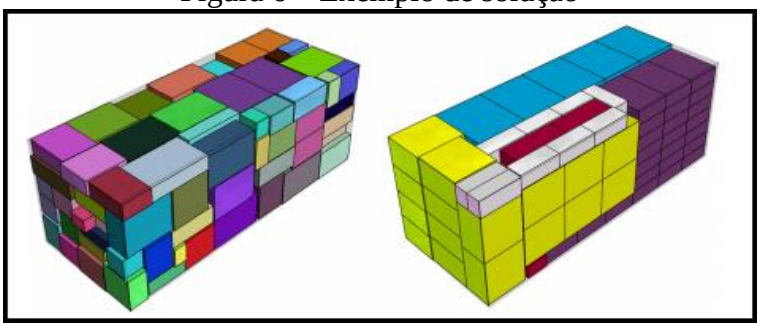
Foram selecionados três trabalhos correlatos, ambos falam da aplicação de AG no problema de empacotamento 3D. O item 2.3.1 descreve “A biased random key genetic algorithm for 2D and 3D bin packing problems” (GONÇALVES; RESENDE, 2013), o item 2.3.2 refere-se ao trabalho “A hybrid genetic algorithm for 3D bin packing problems” (WANG; CHENG, 2010) e o item 2.3.3 relata o trabalho de Li, Zhao e Zhang (2014) denominado “A Genetic Algorithm For The Three-dimensional Bin Packing Problem With Heterogeneous Bins”.

2.3.1 A Biased Random Key Genetic Algorithm For 2D and 3D Bin Packing Problems

Gonçalves e Resende (2013) detalham o desenvolvimento de um algoritmo de chave genética randômica para o problema de empacotamento 2D e 3D. Este algoritmo foi construído visando resolver problemas industriais, tais como: movimentação de carga em veículos de contêiner ou paletes; design de embalagens e na otimização no corte de peças.

Através de AG, foi efetuada uma abordagem o máximo de espaço livre em um contêiner, para mover os pacotes até que se encaixem na maior precisão possível. Desta forma, a ordem de inserção de pacotes será obtida baseada em determinados parâmetros de carga, como por exemplo distância mínima para o topo do contêiner ou distância a ser respeitada do fundo do contêiner. Para esta abordagem, foi utilizado um algoritmo de heurística construtiva, que mantém todos os contêineres abertos enquanto existirem pacotes que ainda não foram encaixados. Além disso, um novo contêiner será aberto apenas quando um determinado pacote não couber nos demais contêineres já abertos. Na figura 6 ilustra-se um exemplo do problema resolvido.

Figura 6 – Exemplo de solução



Fonte: Gonçalves e Resende (2013).

Os testes do algoritmo foram realizados em uma base padrão de referência de Martello, possuindo 320 problemas gerados. As instâncias foram organizadas em 8 classes com 40 instâncias cada, 10 instâncias para cada valor de $n \in \{50, 100, 150, 200\}$. Para classes 1 – 5, o

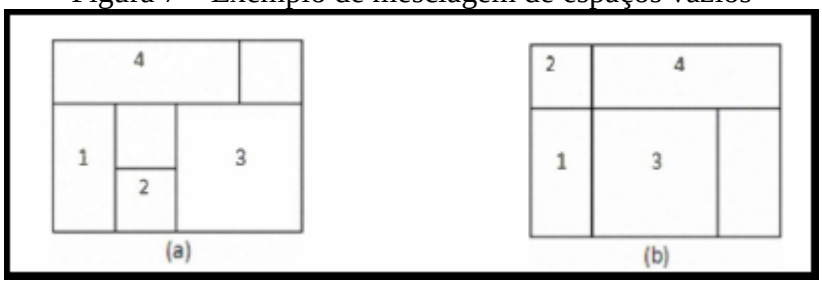
tamanho do contêiner era 100 para largura, altura e comprimento. Para os pacotes, o tamanho era randômico dentro de uma distribuição uniforme de largura, altura e comprimento. Para classes k (k = 1, ..., 5), cada item foi escolhido com probabilidade de 60% e para os outros quatro tipos a probabilidade era 10% cada. Após o término dos testes e das comparações com algoritmos anteriores a este, os resultados provaram que este algoritmo consistentemente iguala e por vezes supera os demais, conseguindo uma solução melhor em apenas 50 gerações, menos do que os demais.

2.3.2 A Hybrid Genetic Algorithm For 3D Bin Packing Problems

Wang e Chen (2010) implementam um algoritmo híbrido genético focado no problema de empacotamento, em que os pacotes fossem reposicionados dentro do contêiner mesclando seus espaços vazios. Foi utilizado uma extensão do algoritmo Deepest Bottom Left With Fill (DBLF), para otimizar o espaço dentro do contêiner, tendo em vista que qualquer mesclagem de espaços vizinhos dentro de um recipiente é benéfico para o encaixe do próximo pacote (WANG; CHEN, 2010).

Na figura 7 ilustra-se uma aplicação simples e estendida do algoritmo DBLF, onde o contêiner (a) possui 4 pacotes alocados de forma randômica, restando 2 espaços vazios distintos e, no contêiner (b), é representado o resultado do algoritmo DBLF estendido, onde os mesmos pacotes do contêiner (a) são reposicionados, otimizando-o, de modo que os espaços vazios distintos fiquem mesclados.

Figura 7 – Exemplo de mesclagem de espaços vazios



Fonte: Wang e Chen (2010).

Os comportamentos dos algoritmos foram testados em uma série de problemas de testes, baseados no método de geração de Bischoff/Ratcliff, onde os tamanhos dos pacotes foram definidos com largura=233, comprimento=235 e altura=1200, e o número total de pacotes foi considerado 60. Foram definidos então três cenários de teste, o primeiro onde os itens não rotacionam, o segundo teste em que os itens podem ser rotacionados horizontalmente (2 rotações permitidas) e o terceiro teste em que os itens podem ser rotacionados como quiserem (até 6 rotações permitidas). A figura 8 exibe o resultado dos

testes, onde a coluna HGA (Hybrid Genetic Algorithm) significa os percentuais de acerto do trabalho anterior a este (considerando margem de erro) e a coluna HGAI (Hybrid Genetic Algorithm Improved) representa os resultados deste trabalho. É possível observar que nos três ambientes de testes (rotações) este trabalho tornou-se mais eficiente e com menor margem de erro, dando destaque para o teste com 2 rotações, onde o percentual de acerto foi de 81,30% para 91,16% e a margem de erro caiu de 3,33% para 0,71%.

Figura 8 – Resultado dos testes

Test Instances	Algorithms		
	HGA	HGAI	*t*-test result
non-rotation	80.43%±1.95%	86.05%±0.85%	*s*+
2-ways rotation	81.30%±3.33%	91.16%±0.71%	*s*+
6-ways rotation	81.06%±3.84%	92.88%±1.38%	*s*+

Fonte: Wang e Chen (2010).

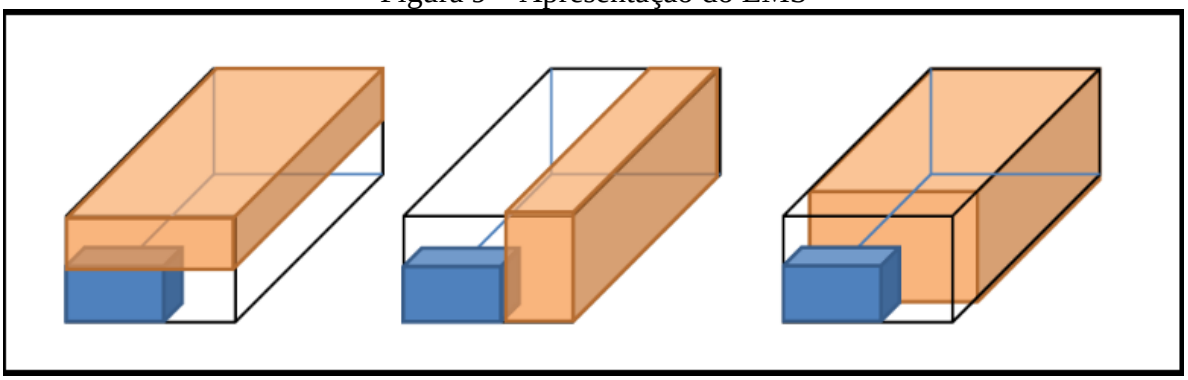
2.3.3 A Genetic Algorithm For The Three-dimensional Bin Packing Problem With Heterogeneous Bins

Li, Zhao e Zhang (2014) implementam um algoritmo genético para o problema de empacotamento considerando caixas heterogêneas, ou seja, caixas de tamanhos diferentes. O algoritmo genético apresentado possuiu uma nova heurística de embalagem, denominada Best Match Heuristic Packing Strategy.

Foram analisados os pontos fortes e fracos do algoritmo DBLF e suas variações, para então ser proposto a solução final do algoritmo. Os principais pontos fracos encontrados no algoritmo DBLF foram que estas heurísticas sempre observam pelo espaço mínimo *vs* maior profundidade para trocar o item atual. Sendo assim, a coordenada *x* sempre fica predominante sobre as demais coordenadas; o item ou o espaço é primeiro determinando e somente depois é selecionado baseado em certas prioridades (LI; ZHAO; ZHANG, 2014).

Para a nova abordagem, foi utilizado o conceito do máximo espaço disponível (EMS – Empty Maximal Spaces) para representar os espaços vazios nos contêineres. Na figura 9 é possível verificar o tratamento implementado, onde ao colocar uma caixa (em azul) no canto esquerdo de um contêiner, uma lista é gerada ordenada pelo maior espaço vazio contida no contêiner.

Figura 9 – Apresentação do EMS

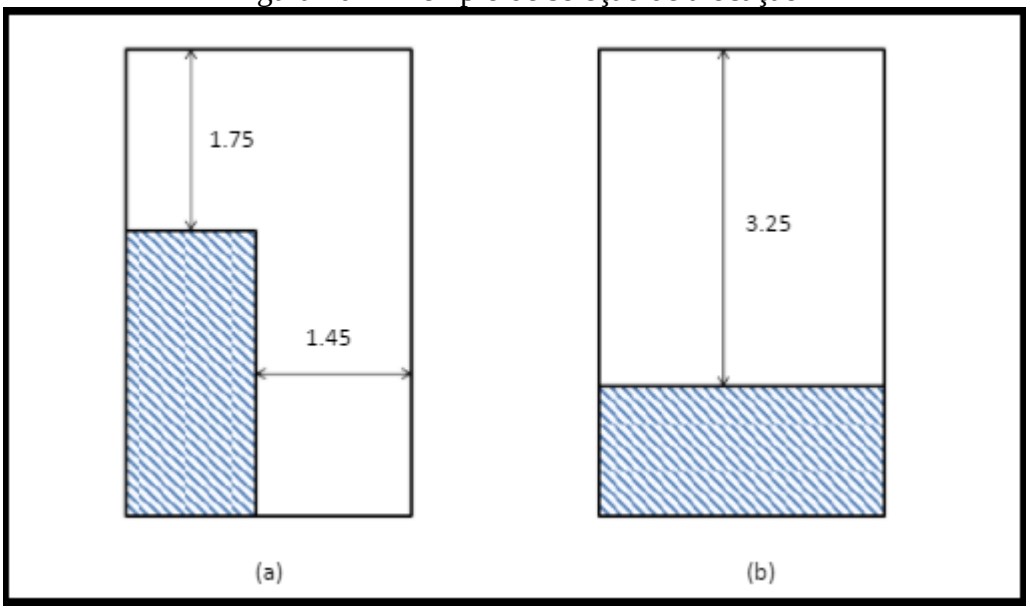


Fonte: Xueping Li et. al (2014).

Depois do processo de EMS é escolhido a prioridade dentre os espaços previamente selecionados. Primeiro são comparadas as menores coordenadas e a coordenada com as menores `vertex` ganharão a prioridade no algoritmo.

Selecionado o espaço, deve-se fazer a seleção de onde alocar a caixa. A posição que ganha prioridade de seleção é a que gera a menor margem em uma direção. A figura 10 mostra um exemplo, onde a figura a possui uma margem de 1.45 - 1.75 e a figura b possui uma margem de 3.45 apenas em uma direção. Neste caso, o algoritmo escolhe a alocação b.

Figura 10 – Exemplo de seleção de alocação



Fonte: Xueping Li et. al (2014).

O algoritmo foi testado baseado primeiramente em 12 instâncias industriais e depois testado em instâncias geradas aleatoriamente. Os resultados demonstram que podem ser encontradas ótimas soluções em tempos razoáveis, onde o método proposto tem uma melhor performance do que o DBLF, porém possui algumas falhas em relação a achar uma solução ótima para instâncias com tamanhos moderados. Foi observado também que utilizando o

método EMS para gestão de espaços vazios é mais eficiente e efetivo do que usar o método *corner points* (LI; ZHAO; ZHANG, 2014).

3 DESENVOLVIMENTO DO SOFTWARE

As seções a seguir descrevem os requisitos, a especificação, a implementação e a operacionalidade, abordando as ferramentas utilizadas nas etapas de desenvolvimento do software. Por fim, são mostrados os resultados obtidos com este trabalho.

3.1 REQUISITOS

O software descrito neste trabalho deverá:

a) permitir seleção de tamanho do contêiner (Requisito Funcional – RF);
b) permitir seleção de tamanho dos pacotes (RF);
c) otimizar a alocação de pacotes dentro do contêiner, alocando a maior quantidade de pacotes possíveis (RF);
d) disponibilizar uma tela para exibir a aptidão atual atingida e o número de gerações do algoritmo (RF);
e) disponibilizar uma tela em três dimensões para visualização do resultado do algoritmo (RF);
f) permitir a rotação de câmera na exibição 3D (RF);
g) ser implementado utilizando Algoritmos Genéticos (Requisito Não-Funcional – RNF);
h) ser implementado utilizando a plataforma Java 8 (RNF);
i) ser implementado utilizando OpenGL (RNF).

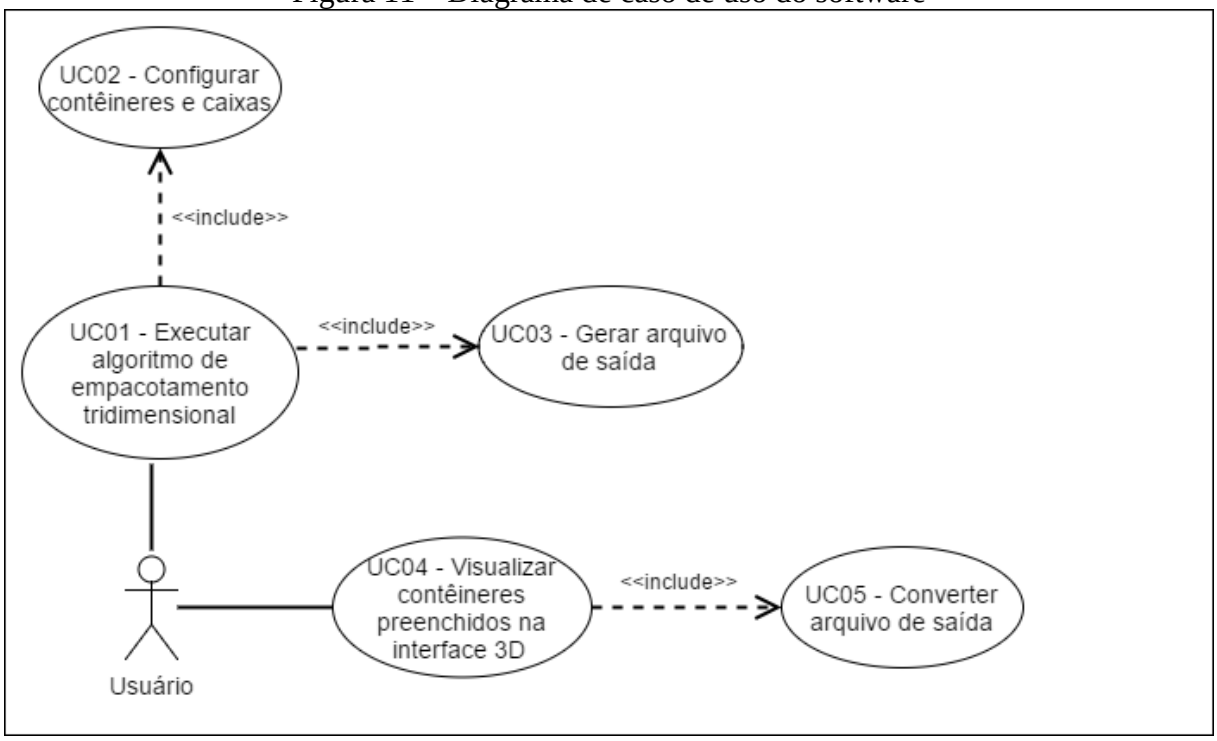
3.2 ESPECIFICAÇÃO

Para especificação do trabalho foi utilizada a ferramenta Draw.io para a criação de diagramas de caso de uso, classes e pacotes.

3.2.1 Casos de uso

O software implementado atenderá cinco casos de uso (UC): `UC01 – Executar algoritmo de empacotamento tridimensional`; `UC02 – Configurar contêineres e caixas`; `UC03 - Gerar arquivo de saída`; `UC04 – Visualizar contêineres preenchidos na interface 3D` e `UC05 - Converter arquivo de saída`. A figura 11 demonstra os casos de uso do software.

Figura 11 – Diagrama de caso de uso do software



Fonte: elaborado pelo autor.

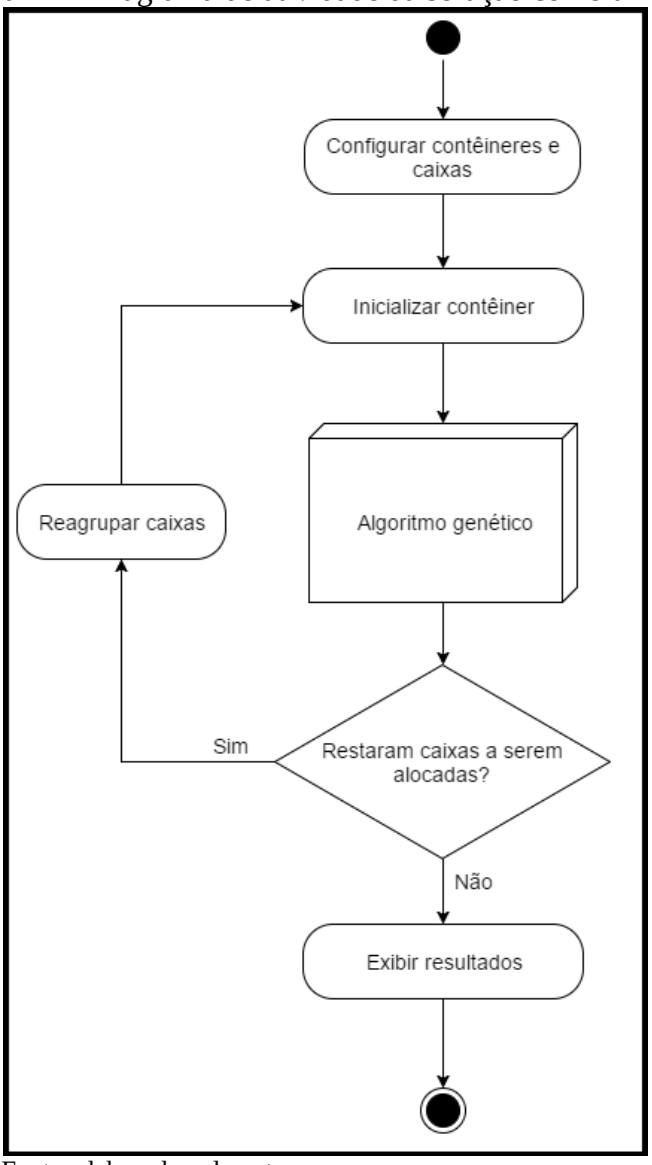
O primeiro caso de uso é a execução do software em si, ou seja, a execução do algoritmo genético que por si só inclui outros dois casos de uso: UC02 e UC03. O UC02 é responsável pelas configurações que o usuário pode fazer antes de executar o algoritmo, que é definir a sequência de caixas que precisam ser alocadas, colocar os parâmetros do algoritmo genético e configurar as dimensões do contêiner. O UC03 trata da geração do arquivo de saída do algoritmo. O arquivo de saída gerado segue no padrão de representação de preenchimento de contêineres.

O UC04 compreende a exibição dos contêineres preenchidos na interface 3D. O UC05 trata de ler o arquivo gerado pelo UC03 e converter para os objetos gráficos utilizados no UC04. O UC04 engloba tanto a abertura da saída automática do sistema quanto a reabertura de uma execução anterior, tendo em vista que os arquivos de saída ficam salvos no disco.

3.2.2 Diagrama de atividade

Nesta seção é apresentado o diagrama de atividade da solução macro de todo o sistema e o diagrama da solução de AG proposta neste trabalho.

Figura 12 – Diagrama de atividade da solução como um todo

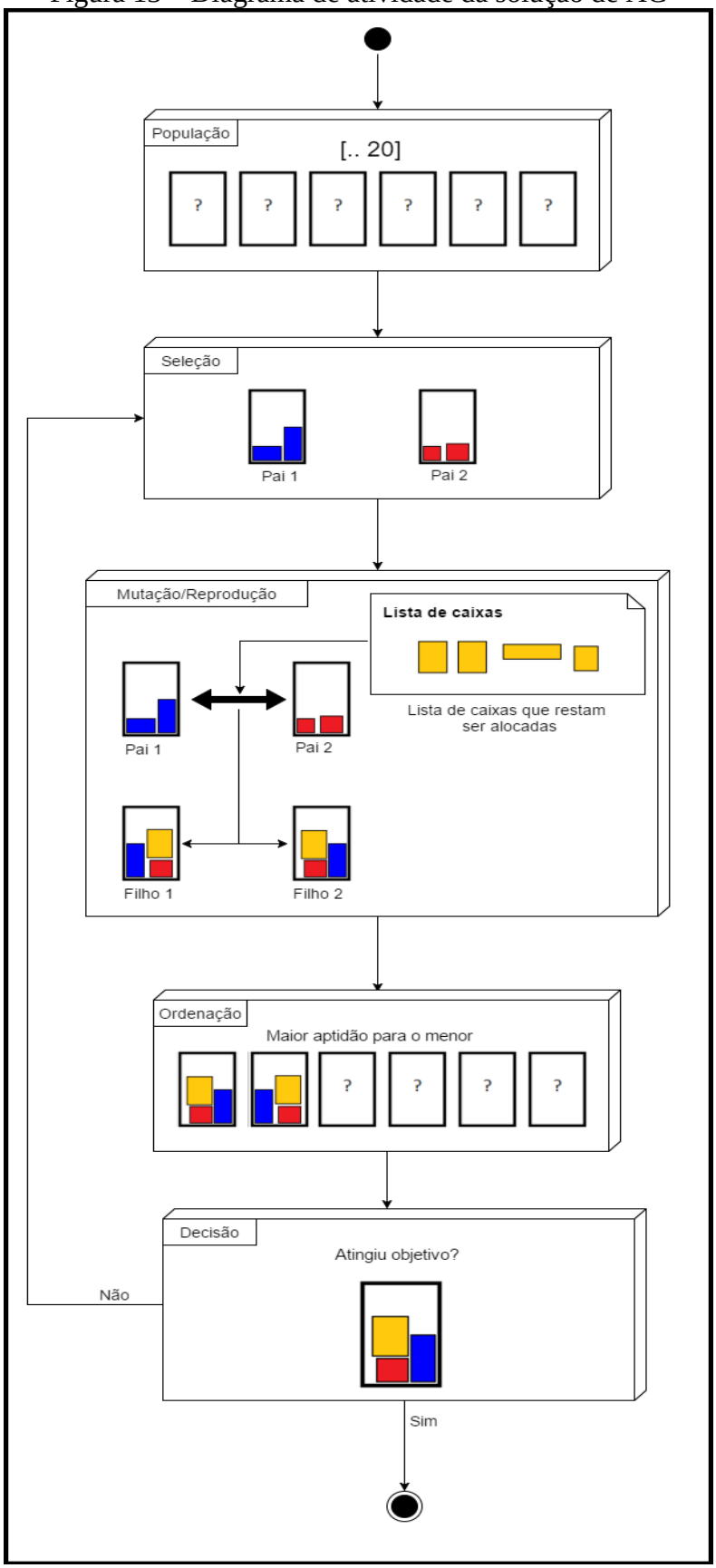


Fonte: elaborado pelo autor.

A figura 12 demonstra o fluxo do sistema como um todo, onde na primeira ação o usuário irá configurar a dimensão do contêiner e as caixas que deverão ser alocadas. Em seguida, o sistema irá inicializar o contêiner baseado nas especificações. Feito isto, é o momento de submeter o contêiner inicializado e a lista de caixas configuradas para o algoritmo genético. Sua saída será o contêiner otimizado ao máximo possível.

Após otimizar o contêiner, o sistema irá decidir se continua ou não a execução. Caso ainda restam caixas na lista, irá reagrupar estas caixas numa lista nova e começar tudo novamente. Caso não restem mais caixas a serem alocadas, irá exibir os resultados ao usuário e depois finalizar a execução. A figura 13 detalha o bloco `Algoritmo genético` encontrado na figura 12.

Figura 13 – Diagrama de atividade da solução de AG



Fonte: elaborado pelo autor.

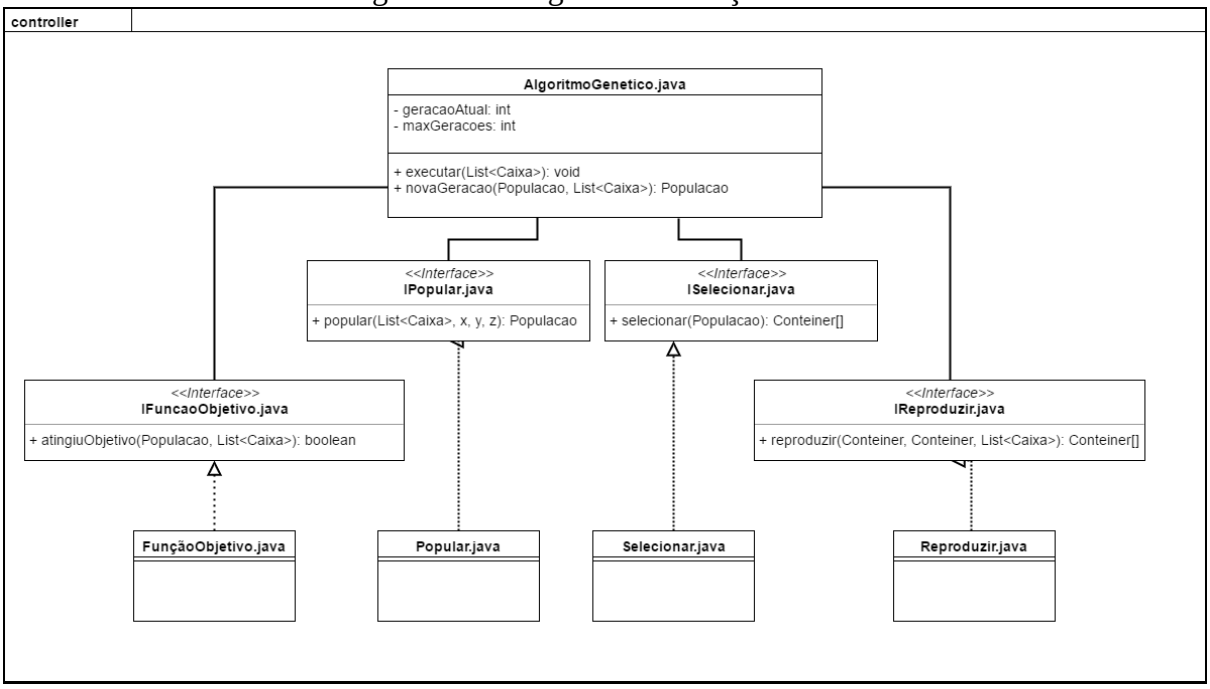
O processo do AG começa iniciando uma população randômica. Em seguida é efetuado o processo de seleção dos dois contêineres pais. Selecionado os pais, o próximo passo é a mutação e reprodução que são feitas em conjunto. Gerado a população filha, é efetuado um passo de ordenação, simplesmente para manter o melhor indivíduo na primeira posição do `array`. Isto implica diretamente no passo seguinte, que é definido por verificar se o algoritmo atingiu ou não o objetivo, seja feito de forma mais fácil e rápida.

Os processos definidos nas figuras 12 e 13 são explanadas mais profundamente na seção 3.3.

3.2.3 Diagrama de classes

Nesta seção é apresentado os diagramas de classe do sistema. A figura 14 apresenta o diagrama de classes da parte controladora do algoritmo genético no sistema.

Figura 14 – Diagrama da solução de AG



Fonte: elaborado pelo autor.

A classe `AlgoritmoGenetico` é responsável por agrupar quatro interfaces: `IFuncaoObjetivo`, `IPopular`, `IReproduzir` e `ISelecionar`. Estas interfaces têm por objetivo deixar a solução dinâmica para permitir ter diferentes implementações para um único passo do algoritmo.

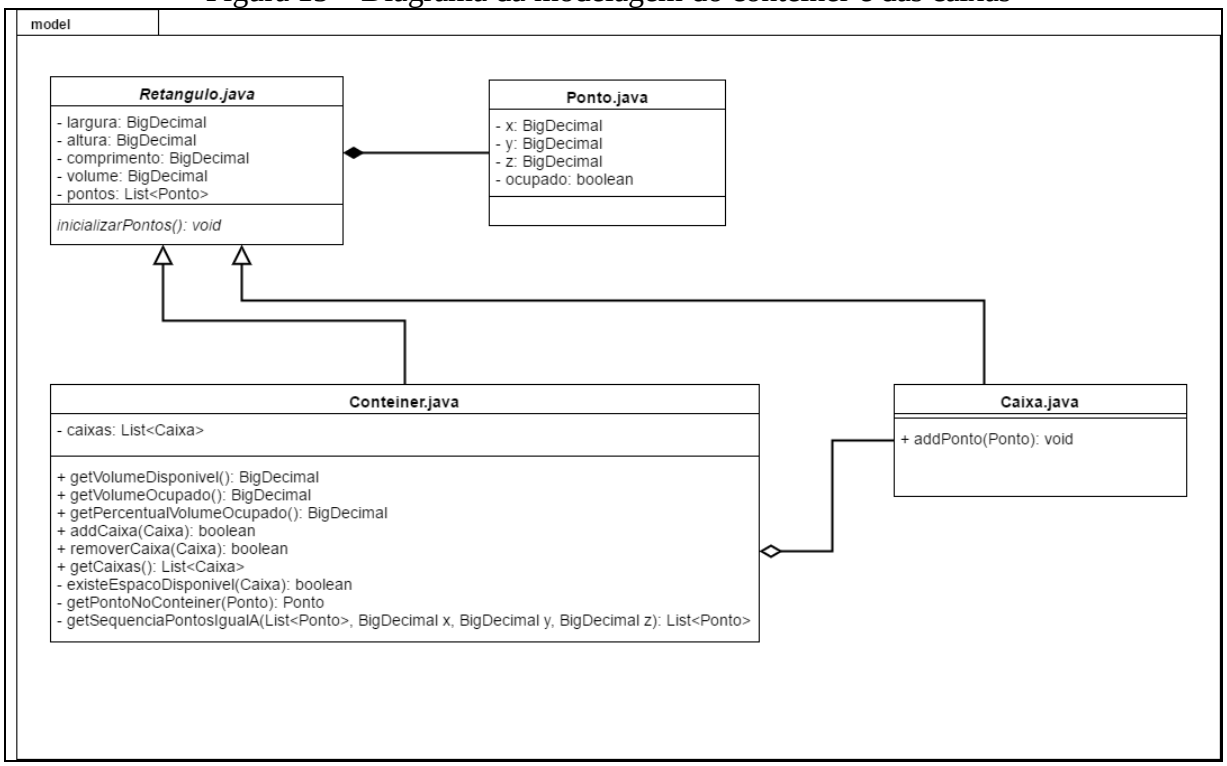
A implementação padrão das interfaces está contida nas classes: `FuncaoObjetivo`, `Popular`, `Reproduzir` e `Selecionar`.

O primeiro passo do `AlgoritmoGenetico` é incluir uma população inicial e para isto é utilizado a classe `Popular`. Após esta etapa, o algoritmo entrará num laço de repetição até

que a `FuncaoObjetivo` seja atingida. O laço consiste em efetuar a seleção dos pais na classe `Selecionar` e logo em seguida efetuar a mutação que está contida na classe `Reproduzir`.

A figura 15 representa como é feita a modelagem do contêiner e das caixas no sistema.

Figura 15 – Diagrama da modelagem do contêiner e das caixas



Fonte: elaborado pelo autor.

A classe `Retangulo` é uma classe abstrata representando os atributos e método comuns entre contêineres e caixas. É representada principalmente pelos atributos de largura, altura, comprimento, volume e é definida através de uma lista de pontos.

A classe `Ponto` representa uma coordenada num ambiente tri dimensional, contendo os atributos x, y, z e um indicador se este ponto está ocupado ou não.

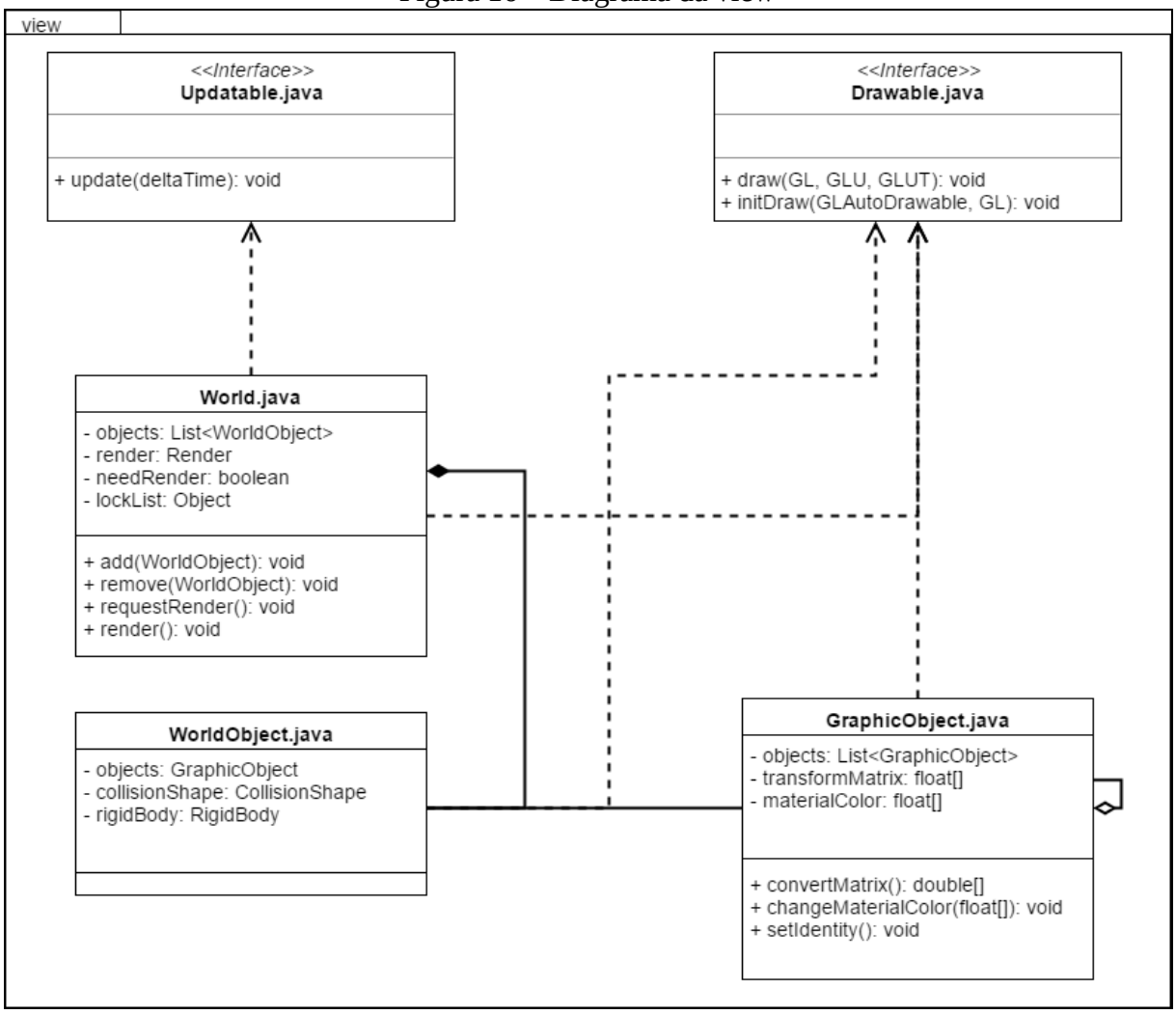
`Conteiner` e `Caixa` estendem de `Retangulo`, onde o `Conteiner` contém uma lista de `Caixa`, que irá depois representar quais caixas foram alocadas neste contêiner.

Apesar de ambos possuírem a lista de `Ponto` herdada do `Retangulo`, para cada um possui um significado diferente. Para o contêiner, a lista representa todo seu espaço interno e para caixa é representa uma referência para o ponto contido no contêiner.

A `view`, representada na figura 16, é composta por objetos que implementam a classe `Drawable` do JavaOpenGL. O World representa o mundo, ou seja, o agrupador das caixas. Cada objeto no cenário, é um `GraphicObject` que está encapsulado na classe `WorldObject`, que por sua vez está contida numa lista dentro de `World`. A classe `World` pode requisitar

renderização através do método `requestRender()`, que é utilizado na criação do cenário e se por ventura um objeto for adicionado em `runtime`.

Figura 16 – Diagrama da view



Fonte: elaborado pelo autor.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

O ambiente de desenvolvimento utilizado para a implementação do sistema foi o Eclipse Mars na versão 4.5.2. Essa versão foi utilizada pois, além das diversas facilidades para implementação do sistema, é uma versão que foi feita para funcionar com Java na versão 8, que é um dos requisitos não funcionais do software. Para controle de versão dos arquivos foi utilizado um servidor GIT no Bitbucket, que foi manuseado através do programa cliente SourceTree.

Para desenvolvimento da interface do sistema foi utilizado o Java Swing, que é padrão do sistema Java e para a representação 3D dos resultados foi utilizado a biblioteca `jogl.jar` (Java OpenGL), que facilita a implementação do OpenGL em Java, juntamente com a biblioteca JBullet, que facilita na modelagem dos objetos além de prover métodos para tratamento de física (que não chegou a ser utilizado pelo sistema).

O desenvolvimento foi divido em três etapas macro: alocação, algoritmo genético e visualização 3D.

3.3.1.1 Alocação

A primeira etapa do software foi desenvolver a alocação de caixas dentro de um contêiner. O método para adicionar uma caixa em um contêiner é definido no quadro 1:

Quadro 1 – Método para adição de caixa

```
01 public boolean addCaixa(Caixa c) {
02       if (c == null) {
03             return false;
04       }
05       if (Utils.isMenorIgual(c.getVolume(), getVolumeDisponivel())) {
06             if (alocarCaixa(c)) {
07                   mapearTrechosDisponiveisNoConteiner();
08                   return caixas.add(c);
09             }
10       }
11       return false;
12 }
```

Fonte: elaborado pelo autor.

O método faz uma validação prévia na linha 5 para saber se ainda existe volume disponível dentro do contêiner para alocação de uma nova caixa. Após desta validação prévia, o software irá chamar o método `alocarCaixa(...)` na linha 6.

A alocação de caixa é o principal método executado para verificar se uma caixa cabe ou não e onde deve ser alocada dentro do contêiner. Caso este método retornar `true`, a caixa é adicionada no atributo `caixas` do contêiner.

Em caso de sucesso na alocação da caixa, é remapeado no contêiner seus espaços disponíveis, que é explicado mais abaixo na seção 3.3.1.1.1.

No quadro 2 é apresentado como foi desenvolvido o método `alocarCaixa(...)`. O método executa três passos: análise de trechos disponíveis na linha 5; análise do espaço disponível na linha 7 e marcação de pontos no contêiner na linha 10 no caso de sucesso na análise.

Quadro 2 – Método para verificação de alocação de caixa

```
01 private boolean alocarCaixa(Caixa caixa) {
02       List<Ponto> listaMarcaveis = new ArrayList<>();
03
04       Map<BigDecimal, List<List<Ponto>>> mapaTrechosDisponiveis = null;
05       mapaTrechosDisponiveis = mapearTrechosDisponiveis(caixa,
06 mapaTrechosDisponiveisConteiner);
07       analisarEspacoDisponivel(mapaTrechosDisponiveis, listaMarcaveis,
08 caixa);
09       if (!listaMarcaveis.isEmpty()) {
10             marcarPontosNoConteiner(caixa, listaMarcaveis);
11             return true;
12       }
13       return false;
14 }
```

Fonte: elaborado pelo autor.

3.3.1.1.1 Análise de trechos disponíveis

A análise de trechos disponíveis é necessária para separar, por largura, pequenos trechos disponíveis para alocação da caixa em questão. Esta separação é feita em duas etapas, a primeira delas é o mapeamento dentro do contêiner e a segunda é um mapeamento baseado no retorno da primeira etapa. Ambos têm o mesmo sentido, porém o primeiro é realizado apenas para melhoria de performance.

O pseudocódigo apresentado no quadro 3 exibe como é feito o mapeamento dentro do contêiner de trechos disponíveis por largura. O método procura uma sequência de pontos disponíveis, por largura, e adiciona-os numa lista. Esta lista é armazenada dentro de um HashMap, que por sua vez tem como chave a altura da dimensão observada. Na saída deste método, o mapa terá n listas de pontos disponíveis por altura.

Quadro 3 – Pseudocódigo para separar trechos por largura no contêiner

```
01 private void mapearTrechosDisponiveisNoConteiner() {
02       BigDecimal altura = BigDecimal.ZERO;
03       BigDecimal comprimento = BigDecimal.ZERO;
04       int idxListaPorAltura = 0;
05       do {
06             do {
07             pontosPorAlturaComprimento = getPontosPorAlturaComp(...);
08                   for (Ponto ponto : pontosPorAlturaComprimento) {
09                         if (ponto.isDisponivel()) {
10                               adicionaNoMapa();
11                         } else {
12                               if (chegouNoLimiteDoMapa()) {
13                                     idxListaPorAltura++;
14                               }
15                         }
16                   }
17                   if (chegouNoLimiteDoMapa()) {
18                         idxListaPorAltura++;
19                   }
20                   comprimento = Utils.somaUm(comprimento);
21             } while (comprimento.compareTo(getComprimento()) < 0);
22             comprimento = BigDecimal.ZERO;
23             altura = Utils.somaUm(altura);
24             idxListaPorAltura = 0;
25       } while (altura.compareTo(getAltura()) < 0);
26 }
```

Fonte: elaborado pelo autor.

Na mesma ideia do método apresentado no quadro 3, o pseudocódigo do quadro 4 exibe como é feito quando uma caixa solicita o mapeamento de trechos disponíveis por altura. O método toma por base o mapa gerado no método anterior e percorre todas as listas de trechos disponíveis limitando-as num tamanho suficiente apenas para alocar a caixa em questão. Exemplo: O contêiner tem 5 trechos de 10 pontos na altura 1 e é solicitado o mapeamento para uma caixa que ocupa 5 pontos de largura, então o mapa irá armazenar, na altura 1, 10 trechos de 5 pontos cada.

Quadro 4 – Pseudocódigo para análise de trechos disponíveis da caixa

```
01 private Mapa mapearTrechosDisponiveis(...) {
02       para_cada_registro_no_mapa {
03             trechoPontos = mapa.getValue();
04             int idxListaPorAltura = 0;
05             para_cada_trecho {
06                   int larguraAlocada = 0;
07                   if (trecho.size() >= larguraNecessaria) {
08                         if (chegouNoLimiteDoMapa()) {
09                               armazenaNovaListaNoMapa();
10                         }
11                         para_cada_ponto {
12                               if (larguraAlocada < larguraNecessaria) {
13                                     larguraAlocada++;
14                               } else {
15                                     idxListaPorAltura++;
16                                     armazenaNovaListaNoMapa();
17                                     larguraAlocada = 1;
18                               }
19                               adicionarPontoNaListaAtualDoMapa();
20                         }
21                         if (mapaContemTrechoMenorQueNecessario()) {
22                               removeTrechoDoMapa();
23                               idxListaPorAltura--;
24                         }
25                         idxListaPorAltura++;
26                   }
27             }
28       }
29       return mapaTrechosDisponiveisParaCaixa;
30 }
```

Fonte: elaborado pelo autor.

3.3.1.1.2 Análise de espaços disponíveis

A análise de espaços disponíveis é um dos pontos principais da verificação para alocação da caixa no contêiner. É nela que é percorrido todos os espaços previamente analisados e verificado se a caixa cabe ou não, de fato, no contêiner. O quadro 5 ilustra o método desta análise.

Quadro 5 – Pseudocódigo do método para análise de espaços disponíveis

```
01 private boolean analisarEspacoDisponivel(...) {
02
03 faca {
04       para_cada_trecho_do_mapa {
05             if (idxIgnorar >= i) {
06                   continue;
07             }
08             para_cada_trecho_por_altura {
09                   listaMarcaveis.add(ponto);
10                   counterComprimento = 1;
11                   zAtual = ponto.getZ();
12                   yAtual = ponto.getY();
13                   if (!verificarAltura(...)) {
14                         idxIgnorar = limparListaMarcaveis(...);
15                         continue para_cada_trecho_do_mapa;
16                   }
17                   faca {
18                         zAtual = Utils.somaUm(zAtual);
19                         pontoVizinho = getPontoNoConteiner(...);
20                         if (pontoOcupado(pontoVizinho) ||
21                                                !verificarAltura(...)) {
22                         idxIgnorar = limparListaMarcaveis(...);
23                         continue para_cada_trecho_do_mapa;
24                         }
25                         listaMarcaveis.add(pontoVizinho);
26                         counterComprimento++;
27                   } enquanto (!comprimentoSuficiente);
28             }
29             if (!listaMarcaveis.isEmpty()) {
30                   return true;
31             }
32       y = Utils.somaUm(y);
33 } enquanto (!chegouAlturaMaxima);
34    return false;
35 }
```

Fonte: elaborado pelo autor.

O método conta com alguns parâmetros: o mapa pré-analisado conforme explicado na seção anterior; uma lista de pontos marcáveis e a caixa a ser alocada. A lista de pontos marcáveis funciona como uma lista de pontos auxiliares, que irá armazenando o histórico de pontos que estão sendo analisados e estão disponíveis para armazenar a caixa. Caso o método encontre uma barreira no caminho da análise, esta lista é zerada (linhas 14 e 22).

Posteriormente, o mapa de trechos disponíveis por largura é percorrido e, para cada ponto contido nele, é analisado também seus `n` pontos vizinhos tanto por comprimento (eixo `z`) quanto por altura (eixo `y`). Caso o método chegue até o fim da análise do comprimento por altura e a lista de marcáveis está preenchida (linha 29), então o método retorna `true`.

3.3.1.1.3 Marcar pontos no contêiner

Após efetuar a análise de espaços disponíveis no contêiner, o software irá analisar a lista de pontos marcáveis. Caso o mesmo não esteja vazio, identificar sua referência no contêiner e alterá-los para indicar `ocupado = true`.

No quadro 6 é ilustrado este funcionamento. Na linha 3 é percorrido os pontos contidos na lista de marcáveis e identificado seu ponto de referência no contêiner (linha 4). Após identificar este ponto, basta marca-lo como `ocupado = true` e adicioná-los na caixa.

Quadro 6 – Marcação dos pontos no contêiner

```
01 private void marcarPontosNoConteiner(Caixa caixa, List<Ponto>
02 listaMarcaveis) {
03       listaMarcaveis.forEach(p -> {
04             Ponto pontoReferenciaNoConteiner = getPontoNoConteiner(p);
05             pontoReferenciaNoConteiner.setOcupado(true);
06             caixa.addPonto(pontoReferenciaNoConteiner);
07       });
08 }
```

Fonte: elaborado pelo autor.

A adição do ponto na caixa não serve para as alocações seguintes, é apenas uma referência direta aos pontos que a caixa está ocupando no contêiner, para facilitar depois na saída em arquivo gerada pelo software.

3.3.1.2 Algoritmo genético

A segunda parte do desenvolvimento do trabalho, foi o algoritmo genético. A ideia central desta parte era deixá-la de tal forma que cada etapa do algoritmo pudesse ser trocada e/ou configurada de diversas formas pelo programador, sem necessidade de refatorar o código.

O processo foi dividido nas etapas: população; seleção; mutação; reprodução; ordenação e decisão. O processo total é ilustrado no diagrama de atividade (figura 12) encontrado na seção 3.2.2.

O contêiner foi definido como indivíduo do problema, onde as caixas alocadas nele representam os genes. A combinação de genes foi definida por combinar genes de dois indivíduos selecionados (pais) e então adicionar caixas que ainda não foram alocadas até o indivíduo filho possuir o número de genes igual ao pai + 1. Isto faz com que o algoritmo evolua a cada execução.

O software possui algumas parametrizações, sendo uma delas o percentual de ponto de corte, que é por padrão 90%. Este percentual indica quantos genes irá manter-se do indivíduo original, por exemplo, um indivíduo com 20 caixas, irá efetuar o corte no índice 18, mantendo

as primeiras 18 caixas. Outra parametrização é quanto ao tamanho da população, que se mantém fixa até o final do algoritmo. Trabalhando junto com a função objetivo do algoritmo, é possível parametrizar também o número máximo de gerações, que têm por objetivo parar de alocar caixas no contêiner caso o algoritmo atinja o número definido no parâmetro, para evitar que o algoritmo fique executando por demasiado tempo.

A primeira etapa do processo, é definido pelo `looping` principal do algoritmo genético. O quadro 7 ilustra o `looping` principal do software. O algoritmo começa chamando o código responsável pela população inicial randômica (linha 1), e em seguida segue num `looping` até que a função de objetivo seja atingida ou o número máximo de gerações seja atingido. O tamanho da população inicial respeita a configuração, que por padrão é 20.

Quadro 7 – Pseudocódigo do looping principal do sistema

```
01 Populacao populacao = metodoPopulacao.popular(listaCaixas, x, y, z);
02
03 while (!funcaoObjetivo.atingiuObjetivo(melhorIndividuo, listaCaixas) &&
04         geracaoAtual <= maxGeracoes) {
05
06       populacao = novaGeracao(populacao, listaCaixas);
07       melhorIndividuo = populacao.getIndividuo(0);
08       geracaoAtual++;
09 }
10 if (!alocouTodasCaixas(listaCaixas, melhorIndividuo)) {
11       filtrarCaixasAlocadas(listaCaixas, populacao);
12       novaExecucao.executar(listaCaixas);
13 } else {
14       exibirResultados();
15 }
```

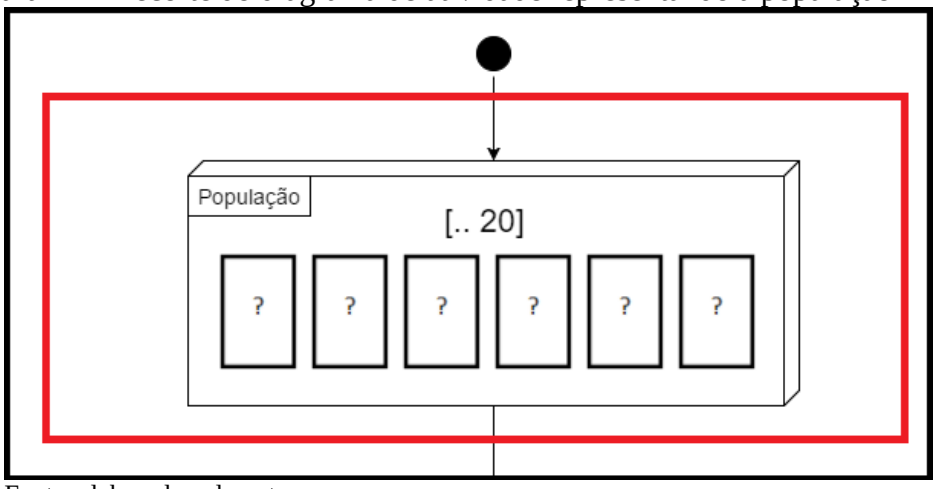
Fonte: elaborado pelo autor.

A cada iteração, o algoritmo passa pelo processo de criar uma nova geração (linha 6), identifica o melhor indivíduo e incrementa o número de gerações. Ao fim, identifica se todas as caixas pré-definidas pelo usuário já foram alocadas (linha 10). Caso já foram, exibe os resultados, caso não foram filtra as caixas restantes e inicia uma nova execução (linhas 11 e 12).

3.3.1.2.1 População inicial

A população inicial, primeira etapa do diagrama de atividade (figura 13 da seção 3.2.2) demonstrada na figura 17, é definida por popular aleatoriamente duas caixas da lista de caixas que estão pendentes de alocação no contêiner.

Figura 17 – Recorte do diagrama de atividade representando a população inicial



Fonte: elaborado pelo autor.

O quadro 8 ilustra a classe criada para popular randomicamente a população inicial que será utilizada pelo algoritmo. Enquanto não atingir o número definido de tamanho da população, o algoritmo embaralha aleatoriamente a lista de caixas (linha 5) e atribui duas caixas não alocadas no contêiner (linhas 6 e 7). Ao final, a população é submetida a ordenação (linha 10) para garantir que o melhor indivíduo esteja na primeira posição do `array`.

Quadro 8 – Pseudocódigo para população randômica

```
01 public Populacao popular(listaCaixas, x, y, z) {
02       Populacao pop = new Populacao();
03       while (pop.getTotalIndividuos() < TAMANHO_POPULACAO) {
04             Conteiner c = new Conteiner(x, y, z);
05             Collections.shuffle(listaCaixas);
06             c.addCaixa(getCaixaDaLista(c, listaCaixas));
07             c.addCaixa(getCaixaDaLista(c, listaCaixas));
08             pop.addIndividuo(c);
09       }
10       pop.ordenarPopulacao();
11       return pop;
12 }
```

Fonte: elaborado pelo autor.

3.3.1.2.2 Nova geração

O processo de criar uma nova geração (quadro 9), começa instanciando uma nova `Populacao`, mantendo o melhor indivíduo da população anterior (linha 4). Depois disto, irá efetuar os métodos de seleção (linha 6), mutação e reprodução (linhas 9 e 10) até a população que está sendo gerada atingir o número máximo da população, que é configurável.

Após a geração da nova população, esta é submetida à ordenação na linha 14. A ordenação irá manter o melhor indivíduo, ou seja, o que possui um maior índice de aptidão, na primeira posição e os demais subsequentemente. O melhor indivíduo ficará automaticamente na próxima geração e os demais servirão apenas para a seleção randômica de pais.

Quadro 9 – Pseudocódigo do método para nova geração

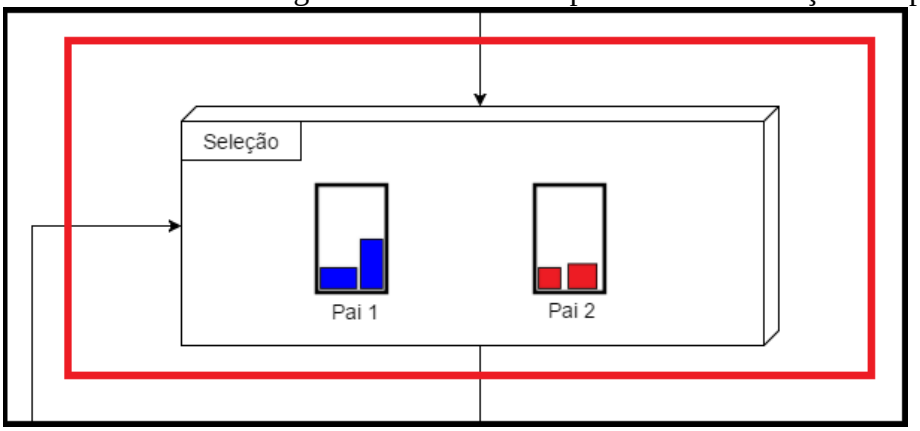
```
01 public Populacao novaGeracao(populacao, listaCaixas) {
02       Populacao novaPopulacao = new Populacao();
03       // Mantém o melhor indivíduo
04       novaPopulacao.addIndividuo(populacao.getIndividuo(0));
05       while (novaPopulacao.getTotalIndividuos() < TAMANHO_POPULACAO) {
06             pais = this.metodoSelecao.selecionar(populacao);
07             filhos = new Conteiner[2];
08             Collections.shuffle(listaCaixas);
09             filhos = metodoReproducao.reproduzir(pais[0], pais[1],
10 listaCaixas);
11             novaPopulacao.addIndividuo(filhos[0]);
12             novaPopulacao.addIndividuo(filhos[1]);
13       }
14       novaPopulacao.ordenarPopulacao();
15       return novaPopulacao;
16 }
```

Fonte: elaborado pelo autor.

3.3.1.2.3 Seleção dos pais

O próximo passo do algoritmo é a seleção dos pais para depois efetuar a mutação e reprodução. Este passo é exibido na figura 18, originalmente encontrada no diagrama de atividade (figura 13 da seção 3.2.2).

Figura 18 – Recorte do diagrama de atividade representando a seleção dos pais



Fonte: elaborado pelo autor.

A seleção implementada foi híbrida (quadro 10), realizando uma seleção por torneiro (POLI et al., 2008) em uma população elitista que contém o melhor indivíduo da população anterior. É gerado dois índices aleatórios, recuperados os indivíduos da população e atribuídos ao `array` de retorno nas linhas 6 e 7.

Quadro 10 – Classe de seleção

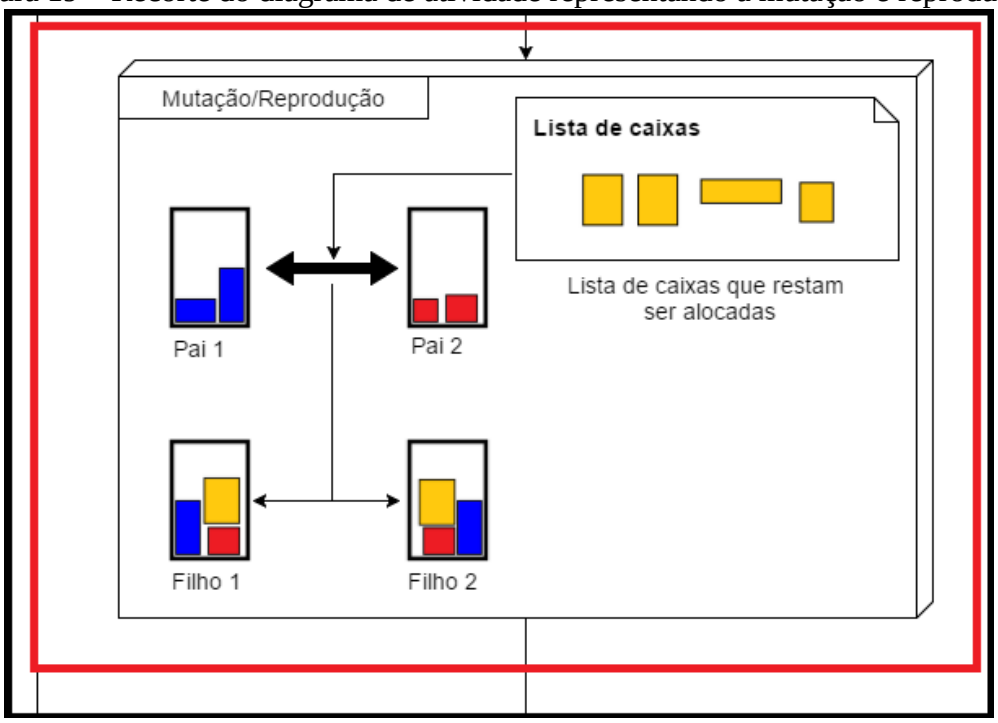
```
01 public Conteiner[] selecionar(Populacao populacao) {
02       final Random r = new Random();
03       final int totalIndividuos = populacao.getTotalIndividuos();
04
05       final Conteiner[] pais = new Conteiner[2];
06       pais[0] = populacao.getIndividuo(r.nextInt(totalIndividuos));
07       pais[1] = populacao.getIndividuo(r.nextInt(totalIndividuos));
08
09       return pais;
10 }
```

Fonte: elaborado pelo autor.

3.3.1.2.4 Mutação e Reprodução

Após a seleção dos pais, é a hora da reprodução (recorte ilustrado pela figura 19). A mutação foi implementada juntamente no método de reprodução, pois não foi identificada a necessidade de separar as duas coisas, tendo em vista que os genes são determinados simplesmente pelas caixas.

Figura 19 – Recorte do diagrama de atividade representando a mutação e reprodução



Fonte: elaborado pelo autor.

Esta etapa é a mais importante do algoritmo. É nela que são feitas as trocas de genes entre os pais e a inserção das caixas pré-configuradas pelo usuário nos filhos, além da própria geração dos filhos em si.

Durante o desenvolvimento do trabalho, existiram duas formas de reprodução e mutação. A primeira delas reproduzia e mutava definindo um ponto de corte fixo, mantendo as caixas do índice 0 até o índice do ponto de corte do pai 1 e a partir do índice do ponto de corte até o final da lista de caixas do pai 2, para ambos os filhos de maneira alternada, e então

era efetuada a tentativa de adicionar uma caixa da lista de caixas não alocadas. Foi visto que este método, por manter a posição das caixas no contêiner idênticas as posições dos pais, estava gerando muita segmentação dentro do contêiner, fazendo com que o algoritmo ficasse pouco otimizado. A segunda abordagem, e definitiva, eliminou boa parte da segmentação dentro do contêiner. O quadro 11 contém o pseudocódigo que detalha este processo.

O método de mutação foi adaptado do método Subtree Mutation (POLI et al., 2008) e seu princípio é definir um ponto de corte para cada pai selecionado (linhas 6 e 7), baseado no parâmetro de ponto de corte, que é definido por padrão em 90%. Definido o ponto de corte, são gerados os filhos e executado o método de mutação. Baseado no ponto de corte, a mutação irá misturar genes do pai 1 com o pai 2 nos filhos 1 e 2, conforme visto nas linhas 13 até 16. Para cada gene irá verificar se atingiu ou não o ponto de corte (linha 32) e, caso ainda não tenha atingido, chama o método de adicionar caixa (linha 35). O método de adicionar caixa eliminou o problema de segmentação comentado anteriormente, porque cada caixa dos seus genes (herdados ou não), serão realocados de acordo com o contexto atual do contêiner.

Feito a mutação, é necessário adicionar caixas da lista de caixas não alocadas nos filhos, de modo que os filhos fiquem com 1 caixa a mais alocada em relação aos pais. Esta conta é feita nas linhas 18 e 19. Nestas linhas são definidas quantas caixas da lista precisam ser alocadas pra geração filha atingir o tamanho dos genes dos pais + 1. Lembrando que o método de adicionar caixa pode retornar `false` em ambos os casos, indicando que não foi possível alocar a caixa. Desta maneira é possível ter uma geração filha com menos caixas, mas não necessariamente com menor aptidão.

Quadro 11 – Pseudocódigo do método de mutação e reprodução

```
01 public Conteiner[] reproduzir(pai1, pai2, listaCaixas) {
02       genesPai1 = pai1.getCaixas();
03       genesPai2 = pai2.getCaixas();
04       removerCaixasIguais(genesPai1, genesPai2);
05
06       int pontoDeCorte1 = (int) (genesPai1.size() * PONTO_CORTE);
07       int pontoDeCorte2 = (int) (genesPai2.size() * PONTO_CORTE);
08
09       final Conteiner[] filhos = new Conteiner[2];
10       filhos[0] = new Conteiner(pai1);
11       filhos[1] = new Conteiner(pai2);
12
13       mutar(pontoDeCorte1, genesPai1, filhos[0]);
14       mutar(pontoDeCorte2, genesPai2, filhos[0]);
15       mutar(pontoDeCorte2, genesPai2, filhos[1]);
16       mutar(pontoDeCorte1, genesPai1, filhos[1]);
17
18       int caixasAdd1 = (genesPai1.size() - pontoDeCorte1) + 1;
19       int caixasAdd2 = (genesPai2.size() - pontoDeCorte2) + 1;
20       for (int i = 0; i < caixasAdd1; i++) {
21             filhos[0].addCaixa(getCaixaNaoAlocada(listaCaixas));
22       }
23       for (int i = 0; i < caixasAdd2; i++) {
24             filhos[1].addCaixa(getCaixaNaoAlocada(listaCaixas));
25       }
26       return filhos;
27 }
28
29 private void mutar(int pontoDeCorte, genesPai, filho) {
30       int contador = 0;
31       for (Caixa caixa : genesPai) {
32             if (contador == pontoDeCorte) {
33                   break;
34             }
35             filho.addCaixa(caixa);
36             contador++;
37       }
38 }
```

Fonte: elaborado pelo autor.

3.3.1.2.5 Ordenação

Por fim do ciclo da nova geração, é efetuada uma ordenação da população, que é necessária apenas para ganhos de performance do algoritmo. A ordenação é representada pelo recorte exibido na figura 20.

Figura 20 – Recorte do diagrama de atividade representando ordenação

Fonte: elaborado pelo autor.

A ordenação foi implementada de maneira simples, o quadro 12 representa sua implementação. É percorrida a lista de indivíduos contida na população e então comparado de dois em dois indivíduos. Quem tiver o menor volume (linha 12) troca de posição com o maior volume, sendo que o de maior volume irá ocupar a posição menor. O algoritmo para quando não trocou nenhum indivíduo de posição.

Quadro 12 – Método de ordenação da população

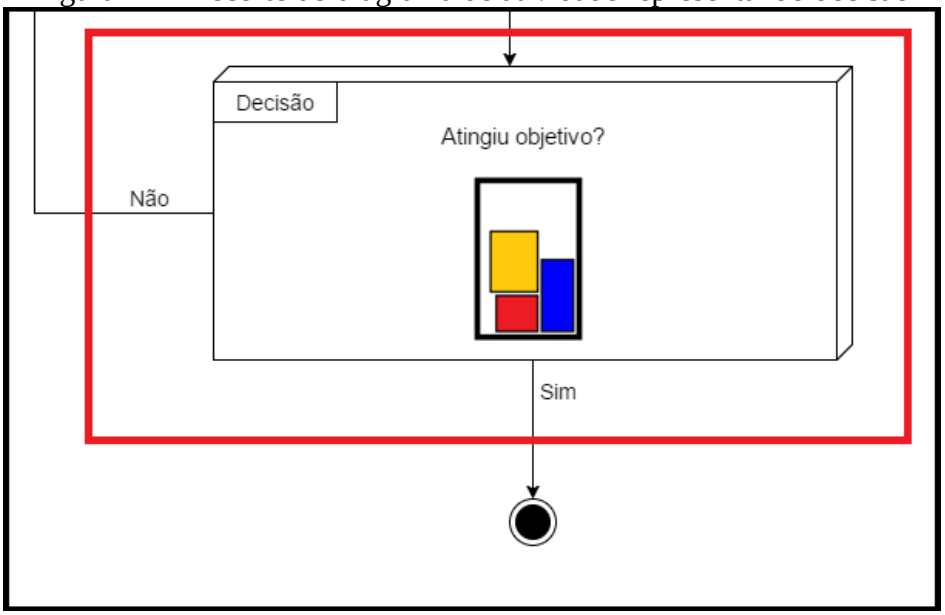
```
01 public void ordenarPopulacao() {
02       boolean trocou = true;
03       while (trocou) {
04             trocou = false;
05             for (int i = 0; i < individuos.length - 1; i++) {
06                   final Conteiner individuo1 = individuos[i];
07                   final Conteiner individuo2 = individuos[i + 1];
08                   final BigDecimal volume1 = individuo1 == null ?
09 BigDecimal.ZERO : individuo1.getPercentualVolumeOcupado();
10                   final BigDecimal volume2 = individuo2 == null ?
11 BigDecimal.ZERO : individuo2.getPercentualVolumeOcupado();
12                   if (Utils.isMenor(volume1, volume2)) {
13                         final Conteiner temp = individuo1;
14                         individuos[i] = individuo2;
15                         individuos[i + 1] = temp;
16                         trocou = true;
17                   }
18             }
19       }
20 }
```

Fonte: elaborado pelo autor.

3.3.1.2.6 Decisão

Por fim da execução do algoritmo genético, é a hora da decisão do algoritmo se continua a gerar novas gerações ou se para a execução e torna disponível ao usuário a visualização 3D. Este passo no diagrama de atividade (figura 13 da seção 3.2.2) é exibido de forma recortada na figura 21.

Figura 21 – Recorte do diagrama de atividade representando decisão



Fonte: elaborado pelo autor.

Sua implementação é descrita através do pseudocódigo encontrado no quadro 13. A função de objetivo será atingida caso o melhor indivíduo da população atingir um percentual pré-definido (configurável) de volume preenchido ou já possuir todas as caixas da lista pré-configurada pelo usuário alocadas (linha 13).

Quadro 13 – Pseudocódigo do método de decisão

```
01 public boolean atingiuObjetivo(melhorIndividuo, listaCaixas) {
02     return melhorIndividuo.atingiuTaxaVolume() ||
03             contemTodasCaixas(listaCaixas);
04 }
05
06 private boolean contemTodasCaixas(melhorIndividuo, listaCaixas) {
07     return melhorIndividuo.getCaixas().size() == listaCaixas.size();
08 }
```

Fonte: elaborado pelo autor.

3.3.1.3 Visualização 3D

A visualização 3D foi o último passo desenvolvido no software. Utilizando o JavaOpenGL foi criado o mundo gráfico, implementado a interface `Drawable` disponibilizada pela biblioteca. O mundo gráfico no software, definido pela classe `World`, é um agrupador de `WorldObject` (ou objeto gráfico), que também implementa a classe `Drawable` da biblioteca JavaOpenGL.

O quadro 14 representa a adição de um objeto no mundo. É necessário sincronizar esta adição (linha 2) para que funcione sem erros em processamento com múltiplas *threads*.

Quadro 14 – Adição de um objeto no mundo gráfico

```
01 public void add(WorldObject worldObject) {
02       synchronized (lockList) {
03             objects.add(worldObject);
04       }
05 }
```

Fonte: elaborado pelo autor.

Após a sua adição no mundo gráfico, é necessário que este objeto seja renderizado na tela e, para isto, é implementado os métodos `draw` e `update` da interface `Drawable`, exibidos no quadro 15.

Quadro 15 – Métodos implementados para desenhar os objetos

```
01 @Override
02 public void draw(GL gl, final GLU glu, GLUT glut) {
03       synchronized (lockList) {
04             objects.forEach(o -> o.draw(gl, glu, glut));
05       }
06 }
07
08 @Override
09 public void update(float deltaTime) {
10       if (needRender) {
11             needRender = false;
12             render();
13       }
14       synchronized (lockList) {
15             objects.forEach(o -> o.update(deltaTime));
16       }
17 }
```

Fonte: elaborado pelo autor.

Para a renderização foi criada uma classe específica para isto, denominada `Render` e esta possui uma lista de `Drawables`, que nada mais são que objetos gráficos desenháveis. No quadro 16 é representado o método de inicialização da renderização dos objetos na tela. O método começa inicializando as classes do OpenGL responsáveis pela renderização. Logo em seguida é ligada a luz do ambiente (linha 10) e depois para cada objeto gráfico é chamado o método `initDraw`, que irá desenhá-lo no mundo.

Quadro 16 – Método responsável pela inicialização da renderização

```
01 public void init(GLAutoDrawable drawable) {
02       glDrawable = drawable;
03       gl = drawable.getGL();
04       glu = new GLU();
05       glut = new GLUT();
06       glDrawable.setGL(new DebugGL(gl));
07
08       gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
09
10       turnOnLight();
11
12       gl.glEnable(GL.GL_CULL_FACE);
13
14       gl.glEnable(GL.GL_DEPTH_TEST);
15
16       for (Drawable d : drawings) {
17             d.initDraw(drawable, gl);
18       }
19 }
```

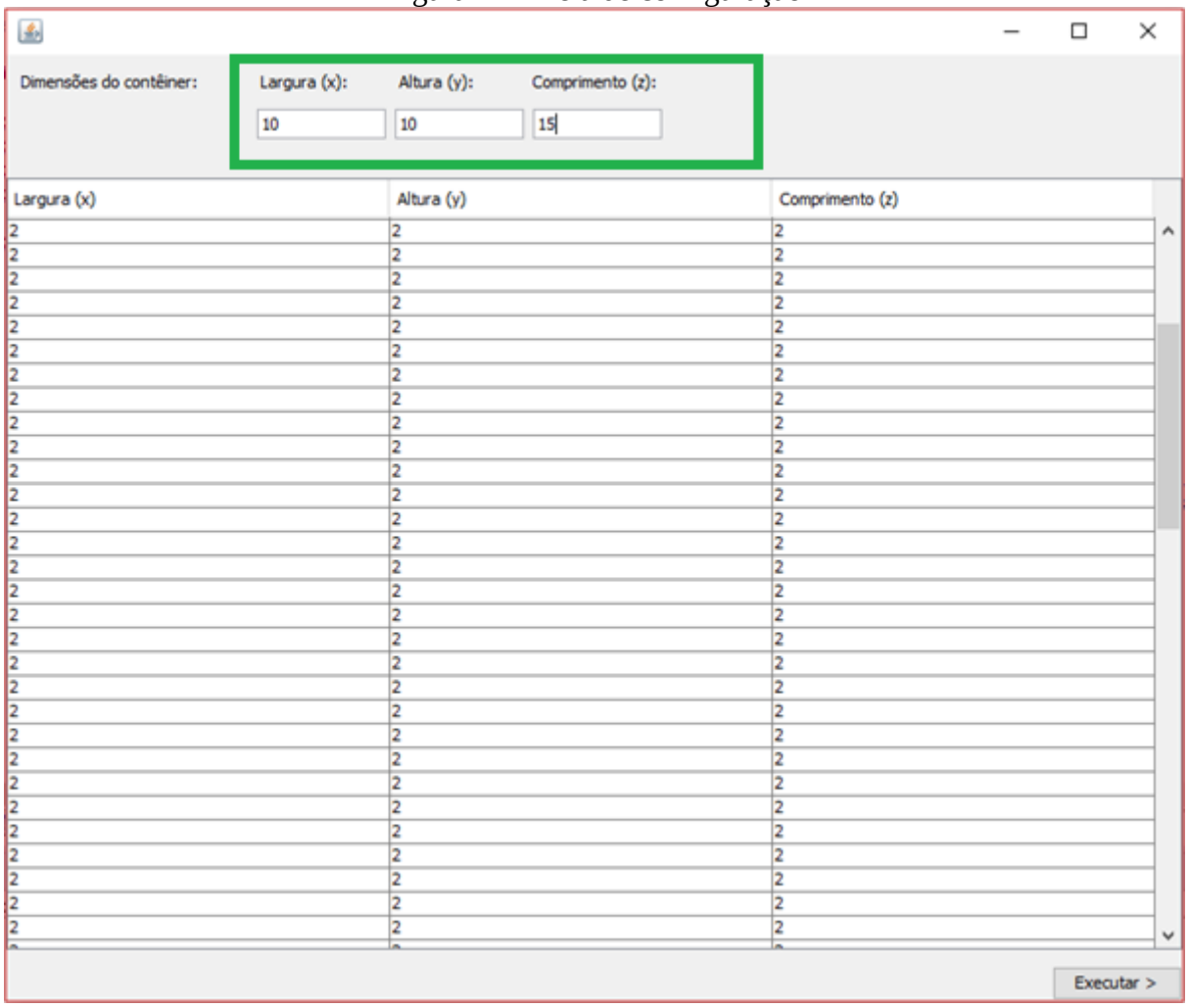
Fonte: elaborado pelo autor.

3.3.2 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade da implementação, com as suas funcionalidades. É demonstrado como realizar as configurações e como operar a saída do software.

A figura 22 exibe a tela de configuração. No quadro destacado em verde está contido as configurações de dimensão no contêiner, onde é possível ao usuário configurar a largura (x), altura (y) e o comprimento (z). No meio da tela contém uma tabela com as caixas que deverão serão alocadas nos contêineres. No fim da tela, é encontrado o botão de `Executar`, que irá pegar as configurações e começar a execução do algoritmo genético.

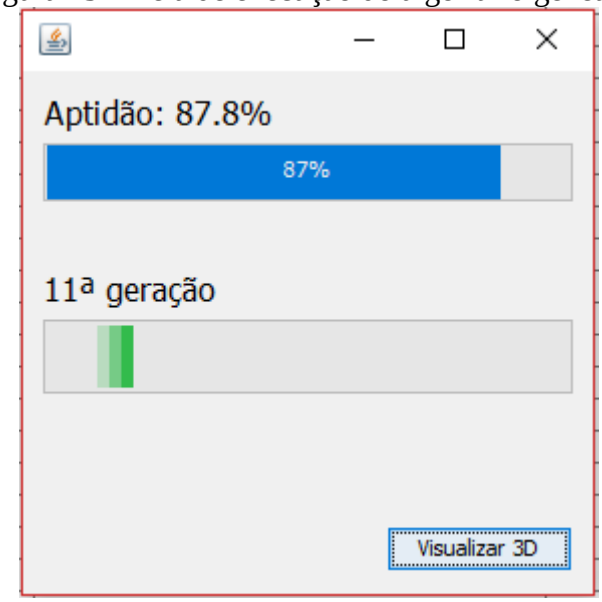
Figura 22 – Tela de configuração



Fonte: elaborado pelo autor.

Após clicar no botão `Executar`, é exibida a tela da figura 23. Esta tela tem por objetivo exibir ao usuário um status da execução atual do algoritmo. A primeira barra de progresso representa a aptidão atual alcançada pelo algoritmo e a segunda barra tem por objetivo indicar que o algoritmo está executando. Existe também a representação de aptidão textual, logo acima da primeira barra de progresso e logo abaixo possui a geração atual do algoritmo. Nesta mesma tela está contido um botão `Visualizar 3D`, que serve como visualização prévia do algoritmo sendo executado.

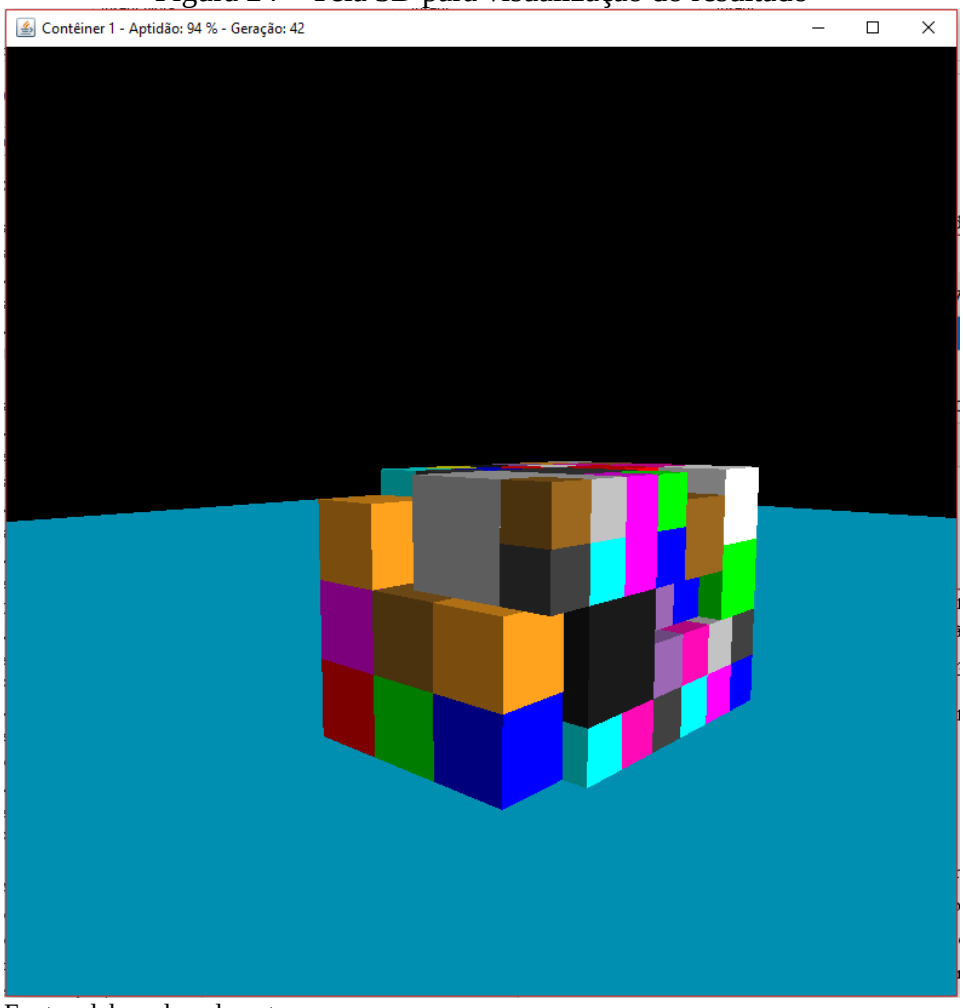
Figura 23 – Tela de execução do algoritmo genético



Fonte: elaborado pelo autor.

Após o algoritmo atingir a função objetivo, é exibido a tela representada na figura 24, onde o usuário pode ficar navegando para visualizar o contêiner de todos os lados.

Figura 24 – Tela 3D para visualização do resultado



Fonte: elaborado pelo autor.

3.4 ANÁLISE DOS RESULTADOS

Os testes deste trabalho foram adaptados dos testes de Silva e Soma (2003), onde os cenários são divididos em classes. Para este software, os testes foram divididos nas seguintes classes:

a) classe 1: A maioria dos itens (70%) são muito altos e profundos (L=(2~3), A=(5~7), C=(5~7));
b) classe 2: A maioria dos itens (70%) são muito largos e profundos (L=(5~7), A=(2~3), C=(5~7));
c) classe 3: A maioria dos itens (70%) são muito altos e largos (L=(5~7), A=(5~7), C=(2~3));
d) classe 4: A maioria dos itens (70%) têm grandes dimensões (L=(5~7), A=(5~7), C=(5~7));
e) classe 5: A maioria dos itens (70%) têm pequenas dimensões (L=(2~3), A=(2~3), C=(2~3)).

Por classe foram efetuados testes alternando as configurações de número de população e percentual de ponto de corte onde, por configuração, foi executado o algoritmo 3 vezes e extraído sua média de execução para comparação. O número de caixas por classe foi fixado em 50. Para cada teste as dimensões do contêiner foram definidas em: largura=10; altura=10 e comprimento=15, onde todos os contêineres possuem as mesmas dimensões. Os testes foram executados em um computador desktop com processador Intel® Core™ i5-4460 CPU @ 3.20GHz, memória RAM de 8GB e sistema operacional Windows 10 64 bits.

3.4.1 Análise dos resultados – Classe 1

A classe 1 é definida por 70% dos itens muito altos e largos e os demais aleatórios. A tabela 1 demonstra as médias de execução deste cenário variando o parâmetro de número de população, com ajuste de ponto de corte em 90%.

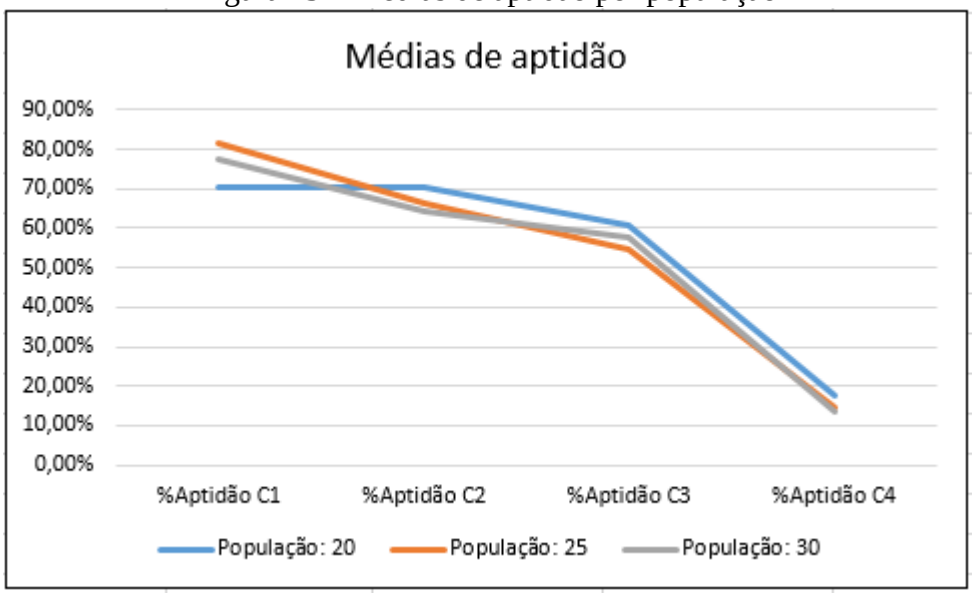
Tabela 1 – Médias de aptidão por população

População	%Aptidão contêiner 1	%Aptidão contêiner 2	%Aptidão contêiner 3	%Aptidão contêiner 4	Tempo total
20	70,66%	70,60%	60,66%	17,53%	288
25	81,59%	66,17%	54,88%	14,55%	380
30	77,37%	64,44%	57,46%	13,70%	488

Fonte: elaborado pelo autor.

É possível observar que aumentando o número da população ocorreu um aumento significativo no percentual de aptidão do primeiro contêiner e uma redução nos demais, por consequência. A figura 25 exibe esta melhora graficamente.

Figura 25 – Médias de aptidão por população

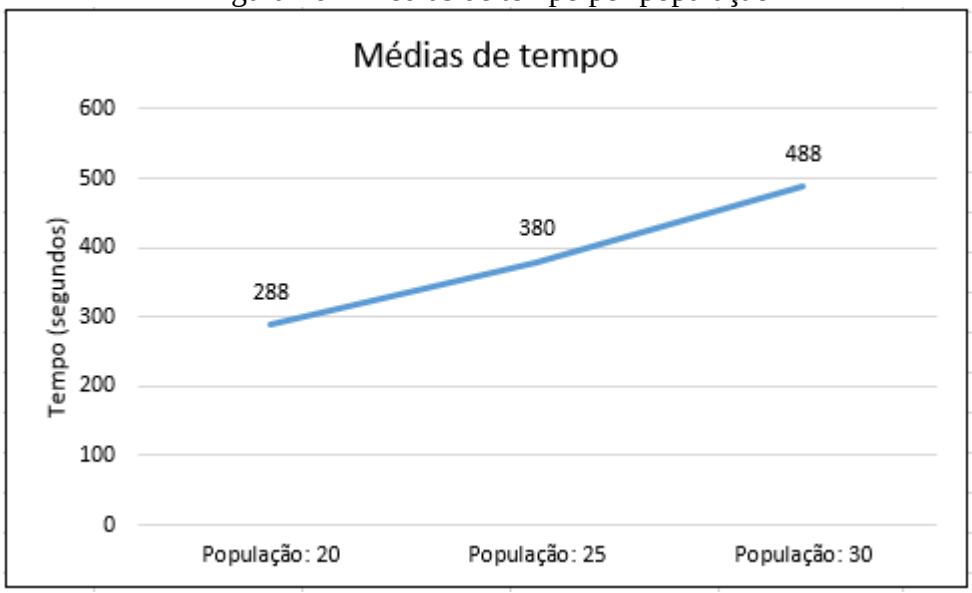


Fonte: elaborado pelo autor.

Nota-se que para este cenário, a melhor configuração foi com população 25, onde gerou um percentual de aptidão para o primeiro contêiner de 81,59%. Os demais contêineres tiveram uma redução no percentual que ocorre naturalmente quando o primeiro contêiner consegue um atingimento maior. Outro detalhe interessante é que a primeira configuração teve um destaque bem grande nos contêineres 2, 3 e 4, gerando uma ocupação bem maior.

Todavia, os tempos foram significativamente maiores de uma configuração de população para outra. Nas médias apresentadas na figura 26, é possível observar um crescente linear da população 20 até a 30.

Figura 26 – Médias de tempo por população



Fonte: elaborado pelo autor.

Outro fator testado foi o percentual de ponto de corte, onde para termos de comparação foi utilizado de base a média da geração de população 20, representada na primeira linha da tabela 2, efetuada no teste anterior.

Tabela 2 – Médias de aptidão por ponto de corte

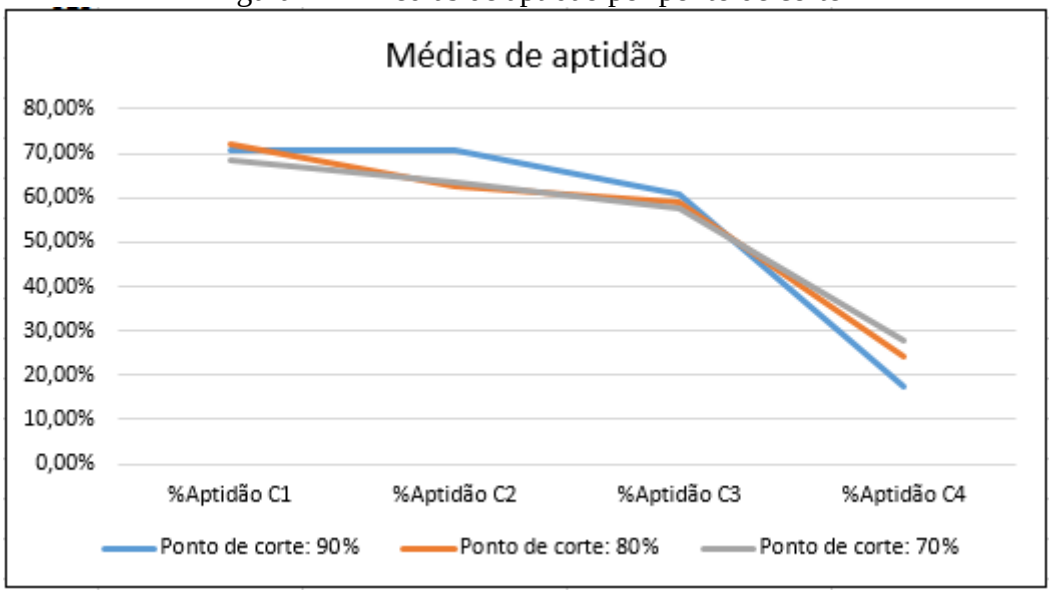
Ponto de corte	%Aptidão contêiner 1	%Aptidão contêiner 2	%Aptidão contêiner 3	%Aptidão contêiner 4	Tempo total
90%	70,66%	70,60%	60,66%	17,53%	288
80%	71,95%	62,77%	58,92%	24,11%	306
70%	68,68%	63,44%	57,64%	27,66%	296,33

Fonte: elaborado pelo autor.

Pela tabela 2 nota-se que não houve uma mudança significativa das médias de aptidão variando o ponto de corte. Nota-se que com o parâmetro fixado em 90%, o algoritmo conseguiu manter um bom nível de aptidão nos contêineres 2 e 3, fazendo com que o 4° contêiner fosse gerado com um número menor de volume ocupado.

A figura 27 mostra o caso em que é notável o ponto de corte em 90% destacando-se nos contêineres intermediários.

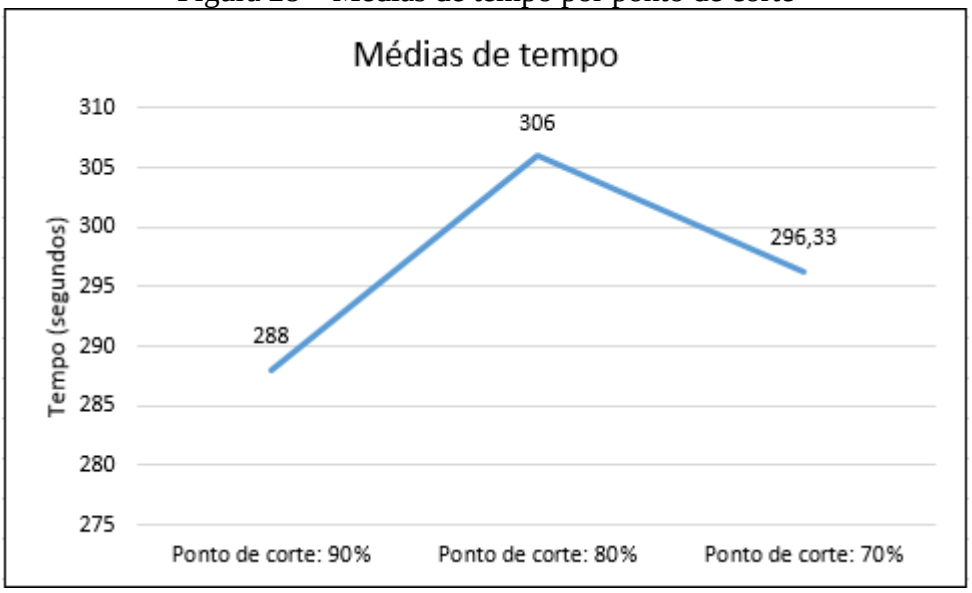
Figura 27 – Médias de aptidão por ponto de corte



Fonte: elaborado pelo autor.

Em relação ao tempo, o ajuste em 80% mostrou-se a pior configuração. A figura 28 exibe graficamente os resultados. O ajuste em 90% conseguiu as melhores médias de tempo e, o ajuste em 80%, as maiores.

Figura 28 – Médias de tempo por ponto de corte



Fonte: elaborado pelo autor.

3.4.2 Análise dos resultados – Classe 2

A classe 2 é definida por 70% dos itens muito largos e profundos e os demais aleatórios. A tabela 3 demonstra as médias de execução deste cenário variando o parâmetro de número de população, com ajuste de ponto de corte em 90%.

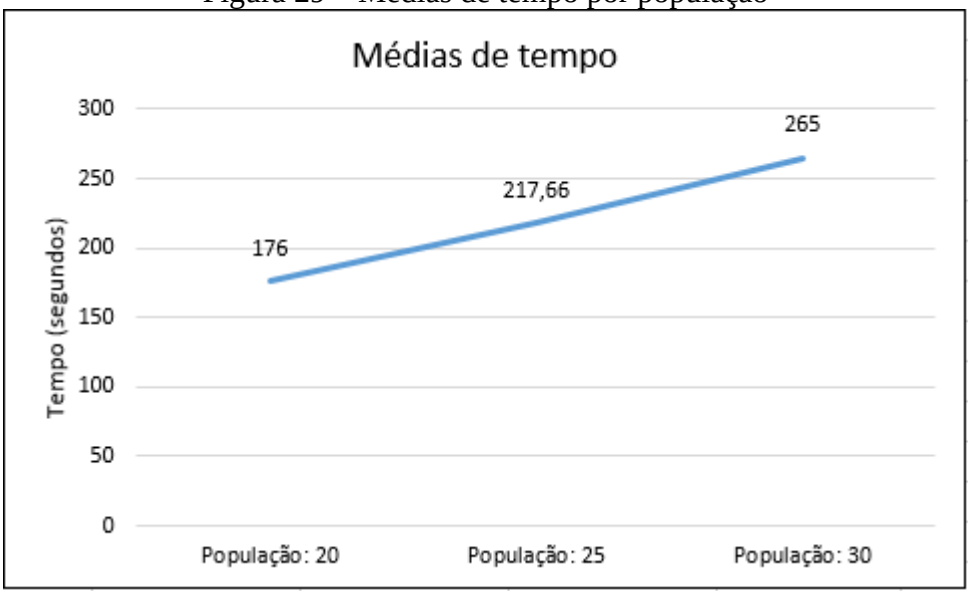
Tabela 3 – Médias de aptidão por população

População	%Aptidão contêiner 1	%Aptidão contêiner 2	%Aptidão contêiner 3	%Aptidão contêiner 4	Tempo total
20	77,17%	70,77%	57,02%	23,73%	176
25	75,04%	65,15%	57,15%	15,60%	217,66
30	73,50%	64,59%	57,84%	27,00%	265

Fonte: elaborado pelo autor.

É possível observar que aumentando o número da população ocorreu uma diminuição no percentual de aptidão do primeiro e segundo contêineres. O tempo seguiu uma linha crescente, de acordo com a população, conforme esperado. A figura 29 exibe graficamente o aumento dos tempos.

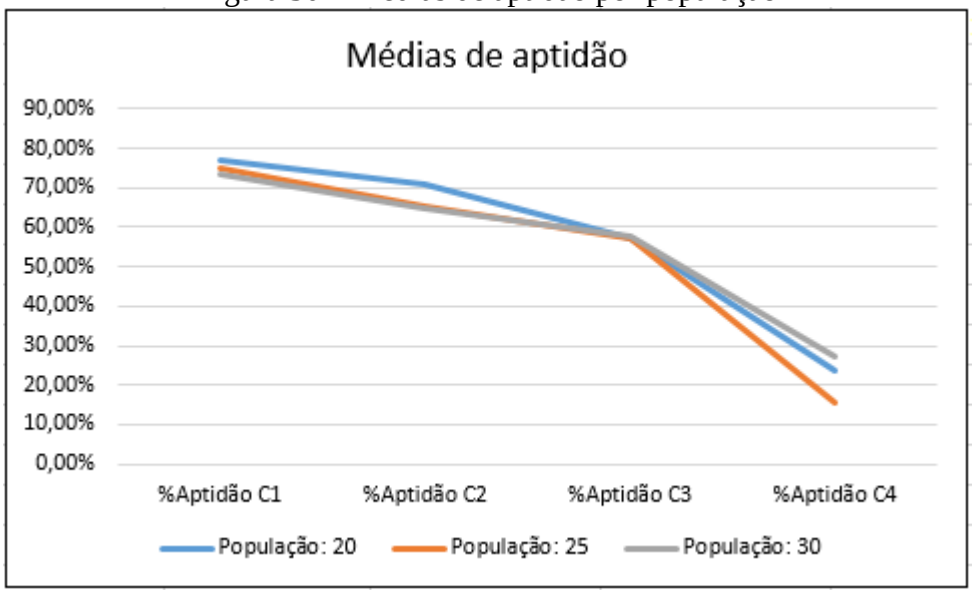
Figura 29 – Médias de tempo por população



Fonte: elaborado pelo autor.

A figura 30 ilustra os percentuais de aptidão por configuração de população. A linha destacada em azul exibe os resultados superiores atingidos pela população 20. Considerando o tempo (figura 29) versus aptidão (figura 30), é possível concluir que para este cenário a melhor configuração de população é 20.

Figura 30 – Médias de aptidão por população



Fonte: elaborado pelo autor.

Em relação ao percentual de ponto de corte, assim como no teste anterior, foi utilizado de base a média da geração de população 20. A tabela 4 exibe os resultados.

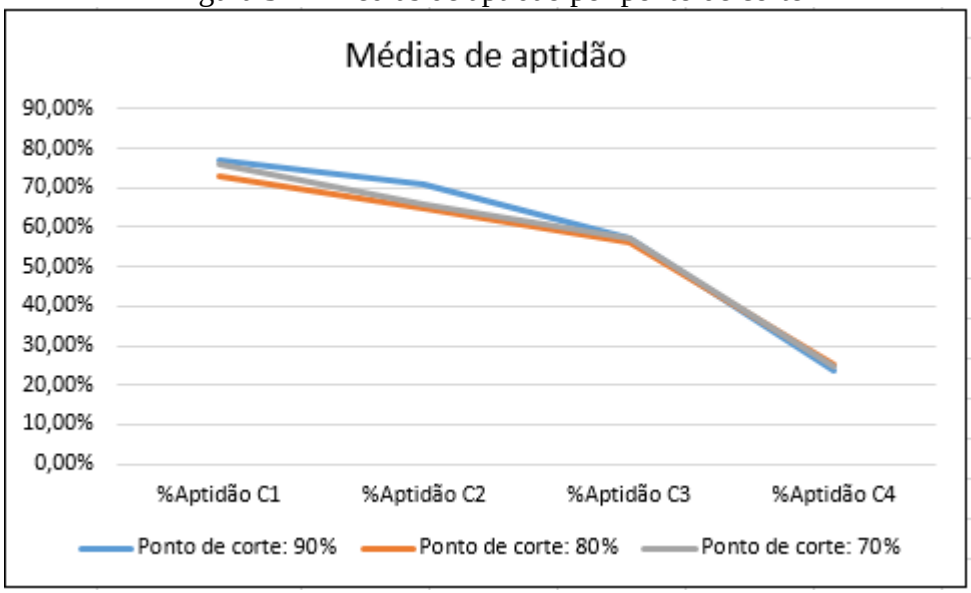
Tabela 4 – Médias de aptidão por ponto de corte

Ponto de corte	%Aptidão contêiner 1	%Aptidão contêiner 2	%Aptidão contêiner 3	%Aptidão contêiner 4	Tempo total
90%	77,17%	70,77%	57,02%	23,73%	176
80%	73,02%	65,02%	56,31%	25,20%	175
70%	75,93%	65,75%	57,13%	24,75%	163

Fonte: elaborado pelo autor.

Pela tabela 4 nota-se que não houve uma mudança significativa das médias de aptidão variando o ponto de corte, assim como no cenário 1. Do ponto de vista de aptidão, o melhor cenário foi o percentual de 90%. A figura 31 exibe as médias de aptidão dos contêineres por configuração de ponto de corte.

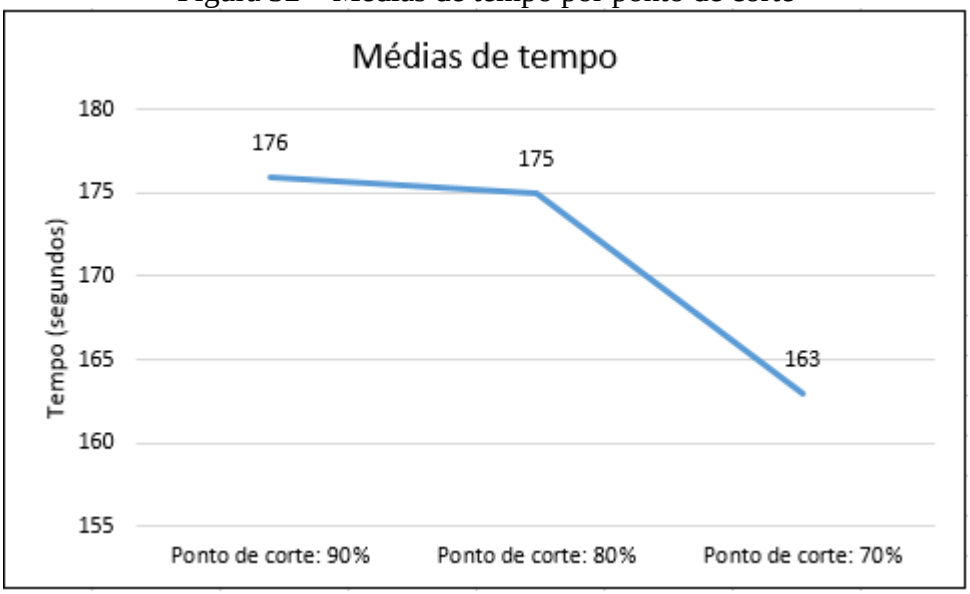
Figura 31 – Médias de aptidão por ponto de corte



Fonte: elaborado pelo autor.

Porém o algoritmo obteve ganhos de performance com um percentual de ponto de corte menor que 90%, como mostra a figura 32. Nota-se que a média de tempo da configuração em 70% em relação à média em 90% teve uma redução significativa de 13 segundos.

Figura 32 – Médias de tempo por ponto de corte



Fonte: elaborado pelo autor.

3.4.3 Análise dos resultados – Classe 3

A classe 3 é definida por 70% dos itens muito altos e largos e os demais aleatórios. A tabela 5 demonstra as médias de execução deste cenário variando o parâmetro de número de população, com ajuste de ponto de corte em 90%.

Tabela 5 – Médias de aptidão por população

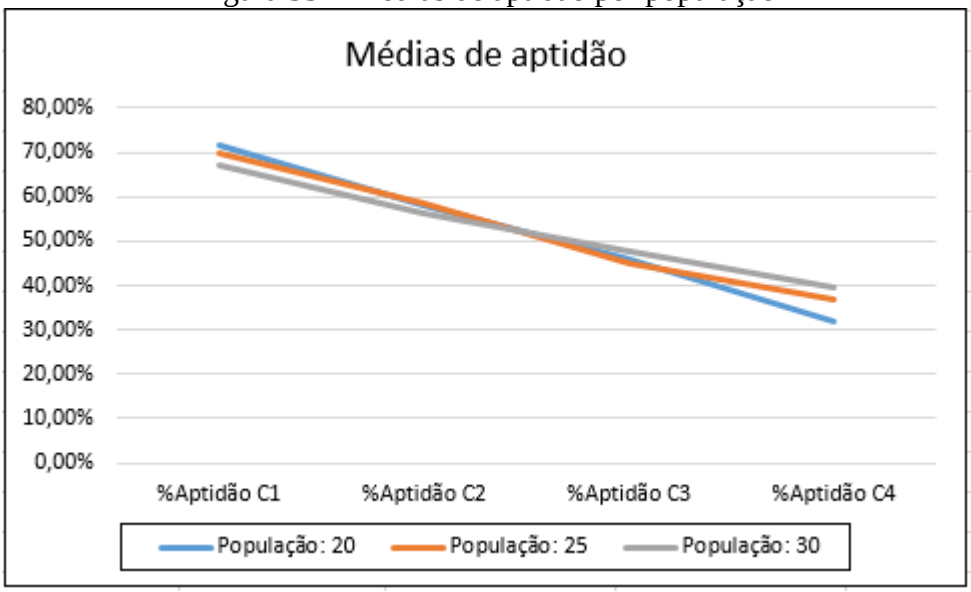
População	%Aptidão contêiner 1	%Aptidão contêiner 2	%Aptidão contêiner 3	%Aptidão contêiner 4	Tempo total
20	71,73%	58,22%	46,02%	31,87%	256,33
25	69,98%	58,70%	45,08%	36,70%	191,3
30	67,17%	56,11%	47,75%	39,47%	272,33

Fonte: elaborado pelo autor.

É possível observar que aumentando o número da população ocorreu uma queda de aptidão do primeiro contêiner. A população em 30 refletiu os piores percentuais e pior tempo total, comprovando que para este cenário o ajuste no número da população deve ser baixo.

É possível observar também, que o fato do primeiro contêiner estar maior ou menor ocupado fez com que os demais contêineres ficassem proporcionalmente inversos, como por exemplo é possível observar na linha destacada em cinza da figura 33.

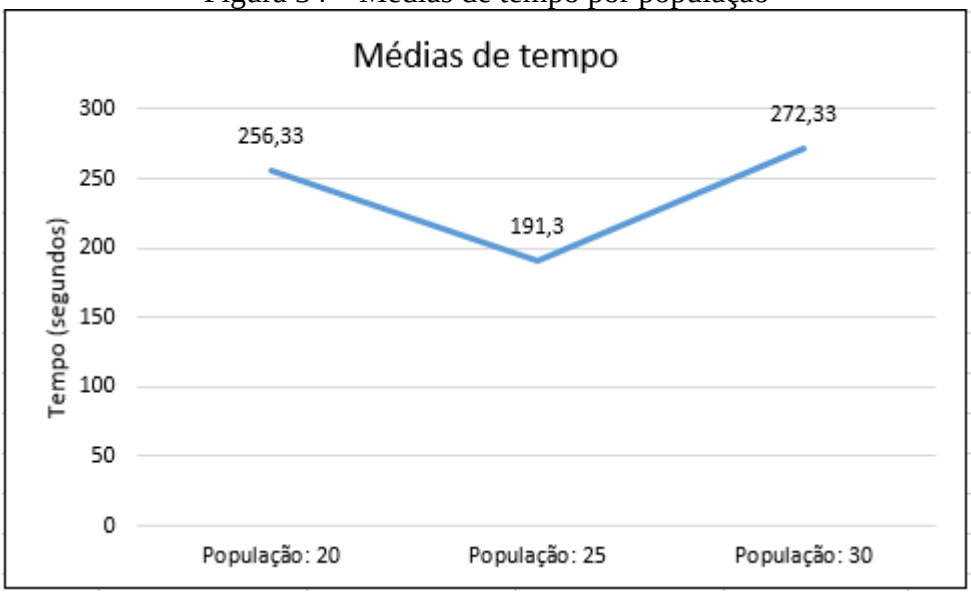
Figura 33 – Médias de aptidão por população



Fonte: elaborado pelo autor.

Em relação ao tempo, houve uma queda brusca quando a população foi configurada com um número intermediário, no caso 25. A figura 34 exibe graficamente esta queda. O pior tempo ficou com a maior configuração de população.

Figura 34 – Médias de tempo por população



Fonte: elaborado pelo autor.

O percentual de ponto de corte, assim como no teste anterior, foi utilizado de base a média da geração de população 20. A tabela 6 exibe os resultados.

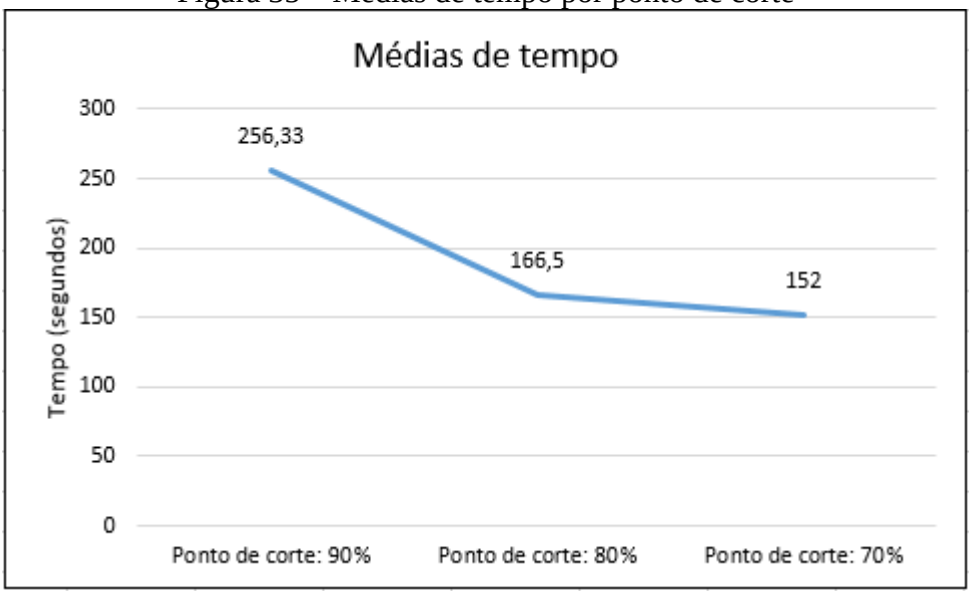
Tabela 6 – Médias de aptidão por ponto de corte

Ponto de corte	%Aptidão contêiner 1	%Aptidão contêiner 2	%Aptidão contêiner 3	%Aptidão contêiner 4	Tempo total
90%	71,73%	58,22%	46,02%	31,87%	256,33
80%	63,06%	56,87%	46,73%	40,70%	166,5
70%	59,13%	58,93%	52,37%	40,37%	152

Fonte: elaborado pelo autor.

Pela tabela 6 nota-se uma queda brusca no tempo de execução quanto menor for o parâmetro de percentual de ponto de corte. A diferença chega a 104,33 segundos entre a primeira média e a última, o que torna o algoritmo significativamente mais rápido com percentual de ponto de corte menor. A figura 35 exibe esta diferença graficamente.

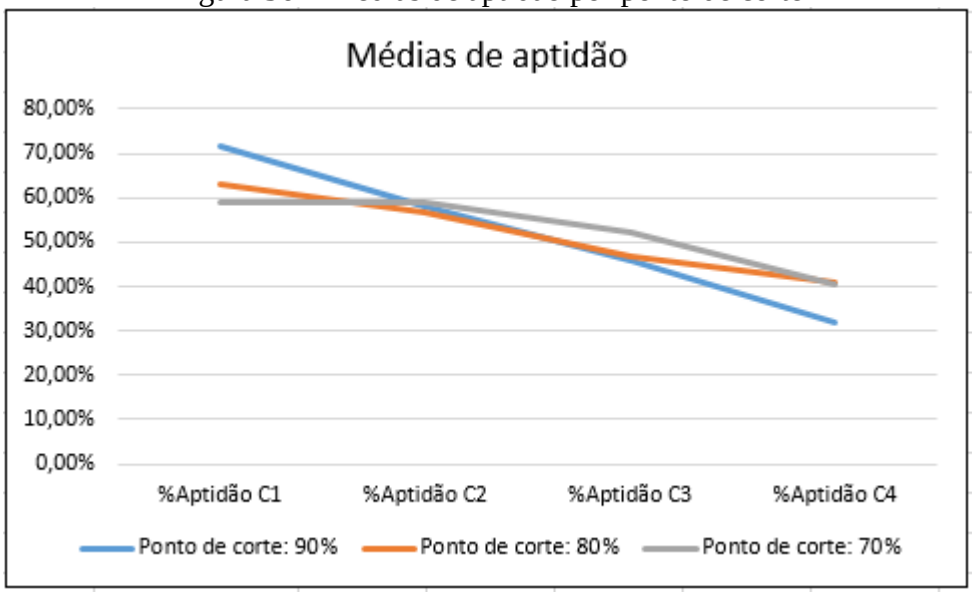
Figura 35 – Médias de tempo por ponto de corte



Fonte: elaborado pelo autor.

Entretanto, o percentual de aptidão do primeiro contêiner também sofreu uma queda brusca, de 71,73% para 59,13% (12,6%), tornando o resultado final menos otimizado. A figura 36 exibe esta diferença. É possível observar que em termos de otimização, a linha destacada em azul (ponto de corte 90%) teve resultados muito melhores que a linha destacada em cinza (ponto de corte 70%).

Figura 36 – Médias de aptidão por ponto de corte



Fonte: elaborado pelo autor.

3.4.4 Análise dos resultados – Classe 4

A classe 4 é definida por 70% dos itens com grandes dimensões e os demais com dimensões aleatórias. A tabela 7 demonstra as médias de execução deste cenário variando o parâmetro de número de população, com ajuste de ponto de corte em 90%.

Tabela 7 – Médias de número de contêineres e tempo por população

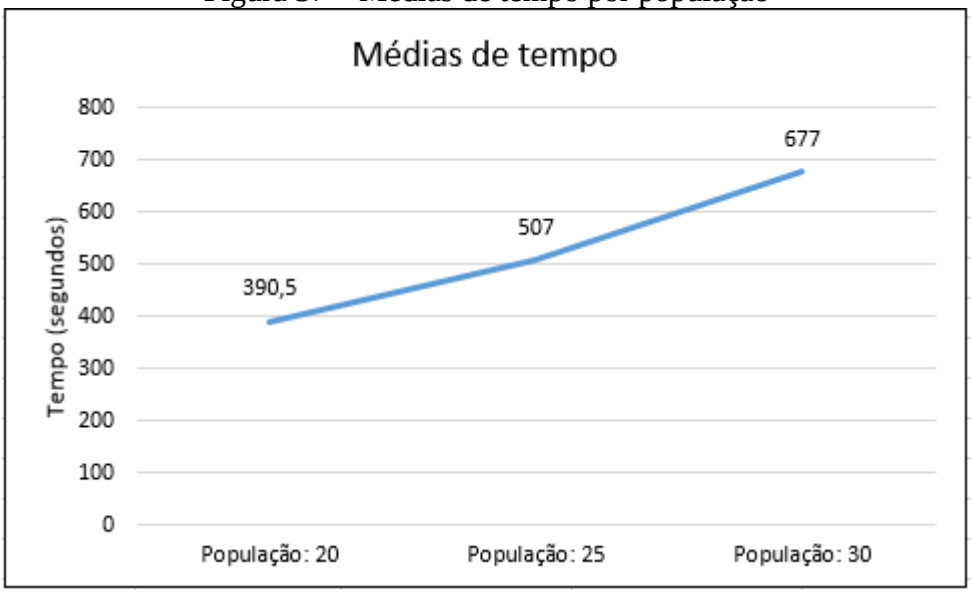
População	Número de contêineres	Tempo total
20	11,5	390,5
25	12,5	507
30	12,5	677

Fonte: elaborado pelo autor.

Por tratar-se de um cenário onde o resultado final irá gerar inúmeros contêineres, a forma de comparação foi um pouco diferente das classes anteriores. Na tabela 7 é demonstrado o número de contêineres utilizados até a solução final e o tempo total de execução. É possível notar que o número de contêineres não sofreu alterações bruscas da população 20 até 30, tendo apenas um contêiner de margem.

A figura 37 exibe graficamente as alterações sofridas no tempo total de execução por população. É possível observar que neste cenário onde o software possui uma lista de caixas enormes e diversos contêineres, qualquer adição no número da população torna-se o tempo bem maior. Da população 30 para 20 ocorreu uma diferença de 286,5 segundos, que é mais que a metade do tempo médio executado com população 20.

Figura 37 – Médias de tempo por população



Fonte: elaborado pelo autor.

Em relação ao percentual de ponto de corte, não houve alterações na maneira como o teste foi efetuado, assim como no teste anterior, foi utilizado de base a média da geração de população 20. A tabela 8 exibe os resultados.

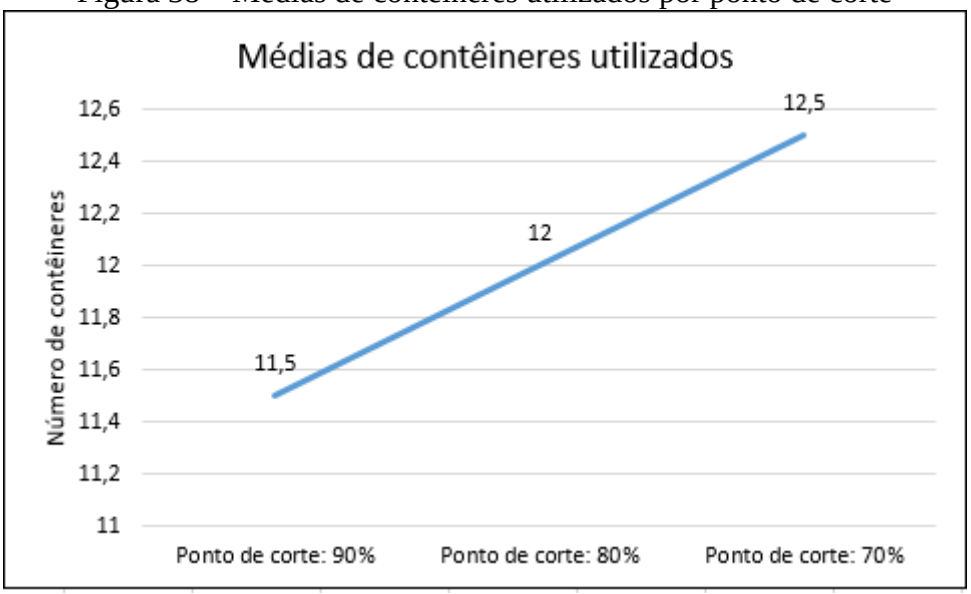
Tabela 8 – Médias de número de contêineres e tempo por ponto de corte

Ponto de corte	Número de contêineres	Tempo total
90%	11,5	390,5
80%	12	420
70%	12,5	451

Fonte: elaborado pelo autor.

Pela tabela 8 nota-se um aumento gradual de números de contêineres a medida que o ponto de corte diminui. A figura 38 exibe esta situação.

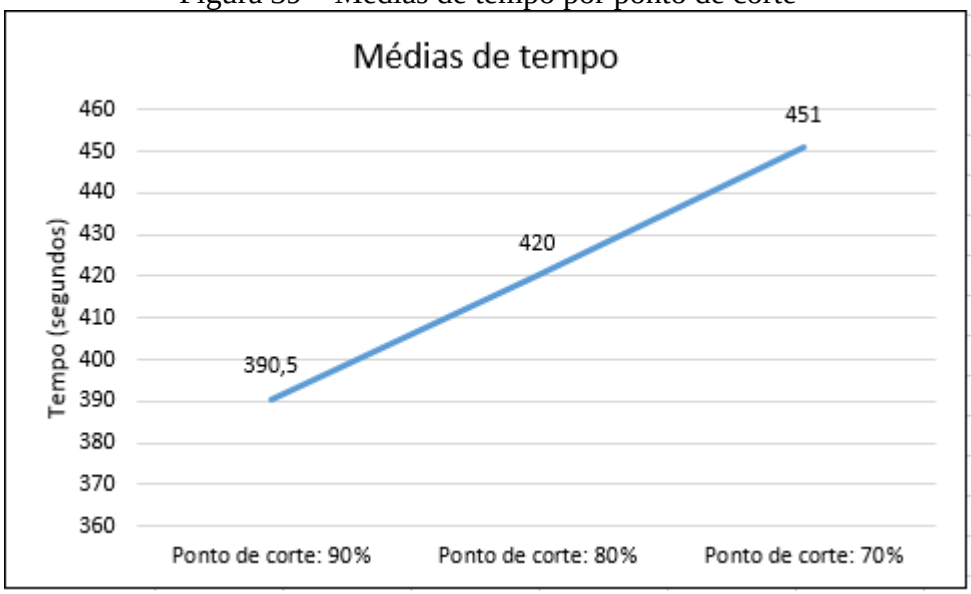
Figura 38 – Médias de contêineres utilizados por ponto de corte



Fonte: elaborado pelo autor.

Em relação ao tempo, assim como no teste por população, há um aumento significativo a medida que o ponto de corte diminui. A figura 39 ilustra esta situação.

Figura 39 – Médias de tempo por ponto de corte



Fonte: elaborado pelo autor.

3.4.5 Análise dos resultados – Classe 5

A classe 5 é definida por 70% dos itens com pequenas dimensões e os demais com dimensões aleatórias. A tabela 9 demonstra as médias de execução deste cenário variando o parâmetro de número de população, com ajuste de ponto de corte em 90%.

Tabela 9 – Médias de geração objetivo e tempo por população

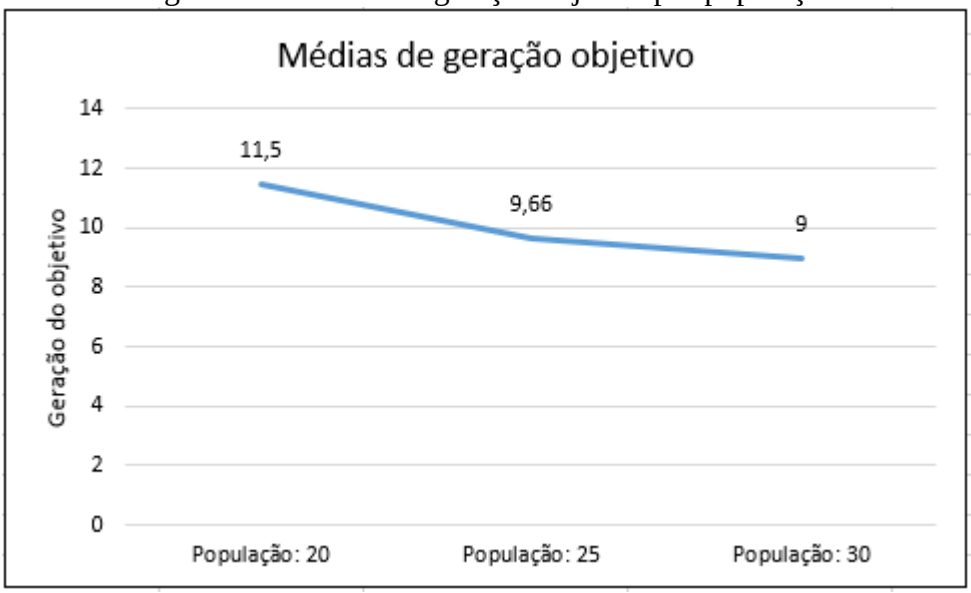
População	Geração objetivo	Tempo total
20	11,5	8,66
25	9,66	8,66
30	9	9,33

Fonte: elaborado pelo autor.

Por tratar-se de um cenário onde o resultado final irá gerar somente um contêiner, a forma de comparação foi um pouco diferente das classes anteriores. Na tabela 9 é demonstrado o número médio de gerações necessárias para atingir o objetivo e o tempo total de execução. É possível notar que o tempo total não sofreu alterações relevantes ao ser aumentado o número da população.

A figura 40 exibe graficamente as alterações sofridas na geração de objetivo atingida por cada configuração de população. É possível observar que neste cenário onde o software possui uma lista de caixas pequenas, onde todas elas cabem em um único contêiner, qualquer adição no número da população faz com que o algoritmo atinja o resultado final em um número menor de gerações.

Figura 40 – Médias de geração objetivo por população



Fonte: elaborado pelo autor.

Em relação ao percentual de ponto de corte, não houve alterações na maneira como o teste foi efetuado, assim como no teste anterior, foi utilizado de base a média da geração de população 20. A tabela 10 exibe os resultados.

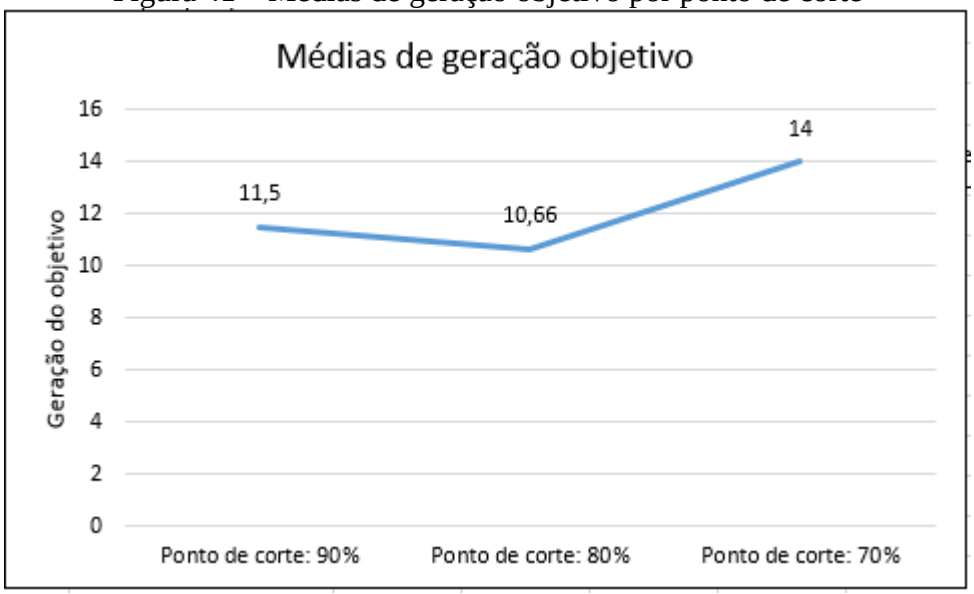
Tabela 10 – Médias de número de contêineres e tempo por ponto de corte

Ponto de corte	Geração objetivo	Tempo total
90%	11,5	8,66
80%	10,66	6,66
70%	14	9,33

Fonte: elaborado pelo autor.

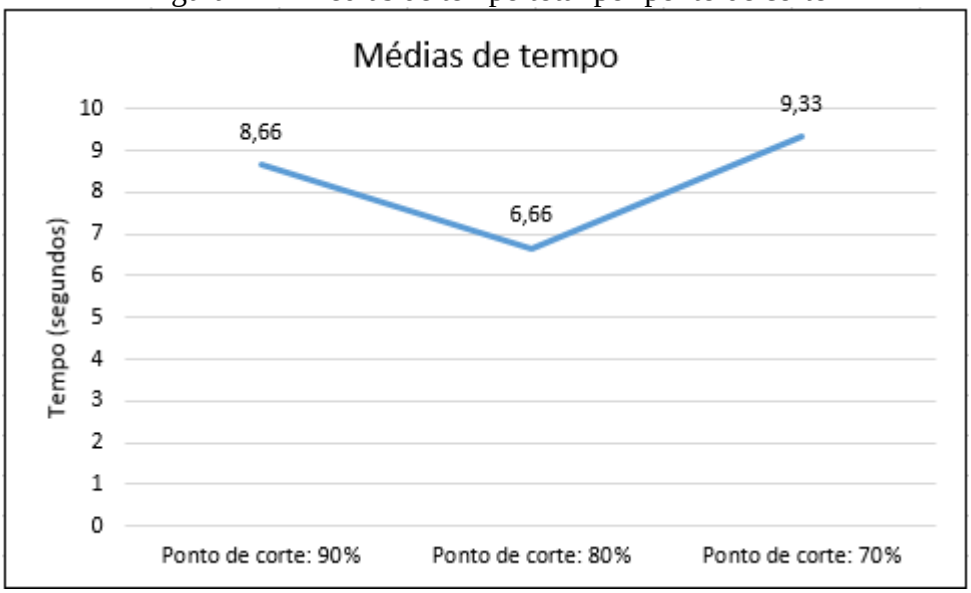
Olhando para a tabela 10 é possível observar que o ponto de corte em 80% foi o mais performático em ambos os sentidos. A figura 41 ilustra a diferença entre o número médio de gerações necessárias para atingir o objetivo e a figura 42 exibe a diferença entre o tempo total de execução com as configurações de ponto de corte diferentes.

Figura 41 – Médias de geração objetivo por ponto de corte



Fonte: elaborado pelo autor.

Figura 42 – Médias de tempo total por ponto de corte



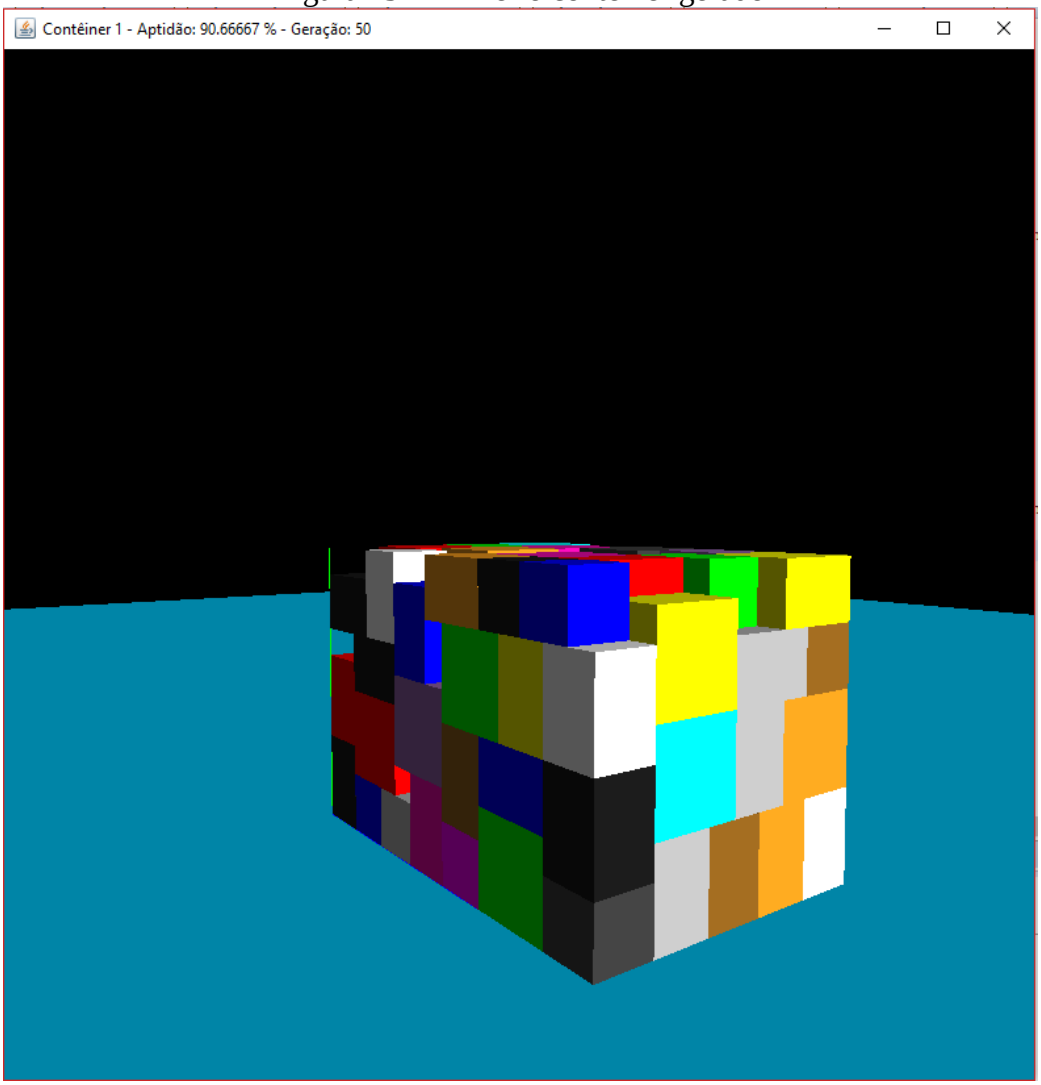
Fonte: elaborado pelo autor.

3.4.6 Análise gráfica do preenchimento de contêineres

Para o teste de análise gráfica do preenchimento de contêineres foi utilizado a classe 5 de testes, com configurações de ponto de corte em 90% e tamanho de população 20. O número de caixas foi fixado em 150 para que o resultado final gerasse mais de um contêiner de saída.

A figura 43 exibe como ficou a saída gerada pelo algoritmo do primeiro contêiner gerado. É possível observar que não houve sobreposições no contêiner gerado e o algoritmo atingiu 90,66% de aptidão na geração 50 (geração definida como limite).

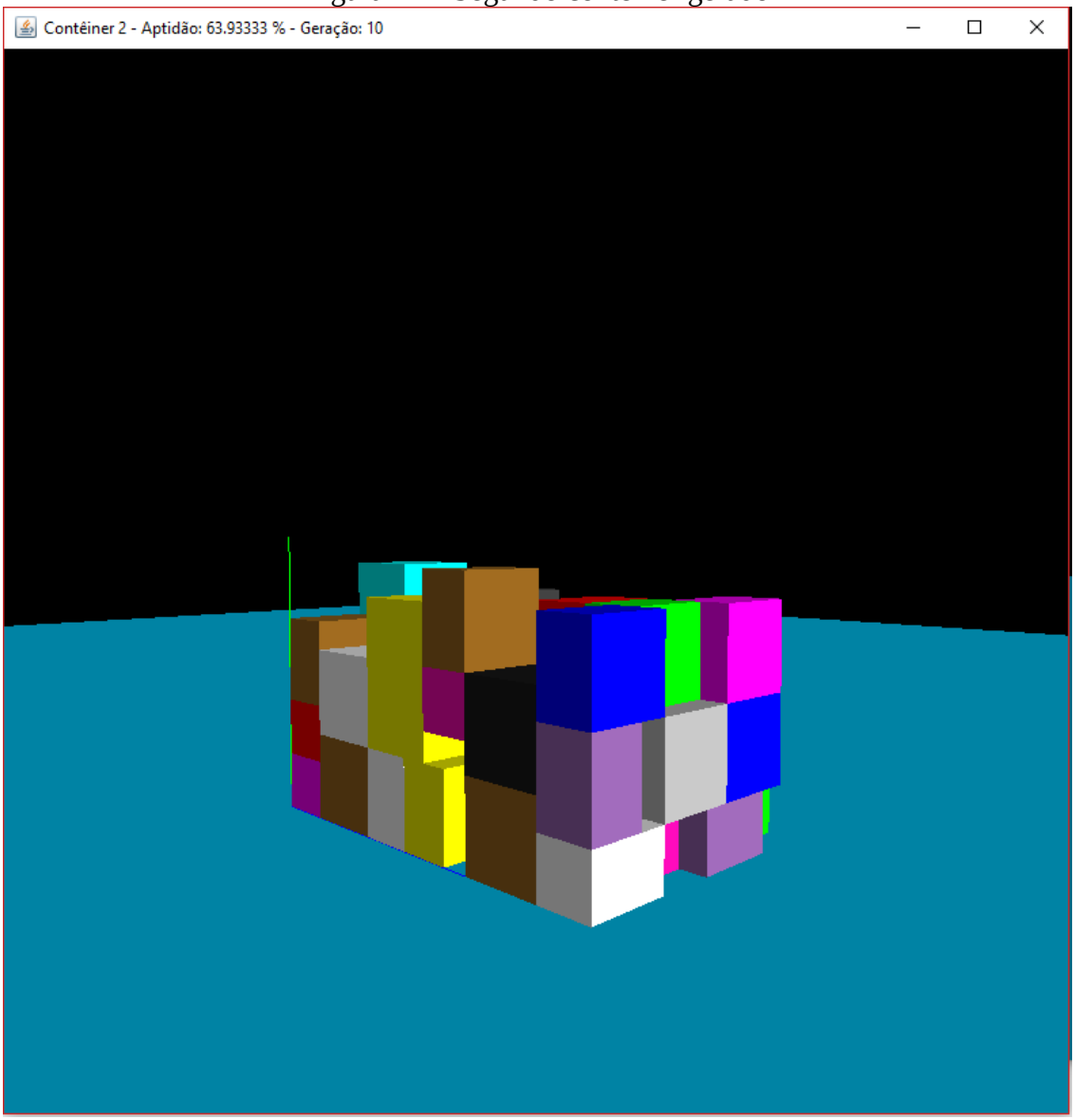
Figura 43 – Primeiro contêiner gerado



Fonte: elaborado pelo autor.

Na figura 44 é exibido a saída do segundo contêiner gerado pelo algoritmo. Este contêiner ficou com as demais caixas, tornando sua geração de atingimento bem mais rápida (geração 10) e todas as caixas foram alocadas quando sua aptidão chegou em 63,93% (aptidão máxima para este conjunto de caixas restante).

Figura 44 – Segundo contêiner gerado



Fonte: elaborado pelo autor.

Andando pelo cenário da figura 44, nota-se porém que o algoritmo não trata problemas de física. A figura 45 mostra a imagem rotacionada, destacando com um círculo vermelho uma caixa de cor roxa, onde foi alocada de uma forma que apenas uma pequena área dela ficou posicionada em cima de outra caixa (não gerando sobreposição). Esta saída gerada pelo algoritmo não seria aplicável no mundo real, pois esta caixa não iria manter-se reta nesta posição.

Figura 45 – Segundo contêiner gerado - rotacionado

Fonte: elaborado pelo autor.

3.4.7 Comparativo entre o trabalho desenvolvido aos correlatos

O quadro 17 apresenta a comparação entre os trabalhos correlatos apresentados e o proposto.

Quadro 17 – Trabalhos correlatos

Características	Gonçalves e Resende (2013)	Wang e Chen (2010)	Li, Zhao e Zhang (2014)	Trabalho desenvolvido
Possui interface gráfica	Sim	Não	Não	Sim
Permite parametrizações	Sim	Não	Sim	Sim
Solução 2D	Sim	Não	Não	Não

Solução 3D	Sim	Sim	Sim	Sim
Heurística	Construtiva	Comparativa	Best Match Heuristic Packing Strategy	First Match Packing Strategy

Fonte: Elaborado pelo autor.

Dos trabalhos correlatos os únicos que possibilitam visualização em interface gráfica são o trabalho desenvolvido e o trabalho de Gonçalves e Resende (2013), porém não permitindo ao usuário andar pelo ambiente, requisito atendido pelo trabalho desenvolvido. As parametrizações são suportadas pelos trabalhos de: Gonçalves e Resende (2013); Li, Zhao e Zhang (2014) e o desenvolvido. A solução do problema de empacotamento 2D é suportado apenas pelo trabalho de Gonçalves e Resende (2013). A solução em 3D, requisito principal do trabalho proposto, é atendida por todos os trabalhos correlatos. Em termos de heurística todos os trabalhos utilizam técnicas diferentes, sendo que todos alteram a forma literária de sua heurística, fazendo ajustes propostos pelo trabalho. A heurística que mais se assemelha ao trabalho desenvolvido é a heurística utilizada por Li, Zhao e Zhang (2014), sendo a principal diferença que o trabalho proposto as caixas são alocadas na primeira posição que cabem e no trabalho de Li, Zhao e Zhang (2014) as caixas são alocadas na melhor posição possível, determinada pela heurística desenvolvida por eles.

4 CONCLUSÕES

O objetivo deste trabalho foi desenvolver um software utilizando algoritmo genético para resolver o problema de empacotamento tridimensional heterogêneo e através dos resultados apresentados, pode se concluir que este objetivo foi alcançado. Com os resultados obtidos o software permite selecionar as dimensões dos contêineres e das caixas, realizando a alocação destas considerando o maior preenchimento dos contêineres e permitindo visualizar o resultado do empacotamento em uma interface 3D.

As ferramentas utilizadas se mostraram adequadas para o objetivo proposto. A representação 3D foi de fácil manipulação, pois foi utilizado a biblioteca `jogl` (JavaOpenGL) que facilitou bastante a implementação. O Java 8 também se mostrou útil, principalmente por sua fácil manipulação de listas, através dos facilitadores lambdas.

Durante o desenvolvimento do trabalho, as abordagens sofreram mudanças por diversas vezes até atingir o resultado final. Um dos pontos interessantes do trabalho foi a forma de mapeamento das caixas dentro de um contêiner, através de coordenadas `x y z`, que se mostraram indispensáveis para este tipo de solução.

A forma de desenvolvimento altamente orientada a objetos, com interfaces e códigos reaproveitáveis se mostrou essencial. Tendo em vista que um trecho específico de código pode ser alterado sem demandar refatorações, torna-se possível implementar mais de uma solução para determinada parte do algoritmo.

Uma parte do software que pode ser melhorado é a exibição 3D, onde as caixas podem ter uma coloração com um nível maior de transparência e o contêiner ser demarcado de uma maneira que facilite mais a visualização do usuário final. Outra melhoria seria implementar a rotação das caixas a cada geração, podendo ser efetuado a troca do eixo `x` com `y`, por exemplo.

O software pode ser incrementado para suportar outros tipos de contêineres, como por exemplo contêiner refrigerado; contêiner tanque; contêiner com prateleiras e etc. A principal contribuição deste trabalho foi proporcionar um algoritmo inteligente na alocação de caixas de variados tamanhos em múltiplos contêineres e exibi-los de uma maneira atrativa ao usuário final.

4.1 EXTENSÕES

Algumas das extensões possíveis para este trabalho são:

a) considerar mais restrições para o problema, como por exemplo fragilidade das

caixas, agendamento de entrega, outros tipos de contêineres, entre outros;

b) melhorar a exibição 3D demarcando melhor o contêiner e deixando a coloração das caixas transparente;
c) incrementar o método de mutação para possibilitar rotação de caixas, ou seja, troca do eixo x pelo eixo y;
d) migrar a visualização 3D para interface *web*;
e) tornar a execução do algoritmo `multi-thread` para melhorias de performance.

REFERÊNCIAS

CHEN, Yi et al. Chemical componentes determination via terahertz spectoscopic statistical analysis using microgenetic algorithm. **Society of Photo-Optical Instrumentation Engineers**, v. 50, n. 3, mar. 2011. Disponível em: < http://opticalengineering.spiedigitallibrary.org/article.aspx?articleid=1157793>. Acesso em: 22 jun.

FALKENAUER, Emanuel. A hybrid grouping genetic algorithm for bin packing. **Journal of Heuristics**, v. 2, n.1, dez. 1996. Disponível em: < https://link.springer.com/article/10.1007/BF00226291>. Acesso em: 12 set.

GONÇALVES, José F.; RESENDE, Mauricio G.C. A biased random key genetic algorithm for 2D and 3D bin packing problems. **International Journal of Production Economics**, v. 145, n. 2, set./out. 2013. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0925527313001837>. Acesso em: 12 set.

HARTMANN, Sönke. Packing problems and project scheduling models: na integrating perspective. **Journal of the Operational Research Society**, v. 51, n. 9, set. 2000. Disponível em: < http://link.springer.com/article/10.1057/palgrave.jors.2601011>. Acesso em: 05 nov.

KARMARKAR, Narendra; KARP, Richard M. An efficient approximation scheme for the one-dimensional bin-packing problem. **Proceedings of the 23rd Annual Symposium on Foundations of Computer Science**, p. 312-320, nov. 1982. Disponível em: < http://dl.acm.org/citation.cfm?id=1382768>. Acesso em: 22 jun.

LI, Xueping; ZHAO, Zhaoxia; ZHANG, Zhang. A genetic algorithm for the three-dimensional bin packing problem with heterogeneous bins. **Industrial and Systems Engineering Research Conference**, v. 1, mar. 2014. Disponível em: < https://www.researchgate.net/publication/273121476_A_genetic_algorithm_for_the_three-dimensional_bin_packing_problem_with_heterogeneous_bins>. Acesso em: 22 jun.

LINDEN, Ricardo. **Algoritmos Genéticos**. 2. ed. Rio de Janeiro: Brasport, 2008.

MARTELLO, Silvano; VIGO, Daniele. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. **Management Science**, v. 44, n. 3, mar. 1998. Disponível em: < http://pubsonline.informs.org/doi/abs/10.1287/mnsc.44.3.388>. Acesso em: 22 jun.

MIRANDA, Marcio N. Algoritmos Genéticos: Fundamentos e Aplicações. In: **Projeto Vida Social**, Rio de Janeiro: UFRJ, 2011.

MOURA, Benjamim. **Logística**: conceitos e tendências. Vila Nova de Famalicão: Centro Atlantico, 2006.

NOVAES, Antonio G. **Logística e Gerenciamento da Cadeia de Distribuição**. 4. Ed. Rio de Janeiro: Elsevier Editora Ltda, 2015.

PACHECO, André. **Computação Inteligente**: Máquinas aprendendo a solucionar problemas complexos. 2016 Disponível em: < http://www.computacaointeligente.com.br/algoritmos/o-algoritmo-genetico-ga/>. Acesso em: 22 jun. 2016.

PARK, Kyungchul et al. Modeling and solving the spatial block scheduling problem in a shipbuilding company. **Computers & Industrial Engineering**, v. 30, n. 3, jul. 1996. Disponível em: <http://www.sciencedirect.com/science/article/pii/0360835296000046>. Acesso em: 05 nov.

POLI, Riccardo; LANGDON, William B.; MCPHEE, Nicholas F. **A field guide to genetic programming**. Tennessee: Lightning Source, 2008.

RUSSEL, Stuart; NORVIG, Peter. **Inteligência Artificial:** Tradução da Terceira Edição. 3. Ed. Rio de Janeiro: Elsevier Editora Ltda, 2013.

SILVA, José L. C.; SOMA, Nei Y. Um algoritmo polinomial para o problema de empacotamento de contêineres com estabillidade estática da carga. **Pesquisa Operacional**, Rio de Janeiro, v. 23, n. 1, jan/abr. 2003. Disponível em: < http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382003000100007>. Acesso em: 22 jun.

TONETTO, Leandro M et al. O Papel das heurísticas no julgamento e na tomada de decisão sob incerteza. **Estudos de Psicologia**, Campinas, v. 23, n. 2, abr./jun. 2006. Disponível em: < http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0103-166X2006000200008>. Acesso em: 05 nov.

WANG, Hongfeng; CHEN, Yanjie. A hybrid genetic algorithm for 3D bin packing problems. **IEEE Fifth International Conference**, v. 1, set./nov. 2010. Disponível em: <http://ieeexplore.ieee.org/document/5645211/>. Acesso em: 12 set.

WÄSCHER, Gerhard; HAUBNER, Heike; SCHUMANN, Holger. An improved typology of cutting and packing problems. **European Journal of Operations Research**, v. 183, n. 3, set. 2007. Disponível em: <http://www.mansci.ovgu.de/mansci_media/publikationen/2007/typology-EGOTEC-5t0pvr6fjifln4r4oav60tt612.pdf >. Acesso em: 12 set.