JONATHAN RIBEIRO DOS SANTOS
SANDRO AUGUSTO DE OLIVEIRA

ALGORITMOS GENÉTICOS APLICADOS À LINHA DE PRODUÇÃO DE CALÇAS

UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG
2015

JONATHAN RIBEIRO DOS SANTOS
SANDRO AUGUSTO DE OLIVEIRA

ALGORITMOS GENÉTICOS APLICADOS À LINHA DE PRODUÇÃO DE CALÇAS

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação da Universidade do Vale do Sapucaí como requisito parcial para obtenção do título de bacharel de Sistemas de Informação.

Orientador: Prof. Me. Roberto Ribeiro Rocha

UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG
2015

Santos, Jonathan Ribeiro dos; Oliveira, Sandro Augusto de

Algoritmos genéticos aplicados à linha de produção de calças / Jonathan Ribeiro dos Santos, Sandro Augusto de Oliveira – Pouso Alegre – MG: Univás, 2015.
104 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade do Vale do Sapucaí, Univás, Sistemas de Informação.
Orientador: Prof. Me. Roberto Ribeiro Rocha

1. Algoritmos genéticos. 2. Inteligência Artificial. 3. Java.

JONATHAN RIBEIRO DOS SANTOS
SANDRO AUGUSTO DE OLIVEIRA

ALGORITMOS GENÉTICOS APLICADOS À LINHA DE PRODUÇÃO DE CALÇAS

Trabalho de conclusão de curso defendido e aprovado em 16/11/2015 pela banca examinadora constituída pelos professores:

Prof. Me. Roberto Ribeiro Rocha
Orientador

Prof. Artur Luis Ribas Barbosa
Avaliador

Prof. André Luiz Martins de Oliveira
Avaliador

De Jonathan Ribeiro dos Santos.

Dedico este trabalho aos meus pais pelo apoio e dedicação.

De Sandro Augusto de Oliveira.

Dedico este trabalho a Deus primeiramente pela força e pela luz dada a nós frente aos desafios ao longo deste ano. Aos meus queridos pais, Carlos Alberto de Oliveira e Maria Geralda de Oliveira pela compreensão, amor, dedicação e apoio em todas as etapas da minha vida, pelos conselhos e ensinamentos, não tenho palavras para agradecer a Deus por tê-los em minha vida. A uma pessoa especial em minha vida, Rebecca Emanuelle Leite Neri da Silva pela amizade, companheirismo, dedicação e carinho. Agradeço pelas palavras de incentivo e por me motivar a dar sempre o meu melhor, você fez toda diferença nesta conquista. Ao meu primo irmão e grande amigo Eduardo Oliveira e Silva que, infelizmente nos deixou em outubro deste ano, pela parceria e por sempre me ajudar quando precisei inclusive por ter me emprestado seu computador durante todo este ano para a realização deste trabalho, onde você estiver meu amigo, saiba que nunca te esqueceremos. A minha vó Maria Alves de Oliveira pelo apoio e incentivo e conselhos nos momentos difíceis. A todos meus tios e tias, primos e primas que me apoiaram durante todos estes anos e sempre torceram pelo meu sucesso. Aos meus grandes amigos Alex Jr., Janaína Lorena, Jonathan Ribeiro dos Santos, Andressa Faria, Edilson Justiniano e Roberta Ribeiro que sempre me apoiaram e me ajudaram durante esta caminhada. A todos que de forma direta ou indireta contribuíram para que eu chegasse até aqui, Deus vos abençoe.

AGRADECIMENTOS

De Jonathan Ribeiro dos Santos

Agradeço primeiramente a Deus por ter me dado saúde e força em todos os momentos da minha vida e por ter nos ajudado ao longo deste curso a para superar todas as dificuldades.

A minha família, em especial aos meus pais, que sempre me apoiaram nesses anos de estudos oferendo o suporte necessário para que eu conseguisse prosseguir com mais essa realização.

Agradeço também aos meus amigos e colegas que fizeram parte dessa trajetória, dividindo momentos de descontração, estudos, discussões, experiências e conquistas.

Ao professor orientador Roberto Ribeiro Rocha que nos auxiliou na elaboração deste trabalho demonstrando paciência e compreensão e dando todo suporte necessário.

Obrigado também a todos os professores e a coordenação do curso de Sistemas de Informação pela cooperação, compreensão, paciência e dedicação desde o primeiro semestre deste curso.

De Sandro Augusto de Oliveira

Agradeço a Deus em primeiro lugar pela saúde que tem me dado e pelo ânimo e por ter me ajudado em todos os momentos da minha vida.

Ao nosso orientador Roberto Ribeiro Rocha pela amizade, apoio e incentivo na execução deste trabalho, pelas correções minuciosas da parte técnica do trabalho escrito.

À professora Joelma Faria por seu empenho e dedicação, nos ajudando a sempre melhorar este trabalho.

Ao professor Artur Barbosa pelas valiosas contribuições na modelagem da solução implementada neste trabalho.

Ao professor André Martins por suas valiosas sugestões e conselhos não só referente ao presente trabalho mas também para a vida profissional.

Ao coordenador José Luiz pelo apoio, dedicação e orientação.

Ao meu amigo Jonathan Ribeiro, parceiro neste trabalho, pela parceria e ajuda.

Aos meus amigos Edilson Justiniano e Andressa Faria pela ajuda e suporte durante o desenvolvimento deste trabalho.

Aos meus familiares e amigos por todo amor e dedicação. Em especial aos meus pais por serem minha base em tudo.

À minha querida Rebecca Leite por ser uma pessoa tão incrível e por me incentivar a sempre buscar o meu melhor.

Ao meu primo e amigo de coração Eduardo Oliveira por ser uma pessoa sensacional e de bom coração.

Aos colegas de classe e demais professores que de alguma forma contribuíram para a realização deste sonho. Deus abençoe todos vocês!

"e não vos conformeis com este mundo, mas transformai-vos pela renovação da vossa mente, para que proveis qual é a boa, agradável e perfeita vontade de Deus.

(Romanos 12:2)

SANTOS, Jonathan Ribeiro dos; Oliveira, Sandro Augusto de. **Algoritmos genéticos aplicados à linha de produção de calças**. 2015. Monografia – Curso de SISTEMAS DE INFORMAÇÃO, Universidade do Vale do Sapucaí, Pouso Alegre – MG, 2015.

RESUMO

Os algoritmos genéticos, inspirados na genética e na teoria da evolução das espécies de Charles Darwin, são algoritmos probabilísticos que, aleatoriamente, promovem uma busca paralela pela melhor solução de um problema e são desenvolvidos baseados no princípio de sobrevivência dos mais aptos, na reprodução e na mutação dos indivíduos, de forma a fazer com que soluções evoluam e se tornem cada vez melhores. A presente pesquisa apresenta uma visão geral sobre o conceito de algoritmos genéticos, demonstrando seus fundamentos, operadores e suas configurações, realizando um comparativo com os conceitos semelhantes na natureza. Para ilustrar a utilização desta técnica, foi desenvolvida uma aplicação, em plataforma WEB, juntamente com um algoritmo genético, que visa encontrar soluções otimizadas para distribuição das atividades de um sistema de produção de calças de modo que permita aos empregados trabalharem em suas casas e que cada parte da calça seja confeccionada separadamente. O software recebe, como entrada, dados de tempo e preço de produção por peça de cada empregado e, após a execução do algoritmo genético sobre estes dados, apresenta como saída uma boa forma de distribuição das atividades que permitem que a produção seja realizada com o menor tempo e tenha o menor custo possível dentro do prazo de entrega. Esta pesquisa é do tipo aplicada, pois não visa modificar os processos de produção e sim apresentar formas de otimizá-los.

Palavras-chave: Algoritmos genéticos. Inteligência Artificial. Java. Produção.

SANTOS, Jonathan Ribeiro dos; Oliveira, Sandro Augusto de. **Algoritmos genéticos aplicados à linha de produção de calças**. 2015. Monografia – Curso de SISTEMAS DE INFORMAÇÃO, Universidade do Vale do Sapucaí, Pouso Alegre – MG, 2015.

ABSTRACT

The Genetic algorithms, inspired in genetic and in the Charles Darwin's theory of evolution of species, are probabilist algorithms which is used to, randomly, make a parallel search for the best solution for a given problem. It is implemented based on principle of survival of the fittest, on the reproduction and on mutation of individuals and so it makes possible the evolution of the solutions which can make them even better. This research presents a general view of genetic algorithms concept, it demonstrate its fundamentals, operators and configurations and also makes a comparison with similar concepts in nature. To illustrate how this technique works it was developed an WEB based application with a genetic algorithm which was developed to find optimized solutions for tasks distribution in a production system of pants in which the employees work in their own houses and each part of the pants is produced separately. The software receives as input the time and the price of production per part of each employee and, after execution of the genetic algorithm over this data, it shows one good form to distribute the tasks in order to make the production in the shortest time and in the shortest cost within the planned deadline. This kind of research is applied because it will not modify the production processes, it will just optimize them.

Key words: Genetic Algorithms. Artificial Intelligence. Java. Production.

LISTA DE FIGURAS

Figura 1 – Demonstração da execução de um AG 20
Figura 2 – Demonstração do método de seleção Roleta 22
Figura 3 – Demonstração do Modelo MVC 27
Figura 4 – Demonstração de um processo de fabricação de uma calça. 39
Figura 5 – Modelo do processo de fabricação no banco de dados. 41
Figura 6 – Classes `Atividade` e `AtividadeOrdem`. 42
Figura 7 – Fluxo básico de execução. 43
Figura 8 – Exemplo de distribuição aleatória de lotes para as costureiras. 46
Figura 9 – Classe `ProcessoChromosome`. 47
Figura 10 – Armazenamento de dados das costureiras. 49
Figura 11 – Classe `CostureiraHabilidade`. 49
Figura 12 – Classe `ProcessoIndividual`. 50
Figura 13 – Distribuição em porcentagem. 51
Figura 14 – Demonstração de costureiras e habilidades. 55
Figura 15 – Estrutura de representação da ordem de precedência. 57
Figura 16 – Exemplo de solicitação de lotes predecessores. 63
Figura 17 – Distribuição demonstrando o tempo. 64
Figura 18 – Representação do processo de seleção dos indivíduos. 70
Figura 19 – Cruzamento dos indivíduos. 72
Figura 20 – Exemplo de mutação. 75
Figura 21 – Menu Distribuir tarefas. 78
Figura 22 – Tela de distribuição de tarefas. 79
Figura 23 – Tela de resultado de distribuição de tarefas. 83
Figura 24 – Criação de um processo. 85
Figura 25 – Detalhes do processo cadastrado. 86
Figura 26 – Inserindo costureiras na habilidade Finalização. 86
Figura 27 – Tela de distribuição de tarefas. 87
Figura 28 – Resultado da distribuição de lotes considerando o tempo de produção. 87
Figura 29 – Alteração no tempo de produção entre as costureiras. 88
Figura 30 – Resultado da distribuição de lotes após a alteração do tempo de produção entre as costureiras. 88

Figura 31 – Custo entre as costureiras da atividade Finalização. 89
Figura 32 – Resultado da distribuição de lotes considerando o custo. 89
Figura 33 – Resultado da distribuição de lotes com configurações iguais entre as costureiras. 90
Figura 34 – Configuração das costureiras da atividade Finalização. 91
Figura 35 – Resultado da distribuição de lotes com prazo longo. 91
Figura 36 – Resultado da distribuição de lotes com prazo reduzido. 92
Figura 37 – Tela de cadastro de costureiras. 92
Figura 38 – Configuração das costureiras da habilidade Finalização. 93
Figura 39 – Resultado da distribuição de lotes considerando o tempo de transporte. 93
Figura 40 – Distribuição das atividades. 94
Figura 41 – Adição de uma atividade ao processo. 95
Figura 42 – Adição de uma atividade ao fluxo do processo. 95
Figura 43 – Adição de costureiras na habilidade Frente. 96
Figura 44 – Resultado da distribuição de lotes com mais uma atividade. 96
Figura 45 – Distribuição das atividades. 97
Figura 46 – Resultado da distribuição de lotes após executar novamente a distribuição. 97
Figura 47 – Distribuição das atividades. 98
Figura 48 – Configuração das costureiras da atividade Finalização. 98
Figura 49 – Resultado da distribuição de lotes com prazo não atendido. 99
Figura 50 – Alteração no custo das costureiras da habilidade Finalização. 99
Figura 51 – Resultado da distribuição após alteração no custo da costureira Duda. . 100

LISTA DE CÓDIGOS

Código 1 **Método** `iniciarDistribuicao()` **da classe** `GeneticAlgorithmManagement` 43
Código 2 **Método** `getInformacoesCostureiras()` 44
Código 3 **Método** `createInitialPopulation()` 45
Código 4 **Criação de cromossomos (Primeira Abordagem)** 50
Código 5 **Distribuição em lotes diretamente.** 53
Código 6 **Método** `calculateValue()` 57
Código 7 **Construtor da classe** `Node` 58
Código 8 **Método** `getValorTotal()` 59
Código 9 **Método** `getCromossomeValue()`. 60
Código 10 **Método** `getTempoRecebimentoPecas()`. 60
Código 11 **Método** `getValueChromosomosPredecessores()`. 61
Código 12 **Método** `calcularTempoEntreCostureiras()`. 63
Código 13 **Método** `classify()` 65
Código 14 **Método** `compareTo()` 66
Código 15 **Método** `execute()` 67
Código 16 **Método** `doSelection()` 69
Código 17 **Método** `doSelectionByClassification()` 69
Código 18 **Método** `doCrossing()` 72
Código 19 **Método** `doPermutationCrossing()` 72
Código 20 **Método** `createIndividual()` **utilzando cromossomos** 74
Código 21 **Construtor de** `ProcessoIndividual` **utilizando cromossomos** 74
Código 22 **Método** `doMutation()` 75
Código 23 **Método** `doMutationPermutation()` 76
Código 24 **Método** `doMutationOnChromossome()` 77
Código 25 **Documento XHTML da tela de distribuição** 79
Código 26 **Método** `iniciarDistribuicao()` 80
Código 27 **Método** `calcularPrazo()` 82

SUMÁRIO

INTRODUÇÃO **14**

2 QUADRO TEÓRICO **17**
2.1 Algoritmos Genéticos **17**
2.1.1 Fundamentos **17**
2.1.2 Características dos Algoritmos Genéticos **19**
2.2 Tecnologias **24**
2.2.1 Linguagem de programação Java **24**
2.2.2 Interface Gráfica - JSF **27**
2.2.3 Armazenamento de dados **29**

3 QUADRO METODOLÓGICO **31**
3.1 Tipo de pesquisa **31**
3.2 Contexto de pesquisa **32**
3.3 Instrumentos **32**
3.3.1 Entrevistas **33**
3.3.2 Reuniões **33**
3.4 Procedimentos **35**
3.4.1 Framework de desenvolvimento **35**
3.4.2 Representação do processo de produção **39**
3.4.3 População inicial: Distribuição das atividades, Indivíduos e Cromossomos **42**
3.4.4 Função de avaliação **55**
3.4.5 Classificação dos indivíduos **65**
3.4.6 Indivíduos estrangeiros e elitismo **67**
3.4.7 Seleção de indivíduos e cruzamento/reprodução **69**
3.4.8 Mutação **75**
3.4.9 Interface gráfica de distribuição **78**

4 DISCUSSÃO DE RESULTADOS **84**
4.1 Teste considerando somente o tempo de produção **85**
4.2 Teste considerando o custo de produção **88**
4.3 Teste considerando tempo x custo x prazo de entrega **90**
4.4 Teste considerando o tempo de transporte **92**
4.5 Teste adicionando mais atividades ao processo **94**
4.6 Teste quando o prazo não é atendido **98**

5 CONCLUSÃO **101**

REFERÊNCIAS **104**

INTRODUÇÃO

Sabe-se que atualmente existem diversos softwares disponíveis no mercado para facilitar o dia a dia nas empresas. Programas que vão desde aqueles desenvolvidos sob medida até softwares genéricos popularmente chamados de "softwares de prateleira". Porém, no cenário corporativo, levando em consideração que empresas buscam constantemente se tornarem mais competitivas, é necessário que um sistema ofereça suporte para que essas possam se tornar mais eficientes, como por exemplo, auxiliar na busca pela diminuição dos custos operacionais, para que assim seja possível alcançar os melhores preços de venda (LAUDON; LAUDON, 2009).

Neste contexto, a ideia de agregar características semelhantes à inteligência humana aos programas se torna uma alternativa interessante, pois segundo Laudon e Laudon (2009, p.329),

> Técnicas inteligentes ajudam os tomadores de decisão capturando o conhecimento coletivo e individual, descobrindo padrões e comportamentos em grande quantidade de dados e gerando soluções para problemas amplos e complexos demais para serem resolvidos por seres humanos.

O conceito por trás deste pensamento é denominado Inteligência Artificial - IA[1] - e é definido por Luger e Stubblefield (1993) como uma área da ciência da computação que abrange a automatização da inteligência.

Para Luque e Silva (2010, p.44), "a IA é inspirada em processos naturais e está relacionada à reprodução de capacidades normalmente associadas à inteligência humana, como aprendizagem, adaptação, raciocínio, entre outras". Ainda segundo os mesmos autores, várias abordagens surgiram ao longo da história, tais como a abordagem conexionista, inspirada nos neurônios biológicos; a simbolista, baseada na inferência humana e a evolutiva, fundamentada na teoria de evolução das espécies.

Na busca por otimizar seu processo de produção, uma empresa da região, que fabrica calças e aloca costureiras que trabalham em suas casas, deseja saber qual a melhor forma de distribuir o trabalho para que um determinado lote de seu produto seja produzido dentro de um prazo com o menor custo possível. Para a resolução desse problema, pode-se fazer uso de IA através de um dos ramos da abordagem evolutiva denominado Algoritmos Genéticos - AGs[2] - pois, como afirma Fernandes (2003), os AGs resolvem problemas de

otimização através de um processo que oferece como saída a melhor solução dentro de várias possíveis formas de se resolver um problema.

O presente trabalho tem por objetivo geral desenvolver uma aplicação utilizando técnicas de inteligência artificial capaz de realizar a alocação de empregados em uma linha de produção de forma otimizada.

Para a realização dessa pesquisa, foram colocados os seguintes objetivos específicos: a) Demonstrar o uso de algoritmos genéticos; b) Projetar uma aplicação em plataforma WEB que distribua as atividades de uma linha de produção de calças de forma inteligente, a fim de se obter um tempo de produção dentro de um prazo estipulado, procurando obter o menor custo.

Para Linden (2012), AGs é definido como uma técnica de otimização e busca que se baseia na teoria do processo de evolução e seleção natural, proposto por Charles Darwin em seu livro A Origem das Espécies, que afirma que indivíduos com melhor capacidade de adaptação ao seu ambiente possuem maior chance de sobreviver e gerar descendentes.

Segundo Fernandes (2003), o termo foi proposto por Holland, em 1975, e por imitação à teoria da evolução, é representado por uma população de indivíduos que representam soluções para um determinado problema e então tais soluções podem evoluir até se chegar a uma solução ótima.

Vários trabalhos foram desenvolvidos utilizando a robustez de AGs, entre eles o trabalho de Santos et al. (2007), que faz a seleção de atributos usando AGs para classificação de regiões; o trabalho de Silva (2001), que descreve a otimização de estruturas de concreto armado utilizando AGs e o trabalho de Freitas et al. (2007), que descreve uma ferramenta baseada em AGs para a geração de tabela de horário escolar.

O sistema desenvolvido neste trabalho, assim como nos outros citados, tem o mesmo conceito, o de otimizar processos. No caso da fábrica de calças, tal otimização irá promover a diminuição do custo de produção das calças confeccionadas, permitindo então que essas possam ser vendidas também por um preço melhor, beneficiando assim, os consumidores da região. Além disso, a ideia de otimizar procedimentos, muitas vezes, também contribui para a preservação de recursos naturais devido ao fato de que processos otimizados podem significar economia de energia e diminuição de emissão de gases. No caso da empresa de calças, se o gestor tiver sempre em mãos a solução mais otimizada, terá consequentemente um menor tempo de transporte de materiais, o que reduzirá

[1] O termo Inteligência Artificial será referenciado pela sigla IA a partir deste ponto do trabalho.
[2] O termo Algoritmos Genéticos será referenciado pela sigla AGs a partir deste ponto do trabalho.

a emissão de gases dos veículos utilizados no transporte de materiais e peças entre as costureiras.

No âmbito acadêmico, o trabalho agregará à base de conhecimento da Univás um material que faça referência a tecnologias e conceitos de inteligência artificial, hoje presentes em diversos sistemas críticos de apoio à decisão e otimização de processos nas empresas.

2 QUADRO TEÓRICO

Neste capítulo, serão listados os conceitos e as tecnologias utilizados no desenvolvimento deste trabalho. Para tal, será apresentada a definição, o histórico e as aplicabilidades de cada um deles, tomando por base autores fundantes e seus comentaristas. É importante ressaltar que o texto desta seção, quando descreve a teoria da evolução das espécies, não tem como objetivo levantar questões sobre a origem dos seres vivos.

2.1 Algoritmos Genéticos

Nesta seção será descrito como surgiu, conceitos e algumas características dos Algoritmos Genéticos, tema principal deste trabalho e fundamental para o desenvolvimento da aplicação.

2.1.1 Fundamentos

Para Melanie (1999), desde o começo da era computacional, cientistas pioneiros, tais como Alan Turing, John von Neumann, Norbert Wiener e outros, tinham o objetivo de dotar os computadores de inteligência de maneira que eles pudessem tomar decisões, se adaptar a determinadas situações e até mesmo ter a capacidade de aprender. Com esta motivação, estes cientistas se interessaram por outras áreas, além da eletrônica, como a biologia e a psicologia, e começaram então a realizar pesquisas para simular os sistemas naturais no mundo computacional a fim de alcançarem suas metas.

Vários conceitos computacionais baseados na natureza surgiram então ao longo do tempo, entre eles, a computação evolucionária inspirada na teoria da evolução natural, da qual o exemplo mais proeminente são os AGs que foram introduzidos por Jhon Holland, seu aluno David Goldberg e outros estudantes da universidade de Michigan. Goldberg (1989) define os AGs como métodos de busca baseados na genética e no mecanismo de seleção natural que permitem a possibilidade de obter robustez e eficácia na tarefa de encontrar uma boa solução para um problema em um espaço de busca complexo, em um tempo aceitável.

Segundo Linden (2012), a teoria da evolução foi proposta pelo naturalista inglês Charles Darwin por volta de 1850, quando este, em uma viagem de navio, visitou vários lugares e, por ser uma pessoa com uma grande habilidade de observação, percebeu que indivíduos de uma mesma espécie, vivendo em lugares diferentes, possuíam também características distintas, ou seja, cada indivíduo possuía atributos específicos que lhe permitia uma melhor adaptação em seu ecossistema.

O autor afirma então que, com base nesta observação, Darwin propôs a existência de um processo de seleção natural, afirmando que, como os recursos na natureza, tais como água e comida, são limitados, os indivíduos competem entre si e aqueles que não possuem atributos necessários à adaptação ao seu ambiente tendem a ter uma probabilidade menor de reprodução e irão ser extintos ao longo do tempo e, por outro lado, aqueles com características que os permitem obter vantagens competitivas no meio onde vivem, acabam tendo mais chances de sobreviver e gerar indivíduos ainda mais adaptados.

A teoria ressalta porém que o processo não tem o objetivo de maximizar algumas características das espécies, pois os novos indivíduos possuem atributos que são resultados da mesclagem das características dos reprodutores, o que faz com que os filhos não sejam exatamente iguais aos pais, podendo assim ser superiores, uma vez que estes herdem as qualidades de seus pais, ou inferiores, se os descendentes herdarem as partes ruins de seus reprodutores (LINDEN, 2012).

Para entender a relação entre AGs e a evolução natural é necessário conhecer as principais terminologias biológicas, sendo importante ressaltar porém que, de acordo com Melanie (1999), apesar da analogia a certos termos da biologia, a forma com que os AGs são implementados é relativamente simples se comparado ao funcionamento biológico real.

Melanie (1999) afirma que todos os seres vivos são compostos de células e estas possuem um ou mais cromossomos que, basicamente, são manuais de instruções que definem as características do organismo. O cromossomo é formado por um conjunto de genes que, em grupo ou individualmente, são responsáveis por um determinado atributo do indivíduo como por exemplo, a cor do cabelo, a altura, etc. Cada gene possui uma localização dentro do cromossomo denominada *locus* e, por fim, o conjunto de todos os cromossomos dentro da célula é definido como genoma.

Considerando isto, Linden (2012, p.33) afirma que,

> Um conjunto específico de genes no genoma é chamado de genótipo. O genótipo é a base do fenótipo, que é a expressão das características

> físicas e mentais codificadas pelos genes e modificadas pelo ambiente, tais como cor dos olhos, inteligência, etc. Daí, podemos concluir: nosso DNA codifica toda a informação necessária para nos descrever, mas esta informação está sob controle de uma grande rede de regulação gênica que, associada às condições ambientais, gera as proteínas na quantidade certa, que farão de nós tudo aquilo que efetivamente somos.

Uma vez descrita a complexidade dos organismos é necessário discorrer, de forma básica, sobre o processo de reprodução responsável pela transmissão da informação genética de geração para geração.

Linden (2012) afirma que existem dois tipos de reprodução, a assexuada, em que não é necessária a presença de um parceiro e a sexuada, que exige a presença de dois organismos. Os AGs simulam a reprodução sexuada, em que cada um dos organismos envolvidos oferece um material genético denominado gametas. Os gametas são formados por um processo denominado *crossing-over* ou *crossover*, que tem início com a divisão de cada cromossomo em duas partes as quais irão se cruzar uma com a outra para formar dois novos cromossomos, que receberão um pedaço de cada uma das partes envolvidas no cruzamento.

Ainda segundo o autor, o resultado deste processo serão, então, quatro cromossomos potencialmente diferentes que irão compor os gametas e farão parte dos novos indivíduos. Neste processo, podem ocorrer mutações que são resultados de alguns erros ou da influência de algum fator externo, como a radiação, por exemplo. Estas mutações são pequenas mudanças nos genes dos indivíduos, podendo ser boas, ruins ou neutras.

E assim a informação genética é passada dos pais para os filhos, e como os componentes dos cromossomos definem as características do organismo, os filhos herdarão características dos pais, porém serão ligeiramente diferente deles, como foi descrito anteriormente, o que permite que os novos indivíduos herdem características melhores ou piores que seus progenitores, porém, se os pais possuem características positivas, a probabilidade de gerarem filhos ainda melhores são maiores (LINDEN, 2012).

2.1.2 Características dos Algoritmos Genéticos

De acordo com Linden (2012), a analogia dos AGs com os processos biológicos se dá por meio da representação de cada termo descrito anteriormente em um modelo computacional voltado a encontrar soluções para um determinado problema em um processo aleatório. O fluxo de execução deste processo inicia-se com a criação aleatória de uma

população inicial. Uma população contém um conjunto de indivíduos sendo que cada indivíduo representa uma possível solução para o problema.

O autor afirma ainda que um indivíduo é formado por cromossomos que guardam as características da solução, ou seja, uma forma de resolver o problema.

Segundo Melanie (1999), a execução de um algoritmo genético é basicamente realizada conforme os itens a seguir:

1. Definição da população inicial;
2. Avaliação e classificação dos indivíduos da população;
3. Seleção de acordo com a qualidade;
4. Cruzamento para geração de novos descendentes;
5. Mutação aleatória dos novos indivíduos;
6. Repetição do processo de seleção, cruzamento e mutação até se formar uma nova população;
7. Avaliação e classificação dos indivíduos da nova população;
8. A nova população substitui a anterior e o processo continua a partir do item 2 até que o número de populações criadas (gerações) atinja um limite, que é definido previamente, ou até atingir outra condição definida pelo programador.

A Figura 1 ilustra os passos do algoritmo.

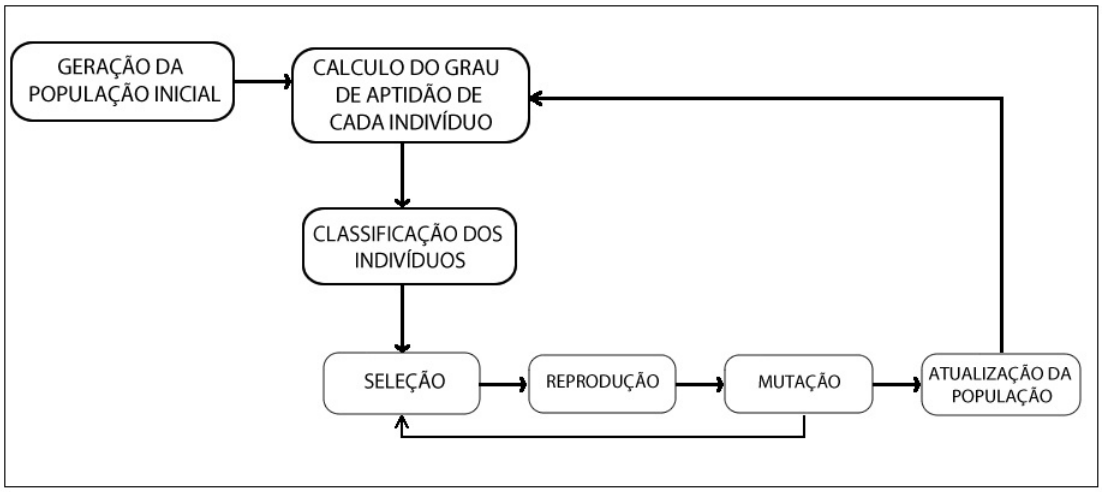


Figura 1 – Demonstração da execução de um AG **Fonte:** Desenvolvido pelos autores.

Como demonstrado acima, existem dois `loops`, o `loop` que acontece até que uma nova população seja formada e outro responsável por iniciar a criação de uma nova população até que o número máximo de gerações seja atingido ou até que outra condição definida seja atingida.

2.1.2.1 População Inicial

De acordo com Melanie (1999), a população inicial é o conjunto dos primeiros indivíduos candidatos à resolução do problema. Estes indivíduos devem ser criados aleatoriamente, seguindo a lógica definida pelo programador com base no contexto do problema a ser resolvido. Um exemplo seria a população inicial do algoritmo desenvolvido neste trabalho, em que a população inicial será formada por possíveis formas de se produzir um determinado lote de calças, ou seja, será formada uma população de possíveis formas de dividir o trabalho entre as costureiras.

2.1.2.2 Função de avaliação

Segundo Linden (2012), após a criação da população inicial, essa é então avaliada através de uma função de avaliação que mede a qualidade de cada uma de suas soluções e em seguida é realizada uma classificação que ordena as soluções das melhores para as piores, para assim iniciar a formação de uma nova população. A nova população pode conter, já inicialmente, os dois melhores indivíduos existentes na população inicial, este mecanismo é denominado elitismo e pode ser utilizado ou não.

Ainda segundo o autor, a função de avaliação ou função de aptidão penaliza as soluções inviáveis para a solução do problema, ou seja, ao verificar que uma certa solução não satisfaz o grau de aptidão necessário para o problema proposto, esta solução é descartada e, assim, só sobrevivem as soluções com mais chance de resolver o problema e por isso torna-se o componente mais importante de qualquer algoritmo genético.

Devido à generalidade encontrada nos AGs, a função de avaliação torna-se em muitos casos a única ligação verdadeira entre o programa e o problema real, pois ela irá analisar os cromossomos de cada indivíduo, convertendo-os, assim, em uma proposta de solução para o problema (LINDEN, 2012).

2.1.2.3 Seleção

Segundo Linden (2012), o próximo passo para a formação da nova população é o cruzamento, que se inicia com a seleção de dois indivíduos, realizada de acordo com a qualidade deles, porém, como é fundamental que esta escolha não despreze completamente os indivíduos com uma qualidade muito baixa, a seleção é feita de forma probabilística, ou seja, indivíduos com boa qualidade possuem mais chances de serem selecionados e, por outro lado, indivíduos com menor nota de avaliação terão menos chance se reproduzirem.

O autor ressalta que a razão do mecanismo de seleção não escolher apenas as melhores soluções é devido ao fato de aquelas com menor grau de avaliação também serem importantes para que se tenha uma maior diversidade de características na população envolvida para a solução do problema, possibilitando assim que essa possa evoluir de forma satisfatória, pois se as novas populações forem constituídas somente das melhores soluções, elas serão compostas de indivíduos cada vez mais semelhantes, impedindo, assim, que novas soluções ainda melhores sejam concebidas.

Existem vários tipos de seleção. Um dos métodos utilizados para este fim é o método roleta, muito empregado entre a maioria dos pesquisadores de AGs. Metaforicamente, cada indivíduo da população ocupa uma porção da roleta, proporcional ao seu índice de aptidão, ou seja, os indivíduos que possuírem maior aptidão ocuparão uma maior porção do que aqueles com menor aptidão. A roleta, então, assim como mostra a Figura 2, é girada e o indivíduo escolhido é aquele em que a seta parar sobre ele (FILITTO, 2008).

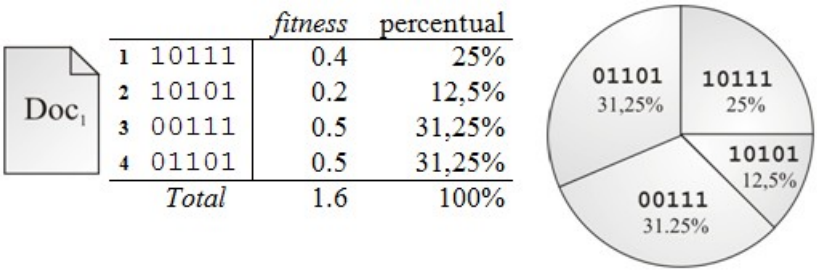


Figura 2 – Demonstração do método de seleção Roleta **Fonte:** "http://www.dgz.org.br/fev09/Art04.htm".

2.1.2.4 Cruzamento

Segundo Linden (2012), após a seleção dos pais, irá ocorrer então o processo de cruzamento ou *crossover* e, como no cruzamento natural, nesse processo dois novos indivíduos, isto é, duas novas soluções são formadas a partir de características daquelas que se cruzaram, ou seja, serão geradas duas novas soluções que conterão alguns cromossomos de uma solução e alguns cromossomos de outra.

Para Lacerda e Carvalho (2015), o operador *crossover* é um mecanismo de busca dos AGs para explorar regiões desconhecidas do espaço de busca. Ele é aplicado a cada par de cromossomos dos indivíduos que passaram pelo processo de seleção, gerando assim, dois novos indivíduos.

Para Filitto (2008), o cruzamento combina os cromossomos pais a fim de gerar cromossomos filhos e para isso existem vários tipos de cruzamento.

Um dos cruzamentos muito utilizados é o cruzamento de um ponto, que divide a lista de cromossomos selecionados em um ponto de sua cadeia, esse ponto é escolhido aleatoriamente. Após essa divisão, é copiada uma parte dos cromossomos de cada pai para definir os cromossomos dos indivíduos filhos. Neste método, é comum os pais gerarem dois novos filhos (FILITTO, 2008).

Um outro cruzamento muito utilizado é o cruzamento uniforme, que ainda segundo o autor, gera cada gene do descendente copiando o gene em questão de um dos pais, em que se usa uma máscara de cruzamento que é gerada aleatoriamente para fazer a escolha. Para criar cada cromossomo do novo indivíduo é feita uma iteração em todas as posições da máscara fazendo uma análise dos seus valores; quando o valor da posição for 1, o gene do primeiro pai referente à mesma posição da máscara é copiado; se o valor for 0, será copiado o gene do segundo pai, depois desse processo é gerado o novo descendente.

Segundo Lacerda e Carvalho (2015), o tipo de cruzamento uniforme difere do cruzamento de um ponto, uma vez que este sempre leva à metade dos *bits* de cada pai.

2.1.2.5 Mutação

Para Linden (2012), também pode ocorrer a mutação em que, como ocorre na natureza, aleatoriamente o valor dos cromossomos de um indivíduo pode ser alterado. A

mutação ocorre de acordo com uma taxa definida. Basicamente é definida uma porcentagem baixa e então um número de 0 a 1 é sorteado e multiplicado por 100, se o resultado for menor que a porcentagem definida, irá ocorrer a mutação para aquele indivíduo.

Para Lacerda e Carvalho (2015), a mutação melhora a diversidade dos cromossomos na população. Em contrapartida, depois de realizada a mutação, se perdem as informações contidas no cromossomo. Assim, para assegurar a diversidade, deve-se usar uma taxa de mutação pequena.

Assim como no cruzamento, há vários tipos de mutação, entre eles a mutação de *bit*, que é um tipo de mutação mais simples de ser implementada.

Segundo Filitto (2008), este é o operador mais fácil de trabalhar, podendo ser aplicado em qualquer forma de representação binária dos cromossomos. Este tipo de mutação gera uma probabilidade de mutação para cada *bit* do cromossomo, caso a probabilidade sorteada estiver dentro da taxa de mutação definida, o *bit* sofrerá mutação, recebendo um valor determinado de forma aleatória entre os valores que podem ser assumidos pelo cromossomo.

No problema a ser solucionado na aplicação deste trabalho, há um cenário em que existem diversas soluções para se resolver o problema e deseja-se encontrar a melhor entre elas. Conforme descrito anteriormente, os AGs são uma das melhores opções para se resolver este tipo de problema, por este motivo esta técnica foi escolhida.

2.2 Tecnologias

Abaixo serão listadas as tecnologias que serão utilizadas no desenvolvimento do projeto.

2.2.1 Linguagem de programação Java

Segundo Oracle (2015b), a Linguagem Java foi projetada para permitir o desenvolvimento de aplicações seguras, portáteis e de alto desempenho para a mais ampla gama de plataformas de computação.

De acordo com Schildt (2007), o Java foi criado em 1991 pela *Sun Microsystems* e foi baseado em uma linguagem já existente, o C++, que foi escolhida por ser orientada a

objetos e por gerar códigos compactados, o que era exatamente o que eles precisavam para implantar em pequenos aparelhos. Além dessas características, um requisito desejável era que a nova linguagem fosse independente de plataforma, para que fosse executado em qualquer arquitetura, tais como TVs, telefones, entre outros, e então, para atender esta exigência, foi criado o conceito de máquina virtual, que ficou conhecido como *Java Virtual Machine* - JVM[2].

O ponto chave que permite ao Java resolver o problema de portabilidade é o fato de o código ser compilado em *Bytecode*, que é um conjunto genérico de instruções altamente otimizado que é executado pela JVM, e essa, por sua vez, traduz o mesmo para a arquitetura na qual ela está instalada, o que possibilita a execução do programa em várias plataformas (SCHILDT, 2007).

Além da portabilidade, o Java também é *multithread*, ou seja, permite a execução de múltiplas tarefas simultaneamente. A linguagem também conta com um *automatic garbage collector* que consiste em um mecanismo de gerenciamento e limpeza de memória que a JVM acomoda. Além disso, o Java suporta uma extensa biblioteca de rotinas que facilitam a interação com protocolos TCP/IP, como HTTP e FTP (SCHILDT, 2007).

De acordo com Junior (2007),

> Atualmente a linguagem está organizada em três segmentos principais:
>
> - JavaMe (*Java Micro Edition*) - Destinado a pequenos dispositivos computacionais móveis, tais como celulares, PDAs e *set-top boxes*. É composto de máquina virtual otimizada para ambientes mais restritos, com especificações de funcionalidades e uma API mais compacta;
>
> - JavaSE (*Java Standard Edition*) - Integra os elementos padrão da plataforma e permite o desenvolvimento de aplicações de pequeno e médio porte. Inclui todas as APIs consideradas de base, além da máquina virtual padrão;
>
> - JavaEE (*Java Enterprise Edition*) - Voltada para o desenvolvimento de aplicações corporativas complexas. Adiciona APIs específicas aos elementos padrão da plataforma.

Para Santos (2003), Java é uma linguagem Orientada a Objetos (OO), este paradigma usa objetos, criados a partir de modelos, também chamados de classes, para representar e processar dados em aplicações computacionais. Basicamente, uma classe ou modelo é um conjunto de especificações de um objeto, ou seja, que tipo de dados esse deve ter e quais operações ele terá e então, após a criação da classe, o programador pode criar um objeto dessa. Uma analogia simples seria a planta de uma casa e sua constru-

[2] O termo Java Virtual Machine será referenciado pela sigla JVM a partir deste ponto do trabalho.

ção, a planta representa a classe, ou seja, como a casa deve ser feita, e a casa, depois de construída, representa o objeto.

Os dados e operações são guardados em objetos e estes são os elementos centrais do programa. Assim, a aplicação como um todo é vista como uma coletânea de objetos que se relacionam uns com os outros. Cada objeto representa um conceito real de uma parte do problema. Isso permite que o desenvolvimento se torne menos complexo, pois os conceitos são familiares às pessoas envolvidas no projeto pelo fato de que a aplicação não está organizada em processos estruturados, mas sim em objetos que espelham o mundo real e interagem entre si (MANZONI, 2005).

Para Santos (2003), os dados pertencentes aos modelos são representados por tipos nativos, característicos das linguagem de programação e podem também ser representados por outros modelos criados pelo programador.

Para o desenvolvimento do sistema de informação deste trabalho, o paradigma de orientação a objetos, que será implementado na linguagem Java, será imprescindível devido à alta complexidade do problema a ser resolvido, pois cada objeto representará partes do algoritmo genético que será desenvolvido, tais como a representação do Indivíduo, Cromossomos, etc., o que facilitará muito o desenvolvimento.

O produto resultante deste trabalho faz uso do Java EE e, como já foi dito em seções anteriores, será implementado em plataforma WEB. Para tal, faz-se necessário o uso de um servidor de aplicações WEB. Eentre as opções disponíveis, foi escolhido o *Tomcat*, pelo fato de ser o mais simples e ter os atributos mínimos necessários para a aplicação que será desenvolvida.

Segundo Vukotic e Goodwill (2011), o *Tomcat* é um servidor *open source* e *container* de aplicações *web* baseados em Java. Ele foi criado para executar *servlets*, que também é uma denominação para classe dentro da especificação do Java para WEB, e arquivos de marcação que definem os elementos visuais da tela. O *Tomcat* foi criado como um subprojeto da *Apache-Jakarta*, porém, como ficou muito popular entre os desenvolvedores, a *Apache* o denominou como um projeto separado e vem sendo melhorado e apoiado por um grupo de voluntários da comunidade *Java Open Source*, que faz dele uma excelente solução para desenvolvimento de uma aplicação *web* completa.

2.2.2 Interface Gráfica - JSF

Para a construção da interface gráfica (páginas web) da aplicação desenvolvida neste trabalho, foi utilizado um *framework* nativo nas novas versões do Java, denominado *Java Server Faces* (JSF).

Bergsten (2004) afirma que o JSF é um *framework server-side* baseado em componentes *web*, cuja principal função é abstrair os detalhes de manipulação dos eventos e organização dos componentes na página *web*. Por meio dele, é possível desenvolver páginas mais sofisticadas de forma simples, abstraindo, inclusive, o tratamento de requisições e respostas. Isto permite ao desenvolvedor focar-se no *back-end* da aplicação, ou seja, na lógica, e não se preocupar com detalhes a respeito de requisições e respostas HTTP e como obter as informações recebidas e/ou enviadas através deste protocolo.

De acordo com Oracle (2015a), o JSF é de fácil aprendizado e utilização, pois possui sua arquitetura claramente definida, sendo dividida entre a lógica da aplicação e apresentação. Essa divisão é possível pois ele utiliza o padrão de projeto *Model-View-Controller* - MVC[3], tornando-o um importante *framework* para desenvolvimento de aplicações utilizando a plataforma Java *Web*.

Segundo Gamma et al. (2009), o padrão de projeto MVC é dividido em três partes. O *Model* é a lógica e acesso aos dados da aplicação, a *View* é camada de apresentação e, por último, o *Controller* é responsável por definir a interface entre a lógica e a apresentação. Portanto, todo tipo de requisição ou resposta deve ser obrigatoriamente enviada ao *Controller* que, por sua vez, encaminhará para a camada de visão ou de lógica. A Figura 3 demonstra um exemplo do modelo MVC utilizando o JSF.

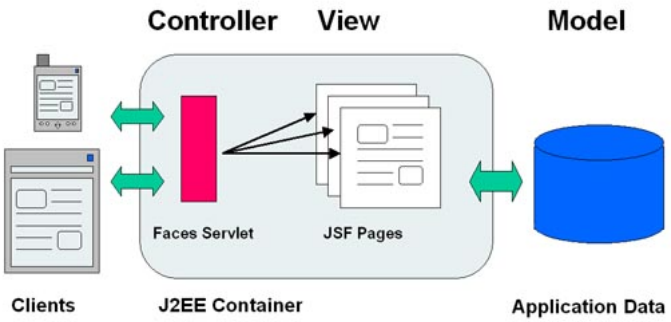


Figura 3 – Demonstração do Modelo MVC. **Fonte:** http://www.javabeat.net/jsf-2/.

[3] MVC: *Model-View-Controller - Design pattern*, padrão de arquitetura de software que separa a informação (e as suas regras de negócio) da interface com a qual o usuário interage.

Ao utilizar o JSF, toda e qualquer interação que o usuário realizar com a aplicação será executada por um *servlet* chamado *Faces Servlet*. Ele é responsável por receber tais requisições da camada de visão e redirecioná-las à lógica da aplicação e, posteriormente, enviar a resposta ao usuário (FARIA, 2013).

Segundo Lucknow e Melo (2010), a especificação do *JavaServer Faces* foi definida pela *Java Comunity Process* - JCP[4], que é uma entidade cujo objetivo é especificar a evolução do Java. E, por este motivo, grandes empresas como *Apache*, *IBM*, *Oracle*, entre outras, participaram desta definição. As implementações mais conhecidas da especificação do JSF são:

- *Sun Mojarra* (antes JSF R1) – implementação de referência (<http://javaserverfaces.java.net/>)
- *MyFaces* da *Apache* (<http://myfaces.apache.org/>)

Com essas implementações é possível utilizar todos os recursos do padrão JSF, como formulários, tabelas, *layout*, conversão e validação de eventos, além de toda a inteligência para a interpretação dos arquivos de configuração e interação com o *container* Java. Como o JSF é um padrão de mercado, várias empresas investem no desenvolvimento de bibliotecas de componentes como calendário, barras de progresso, menus, efeitos de transição, entre outros (LUCKNOW; MELO, 2010).

Algumas das principais bibliotecas de componentes são:

- *Trinidad*, da *Apache MyFaces* (<http://myfaces.apache.org/trinidad/>);
- *Tobago*, da *Apache MyFaces* (<http://myfaces.apache.org/tobago/>);
- *ICEFaces*, da *ICESoft* (<http://www.icefaces.org/>);
- *RichFaces*, da *JBoss* (<http://www.jboss.org/richfaces/>);
- *Tomahawk*, da *Apache MyFaces* (<http://myfaces.apache.org/tomahawk/>);
- *PrimeFaces* (<http://www.primefaces.org/>).

Neste trabalho, é utilizada a biblioteca PrimeFaces, que é uma biblioteca de componentes que implementa a especificação do JSF. Ela possui uma gama de componentes

[4] O termo *Java Comunity Process* será referenciado pela sigla JCP a partir deste ponto do trabalho.

disponíveis para auxiliar o desenvolvimento de interfaces *web* ricas, além de possuir o seu código fonte aberto (ROSS; BORSOI, 2012).

Para Juneau (2014), uma das grandes vantagens do Primefaces é a facilidade de integração entre ele e o JSF, bastando apenas incluir a biblioteca do Primefaces no projeto JSF, salvo alguns componentes específicos, como o *file upload*, que necessita de pequenas configurações adicionais. Essas mudanças, quando necessárias, devem ser realizadas no arquivo de configuração da aplicação que, por padrão, é chamado de *web.xml*, porém, o mesmo pode ser alterado pelo desenvolvedor.

Por possuir as vantagens descritas acima e uma simples configuração, além de ser um *framework cross-browser*[5], o JSF foi escolhido para auxiliar no desenvolvimento das página *web* deste trabalho.

2.2.3 Armazenamento de dados

A aplicação desenvolvida neste trabalho irá demandar o armazenamento de informações das costureiras e da localização das mesmas. Além disso, os processos e suas atividades juntamente com a ordem de execução das mesmas também devem ser armazenados, fazendo-se necessário o uso de um banco de dados.

O gerenciador de banco de dados a ser utilizado nesta aplicação é o MySQL, que foi escolhido por ser uma ferramenta robusta e *open source*. O MySQL é um banco de dados relacional estável, multi-tarefa e multi-usuário, que pode ser usado em sistemas web, sistemas de produção com alto volume de dados e missão critica. O conjunto de recursos do MySQL inclui praticamente tudo que uma aplicação de nível corporativo precisa (SUEHRING, 2002).

Para Date (2004) , "um banco de dados é um sistema computadorizado cuja funcionalidade geral é armazenar informações e permitir que os usuários busquem e atualizem essas informações quando as solicitar". Já o conceito de banco de dados relacional é definido por Price (2008, p.30) como:

> uma coleção de informações relacionadas, organizadas em tabelas. Cada tabela armazena dados em linhas; os dados são organizados em colunas. As tabelas são armazenadas em esquemas de banco de dados, que são áreas onde os usuários podem armazenar suas próprias tabelas.

Para manipular e acessar as informações em um banco de dados relacional é usada uma linguagem denominada SQL (*Structured Query Language*), que foi projetada espe-

[5] *Cross-Browser* - Compatibilidade com todos os tipos de dispositivos e navegadores

cificamente para este fim. A linguagem foi desenvolvida pela IBM, por volta de 1970, que tomou como base o trabalho do Dr. Edgar Frank Codd e possui uma sintaxe simples de fácil aprendizado e utilização (PRICE, 2008).

3 QUADRO METODOLÓGICO

O quadro metodológico descreve os passos realizados para a execução do trabalho. Neste capítulo serão listados, em suas seções, os itens essenciais no desenvolvimento do trabalho: sendo eles as técnicas, os procedimentos, as práticas e instrumentos utilizados, o contexto de aplicação e o tipo de pesquisa.

3.1 Tipo de pesquisa

Para Gil (1999, p.42), a pesquisa tem um caráter pragmático, é um “processo formal e sistemático de desenvolvimento do método científico. O objetivo fundamental da pesquisa é descobrir respostas para problemas mediante o emprego de procedimentos científicos”.

Este trabalho terá como base a metodologia de pesquisa aplicada, pois será desenvolvida uma aplicação inteligente utilizando Algoritmos Genéticos para o auxílio na tomada de decisão sobre o processo de produção de calças de uma confecção. Esta pesquisa consiste em procurar respostas para problemas propostos baseados em padrões e conhecimentos já existentes.

Gerhardt e Silveira (2009, p.35) afirmam que o método de pesquisa aplicada "objetiva gerar conhecimentos para aplicação prática, dirigidos à solução de problemas específicos. Envolve verdades e interesses locais."

Segundo Zanella (2009), a pesquisa aplicada tem como motivação básica a solução de problemas concretos, práticos e operacionais e também pode ser chamada de pesquisa empírica, pois o pesquisador precisa ir a campo, conversar com pessoas e presenciar relações sociais.

Como citam Marconi e Lakatos (2009), a pesquisa aplicada caracteriza-se por possuir um interesse prático, quando os resultados serão aplicados ou utilizados na solução de problemas que ocorrem na realidade, sempre visando gerar conhecimento para solucionar situações específicas.

Como já foi explicado o tipo de pesquisa em que se enquadra este trabalho, ela deve ser aplicada a um determinado contexto, conforme será explicado na seção a seguir.

3.2 Contexto de pesquisa

Sabe-se que, com a alta competitividade no mercado, as empresas, cada vez mais, buscam diferenciais para seus produtos e, neste cenário, a ideia de redução de custos se torna essencial, uma vez que tal redução pode ser refletida no preço dos produtos, permitindo que eles se diferenciem dos demais. Entre os fatores que viabilizam tais reduções está a otimização de processos, que consiste em organizar os procedimentos relacionados à produção, de forma que esses se tornem mais eficazes.

O software desenvolvido neste trabalho visa organizar uma linha de produção de forma que ela se torne o mais eficiente possível. Será utilizada como base uma fábrica de confecção de calças situada na cidade de Cachoeira de Minas - MG, porém a base de conhecimento pode ser aplicada a outros tipos de negócios que seguem o mesmo padrão de desenvolvimento de produtos.

Como cada funcionário trabalha em sua casa, é preciso ter uma boa forma de distribuir o trabalho, permitindo que a produção possa ser feita dentro de um prazo e com custo reduzido. Para isso, é necessário que o software faça a distribuição dos lotes a serem confeccionados entre as costureiras, de forma que elas sejam alocadas e recebam a quantidade de peças de acordo com sua capacidade de trabalho, considerando também o tempo que cada costureira irá gastar para pegar as peças e o material necessário para realizar seu trabalho. Além disso, o software considera o preço cobrado por cada uma das delas.

A aplicação cruza então todas essas informações de forma a se produzir soluções para o problema e, no final, a melhor dessas soluções será aquela que ofereça o menor custo dentro do prazo de entrega. Porém, sabe-se que, em algoritmos genéticos, não há garantias de que a solução encontrada seja a melhor para se resolver o problema, mas a tendência é que a solução proposta pelo algoritmo seja muito boa e que seria muito difícil encontrá-la manualmente.

3.3 Instrumentos

Segundo Faria (2015), instrumentos de pesquisa são a forma pela qual os dados são coletados para a realização do trabalho, podendo ser, dentre outras, por meio de reuniões,

questionários e entrevistas. Para este trabalho foram utilizados os instrumentos descritos nas subseções a seguir.

3.3.1 Entrevistas

Segundo Haguette (1997), entrevista é uma interação entre duas pessoas em que uma representa o entrevistador que, através de perguntas, obtém informações por parte de outra pessoa que representa o entrevistado.

Foi realizada uma entrevista com o dono da empresa de confecção com o objetivo de entender seu modelo de negócio para que então fosse possível começar a fazer o levantamento dos requisitos do sistema. Para Pressman (2011, p.128), levantamento de requisitos de software consiste em

> "perguntar ao cliente, aos usuários e aos demais interessados quais são os objetivos para o sistema ou produto, o que deve ser alcançado, como o sistema ou produto atende às necessidades da empresa e, por fim, como o sistema ou produto deve ser utilizado no dia a dia".

A entrevista ocorreu no dia 09/05/2015, para conhecer mais sobre o processo de produção da fábrica. Nesta entrevista, ficou esclarecido todo processo e também se teve acesso à forma como era controlada a distribuição da produção entre os funcionários. Toda produção era controlada por meio de planilhas Excel, gerenciadas e alimentadas pelo proprietário. Tais planilhas contemplavam as estimativas de produção, as datas de entrega dos lotes encomendados, levando em consideração a quantidade de peças por lote, o corte e também o tempo que cada lote levaria para ser entregue.

3.3.2 Reuniões

De acordo com Carvalho (2012), reunião é o ajuntamento de pessoas para se tratar de um determinado assunto em que é necessário que se tenha conclusões sobre as questões que foram discutidas.

Durante o desenvolvimento do trabalho seriam realizadas várias reuniões com o proprietário da fábrica de calças para sanar as dúvidas, sugestões e outros assuntos que poderiam surgir. Todavia foi realizada apenas uma reunião com o proprietário da empresa para poder entender como funciona o processo de produção, pois foi constatado nesta

reunião que o processo de produção havia sido alterado. No processo inicial, o qual foi a base para este trabalho, cada funcionário trabalhava em sua residência e as peças eram distribuídas entre eles. Atualmente o processo passou por muitas mudanças, uma delas é que a produção é feita em um lugar apenas, sem a necessidade de transportar as peças entre as casas dos funcionários. Segundo o proprietário, isso gerou um ganho de tempo bem expressivo, pois as peças circulavam dentro de um mesmo local e não pela cidade.

Considerando essa mudança, não foram feitas outras reuniões com o proprietário da fábrica, pois o trabalho não atende mais o processo de produção atual da fábrica, porém o mesmo pode ser usado em outras empresas que seguem a forma de produção pesquisada. Assim, foi definido um escopo para o desenvolvimento da aplicação baseando-se no processo inicial da fábrica de calças, que se resume em construir uma aplicação levando em consideração que:

- o processo de desenvolvimento das calças deveria ser dividido em atividades com ordem de precedência;
- cada atividade poderia ser feita por uma ou mais costureiras, de acordo com a habilidade de cada uma;
- cada costureira gasta um tempo, medido em segundos, para fabricar uma peça;
- cada costureira cobra um preço por peça produzida;
- o usuário deveria ser capaz de cadastrar um novo processo, costureiras e habilidades;
- o usuário deveria cadastrar o tempo e o preço de produção, por peça, para cada costureira, além de sua posição geográfica;
- o total de peças deveria ser dividido em lotes e cada costureira deveria receber uma quantidade de lotes distribuída aleatoriamente;
- o usuário deveria informar uma data de entrega das peças de um processo e uma data de início de execução do mesmo para que se pudesse calcular o prazo;
- o software deveria então oferecer como saída a melhor distribuição de forma a se produzir no menor tempo dentro do prazo, com o menor custo, considerando o tempo e o preço de produção de cada costureira e o tempo de transporte das peças entre elas.

- se o software não conseguisse encontrar nenhuma solução abaixo do prazo, a busca pelo menor custo seria ignorada e o menor tempo encontrado seria retornado.

3.4 Procedimentos

Esta seção descreve os procedimentos realizados na execução do trabalho, definindo primeiramente o *framework* de desenvolvimento e explicando como o algoritmo genético foi desenvolvido para os requisitos definidos no escopo.

3.4.1 Framework de desenvolvimento

Primeiramente é necessário ressaltar que, para o desenvolvimento da aplicação, foi utilizada uma base desenvolvida pelo professor Artur Barbosa durante as aulas de sistemas especialistas, do VII período do curso de sistemas de informação, nesta universidade. Esta base, também denominada *framework*, define regras a serem seguidas no desenvolvimento de cada elemento de um algoritmo genético. Tal *framework* é definido dentro da seguinte estrutura:

- Classe `GAModel`:

  A classe `GAModel` é basicamente a classe mãe de todos os elementos de um algoritmo genético, que representa o modelo que irá armazenar a população de indivíduos, além de ser a classe que armazena os parâmetros que definem as configurações do algoritmo, tais como tipo de cruzamento, tipo de mutação, tamanho da população, etc.

  A classe contém os seguintes atributos:

  - `populationSize`: este atributo define qual será o tamanho da população, ou seja, quantos indivíduos irão formar cada população;
  - `generationQuantity`: como já explicado no quadro teórico, o processo de cruzamento e mutação se repete até que o número de indivíduos, definido no atributo anterior, seja atingido, formando assim uma nova população e então, por sua vez, esse processo de geração de novas populações se repete até que

seja atingido um número de gerações definido pelo programador. Esse atributo representa essa quantidade;

- `elitism`: atributo do tipo *boolean*, que representa se o algoritmo vai ter a função de elitismo. Essa função, como já foi explicada no quadro teórico, quando está ativada (com valor *true*), no momento de começar a se criar uma nova população, os dois melhores indivíduos da população que será substituída já começam a fazer parte da nova população, antes de começar o processo de cruzamento e mutação. Esse mecanismo garante que a nova população terá pelo menos dois indivíduos iguais ao da antiga população, o que irá impedir que a nova população não seja pior que a anterior;
- `foreignIndividualRate`: este atributo define a taxa de novos indivíduos que devem entrar em novas populações e será visto com mais detalhes na seção 3.4.6.
- `mutationRate`: como descrito no quadro teórico, a mutação é o fato de realizar pequenas alterações no indivíduo. a fim de que ele possa se tornar ainda melhor. Este parâmetro define uma porcentagem, geralmente baixa, que define quando o indivíduo sofrerá mutação ou não. Essa questão ficará mais clara na seção 3.4.8.
- `mutationQuantity`: caso a mutação for ocorrer para o indivíduo, a alteração aleatória será feita nos cromossomos. Este parâmetro define quantos cromossomos do indivíduo devem ser alterados pela mutação;
- `selectionType`: conforme descrito no quadro teórico, existem várias formas de seleção dos indivíduos para realizarem o cruzamento. Este parâmetro define qual será a forma de seleção escolhida pelo programador ao implementar o seu problema. No *framework*, esse parâmetro é do tipo `enum` e pode assumir 2 valores: o `ROULETTE`, que define que o método de seleção utilizado será o de roleta e o `CLASSIFICATION`, que define que o método a ser utilizado é o de classificação. Neste projeto, o método padrão que já foi pré-definido no código foi o método de classificação;
- `crossType`: assim como a seleção, existem diversas formas de fazer o cruzamento dos indivíduos. Este atributo, também do tipo `enum`, representa a forma de cruzamento e pode receber os valores `Binary`, `Permutation`, `Uniform` e `Aritmetic`, esses valores definem qual implementação de cruzamento o al-

goritmo deve utilizar, esta aplicação utilizará o método `Permutation`, desta forma somente um método de cruzamento foi implementado, conforme mostra a seção 3.4.7;

  - `mutationType`: Segue a mesma forma que o `selectionType` e o `crossType` e pode assumir os valores `Permutation`, `Binary` e `Numerical`, e neste projeto foi o escolhido o método `Permutation`.

- Classe `Individual`:

  A classe abstrata `Individual` representa a estrutura básica de um indivíduo. A classe contém uma `lista` do tipo `Cromossomo`, que será descrita posteriormente, que contém uma coleção de objetos que representam as características da solução.

  A classe contém ainda um atributo chamado `valor`, que irá armazenar a qualidade (*fitness*), ou seja, qual é o custo da solução representada pelo indivíduo, tal valor é recebido no retorno da operação `calculateValue()` descrita abaixo.

  Com relação as operações, além dos *getters and setters*, a classe contém a operação abstrata `calculateValue()`, que realiza a função de avaliação, que mede a qualidade do indivíduo. Desta forma, ao utilizar este *framework*, a classe que representará o indivíduo do problema deve herdar desta classe `Individual`. Fazendo isso, a classe herdeira passará a ter uma lista de cromossomos e o atributo que representa o seu valor e, obrigatoriamente, terá que implementar a operação `calculateValue()`, permitindo assim que o programador desenvolva a função de avaliação específica para o seu problema.

- Classe `Chromosome`:

  É uma classe abstrata, que possui todos os métodos abstratos, desta forma ela só existe para garantir que os cromossomos do problema irão implementar os métodos necessários para o funcionamento do algoritmo. Estes métodos são:

  - `equals`: necessário para efeito de comparação dos cromossomos;
  - `doMutation`: realiza a mutação. Deve ser implementado pela classe que representa o cromossomo, pois a mutação é feita nos cromossomos.
  - `clone`: devolve um objeto exatamente com os mesmos atributos do objeto, porém em uma instância diferente.

- Classe `IndividualPair`:

  A classe `IndvidualPair` possui uma estrutura simples. Apenas representa dois indivíduos. Ela se torna necessária, pois o processo de cruzamento dos indivíduos retornam dois novos indivíduos, desta forma, como no Java não é possível retornar dois valores, é retornado então um objeto desta classe contendo os dois novos indivíduos criados.

- Classe `GAController`:

  A classe `GAController`, como o próprio nome já diz, é o controlador de todo processo de execução do algoritmo genético. Ela recebe, no seu construtor, o modelo que é do tipo `GAModel` que, como já explicado anteriormente, armazena os parâmetros a serem seguidos na execução do algoritmo. Além disso, através do seu método principal, denominado `execute()`, ela é responsável por criar novas populações a partir de cruzamentos, mutações, elitismo, etc, tendo também a responsabilidade de chamar a função de classificação e avaliação de cada indivíduo.

  Os seguintes passos descrevem basicamente os passos executados dentro do método execute(). Outros detalhes serão vistos mais adiante quando será descrita a implementação do algoritmo da fábrica de calças, pois será necessário realizar algumas adaptações neste *framework*.

  - Criação da população inicial, através do método `createInitialPopulation()` do objeto que será extendido da classe GAModel;
  - Classificação e avaliação da população inicial através do método `classify()`;
  - Realização do processo de elitismo, através do método `doElitism()`;
  - Inserção de indivíduos estrangeiros na população;
  - Realização do processo de seleção de indivíduos, através do método `doSelection()`;
  - Execução do processo de cruzamento e mutação, através do método `doCrossing()` e `doMutaion()` respectivamente.

  Após estes passos, uma nova população foi criada e está armazenada na variável `newGeneration`, assim o método `setPopulation()` do objeto `model` é chamado para então substituir a antiga população pela nova. Como a execução está dentro de uma estrutura de repetição `for`, ocorre um `loop` e então é recomeçado o processo de criação de uma outra população. Quando o número de gerações definido no

parâmetro `generationQuantity` do objeto `model` for atingido é dado o comando `break`, e o `loop` é encerrado.

As próximas seções apresentam a implementação dos requisitos da aplicação e dos elementos do algoritmo genético, seguindo as definições do *framework*.

3.4.2 Representação do processo de produção

Primeiramente foi definido como seria o processo de fabricação. Este foi pensado com base no processo da fábrica, em que a confecção das peças deveria ser dividida em atividades que representam a fabricação de cada parte da calça. Neste contexto, surgiu a necessidade de determinar uma ordem para a execução do processo, devido ao fato de que algumas atividades dependem da finalização de outras para poderem ser realizadas. A Figura 4 demonstra basicamente um exemplo de ordem de execução do processo de confecção, sendo importante ressaltar que a atividade finalização, apesar de ser a atividade final, é que inicia o processo de solicitação das dependências das atividades predecessoras.

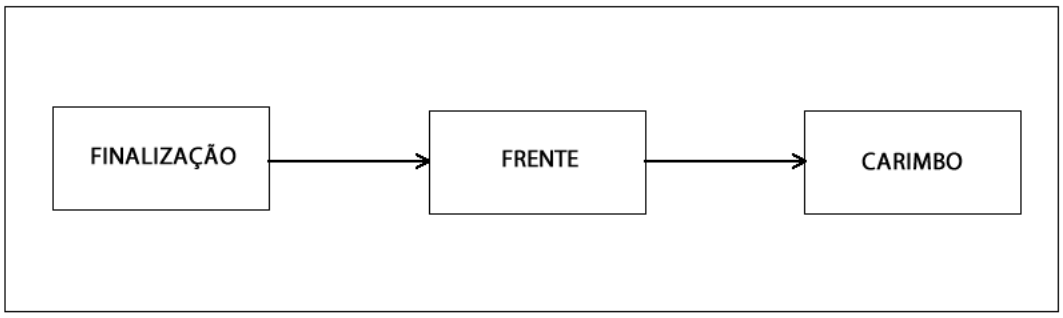


Figura 4 – Demonstração de um processo de fabricação de uma calça. **Fonte:** Desenvolvido pelos autores.

Uma questão importante que foi definida é que, independente da complexidade e tamanho do processo, ele deve sempre começar com a atividade carimbo e finalizar com a atividade Finalização. Isso ocorre pois a finalização é sempre a última atividade de qualquer processo da fábrica e o carimbo é uma atividade simbólica que representa o fato de o dono da fábrica separar o material de costura. O tempo gasto e o custo desta separação não são contabilizados no algoritmo genético, somente o tempo de transporte dos materiais até as costureiras são considerados na distribuição, além disto esta atividade terá somente uma pessoa trabalhando, neste caso será o Marcelo, dono da fábrica, que é o responsável por este trabalho.

Para armazenar este processo e suas atividades no software, foram utilizadas tabelas do banco de dados, conforme mostra a Figura 5.

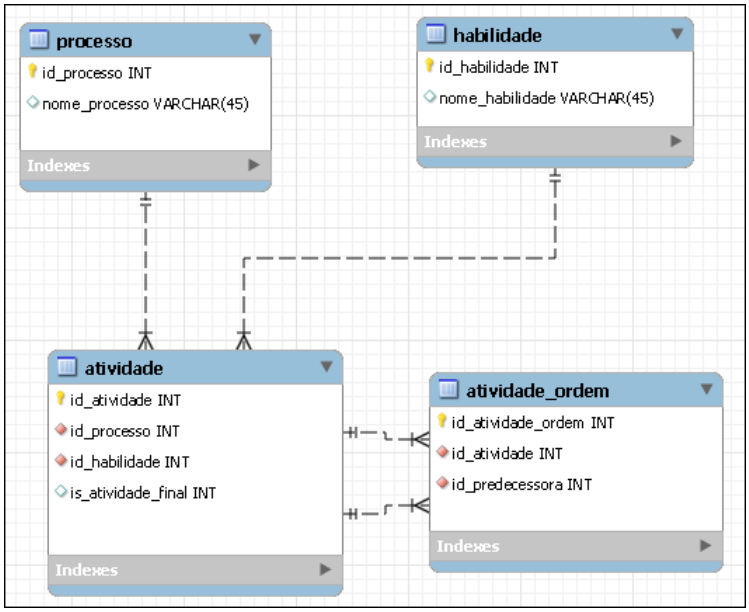


Figura 5 – Representação do processo de fabricação no banco de dados. **Fonte:** Desenvolvido pelos autores.

A tabela `processo` tem como finalidade gerar um código único para representar cada processo de produção, cada processo representa um modelo, cada modelo de calça possui um processo diferente que pode possuir diversas atividades que são representadas na tabela `atividade`, onde é feita a relação que define quais são as atividades de um processo, além de conter quais são as habilidades necessárias para cada atividade, ou seja, cada registro desta tabela representa uma atividade do processo e qual habilidade é necessária para sua execução. O campo `is_atividade_final`, quando tem o valor 1, define que tal atividade é a atividade Finalização. Esta *flag* é importante no momento de calcular o tempo total de execução do processo e será visto com mais detalhes posteriormente. Por fim, a tabela `atividade_ordem` é onde é feita a definição de ordem de execução das atividades.

A Figura 6 demonstra as classes que representam o mapeamento da relação entre a tabela `atividade` e `atividade_ordem` para o Java.

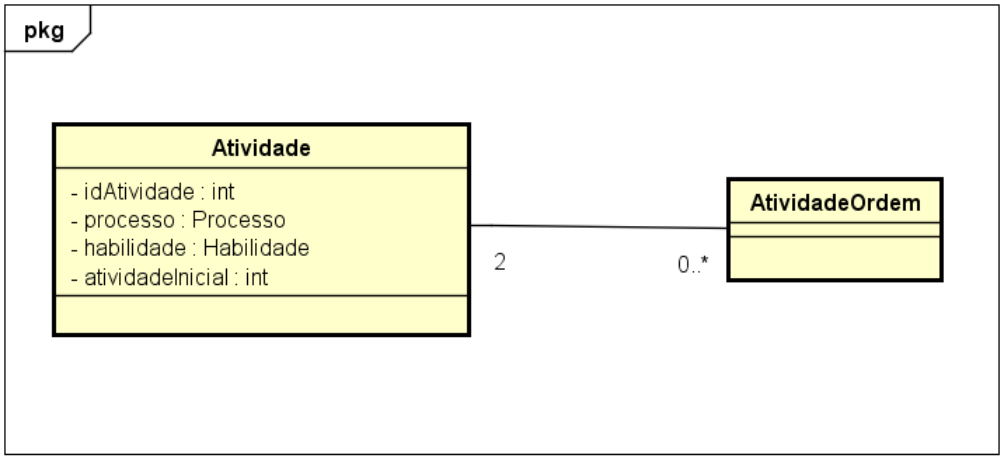


Figura 6 – Classes `Atividade` e `AtividadeOrdem`. **Fonte:** Desenvolvido pelos autores.

A classe `Atividade` pode ter zero ou vários objetos da classe `AtividadeOrdem`. Esta possui dois objetos da própria classe `Atividade`, um representando uma atividade e outro representando a sua predecessora.

Considerando o processo de produção e o modelo de dados apresentados anteriormente, foi desenvolvido um algoritmo genético para a solução do problema de otimização. Conforme já demonstrado no capítulo 2, um algoritmo genético deve seguir uma ordem de execução conforme mostra a Figura 1 (ver página 16).

O primeiro passo para a execução do algoritmo, então, foi o desenvolvimento de uma lógica para a criação da população inicial de indivíduos.

3.4.3 População inicial: Distribuição das atividades, Indivíduos e Cromossomos

Para um melhor entendimento dos conceitos a serem explanados, é necessário conhecer, de forma básica, o fluxo de execução da aplicação. Como será visto na seção 3.4.9, o usuário irá iniciar a execução do algoritmo genético através de uma tela de distribuição de atividades, onde ele deve informar o número de peças que deseja produzir, a quantidade de peças que cada lote deve possuir e uma data de início das atividades. Através da data de início e da data de entrega, informada no cadastro do processo, o controlador da tela calcula o prazo de entrega e o número total de lotes e chama o serviço de gerenciamento da execução do AG, passando o número de lote, o número de peças por lote e o prazo de entrega a ser considerado durante a distribuição. Tal serviço, por sua vez, configura os parâmetros do algoritmo genético e dá início a sua execução. A Figura 7 ilustra o fluxo

básico de execução da aplicação.

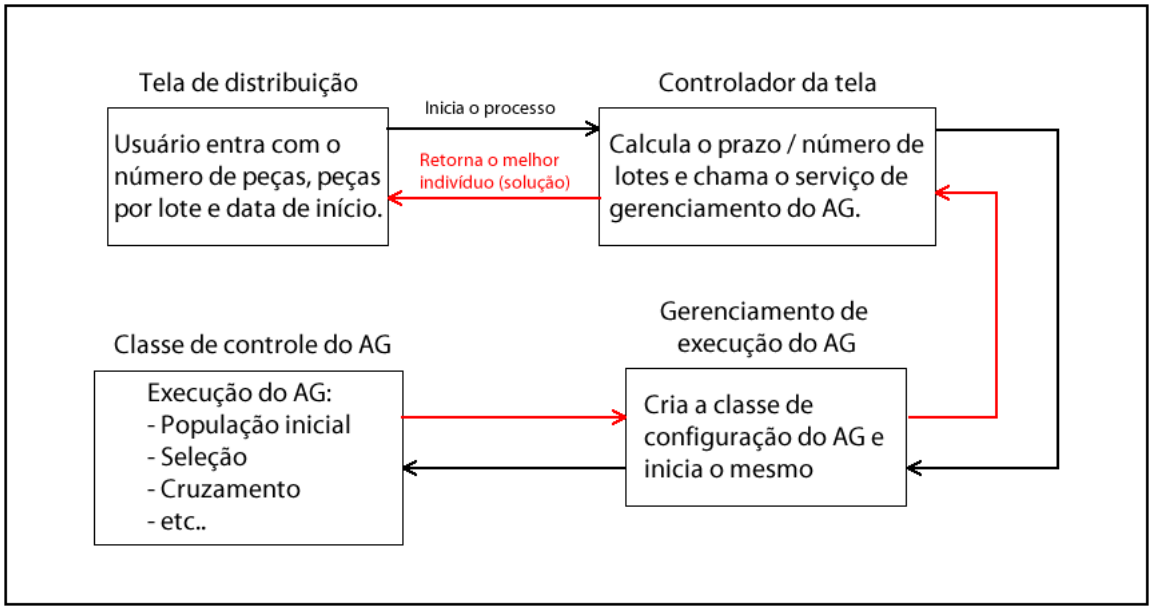


Figura 7 – Fluxo básico de execução. **Fonte:** Desenvolvido pelos autores.

Conforme ilustrado na Figura 7, o processo de criação da população inicial, assim como o processo de seleção, cruzamento e mutação e a chamada da função de avaliação de cada indivíduo, ocorre dentro de uma classe de controle, ou seja, esta classe faz a orquestração de toda a execução do AG. Tal classe é denominada `GAController` e pertence ao *Framework* explanado anteriormente, na seção 3.4.1.

A classe criada para o gerenciamento da execução do algoritmo genético é denominada `GeneticAlgorithmManagement` e, conforme demonstrado na Figura 7, esta é instanciada pelo controlador da tela de distribuição de atividades. Tal classe possui um único método denominado `iniciarDistribuicao()` e recebe nele os dados informados pelo usuário, além disto, nesse método, é criado um objeto de `ProcessoModel`, que herda da classe `GAModel` do *framework*, e, nesse objeto, são definidos os parâmetros a serem usados pelo algoritmo genético. Além disso, os dados informados pelo usuário também são definidos neste objeto, pois são utilizados pela regra de negócio seguida na execução do algoritmo. O Código 1 demonstra o método `iniciarDistribuicao()`.

Código 1 – Método `iniciarDistribuicao()` da classe `GeneticAlgorithmManagement`. **Fonte:** Elaborado pelos autores.

```
1
2 public ProcessoIndividual iniciarDistribuicao(
3       int numeroLote,BigDecimal prazEmSegundos,
4       int pecasPorLote,int idProcesso){
5
6     EntityManager manager = ConFactory.getConn();
7
```

```
8      ProcessoModel model = new ProcessoModel(manager,idProcesso);
9      model.setNumeroLote(numeroLote);
10     model.setPecasPorLote(pecasPorLote);
11     model.setPrazoEmSegundos(prazEmSegundos);
12     model.setGenerationQuantity(10000);
13     model.setPopulationSize(80);
14     model.setElitism(true);
15     model.setSelectionType(GAModel.SelectionType.CLASSIFICATION);
16     model.setCrossType(CrossType.PERMUTATION);
17     model.setForeignIndividualRate(0.3f);
18     model.setMutation(GAModel.MutationType.PERMUTATION);
19     model.setMutationRate(0.05f);
20     model.setMutationQuantity(1);
21
22     GAController controller = new GAController(model);
23     return (ProcessoIndividual) controller.execute();
24 }
```

Tal método retorna um objeto do tipo `ProcessoIndividual`, que será explicado posteriormente, para o controlador da tela de distribuição, este é o melhor indivíduo encontrado, ou seja, a melhor solução encontrada. Tal solução será então apresentada ao usuário conforme mostra a Figura 7.

A classe `ProcessoModel`, ao ser instanciada, recebe em seu construtor uma conexão para o banco de dados e o `ID` do processo que será otimizado pelo algoritmo. Tal construtor então chama o método `getInformacoesCostureiras()`, que tem por finalidade buscar no banco de dados todas as atividades do processo em questão, buscar todas as costureiras que têm a habilidade de fazer cada uma das tarefas e criar um `HashMap`, que possui como chave o `ID` da atividade e como valor uma lista de `CostureiraHabilidade`, feito isto o método também recupera a atividade final (finalização), conforme demonstra o Código 2.

Código 2 – Método `getInformacoesCostureiras()`. **Fonte:** Elaborado pelos autores.

```
1
2  public void getInformacoesCostureiras() {
3    List<CostureiraHabilidade> costureirasHabilidades = null;
4
5    if (atividadesCostureiras != null && atividadesProcesso != null
        ){
6       atividadesCostureiras.clear();
7       atividadesProcesso.clear();
8    }
9
10   atividadesCostureiras =
11      new HashMap<Integer, List<CostureiraHabilidade>>();
12
13   atividadesProcesso =
14      atividadeDao.listAtividadesByProcesso(processo);
15
16   /* Montar um MAP tendo como chave cada atividade do processo
17      e a lista de costureiras que tenham a habilidade relacionada
           .*/
```

```
18   for (Atividade atividade : atividadesProcesso) {
19
20     costureirasHabilidades =
21        cdao.getCostureirasByHabilidade
22           (atividade.getHabilidade().getIdHabilidade());
23
24     atividadesCostureiras.put
25        (atividade.getIdAtividade(),costureirasHabilidades);
26
27     if (atividade.isAtividadeFinal()) atividadeFinal = atividade;
28   }
29 }
```

Então, de acordo com o Código 1, após a definição dos parâmetros, é criado um novo objeto de `GAController`, passando-se o objeto da classe `ProcessoModel` com todos os parâmetros do algoritmo genético, os dados do usuário e informações de atividades e costureiras do processo em questão e então o método `execute()` é chamado, dando-se início à execução do algoritmo genético. A primeira coisa a ser feita, neste método, é a criação da população inicial de indivíduos realizada a partir do método `createInitialPopulation()`, declarado de forma abstrata na classe `GAModel` e implementado pela classe `ProcessoModel`. Tal método basicamente executa um `for` de 0 até o tamanho da população (atributo `populationSize`) e assim para cada iteração é criado um objeto de `ProcessoIndividual` passando a `atividadeFinal`, o `map` que contém as atividades e suas costureiras (`atividadesCostureiras`) criados pelo método `getInformacoesCostureiras()`, além dos atributos `numeroLote`, `pecasPorLote` e `prazoEmSegundos`, conforme mostra o Código 3.

Código 3 – Método `createInitialPopulation()`. **Fonte:** Elaborado pelos autores.

```
1
2 @Override
3   public void createInitialPopulation() {
4     for (int i = 0; i < getPopulationSize(); i++) {
5       population.add
6          (new ProcessoIndividual
7              (atividadeFinal, prazoEmSegundos,
                 atividadesCostureiras,
8               this.numeroLote, this.pecasPorLote));
9   }
10 }
```

Quando se cria um novo indivíduo, a partir da instanciação de um novo objeto de `ProcessoIndividual`, acontece então, no construtor da classe, a distribuição das atividades de forma a representar uma solução para o problema. Essa distribuição foi realizada considerando que o total de peças a ser produzido deveria ser dividido em lotes e então, em cada atividade, esse número de lote deveria ser distribuído entre as costureiras que possuíssem a habilidade em questão. Por exemplo: se a quantidade total de peças de

uma ordem de produção for 500, primeiramente deve-se definir qual será o número de peças por lote, neste caso, se for definido que cada lote deverá ter 50 peças, então o resultado final será 500/50, ou seja, 10 lotes contendo 50 calças cada um.

Nesse sentido, seguindo o exemplo apresentado na Figura 4, a distribuição deverá ser feita de forma que, para cada atividade do processo, seja distribuído o trabalho de 10 lotes, ou seja, o material para a confecção de 10 lotes deve ser enviado para as costureiras que irão fazer a parte da frente e posteriormente essas enviam os 10 lotes para as costureiras que fazem a finalização.

Com base nesses requisitos, para a realização de tal distribuição, inicialmente, o algoritmo irá distribuir, de forma aleatória, o número de lotes definido entre as costureiras de cada atividade, conforme mostra a Figura 8.

Finalização	Frente	Carimbo
Josi: 10	Roberta: 5	Marcelo: 10
-	Tereza: 5	-
-	Maria: 0	-

Figura 8 – Exemplo de distribuição aleatória de lotes para as costureiras. **Fonte:** Desenvolvido pelos autores.

Uma costureira pode não receber lotes, isso permite que a decisão de quem vai participar ou não também fique por conta do algoritmo.

Como já explanado no quadro teórico, a estrutura do algoritmo genético é composta por populações que são formadas por indivíduos, que por sua vez são formados por cromossomos. Cada indivíduo representa uma solução e cada cromossomo do indivíduo representa uma de suas características. Assim, então, é gerada uma população inicial de indivíduos e, a partir dela, um processo de cruzamento e mutação é iniciado a fim de que possam ser gerados novos indivíduos que representem soluções ainda melhores que seus antecessores.

Nesse sentido, para o desenvolvimento do algoritmo de distribuição de lotes, o processo de definição de indivíduo e cromossomo foi o primeiro passo do desenvolvimento da aplicação. Isso se deve ao fato de que esses elementos compõem a parte crucial

para que se possa definir a lógica a ser seguida para a definição da população inicial, o tipo de cruzamento, a função de avaliação, etc.

Nesse caso, cada indivíduo da população irá representar uma forma de distribuir as atividades e a quantidade de lotes distribuídos a determinada costureira em uma determinada atividade irá representar um cromossomo. Tomando como base o exemplo da Figura 8, o quadro, como um todo, representa o indivíduo e cada distribuição, como por exemplo a Roberta, que recebeu 5 lotes para confeccionar a frente, representa um cromossomo.

Para fazer esta representação em Java, primeiramente foi criado uma classe denominada `ProcessoChromosome`, que é representada na Figura 9:

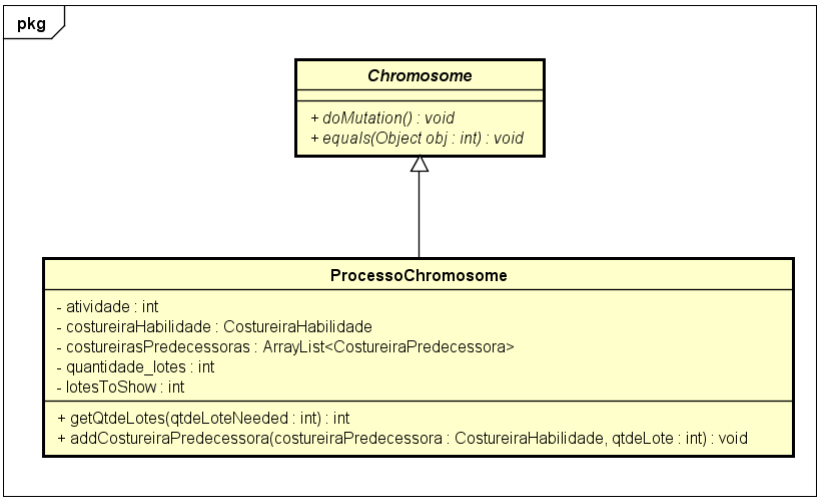


Figura 9 – Classe `ProcessoChromosome`. **Fonte:** Desenvolvido pelos autores.

A classe `ProcessoChromosome` herda de `Chromosome` do *framework* descrito na Figura 9. Por enquanto, é necessário compreender apenas os atributos `atividade`, `costureiraHabilidade` e `quantidade_lotes`, que recebem seus valores pelo construtor, os demais atributos e métodos são utilizados pela função de avaliação e serão explicados mais adiante. O atributo `atividade` representa o `ID` da atividade, à qual se está atribuindo a costureira e a quantidade de lotes, este atributo será passado na criação de cada cromossomo sempre que for necessário se criar um novo indivíduo. O atributo `costureiraHabilidade` é do tipo `CostureiraHabilidade`, que representa o mapeamento da tabela `costureira_habilidade` do banco de dados, onde é feita a relação entre quais habilidades cada costureira possui além do tempo e o preço de cada uma para fazer uma peça de uma determinada parte da calça. Além disso, na tabela `costureira`, foram definidos os campos `posicaoX` e `posicaoY`, que definem a localização da costu-

reira, conforme demonstra a Figura 10. Estes valores serão utilizados posteriormente para calcular a distância entre duas costureiras,

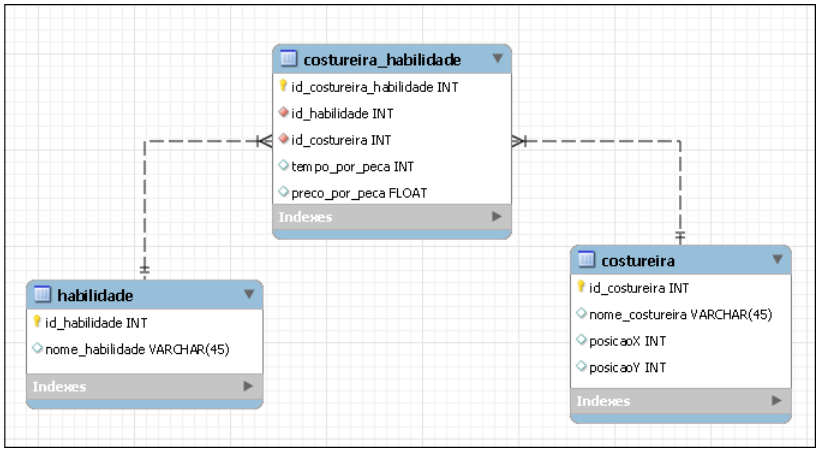


Figura 10 – Armazenamento de dados das costureiras. **Fonte:** Desenvolvido pelos autores.

A classe `CostureiraHabilidade`, conforme ilustra a Figura 11, por sua vez, possui o atributo habilidade, outro que representa a costureira, além de dois atributos que representam o tempo de produção e o custo de cada costureira para confeccionar uma peça de uma determinada parte. Fez-se necessário ter um atributo da classe `CostureiraHabilidade` ao invés de simplesmente ter um objeto do tipo Costureira, pois, na função de avaliação, como será visto mais adiante, é necessário ter o tempo gasto e o preço de cada costureira para se fazer uma peça e essas informações podem variar para uma mesma costureira dependendo de suas habilidades.

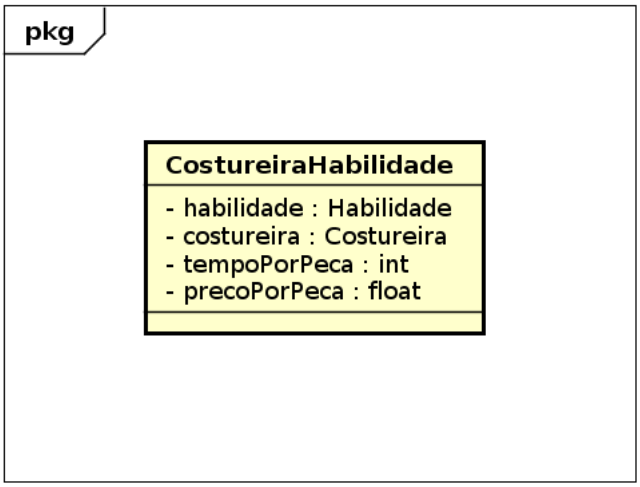


Figura 11 – Classe `CostureiraHabilidade`. **Fonte:** Desenvolvido pelos autores.

Assim, para representar cada característica da solução, tomando como exemplo a Figura 8, o ato de Roberta fazer 5 lotes da parte da frente é representado na implementação do código, criando-se um objeto da classe `ProcessoChromosome`, passando no construtor o `id` da atividade frente, um objeto de `costureiraHabilidade`, cujo atributo `costureira` represente a Roberta, o atributo `habilidade`, que representa a habilidade

em questão e a quantidade de lotes que Roberta deverá confeccionar, que seria, nesse caso, cinco.

A representação do indivíduo foi feita criando-se a classe `ProcessoIndividual`, como demonstra a Figura 12:

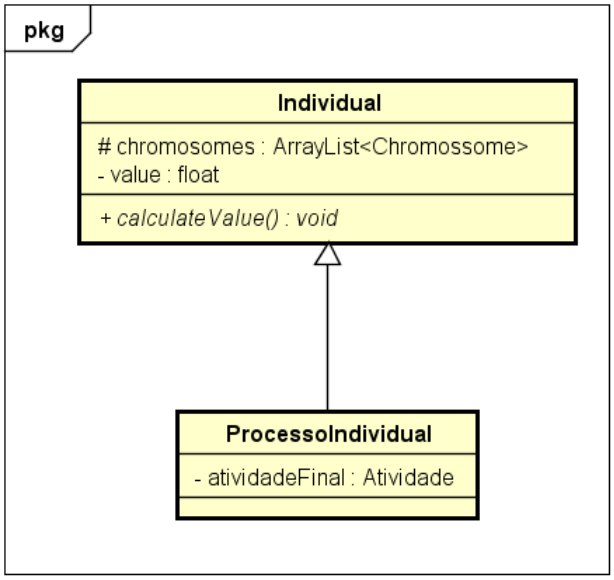


Figura 12 – Classe `ProcessoIndividual`. **Fonte:** Desenvolvido pelos autores.

A classe `ProcessoIndividual` herda da classe `Individual` do *framework*, e, por isso, essa passa a ter um `ArrayList` com objetos do tipo `Chromosome`. Nesse caso, como a classe `ProcessoChromosome` herda de `Chromosome`, este `ArrayList` terá objetos do tipo `ProcessoChromosome`.

A criação dos cromossomos que irão compor o indivíduo é feita através do construtor da classe `ProcessoIndividual` e é neste ponto que os lotes são distribuídos às costureiras de cada atividade. O `construtor` da classe `ProcessoIndividual` recebe como parâmetro um objeto representando a atividade final, que será utilizado pela função de avaliação mais adiante, e um `HashMap` que possui como chave o `ID` de uma atividade e uma lista do tipo `CostureiraHabilidade`, contendo as costureiras e suas respectivas informações de tempo e preço para se fazer tal atividade.

Com base neste `HashMap`, então é feita a criação dos cromossomos do indivíduo. Em um primeiro momento, a distribuição de tarefas entre as costureiras seria feita em forma de porcentagem, ou seja, o algoritmo distribuiria uma porcentagem aleatória para cada costureira de uma determinada atividade, dessa forma a distribuição seria feita conforme demonstra o código 4.

Código 4 – Criação de cromossomos (Primeira Abordagem). **Fonte:** Elaborado pelos autores.

```
1  package edu.univas.edu.tcc.ga_code;
2
3  import java.util.ArrayList;
4
5  public class ProcessoIndividual extends Individual {
6
7    public Atividade atividadeFinal;
8
9    public ProcessoIndividual(Atividade atividadeInicial,
10     Map<Integer, List<CostureiraHabilidade>>
          atividadesCostureiras){
11
12     chromosomes = new ArrayList<Chromosome>();
13     this.atividadeFinal = atividadeInicial;
14
15     for(Integer key: atividadesCostureiras.keySet()){
16        for(CostureiraHabilidade costureira :
17          atividadesCostureiras.get(key)){
18
19          Float porcentagem = (float) (Math.random() *1);
20          chromosomes.add(new ProcessoChromosome(key,
21            costureira, porcentagem));
22       }
23     }
24   }
25 }
```

Feita a distribuição da porcentagem, antes de fazer o cálculo do indivíduo, seria então realizado um cálculo de normalização para que se pudesse encontrar o número de lotes a ser produzido por cada costureira em cada atividade. Tomando como exemplo a Figura 13, este cálculo seria feito da seguinte forma:

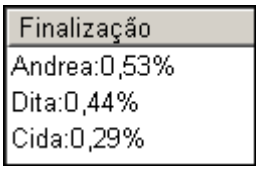


Figura 13 – Distribuição em porcentagem. **Fonte:** Desenvolvido pelos autores.

- Primeiramente deveria ser feita a soma de todas as porcentagens distribuídas, logo:

  `0,53 + 0,44 + 0,29 = 1,26;`

- o segundo passo seria calcular quanto cada porcentagem equivale dentro do total, nesse sentido o cálculo, já fazendo o arredondamento, seria:

  `0,53 / 1,26 = 0,42 | 0,44 / 1,26 = 0,35 | 0,29 / 1,26 = 0,23`

  Logo, nesse caso a Andrea seria responsável por 42%, a Dita por 35% e a Cida por 23%;

- Assim, seria feito um cálculo com regra de 3 com o número total de peças. Supondo que o valor total fosse 500, logo:

  `(500 * 42) / 100 = 210 | (500 * 35) / 100 = 175 | (500 * 23) / 100 = 115`

  Nesse caso então, a Andrea deveria produzir 210 peças, a Dita 175 e a Cida 115 peças;

- Por fim, deveria ser feita uma divisão dos número de peças de cada costureira pela quantidade de peças por lote, que nesse caso poderia ser 50, então realizando o cálculo já com arrendondamento:

  `210 / 50 = 4 | 175 / 50 = 4 | 115 / 50 = 2`

  Assim, a Andrea produziria 4 lotes, a Dita 4 e a Cida 2, dando o total dos 10 lotes a serem produzidos para a atividade de finalização.

Os passos acima para distribuição de atividades seriam então repetidos para cada atividade chave do `HashMap`, realizando tal distribuição para cada costureira da lista de costureiras de cada atividade. No momento de fazer o último cálculo, foi feito um arredondamento, com isso se uma costureira tivesse tido uma porcentagem muito pequena, o valor de lotes para esta seria 0, eliminando-a assim da distribuição.

Todavia, verificou-se que, distribuindo desta forma, em alguns casos, o total de lotes por atividade não era distribuído de forma correta. Devido ao arredondamento, às vezes uma atividade ficava com lotes a menos ou lotes a mais do que o total definido, o que poderia causar erros no cálculo final. Além disso, no momento de definir como seria o cruzamento, surgiu uma questão importante, que é o fato de que todas as vezes que fosse criado um indivíduo a partir de outros, deveria ser realizado o cálculo de normalização, e com isso a distribuição de lotes no novo indivíduo poderia ficar completamente diferente de seus pais, resultando assim na quebra do paradigma de algoritmos genéticos, que descreve que os indivíduos filhos devem ser formados pela mesclagem das características dos pais.

Buscou-se então uma outra alternativa para se realizar a distribuição dos lotes e definiu-se que, ao invés de distribuir a porcentagem, a distribuição já deveria ser feita por lote, sendo esta também realizada de forma aleatória. Os parâmetros do `construtor` da classe `ProcessoIndividuo` permaneceram da mesma forma, alternando somente a

forma com que os lotes são distribuídos entre as costureiras em cada atividade, conforme mostra o Código 5.

Código 5 – Distribuição em lotes diretamente. **Fonte:** Elaborado pelos autores.

```
1  public ProcessoIndividual(Atividade atividadeFinal,
2    BigDecimal prazoEmSegundos,
3    Map<Integer, List<CostureiraHabilidade>> atividadesCostureiras,
4    int numeroLote, int pecasPorLote){
5
6    chromosomes = new ArrayList<Chromosome>();
7    Map<Integer, ProcessoChromosome> chromossomosMap =
8      new HashMap<Integer, ProcessoChromosome>();
9
10   this.atividadeFinal = atividadeFinal;
11   this.numeroLote = numeroLote;
12   this.pecasPorLote = pecasPorLote;
13   this.prazo = prazoEmSegundos;
14
15   boolean distribuiuPorTodasCostureiras = false;
16
17   int qtdeLote = 0;
18   int cont = 0;
19
20   for (Integer key : atividadesCostureiras.keySet()) {
21     qtdeLote = numeroLote;
22     cont = 0;
23     int loteCostureira = 0;
24     distribuiuPorTodasCostureiras = false;
25
26     while (true){
27       /* Verificou-se que quando a qtdeLote era 1 o valor
28          sorteado nunca era zero usando o Math.random com
29          CAST para INT */
30       if(qtdeLote == 1){
31         loteCostureira  = Math.round((float) Math.random() * 1);
32       }else{
33         loteCostureira = (int) (Math.random() * qtdeLote);
34       }
35
36       CostureiraHabilidade costureiraHabilidade =
37         atividadesCostureiras.get(key).get(cont);
38
39       ProcessoChromosome intermediario =
40         chromossomosMap.get(costureiraHabilidade.
             getIdCostureiraHabilidade());
41
42       if(intermediario == null){
43         ProcessoChromosome pc = new ProcessoChromosome
44           (key, costureiraHabilidade,loteCostureira);
45
46         chromossomosMap.put
47           (costureiraHabilidade.getIdCostureiraHabilidade(), pc);
48
49         chromosomes.add(pc);
50       }else{
51         int oldValue = intermediario.getQuantidade_lotes();
52         int newValue = oldValue + loteCostureira;
53         intermediario.setQuantidade_lotes(newValue);
54         intermediario.setLotesToShow(newValue);
55       }
```

```
56
57        if (cont == atividadesCostureiras.get(key).size() - 1) {
58          cont = -1;
59          distribuiuPorTodasCostureiras = true;
60        }
61
62        qtdeLote -= loteCostureira;
63
64        if(qtdeLote == 0 && distribuiuPorTodasCostureiras){
65          break;
66        }
67        cont++;
68      }
69    }
70  }
```

Conforme descrito no Código 5, é feita uma iteração no `HashMap` e, para cada atividade, é feita a distribuição dos lotes para as costureiras desta lista. O algoritmo então atribui, a cada costureira, um valor que pode variar de 0 até `qtdeLote`. Assim são criados objetos da classe `ProcessoChromosome` e colocados na lista de cromossomos do indivíduo. Após a criação de cada cromossomo, a quantidade de lotes é subtraída pelo valor atribuído ao cromossomo recém-criado. Se a iteração passar por todas as costureiras da lista, o contador é reiniciado e então, se ao final sobrar lotes a serem distribuídos, a distribuição recomeça na primeira costureira, acrescentando assim seu número de lotes de acordo com o novo valor sorteado. A iteração termina quando não há mais lotes a serem distribuídos e a execução já passou por todas as costureiras. Em uma primeira versão, quando a execução chegava ao final da lista de costureiras e ainda existiam lotes a serem distribuídos, este restante de lotes era atribuído à última costureira, porém, verificou-se que, dessa forma, o algoritmo tendia a nunca distribuir zero lotes à última costureira, o que causou resultados ineficazes na distribuição. Alterando para a segunda versão, a distribuição passou a ser feita de maneira totalmente uniforme, deixando o resultado coerente. Como é possível perceber, nesse processo uma costureira pode receber aleatoriamente o valor 0, o que irá resultar na sua eliminação do processo, da mesma forma que iria ocorrer na primeira abordagem. Por fim, após a finalização do primeiro `for`, um novo indivíduo terá sido criado, semelhante ao quadro apresentado na Figura 8.

Concluindo, a distribuição das atividades ocorre todas as vezes que se cria um novo indivíduo. Tais indivíduos podem ser criados no processo de criação da população inicial, na criação de indivíduos estrangeiros e no processo de cruzamento, como será descrito nas próximas subseções, ressaltando porém que, no processo de cruzamento, os cromossomos do indivíduo são a mistura dos cromossomos dos pais, já criados anteriormente, e portanto há também um construtor na classe `ProcessoIndividual` que recebe

uma lista de cromossomos para se criar um novo indivíduo. Este processo será demonstrado na seção que descreve o cruzamento.

3.4.4 Função de avaliação

Após a criação da população inicial, esta é então submetida a um processo de avaliação. Assim é feita uma iteração sobre a lista de indivíduos e para cada um é chamado então o seu método `calculateValue()`. Tal método é declarado de forma abstrata na classe mãe `Individual` e implementado na classe `ProcessoIndividual`.

Assim como já foi visto anteriormente, cada costureira sabe fazer uma ou mais partes da calça e gasta um determinado tempo, medido em segundos, além de cobrar um valor para se produzir cada peça, que varia de acordo com a habilidade, além disto, existe um tempo de transporte entre cada costureira que é medido através da distância euclidiana, ou seja, cada costureira recebe um valor X e Y que representam sua localização, e então para se calcular a distância entre duas costureiras, os valores X e Y de cada uma delas são inseridos como variáveis na fórmula euclidiana, obtendo-se assim a distância entre elas. A Figura 14 demonstra exemplo de dados a serem considerados para o cálculo do valor do indivíduo.

Habilidade	Costureira	Tempo/Peça	Preço/Peça	Posição X	Posição Y
Finalização	Josi	3s	R$ 1,50	30	10
Carimbo	Marcelo	0s	R$ 0,00	10	20
Frente	Roberta	4s	R$ 1,40	20	30
Frente	Tereza	2s	R$ 2,50	15	35
Frente	Maria	3s	R$ 1,70	25	43

Figura 14 – Demonstração de costureiras e habilidades. **Fonte:** Desenvolvido pelos autores.

Com base nesses dados é realizado um cálculo a fim de se encontrar o custo e o tempo total de fabricação do número de peças desejado e, ao fim, um valor de tempo e custo de produção será atribuído ao indivíduo. É importante ressaltar que a função de avaliação tem a responsabilidade apenas de definir o valor do indivíduo, que nesse caso é o tempo e o custo de produção, a escolha dos indivíduos mais aptos será feita posteriormente no processo de classificação.

Para o desenvolvimento do cálculo do tempo de produção, foi necessário construir uma estrutura para representar a questão da ordem de precedência entre as atividades. Tal estrutura, conforme é demonstrado na Figura 15, deveria possuir nós representando cada atividade, as costureiras que trabalham em cada atividade e o número de lotes atribuídos a cada uma, aleatoriamente, pelo algoritmo.

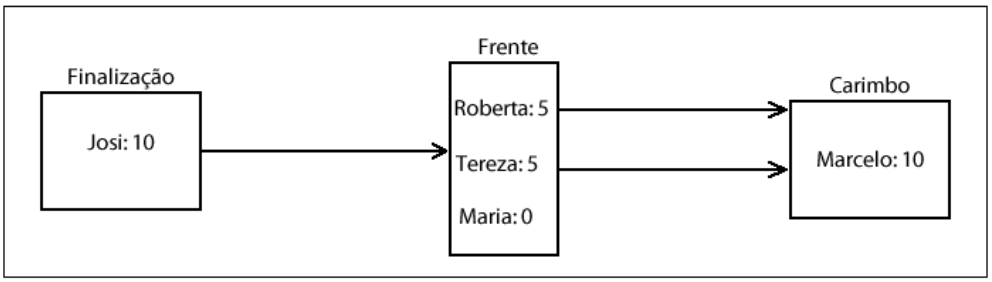


Figura 15 – Estrutura de representação da ordem de precedência. **Fonte:** Desenvolvido pelos autores.

Como foi visto na seção 3.4.2, cada indivíduo possui uma lista de cromossomos, e cada cromossomo, por sua vez, representa a alocação de uma costureira, contendo os lotes que esta deve produzir para uma determinada atividade. Desta forma, para calcular o custo total de produção, foi realizada uma iteração na lista de cromossomos do indivíduo, somando o preço total de confecção dos lotes que cada costureira recebeu aleatoriamente pelo algoritmo, conforme mostra o Código 6.

Para o cálculo do tempo de produção, definiu-se então que tal lista de cromossomos deveria ser dividida de forma que se pudesse agrupar os cromossomos por atividade, estabelecendo, assim, a relação demonstrada na Figura 15, para que, por fim, o cálculo pudesse ser realizado. Para isso, conforme demonstrado no Código 6, primeiramente, a lista de cromossomos foi distribuída em um `HashMap` denominado `atividadeCromossomos`, contendo como chave a atividade e como valor a lista de cromossomos para a respectiva atividade e foi criada uma classe denominada `Node`, sendo ela a responsável por criar a estrutura mostrada na Figura 15.

O método `calculateValue()`, após agrupar os cromossomos por atividade, instancia um objeto da classe `Node` passando a atividade final e o número de peças por lote recebidos na criação do indivíduo, conforme descrito na seção 3.4.3, e o `MAP` `atividadeCromossomos`, conforme mostra o Código 6.

Código 6 – Método `calculateValue()`. **Fonte:** Elaborado pelos autores.

```
1   public void calculateValue() {
2     Map<Integer, List<Chromosome>> atividadeCromossomos
3       = new HashMap<Integer, List<Chromosome>>();
4
5     Integer lastAtividade = null;
6     float custoTotal = 0;
7     int totalPecasAProduzir = 0;
8     List<Chromosome> cromossomos = null;
9
10    /*Calcular o custo total*/
11    for(Chromosome chromosomeCusto : chromosomes){
12      totalPecasAProduzir = 0;
13      ProcessoChromosome processoChromossome =
14        (ProcessoChromosome) chromosomeCusto;
```

```
15
16        totalPecasAProduzir =
17          processoChromossome.getLotesToShow() * this.pecasPorLote;
18
19        custoTotal += totalPecasAProduzir *
20          processoChromossome.getCostureiraHabilidade()
21            .getPrecoPorPeca();
22      }
23
24      for (Chromosome chromosome : chromosomes) {
25      ProcessoChromosome processoChromossome =
26        (ProcessoChromosome) chromosome;
27
28        if (lastAtividade == null || lastAtividade !=
29          processoChromossome.getAtividade()) {
30
31          cromossomos = new ArrayList<Chromosome>();
32
33          atividadeCromossomos.put
34            (processoChromossome.getAtividade(),cromossomos);
35          lastAtividade = processoChromossome.getAtividade();
36        }
37        cromossomos.add(processoChromossome);
38      }
39      node = new Node(
40        atividadeFinal, atividadeCromossomos,this.pecasPorLote);
41
42      setCusto(custoTotal);
43
44
45      /*So deve-se calcular o valor do individuo se
46        ele nao foi calculado ainda porque uma vez calculado
47        o valor o numero de lotes no objeto cromossomo foi
48        decrementado */
49      if (this.getTempoTotalProducao() == 0) {
50        this.setTempoTotalProducao(node.getTempoTotal());
51        setRootNode(node);
52      }
53    }
```

A estrutura da classe `Node` foi realizada de forma a produzir objetos de si mesma de forma recursiva, assim cada vez que ela é instanciada, é como se criasse um nó daqueles mostrados na Figura 15. Então, quando o método `calculateValue()` instancia um objeto `Node`, outros nós são criados a partir do construtor de forma recursiva, e toda estrutura, como foi ilustrada na Figura 15, será criada. O Código 7 mostra a construção de um objeto `Node`.

Código 7 – Construtor da classe `Node`. **Fonte:** Elaborado pelos autores.

```
1   public Node(Atividade atividade,Map<Integer,List<Chromosome>>
2      atividadeCromossomos, int pecasPorLote){
3
4     this.atividade = atividade;
5     this.pecasPorLote = pecasPorLote;
6     cromossomos = atividadeCromossomos.
7       get(atividade.getIdAtividade());
8
```

```
9     Atividade atividadePredecessora = null;
10
11    for(AtividadeOrdem predecessora :
12      atividade.getAtividadeOrdemsForIdAtividade()){
13
14      atividadePredecessora =
15        predecessora.getAtividadePredecessora();
16
17      predecesoras.add(new Node(atividadePredecessora,
18        atividadeCromossomos,pecasPorLote));
19    }
20  }
```

A classe `Node` recebe em seu construtor um objeto de `Atividade` que, na primeira vez em que for instanciado, será a atividade final, o `MAP` com todos os cromossomos e o número de peças por lote, que será utilizado posteriormente. O construtor então armazena as informações e pega do `MAP` somente os cromossomos relacionados à atividade recebida no construtor e, por fim, faz uma iteração na lista de atividades predecessoras de tal atividade e, recursivamente, cria novos nós, construindo assim, a estrutura demonstrada na Figura 15.

Como se pode ver no método `calculateValue()` no Código 6, após criar a estrutura de nós, é chamado o método `getTempoTotal()` do objeto `node`. Este método é responsável por iniciar a sequência lógica que faz o cálculo do tempo total a ser gasto pelo indivíduo, calculando o tempo gasto por cada costureira, definindo quem irá enviar peças para quem e calculando o tempo de transporte de cada envio, conforme demonstra o Código 8 .

Código 8 – Método `getValorTotal()`. **Fonte:** Elaborado pelos autores.

```
1   public long getTempoTotal(){
2     long valor =  0;
3     long maior = 0;
4
5     for(Chromosome chromosome : cromossomos){
6       ProcessoChromosome processoChromosome =
7         (ProcessoChromosome) chromosome;
8
9       if(processoChromosome.getQuantidade_lotes() > 0){
10        valor = getCromossomeValue(processoChromosome,
11          processoChromosome.getQuantidade_lotes());
12      }
13
14      if(valor > maior){
15        maior = valor;
16      }
17    }
18    return maior;
19  }
```

O método faz uma iteração na lista de cromossomos do nó da atividade final e

irá chamar o método `getChromosomeValue()`, passando cada cromossomo e o valor de seus lotes, e irá retornar o valor do maior cromossomo.

Tomando como base a Figura 15, para facilitar o entendimento, o método `getChromosomeVal` será chamado, passando o cromossomo "Josi"e o inteiro 10 na quantidade de lotes. Este método é responsável por calcular o tempo gasto pela costureira para confeccionar os lotes atribuídos a ela. O tempo gasto pela costureira é definido por `NLC * QPL * TP`, onde: `NLC` é o número de lotes atribuídos para a costureira, `QPL` é a quantidade de peças por lote e o `TP` é o tempo que a costureira gasta para fazer cada peça, porém este tempo também é influenciado pelo tempo que se é gasto para receber as partes dependentes, conforme demonstra o Código 9.

Código 9 – Método `getCromossomeValue()`. **Fonte:** Elaborado pelos autores.

```
1
2 public long getCromossomeValue(ProcessoChromosome
    processoChromosome,
3   int qtdeLote){
4
5   long valor = 0;
6   valor = qtdeLote * this.pecasPorLote *
7      processoChromosome.getCostureiraHabilidade().getTempoPorPeca
          ();
8
9   valor += getTempoRecebimentoPecas(processoChromosome,qtdeLote);
10  return valor;
11 }
```

O método `getChromossomeValue()` chama então o método `getTempoRecebimentoPecas(` passando o cromossomo Josi e o inteiro 10 como quantidade de lote. O método chamado tem a função de fazer uma busca nos nós predecessores, buscando encontrar qual o tempo gasto para o recebimento das partes predecessoras da atividade e retornar o maior valor, conforme demonstra o Código 10.

Código 10 – Método `getTempoRecebimentoPecas()`. **Fonte:** Elaborado pelos autores.

```
1
2   public long getTempoRecebimentoPecas(
3     ProcessoChromosome processoChromosome, int qtdeLote){
4
5     long valor = 0;
6     long maior = 0;
7
8     for(Node node : predecesoras){
9       valor = node.getValueChromosomosPredecessores(
10        processoChromosome, qtdeLote);
11
12      if(valor > maior){
13        maior = valor;
14      }
15    }
16    return maior;
```

```
17    }
18 }
```

Neste ponto começa então um processo recursivo, pois é chamado um método da própria classe `Node`, só que de uma outra instância. O método chamado é o `getValueChromosomosPredecessores()`, passando o cromossomo Josi e o inteiro 10 como número de lotes. O código 11 mostra este método.

Código 11 – Método `getValueChromosomosPredecessores()`. **Fonte:** Elaborado pelos autores.

```
1
2   public long getValueChromosomosPredecessores(
3     ProcessoChromosome processoChromosome,int qtdeLote){
4
5     int qtdeEachCromossome = 0;
6     long valor = 0;
7     long maior = 0;
8     long distance = 0;
9
10    for(Chromosome chromosome : cromossomos){
11      ProcessoChromosome processoChromosomeBefore =
12         (ProcessoChromosome) chromosome;
13
14      qtdeEachCromossome =
15         processoChromosomeBefore.getQtdeLotes(qtdeLote);
16
17      if(qtdeEachCromossome > 0){
18        qtdeLote -= qtdeEachCromossome;
19        valor = getCromossomeValue(
20          processoChromosomeBefore, qtdeEachCromossome);
21
22        distance = calcularTempoEntreCostureiras(
23          processoChromosome, processoChromosomeBefore);
24
25        processoChromosome.addCostureiraPredecessora(
26          processoChromosomeBefore.getCostureiraHabilidade(),
27
28        qtdeEachCromossome,distance,valor);
29
30        valor += distance;
31      }
32
33      if(valor > maior){
34        maior = valor;
35      }
36
37      if(qtdeLote == 0){
38        break;
39      }
40    }
41    return maior;
42  }
```

Tal método é responsável por iterar sobre a lista de cromossomos do nó anterior, buscando de qual ou quais costureiras podem ser obtidos os lotes, retornando o maior valor. No exemplo da Figura 16, a atividade anterior é a frente e a primeira costureira da

lista é a Roberta, assim a Josi solicita 10 lotes da parte da frente para a Roberta, porém, neste caso, a Roberta confeccionou somente 5 lotes, então a Josi irá consumir os 5 lotes confeccionados.

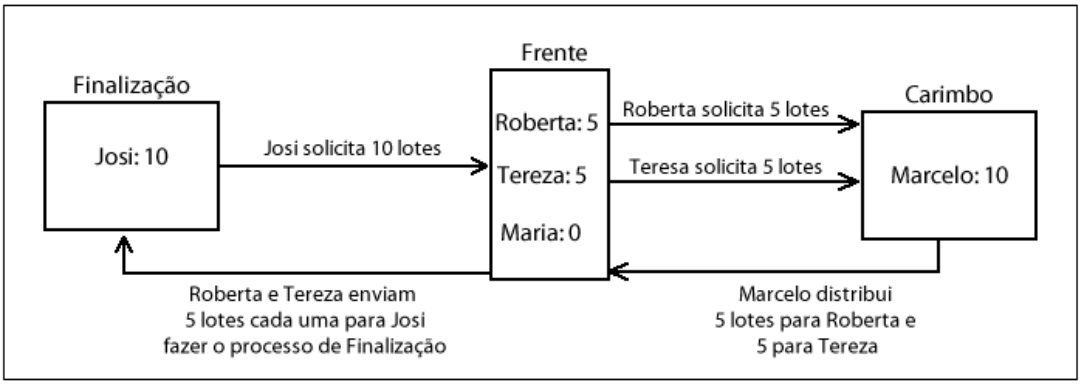


Figura 16 – Exemplo de solicitação de lotes predecessores. **Fonte:** Desenvolvido pelos autores.

Assim, a costureira Roberta é adicionada ao cromossomo Josi como costureira predecessora e novamente é chamado o método `getChromosomeValue()`, só que agora passando o cromossomo Roberta e a quantidade de lote que ela deve produzir para atender a Josi, para que se possa calcular o tempo, além disso, será chamado o método `calcularTempoEntreCostureiras()` que irá retornar o tempo de transporte entre a Josi e a Roberta, e será somado ao valor retornado de `getChromosomeValue()` que foi chamado passando o cromossomo Roberta. O Código 12 demonstra este método `calcularTempoEntreCostureiras()`.

Código 12 – Método `calcularTempoEntreCostureiras()`. **Fonte:** Elaborado pelos autores.

```
1
2   public long calcularTempoEntreCostureiras(
3     ProcessoChromosome processoChromosome,
4     ProcessoChromosome processoChromosomeBefore){
5
6     int posicaoCostureiraX = processoChromosome.
7       getCostureiraHabilidade().
8       getCostureira().getPositionX();
9
10    int posicaoCostureiraY = processoChromosome.
11      getCostureiraHabilidade().
12      getCostureira().getPositionY();
13
14    int posicaoCostureiraBeforeX = processoChromosomeBefore.
15      getCostureiraHabilidade().
16      getCostureira().getPositionX();
17
18    int posicaoCostureiraBeforeY = processoChromosomeBefore.
19      getCostureiraHabilidade().
20      getCostureira().getPositionY();
21
22    long distance = (long) Math.sqrt(
23    Math.pow(posicaoCostureiraX - posicaoCostureiraBeforeX, 2)+
24    Math.pow(posicaoCostureiraY - posicaoCostureiraBeforeY, 2));
25
26    return distance * 100;
27  }
```

Resumindo, o fluxo de execução dos métodos então será `getCromossomeValue()`, `getTempoRecebimentoPecas()`, `getValueChromosomosPredecessores()` e neste ponto o cromossomo Roberta irá solicitar para sua atividade anterior 5 lotes, conforme mostra a Figura 16.

Nesse caso, a atividade anterior é o carimbo, nesse ponto existe uma exceção, pois quando se solicita a atividade carimbo, não são consumidos os lotes dela, pois, conforme definido no escopo explicado anteriormente, a atividade de Carimbo só possui o Marcelo como trabalhador e ele apenas distribui o material de costura, assim, qualquer solicitação feita a ele é correspondida, além disso, o tempo de produção e o custo por peça do Marcelo é zero, sendo assim, será considerado apenas o tempo de transporte entre o Marcelo e a Roberta.

Dessa maneira, todo o processo é feito recursivamente, as costureiras da primeira atividade vão consumindo os lotes das costureiras das atividades predecessoras, conforme mostra a Figura 16. Como a Roberta não conseguiu atender a Josi, uma vez calculado o tempo de produção de 5 peças da Roberta, o algoritmo verifica a próxima costureira da atividade frente, que seria a Tereza neste exemplo e, da mesma forma, calcula o tempo de produção do número de peças solicitadas, que seria 5 nesse caso.

Concluindo, o processo é todo feito recursivamente, no qual as costureiras das primeiras atividades vão consumindo os lotes das costureiras das atividades predecessoras, de forma que o tempo de produção e transporte é calculado permanecendo sempre o maior valor, conforme mostra a Figura 17.

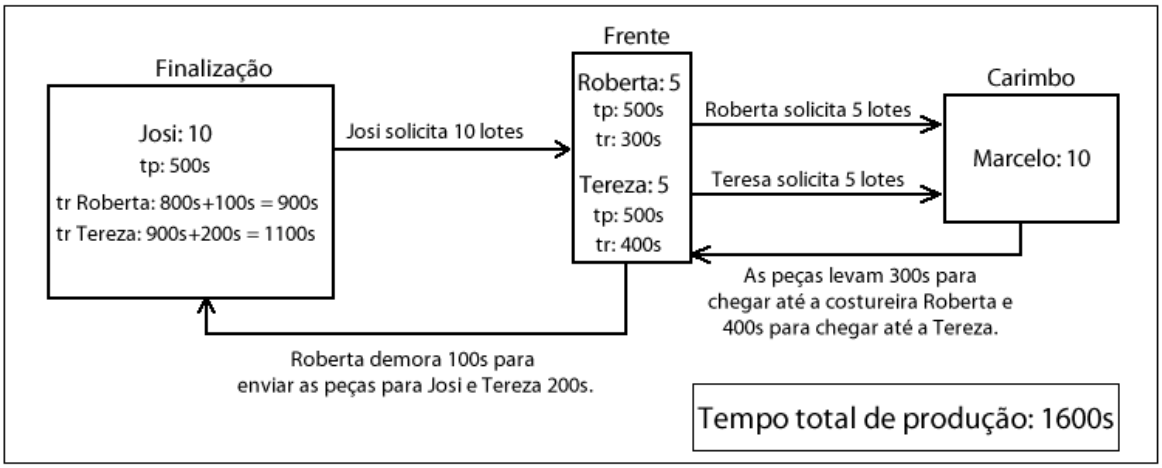


Figura 17 – Distribuição demonstrando o tempo. **Fonte:** Desenvolvido pelos autores.

A recursividade para quando se chega na atividade Carimbo, então os valores começam a ser retornados. No fim, prevalece sempre o maior valor do tempo de recebimento

dos lotes das atividades predecessoras e então tal valor é somado ao tempo de produção da finalização, o que, no exemplo da Figura 17, resultou em 1600 segundos, este valor então é colocado no atributo `tempoTotal` do indivíduo.

A função de avaliação é chamada dentro do método `classify`, dentro da classe `GAController`, tal método, por sua vez, é chamado dentro do método `execute()` após a criação da população inicial.

3.4.5 Classificação dos indivíduos

Uma vez definidos os valores de tempo total de produção e custo do indivíduo, o próximo passo a ser realizado, dentro da ordem de execução do algoritmo genético, é a classificação dos indivíduos. Tal classificação também é realizada no método `classify` da classe `GAController`, conforme mostra o Código 13.

Código 13 – Método `classify()`. **Fonte:** Elaborado pelos autores.

```
1  public void classify(List<Individual> population){
2    List<Individual> populationAux = new ArrayList<Individual>();
3    List<Individual> populationGood = new ArrayList<Individual>();
4
5    for(Individual individual : population){
6      individual.calculateValue();
7    }
8    Collections.sort(population);
9
10
11   for(Individual individual : population){
12
13     ProcessoIndividual pi = (ProcessoIndividual) individual;
14     if(individual.getValue() <= pi.getPrazo().longValue()){
15       individual.setTipoDeClassificacao(
16         Constants.CLASSIFICACAO_POR_CUSTO);
17
18       populationGood.add(individual);
19     }
20   }
21   Collections.sort(populationGood);
22
23   for(Individual individual : populationGood){
24     individual.setTipoDeClassificacao(
25       Constants.CLASSIFICACAO_POR_TEMPO);
26   }
27
28   //Remove todos os objetos com tempo < prazo da populacao
29   population.removeAll(populationGood);
30
31   //Adiciona-os na populacao auxiliar
32   populationAux.addAll(populationGood);
33
34   //adiciona os demais individuos da populacao
```

```
35    populationAux.addAll(population);
36
37    //Redefine a populacaoo
38    population.clear();
39    population.addAll(populationAux);
40
41 }
```

Após chamar a função de avaliação para todos os indivíduos, através do método `calculateValue()`, o método de ordenação `Collections.sort()` é chamado, passando a lista de indivíduos (população). Para que se possa utilizar este método de ordenação, a classe, que representa o tipo da lista, deve implementar a interface `Comparable` e, ao implementá-la, obrigatoriamente o método `compareTo()` deve ser implementado. Este método define qual será o critério de comparação durante a ordenação da lista. No caso do método abstrato `compareTo` da classe `Individual`, implementado na classe `ProcessoIndividual`, foi definido uma *flag* de forma a definir se os indivíduos devem ser ordenados pelo tempo total ou pelo custo, conforme mostra o Código 14.

Código 14 – Método `compareTo()`. **Fonte:** Elaborado pelos autores.

```
1
2    @Override
3    public int compareTo(Individual o) {
4      if(tipoDeClassificacao == Constants.CLASSIFICACAO_POR_CUSTO){
5        return Float.compare(getCusto(), o.getCusto());
6      }else{
7        return Float.compare(getTempo(), o.getTempo());
8      }
9    }
```

O atributo `tipoDeClassificacao` é iniciado com zero, ou seja, o padrão é que a classificação seja realizada levando em consideração o tempo total de produção de cada indivíduo. Desta forma, no Código 13, na linha 8, a ordenação da população será com base no tempo de produção. A questão é que, de acordo com o escopo definido, o objetivo do algoritmo seria encontrar um baixo tempo de produção associado a um custo minimizado, de forma que a melhor solução fosse aquela que permitisse a produção das calças no menor custo com um tempo menor ou igual ao prazo, esse prazo foi passado pelo usuário e atribuído ao indivíduo no construtor da classe `ProcessoIndivíduo`, conforme já explicado na seção anterior.

Dessa forma, de acordo com o Código 13, após a ordenação da população pelo tempo de produção, é feita uma verificação para buscar todos os indivíduos que possuem tempo de produção menor ou igual ao prazo e os indivíduos que antedem este requisito recebem o valor 1 no atributo `tipoDeClassificacao` (ordenação por custo), além disso

são adicionados a uma outra lista denominada populationGood, que também será ordenada, só que agora o critério de ordenação do método compareTo() passou ser o custo. Após esta ordenação, o atributo tipoDeClassificacao é zerado para evitar possíveis inconsistências em futuras ordenações e, com ajuda de listas auxiliares, os objetos da lista populationGood são transferidos para as primeiras posições da população e os demais indivíduos acrescentados ao final.

Sendo assim, os melhores indivíduos (primeiros colococados) irão ter sempre menor custo, procurando manter o tempo dentro do prazo, porém caso nenhum dos indivíduos tenham o tempo total menor ou igual ao prazo, então o melhor tempo é retornado e a ordenação por custo é ignorada.

3.4.6 Indivíduos estrangeiros e elitismo

Após a classificação dos indivíduos, o próximo passo a ser realizado pelo método execute() é verificar se o melhor indivíduo (primeiro da lista) da população atual é melhor que o da população anterior, caso positivo ou caso a população atual seja a população inicial, a variável lastBest recebe este melhor indivíduo e então é realizada uma verificação para checar se o número de gerações (atributo getGenerationQuantity) foi atingido, caso positivo, a execução é interrompida e o indivíduo armazenado na variável lastBest é retornado, ao contrário, a execução irá continuar de forma a iniciar a criação de uma nova população conforme mostra o Código 15.

Código 15 – Método execute(). **Fonte:** Elaborado pelos autores.

```
1   public Individual execute(){
2     model.createInitialPopulation();
3     Individual lastBest = null;
4
5     for(int i = 0; ; i++){
6       ArrayList<Individual> population = model.getPopulation();
7       ArrayList<Individual> newGeneration =
8         new ArrayList<Individual>();
9
10      classify(population);
11      if(lastBest == null || lastBest != population.get(0)){
12        lastBest = population.get(0);
13      }
14
15      //verifica o final da execucao
16      if(i == model.getGenerationQuantity()){
17        break;
18      }
19
20      //A partir deste ponto e iniciado a criacao de
```

```
21        //uma nova populacao
22
23        //elitismo
24        if(model.isElitism()){
25          doElistim(newGeneration);
26        }
27
28        int foreignQuantity = Math.round(
29          model.getPopulationSize() * model.
              getForeignIndividualRate());
30
31        foreignQuantity = foreignQuantity % 2 == 0 ?
32          foreignQuantity : foreignQuantity +1;
33
34        for(int j = 0;j < foreignQuantity;j++ ){
35          newGeneration.add(model.createIndividual());
36        }
37
38        while(newGeneration.size() < model.getPopulationSize()){
39          //selecao
40          Individual individual1 = doSelection();
41          Individual individual2;
42
43          do{
44            individual2 = doSelection();
45          }while(individual1 == individual2);
46
47          //cruzamento
48          IndividualPair pair = doCrossing(individual1, individual2
              );
49
50          //mutacao
51          doMutation(pair.getIndividual1());
52          doMutation(pair.getIndividual2());
53
54          newGeneration.add(pair.getIndividual1());
55          newGeneration.add(pair.getIndividual2());
56        }
57
58        model.setPopulation(newGeneration);
59      }
60      return lastBest;
61    }
```

O processo de criação de uma nova população inicia-se com o processo de elitismo. Tal processo, conforme já explicado anteriormente, consiste em adicionar à nova população os dois melhores indivíduos da população anterior. O próximo passo a ser executado é referente a adição de indivíduos à nova população, seguindo um conceito, existente no *framework*, explicado na seção 3.4.1, denominado indivíduos estrangeiros. Este conceito considera o fato de que, na natureza, durante o processo de geração de uma nova população, indivíduos estrangeiros podem começar a fazer parte de tal população.

No algoritmo genético, tais indivíduos simplesmente são introduzidos à nova população de acordo com uma taxa definida no atributo `foreignIndividualRate` da classe `GAModel` do *framework*, conforme já explicado anteriormente. Conforme demonstrado

na linha 35 do Código 15, os indivíduos estrangeiros são criados utilizando o método `createIndividual()` da classe `GAModel`, que foi implementado na classe `ProcessoModel`, tal método utiliza o mesmo construtor utilizado para criação de indivíduos da população inicial, portanto segue a mesma lógica utilizada, a diferença é que, ao criar tais indivíduos, o *looping* é executado de acordo com a taxa definida, o que irá resultar na criação de um número bem menor de indivíduos, comparado à quantidade de indivíduos criados durante a população inicial.

3.4.7 Seleção de indivíduos e cruzamento/reprodução

Após o processo de elitismo e a criação dos indivíduos estrangeiros, o restante da nova população é criado através do processo de cruzamento, conforme as instruções a partir da linha 39 do Código 15. O cruzamento começa com a seleção dos indivíduos que devem participar desse processo, essa seleção é realizada por meio do método `doSelection()`, mostrado no Código 16.

Código 16 – Método `doSelection()`. **Fonte:** Elaborado pelos autores.

```
1
2 private Individual doSelection() {
3   switch (model.getSelectionType()) {
4     case CLASSIFICATION : return doSelectionByClassification();
5     case ROULETTE: return doSelectionByRoulette();
6   }
7   return null;
8 }
```

O método verifica o atributo de configuração `selectionType` e, neste caso, o valor de tal atributo é `CLASSIFICATION`, conforme mostrado na configuração do algoritmo genético na classe `GeneticAlgorithmManagement`, explanada anteriormente, portanto o método chamado é o `doSelectionByClassification()`, conforme mostra o Código 17.

Código 17 – Método `doSelectionByClassification()`. **Fonte:** Elaborado pelos autores.

```
1   private Individual doSelectionByClassification() {
2     int maxValue = 0;
3
4     for(int i = 0; i< model.getPopulationSize(); i++){
5       maxValue += (i + 1);
6     }
7
8     double index = Math.random() * maxValue;
9     int cursor = 0;
10
```

```
11      for(int i = 0; i < model.getPopulationSize(); i++){
12        cursor += model.getPopulationSize() - 1;
13
14        if(index <= cursor){
15          return model.getPopulation().get(i);
16        }
17      }
18
19      return null;
20    }
```

Tal método é responsável pela seleção de um indivíduo para realização do cruzamento. Primeiramente é feita a soma dos índices da lista de indivíduos (`population`), realizando um `FOR` de 0 até `populationSize`, parâmetro que armazena o tamanho da população, assim, se o tamanho da população for 7, a soma seria 1+2+3+4+5+6+7, o que daria um total de 28. Feita essa soma, é sorteado então um número entre zero até a soma dos valores e então é realizado um novo `FOR` de 0 até `populationSize`, dentro dele é feita uma verificação, se o número sorteado for menor ou igual a ao `populationSize` (`cursor`), que neste caso é 7, então o indivíduo da posição `i` (zero) é escolhido, caso não seja, a variável cursor é incrementada como o valor do tamanho da população, que neste caso resultaria em 14 e assim a verificação é feita novamente até que o número sorteado seja menor ou igual ao cursor e, quando assim for, o indivíduo da posição `i` é retornado. A Figura 18 ilustra este processo.

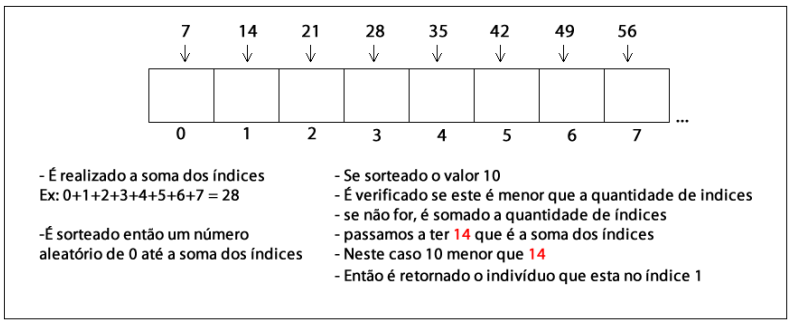


Figura 18 – Representação do processo de seleção dos indivíduos. **Fonte:** Desenvolvido pelos autores.

Ao finalizar este processo, um indivíduo terá sido escolhido, assim, de acordo com a linha 44 do Código 15, um segundo indivíduo deve ser selecionado, porém a instrução está dentro de um `do while`, pois o processo deve ser repetido até que o novo indivíduo selecionado seja diferente do anterior, evitando assim que dois indivíduos iguais façam parte do cruzamento.

O cruzamento é realizado em nível de cromossomos, de forma que os indivíduos filhos recebem a mesclagem dos cromossomos de seus pais. No cruzamento desenvolvido para esta solução, foi necessário realizar um agrupamento dos cromossomos e então criar

novos indivíduos através da mesclagem destes grupos. Isso foi necessário pois o total de lotes distribuídos entre os cromossomos de cada atividade não pode ser diferente do número total de lotes definido, assim o cruzamento então ocorre de forma que os indivíduos filhos recebem a mesclagem de grupos de cromossomos de seus pais referente à cada atividade e assim a cada cruzamento são criados dois novos indivíduos que irão fazer parte da nova população. A Figura 19 ilustra este processo.

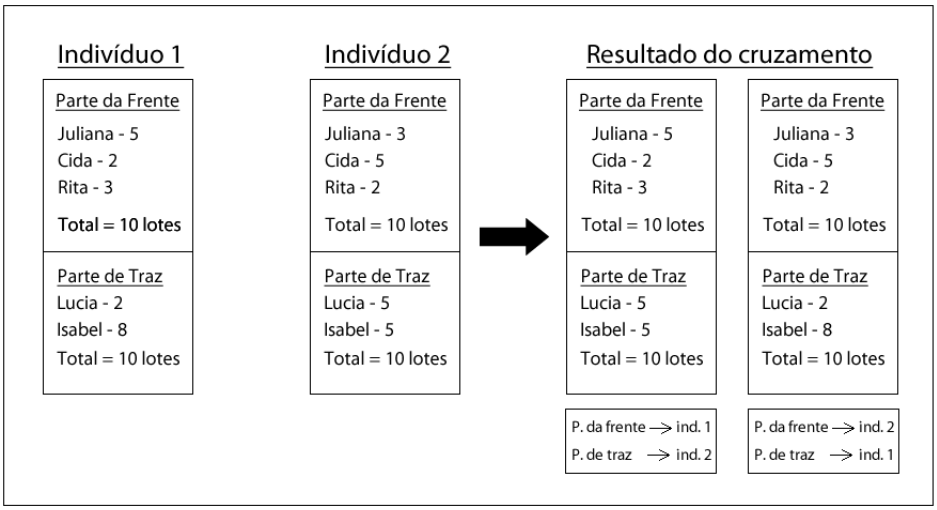


Figura 19 – Cruzamento dos indivíduos. **Fonte:** Desenvolvido pelos autores.

A partir da linha 48 do código Código 15, dois indivíduos já foram selecionados e então o método `doCrossing()` é chamado para realizar o cruzamento. Tal método também verifica o tipo de cruzamento definido, chamando o método correspondente, conforme mostra o Código 18.

Código 18 – Método `doCrossing()`. **Fonte:** Elaborado pelos autores.

```
1   private IndividualPair doCrossing(Individual individual1,
2     Individual individual2){
3
4     switch (model.getCrossType()) {
5       case ARITMETIC: break;
6
7       case BINARY:
8         return doBinaryCrossing(individual1,individual2);
9       case PERMUTATION:
10        return doPermutationCrossing(individual1, individual2);
11
12      case UNIFORM:break;
13    }
14    return null;
15  }
```

Nesse caso, o método escolhido para o cruzamento foi o de permutação, dessa forma o método `doPermutationCrossing()` é chamado. Tal método faz parte do *framework* de desenvolvimento, porém foi modificado para se adaptar ao problema a ser resolvido pela aplicação, conforme demonstrado no Código 19.

Código 19 – Método `doPermutationCrossing()`. **Fonte:** Elaborado pelos autores.

```
1     private IndividualPair doPermutationCrossing(
2       Individual individual1, Individual individual2){
3
4       ArrayList<Chromosome> chromosomes1 =
5         new ArrayList<Chromosome>();
6
```

```
7        ArrayList<Chromosome> chromosomes2 =
8          new ArrayList<Chromosome>();
9
10       Integer lastAtividade = null;
11       float chooseIndividual = 0;
12       int i = 0;
13
14       for(Chromosome chromosome : individual1.getChromosomes()){
15         ProcessoChromosome processoChromosome =
16           (ProcessoChromosome) chromosome;
17
18         if(lastAtividade == null ||
19           !lastAtividade.equals(processoChromosome.getAtividade()
               )){
20
21           chooseIndividual = Math.round((float) Math.random() *
               1);
22           lastAtividade = processoChromosome.getAtividade();
23         }
24
25         if(chooseIndividual == 1){
26           chromosomes1.add(individual1.getChromosomes().
27             get(i).clone());
28
29           chromosomes2.add(individual2.getChromosomes().
30             get(i).clone());
31         }else{
32           chromosomes1.add(individual2.getChromosomes().
33             get(i).clone());
34
35           chromosomes2.add(individual1.getChromosomes().
36             get(i).clone());
37         }
38         i++;
39       }
40
41       Individual newIndividual1 =
42         model.createIndividual(chromosomes1);
43
44       Individual newIndividual2 =
45         model.createIndividual(chromosomes2);
46
47       return new IndividualPair(newIndividual1,newIndividual2);
48     }
```

Conforme visto na Figura 19, os cromossomos estão ordenados na lista de acordo com suas respectivas atividades, considerando isso, foi feita uma lógica para agrupar estes cromossomos por atividade e criar novos indivíduos através da mesclagem dos grupos criados, de forma a sortear de qual indivíduo do cruzamento cada grupo deve ser pego ao criar um novo indivíduo.

Assim, de acordo com o Código 19, primeiramente é feita uma iteração sobre a lista de cromossomos do indivíduo 1, dentro de tal iteração, é realizada uma verificação para checar se um cromossomo é o primeiro da primeira atividade (`lastAtividade == null`) ou se é o primeiro de uma determinada atividade de acordo com a ordem delas.

Caso uma dessas condições for verdadeira, um valor binário é sorteado, se tal valor for um, o primeiro novo indivíduo que será criado terá os cromossomos da atividade em questão, pegos do indivíduo 1 do cruzamento, e o segundo novo indivíduo receberá os cromossomos, referentes à tal atividade, do indivíduo 2, caso o valor sorteado seja zero, o processo acontece de forma inversa.

Assim a iteração continua até a lista de cromossomos dos novos indivíduos possuir todos os grupos de cromossomos referentes a todas as atividades e então, ao fim da iteração, dois novos indivíduos serão criados através do método `createIndividual()`.

Quando se cria indivíduos a partir deste ponto, a classe `ProcessoModel` utiliza um outro construtor para criar o indivíduo, conforme mostra o Código 20.

Código 20 – Método `createIndividual()` utilizando cromossomos. **Fonte:** Elaborado pelos autores.

```
1   @Override
2   public Individual createIndividual(
3     ArrayList<Chromosome> chromosomes) {
4
5     return new ProcessoIndividual(atividadeFinal,
6        prazoEmSegundos, chromosomes,this.numeroLote,
7          this.pecasPorLote);
8   }
```

Nesse caso são passados quase todos os parâmetros enviados durante a criação da população inicial, a diferença é que o indivíduo será criado a partir dos cromossomos conforme mostra o Código 21.

Código 21 – Construtor de `ProcessoIndividual` utilizando cromossomos. **Fonte:** Elaborado pelos autores.

```
1 public ProcessoIndividual(Atividade atividadeInicial,
2               BigDecimal prazoEmSegundos,
3               ArrayList<Chromosome> chromosomes,
4               int numeroLote, int pecasPorLote) {
5
6   super(chromosomes);
7   this.atividadeFinal = atividadeInicial;
8   this.numeroLote = numeroLote;
9   this.pecasPorLote = pecasPorLote;
10  this.prazo = prazoEmSegundos;
11 }
```

Os cromossomos recebidos são inseridos na lista de cromossomos declarada na classe mãe (`Individual`), e os outros parâmetros são atribuídos aos seus respectivos atributos. No final do método `doPermutaionCrossing`, como no Java não é possível retornar dois valores, é criado um objeto da classe `IndividualPair`, passando os dois novos indivíduos, e então este objeto é retornado. Esta classe só possui dois atributos do tipo `Individual` e só é utilizada para retornar os dois novos indivíduos criados.

3.4.8 Mutação

O processo de mutação, conforme já explicado anteriormente, consiste em realizar uma pequena modificação entre os cromossomos de um indivíduo. Esta modificação pode ser boa, melhorando o resultado do indivíduo, ou ruim. No algoritmo genético desenvolvido, a mutação foi realizada trocando o número de lotes recebidos entre duas costureiras escolhidas aleatoriamente dentro de cada grupo de cromossomos (atividades) do indivíduo, conforme mostra a Figura 20.

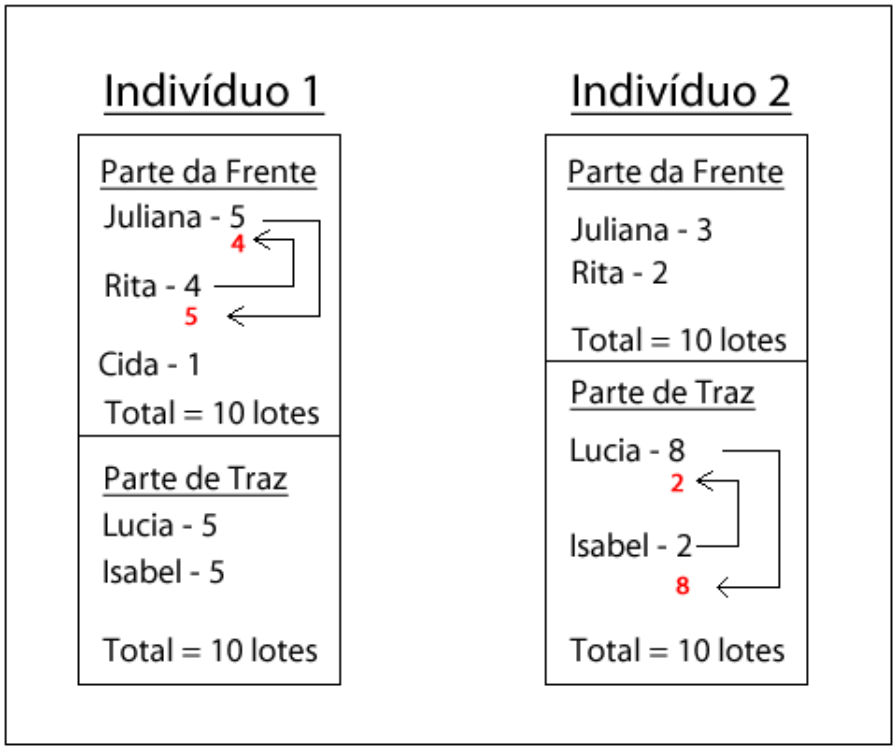


Figura 20 – Exemplo de mutação. **Fonte:** Desenvolvido pelos autores.

O processo de mutação inicia-se logo após cada cruzamento e é realizado nos indivíduos criados nesse processo, dentro de uma determinada taxa, definida anteriormente nos atributos de configuração do algoritmo genético na classe `GeneticAlgorithmManagement`. O método que inicia este processo é o `doMutation()`, que verifica o tipo de mutação definido, chamando o método correspondente, além disso, ele verifica se a mutação deve ser realizada, sorteando um número aleatoriamente e checando se o número sorteado é menor que a taxa de mutação definida, conforme mostra o Código 22.

Código 22 – Método `doMutation()`. **Fonte:** Elaborado pelos autores.

```
1  public void doMutation(Individual individual){
2    if(Math.random() < model.getMutationRate()){
3      for(int i = 0; i < model.getMutationQuantity();i++){
```

```
4          switch (model.getMutation()) {
5            case BINARY: doMutationBinary(individual);break;
6            case NUMERICAL: break;
7            case PERMUTATION: doMutationPermutation(individual);
               break;
8          }
9        }
10     }
11   }
```

Nesse caso, o método escolhido para a mutação foi o `PERMUTATION`, dessa forma o método `doMutationPermutation()` é chamado. Tal método faz parte do `framework` de desenvolvimento, porém também foi modificado para se adaptar ao problema a ser resolvido pela aplicação, conforme demonstrado no Código 23.

Código 23 – Método `doMutationPermutation()`. **Fonte:** Elaborado pelos autores.

```
1   public void doMutationPermutation(Individual individual) {
2
3     Integer lastAtividade = null;
4     ArrayList<ProcessoChromosome> chromossomesToMutate =
5       new ArrayList<ProcessoChromosome>();
6
7     for(Chromosome chromosome : individual.getChromosomes()){
8       ProcessoChromosome processoChromosome =
9         (ProcessoChromosome) chromosome;
10
11      if(lastAtividade == null ||
12           !lastAtividade.equals(processoChromosome.getAtividade
               ())){
13
14        if(!chromossomesToMutate.isEmpty() &&
15             chromossomesToMutate.size() > 1){
16
17          doMutationOnChromossome(chromossomesToMutate);
18        }
19        chromossomesToMutate.clear();
20        lastAtividade = processoChromosome.getAtividade();
21      }
22      chromossomesToMutate.add(processoChromosome);
23    }
24    //Para o ultimo grupo
25    if(!chromossomesToMutate.isEmpty() &&
26         chromossomesToMutate.size() > 1){
27
28      doMutationOnChromossome(chromossomesToMutate);
29    }
30  }
```

A lógica que faz a mutação inicia-se com uma iteração na lista de cromossomos do indivíduo no qual a mutação será realizada, dentro dessa iteração, os cromossomos de uma atividade são adicionados à lista `chromossomesToMutate`, e então, quando todos os cromossomos de uma atividade são adicionados nesta lista, o método `doMutationOnCromossome`, demonstrado no Código 24, é chamado para realizar a mutação, assim, após a mutação

para os cromossomos de tal atividade, a lista `chromossomesToMutate` é zerada para que os cromossomos da próxima atividade sejam adicionados a ela e posteriormente sofram a mutação, logicamente, o grupo de atividades sofrerá mutação somente se possuir mais de um cromossomo.

Código 24 – Método `doMutationOnChromossome()`. **Fonte:** Elaborado pelos autores.

```
1   private void doMutationOnChromossome(
2     ArrayList<ProcessoChromosome> chromossomesToMutate){
3
4     int position1;
5     int position2;
6     int varAux;
7     int varAux2;
8
9     ProcessoChromosome chromosome1 = null;
10    ProcessoChromosome chromosome2 = null;
11
12    position1 = (int) (Math.random() * (chromossomesToMutate.size
        ()));
13
14    do{
15      position2 = (int) (Math.random() *
16        (chromossomesToMutate.size()));
17    }while(position1 == position2);
18
19    chromosome1 = chromossomesToMutate.get(position1);
20    chromosome2 = chromossomesToMutate.get(position2);
21
22    varAux  = chromosome2.getQuantidade_lotes();
23    varAux2 = chromosome2.getLotesToShow();
24
25    chromosome2.setQuantidade_lotes(
26      chromosome1.getQuantidade_lotes());
27
28    chromosome2.setLotesToShow(chromosome1.getLotesToShow());
29
30    chromosome1.setQuantidade_lotes(varAux);
31    chromosome1.setLotesToShow(varAux2);
32  }
```

Nesse método, são selecionados quais cromossomos terão seus valores trocados entre si. O segundo cromossomo é escolhido dentro da estrutura de repetição `do While`, para evitar que o mesmo cromossomo escolhido na primeira vez seja selecionado e então, com auxílio de variáveis auxiliares, os atributos `quantidade_lotes` e `lotesToShow`, têm seus valores trocados entre si. O atributo `lotesToShow` é utilizado para espelhar a quantidade de lotes do indivíduo. Este atributo é necessário pois, no processo de avaliação dos indivíduos, as costureiras (cromossomos) das primeiras atividades consomem lotes dos cromossomos das atividades predecessoras, decrementando assim o atributo `quantidade_lotes` do cromossomo, assim o atributo `lotesToShow` é utilizado para manter o valor de lotes original dos cromossomos para mostrar no resultado da distribui-

ção quantos lotes cada costureira recebeu.

Concluindo, após a finalização do processo de cruzamento e possível mutação dos novos indivíduos, uma nova população terá sido criada, assim esta nova população substitui a população anterior e então todo o processo de classificação, verificação do melhor indivíduo, etc., é reiniciado, dessa forma sempre será criada uma nova população até que o número de gerações seja atingido e quando for, o melhor indivíduo é retornado, conforme explicado anteriormente.

3.4.9 Interface gráfica de distribuição

Para a interação do usuário com o algoritmo de distribuição e possibilidade de cadastro de fluxos de processos e costureiras, bem como suas respectivas habilidades, tempo e preço de produção, foi realizada uma aplicação em plataforma *web* utilizando *JSF* e *Primefaces*. A forma que o mecanismo de cadastro das informações foi realizado não é relevante ser apresentada, sendo importante, porém, explanar sobre o desenvolvimento da tela de distribuição das atividades, a qual é de extrema importância para entender o fluxo de execução da aplicação. No menu "Distribuir tarefas"da aplicação estão disponíveis todos os processos cadastrados e para abrir a tela de distribuição de atividades basta clicar no ícone "Abrir"da coluna opções, conforme mostra a Figura 21.

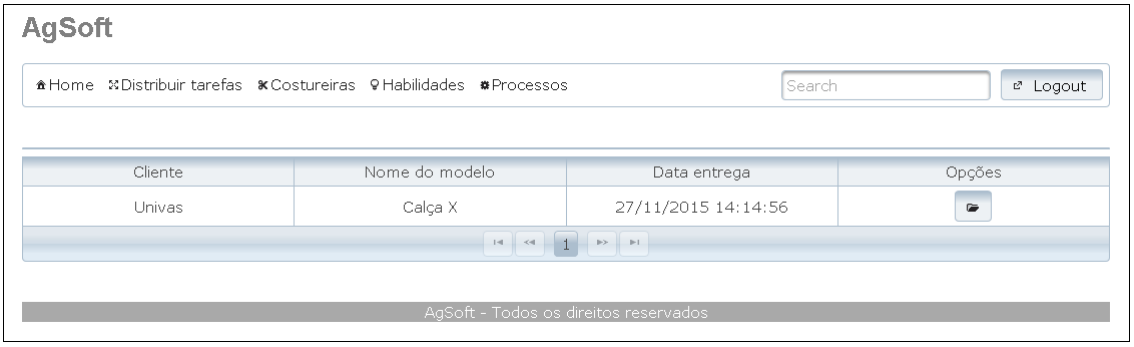


Figura 21 – Menu Distribuir tarefas. **Fonte:** Desenvolvido pelos autores.

Ao clicar no ícone "Abrir", é aberta uma tela de distribuição para aquele processo, que é responsável por captar alguns dados importantes para a regra de negócio do algoritmo genético, além disso, é nela que as informações do melhor indivíduo (melhor solução) são apresentadas. Os dados informados através dessa tela são o número total de peças que se deseja produzir, o número de peças que cada lote irá conter e a data de início da produção, conforme mostra a Figura 22.

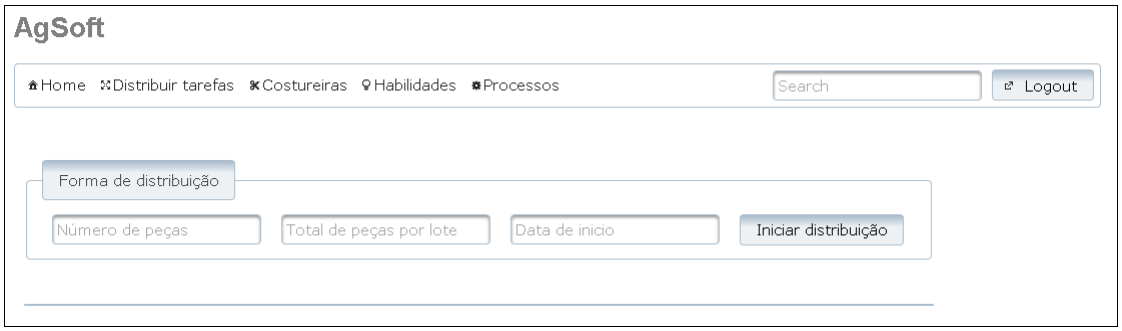


Figura 22 – Tela de distribuição de tarefas. **Fonte:** Desenvolvido pelos autores.

Toda tela criada com JSF é desenvolvida através de documentos XHTML, tais documentos possuem componentes para entrada e apresentação dos dados. Estes componentes podem ser tanto do JSF quanto do *primefaces*, o Código 25 demonstra o documento XHTML da tela demonstrada acima.

Código 25 – Documento XHTML da tela de distribuição. **Fonte:** Elaborado pelos autores.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ui:composition xmlns="http://www.w3.org/1999/xhtml"
3  xmlns:ui="http://java.sun.com/jsf/facelets"
4  xmlns:f="http://java.sun.com/jsf/core"
5  xmlns:h="http://java.sun.com/jsf/html"
6  xmlns:p="http://primefaces.org/ui">
7
8  <h:form id="formAbrirProcessoDis">
9    <p:messages autoUpdate="true"></p:messages>
10
11   <p:panelGrid columns="1">
12
13     <div id="distribuicaoFields">
14       <p:fieldset legend="Forma de distribuicao">
15         <p:panelGrid columns="4">
16           <p:inputText id="nPecas" required="true"
17           value="#{distribuicaoController.totalPecas}"
18           requiredMessage="Por favor informe o total de pecas"
19           maxlength="19" placeholder="Numero de pecas" />
20
21           <p:inputText id="totalPecaPorLote" required="true"
22           value="#{distribuicaoController.totalPecasPorLote}"
23           requiredMessage="Por favor informe o total
24             de pecas por lote"
25
26           maxlength="19" placeholder="Total de pecas por lote"/>
27
28           <p:calendar id="inicio" required="true"
29                 mask="true" effect="fold"
30                 value="#{distribuicaoController.dataInicio}"
31                 requiredMessage="Por favor informe uma data de
                      inicio"
32                 pattern="dd/MM/yyyy HH:mm:ss"
33                   mindate="#{processosController.getCurrentDate()
                        }"
34
35                 maxdate="#{distribuicaoController.getDataEntrega
                      ()}"
```

```
36                 placeholder="Data de inicio"/>
37
38          <p:commandButton action="#{distribuicaoController.
39            iniciarDistribuicao()}"
40
41          value="Iniciar distribuicao"
42          update="formAbrirProcessoDis"
43          ajax="true" onstart="PF('waitDialog').show();"
44          onsuccess="PF('waitDialog').hide();"/>
45
46        </p:panelGrid>
47      </p:fieldset>
48    </div>
49  .
50  .
51  .
```

No JSF, toda página XHTML possui um controlador que consiste em uma classe Java, e para definir que tal classe é uma controladora de páginas, na sua declaração é definida a anotação `@ManagedBean(name = "[nome da classe em minusculo]")`. No caso da tela de distribuição, a classe controladora é denominada `DistribuiçãoController`, portanto possui a notação `@ManagedBean(name = "[distribuiçãoController]")` em sua declaração. As páginas XHTML fazem então um *bind*, ou seja, uma conexão com a classe controladora de forma que os dados do XHTML são disponibilizados em tal classe e vice-versa. No caso da página de distribuição, as *tags* `p:inputText` e `p:calendar` são responsáveis pela entrada de informações, tais *tags* possuem um atributo `value`, neste atributo é informado em qual classe controladora e em qual atributo dela os dados informados pelo usuário serão atribuídos, realizando assim o *bind*. Nesse caso, os dados informados pelo usuário estarão respetivamente disponíveis nos atributos `totalPecas`, `totalPecasPorLote` e `dataInicio` da classe `DistribuiçãoController`.

Após entrar com os dados necessários, o usuário deve clicar no botão "Iniciar Distribuição", este botão é construído com a *tag* `p:commandButton` conforme mostra o Código 25. Esta *tag* possui o atributo `action`, que é responsável por chamar um método da classe controladora para executar uma determinada ação, nesse caso, o método chamado é o `iniciarDistribuicao()`, responsável por iniciar a distribuição das atividades chamando o gerenciador do algoritmo genético, conforme mostra o Código 26.

Código 26 – Método `iniciarDistribuicao()`. **Fonte:** Elaborado pelos autores.

```
1
2 public void iniciarDistribuicao(){
3   int totalPecasInt = Integer.parseInt(totalPecas);
4   int totalPecasPorLoteInt = Integer.parseInt(totalPecasPorLote);
5
6   if(totalPecasInt == 0 || totalPecasPorLoteInt == 0){
7     sendMessageToView(
```

```
8        "Total de pecas ou total de pecas por lote e invalido!",
9          FacesMessage.SEVERITY_ERROR);
10     return;
11   }
12
13   numeroDeLotes = totalPecasInt / totalPecasPorLoteInt;
14   if((numeroDeLotes * totalPecasPorLoteInt) != totalPecasInt){
15     sendMessageToView("Numero de lote nao exato: "+numeroDeLotes+
         " * "
16       + totalPecasPorLote+" = "+numeroDeLotes*
           totalPecasPorLoteInt,
17         FacesMessage.SEVERITY_ERROR);
18     return;
19   }
20
21   if(idProcesso <= 0 || processo == null){
22     sendMessageToView(
23       "Processo invalido", FacesMessage.SEVERITY_ERROR);
24
25     return;
26   }
27
28   prazEmSegundos = calcularPrazo();
29   prazoAtendido = false;
30
31   GeneticAlgorithmManagement gam = new GeneticAlgorithmManagement
       ();
32   melhorIndividuo = gam.iniciarDistribuicao(
33     numeroDeLotes, prazEmSegundos, totalPecasPorLoteInt,
         idProcesso);
34
35   //This is just to manage information messages in the view
36   if(melhorIndividuo.getValue() <= prazEmSegundos.longValue()){
37     prazoAtendido = true;
38   }
39
40   if(melhorIndividuo != null){
41     rootFluxograma = construirArvore(melhorIndividuo.getNode(),
         null);
42     allNodes = new ArrayList<Node>();
43     rootTable  =
44       new DefaultTreeNode(
45         new TabDetailBean("-", "-", "-","-","-","-","-"));
46
47     construirTableDetail(melhorIndividuo);
48     mostrarResult = true;
49   }
50 }
```

Os dados que o algoritmo genético espera receber são: o número de lotes a serem distribuídos, o número de peças por lote e o prazo de entrega. Portanto, uma das funções do método `iniciarDistribuicao()` é calcular a quantidade de lotes a serem produzidos e o prazo de entrega das peças, além disso, o método realiza algumas validações, de forma a somente enviar informações consistentes para o algoritmo genético. O cálculo de número de lotes é realizado então dividindo o número de peças informado pelo usuário pelo número de peças por lote, o resultado é armazenado na variável `numeroDeLotes`.

Para calcular o prazo de entrega da produção é chamado o método `calcularPrazo()`, demonstrado no Código 27.

Código 27 – Método `calcularPrazo()`. **Fonte:** Elaborado pelos autores.

```
1
2 public BigDecimal calcularPrazo(){
3   DateTime dataInical = new DateTime(dataInicio);
4   DateTime dataFinal  = new DateTime(processo.getDataEntrega());
5   return  new BigDecimal(Seconds.secondsBetween(dataInical,
        dataFinal).getSeconds());
6 }
```

É importante observar que o prazo é calculado em segundos, devido ao fato de que o tempo de produção por peça e o tempo de transporte entre as costureiras são definidos em segundos. O método `calcularPrazo()` simplesmente retorna o tempo em segundos entre a data de início, informada pelo usuário e disponível no atributo `dataInicio`, e o prazo de entrega, disponível no atributo `dataEntrega` do processo que foi informada no cadastro do mesmo. É importante ressaltar que o objeto `processo` utilizado neste método foi criado no momento em que o usuário abriu o processo, nesse caso, foi enviado o `ID` do tal processo via `URL` e o controlador buscou no banco de dados as informações do mesmo através deste `ID`, criando assim tal objeto.

Após o calculo do número de lotes e do prazo, é instanciado um objeto da classe `GeneticAlgorithmManagement`, conforme explicado anteriormente, e então o método `iniciarDistribuicao()` é chamado. Tal método, conforme já visto nas seções anteriores, faz o gerenciamento da execução do algoritmo genético e retorna o melhor indivíduo encontrado, tal indivíduo é armazenado no atributo `melhorIndividuo` da classe `distribuiçãoController` e então os métodos `construirArvore()` e `construirTableDetail()` são chamados para realizar a apresentação dos dados do indivíduo na tela após a distribuição, tais métodos simplesmente apresentam ao usuário as atividades e suas respectivas costureiras juntamente com o número de lote que cada uma recebeu, além disto, também são apresentados o tempo e o custo total, além de uma tabela de detalhes da distribuição. A Figura 23 mostra a tela com o resultado da execução do algoritmo.

Os resultados serão vistos com mais detalhes nos casos de testes apresentados na discussão de resultados.

Resultado da distribuição

Total de lotes definidos: 10
Tempo total: 3200.0 segundos - Custo: R$ 750.0
O menor tempo encontrado está associado ao menor custo dentro do prazo

Início: 08/11/2015 19:32:32 - Entrega: 27/11/2015 14:14:56
Prazo total: 1622544 segundos

<Finalização> (Maria: 10 Duda: 0) — <Carimbo> (Marcelo: 10)

Detalhes da distribuição

Atividade/Costureira	Parte	Qtde Lotes	Peç lote	Tot peças	T. peca	T.producao	T. Envio	T. Recebimento	T. total
AT_Finalização	-	-	-	-	-	-	-	-	
Maria	-	10	50	500	5	2500	-	700	3200
Marcelo	Carimbo	-	-	-	-	0	700	-	700
Duda	-	0	50	0	5	0	-	0	0

Figura 23 – Tela de resultado de distribuição de tarefas. **Fonte:** Desenvolvido pelos autores.

4 DISCUSSÃO DE RESULTADOS

Neste capítulo serão apresentados e discutidos os resultados obtidos pela pesquisa e implementação do sistema de otimização do processo de fabricação de calças. Tal discussão será realizada em forma de casos de teste, buscando demonstrar o comportamento do algoritmo genético de diferentes formas.

Desde o princípio, quando começou-se a discutir sobre o tema do trabalho de conclusão de curso, teve-se a ideia de desenvolver algo relacionado à inteligência artificial, por ser um assunto de bastante relevância na área de desenvolvimento de software. Dentro desse campo então, realizando algumas buscas na internet, foi encontrado o assunto de algoritmos genéticos. Coincidentemente foi lecionada, no primeiro semestre deste ano, a disciplina "sistemas especialistas", na qual o assunto foi abordado, o que facilitou bastante o aprendizado.

Assim, como sugestão do professor orientador, foi decidido então desenvolver uma aplicação para otimizar um processo de distribuição de atividades entre costureiras para um microempresário da cidade de Cachoeira de Minas - MG, do ramo de costura. Constatamos que um sistema *web* seria mais cômodo de ser utilizado, por não precisar de nenhuma instalação por parte do usuário e ainda poder ser acessado de qualquer lugar, desde que estivesse conectado à internet.

Assim sendo, foram adotadas como tecnologias para o desenvolvimento em plataforma Web os *frameworks* JSF e Primefaces. Além disso, foi utilizado um *plug-in* denominado `JBoss Tools`, o qual foi de grande utilidade para gerar as classes modelos a partir do banco de dados.

Inicialmente foi tomado como base o caso do microempresário citado acima, todavia, logo após iniciarmos o trabalho, o proprietário fez uma reestruturação de processos em sua empresa. Portanto, sua maneira de trabalhar deixou de ser um cenário no qual algoritmos genéticos pudessem ser aplicados, consequentemente, fechou-se um escopo para que o trabalho pudesse continuar, conforme descrito na seção 3.3.2 do quadro metodológico. Com o escopo definido, o foco passou a ser na definição da estrutura dos elementos do algoritmo genético.

Posto isso, seguindo os passos descritos no quadro metodológico, foi obtido como resultado a aplicação capaz de distribuir atividades de forma a se obter o menor custo e o menor tempo de produção, alcançando assim os objetivos específicos. Com a aplicação

finalizada, foram realizados então casos de testes, a fim de colocar em prova a eficiência da ferramenta para se buscar melhores soluções nas distribuição de tarefas, conforme mostram as seções seguintes.

4.1 Teste considerando somente o tempo de produção

Este teste demonstra a distribuição de lotes levando em consideração o tempo de cada costureira para fabricação das peças. Neste teste será definido o preço por peça igual para as costureiras alterando somente o tempo por peça de cada uma. Para este teste foi cadastrado um processo de produção informando o cliente, o modelo da calça e a data e hora da entrega conforme mostra a Figura 24.

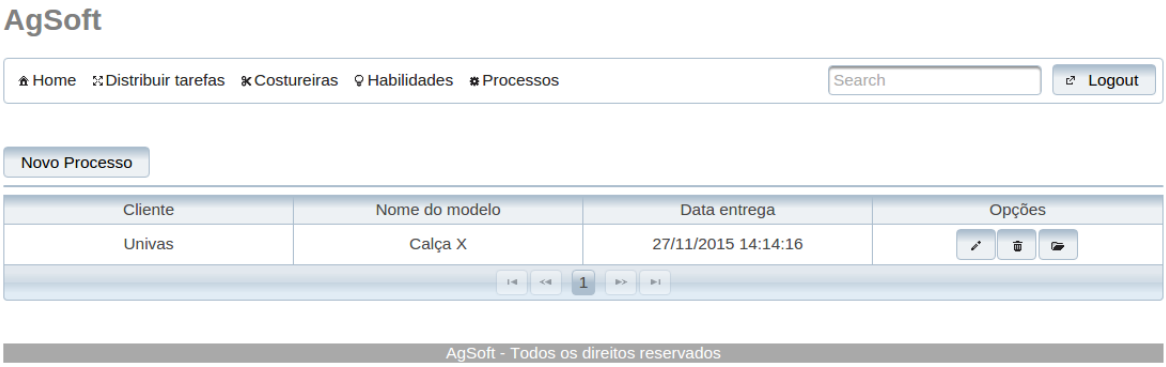


Figura 24 – Criação de um processo. **Fonte:** Desenvolvido pelos autores.

Todo processo ao ser criado, por padrão já contém as atividades principais Carimbo e Finalização conforme mostra a Figura 25.

Após a criação do processo foram definidas quais costureiras possuem a habilidade para realizar as atividades que compõe o processo, sendo definidas para realizar a atividade finalização, as costureiras Maria e Duda, ambas cobram o mesmo valor por peça porém o tempo gasto para fabricar uma peça é diferente conforme mostra a Figura 26. Vale ressaltar que a atividade carimbo é realizada somente pelo proprietário da fábrica pois é a atividade onde serão distribuídas as peças para serem produzidas.

Feito isso, foi aberto o processo para a distribuição das atividades, através do menu "Distribuição de Tarefas". Ao acessar a página todos os processos criados são listados como mostra a Figura 27:

Ao clicar no botão "Abrir"da coluna opções é mostrado a tela para que o usuário insira os dados, como número de peças, total de peças por lote e data de inicio. Depois

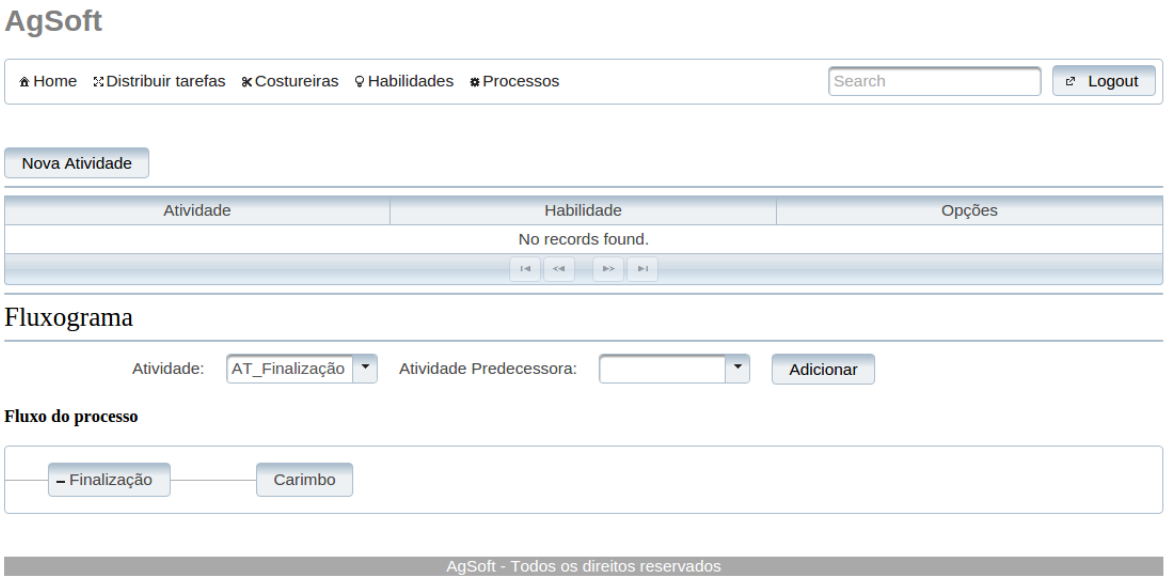


Figura 25 – Detalhes do processo cadastrado. **Fonte:** Desenvolvido pelos autores.

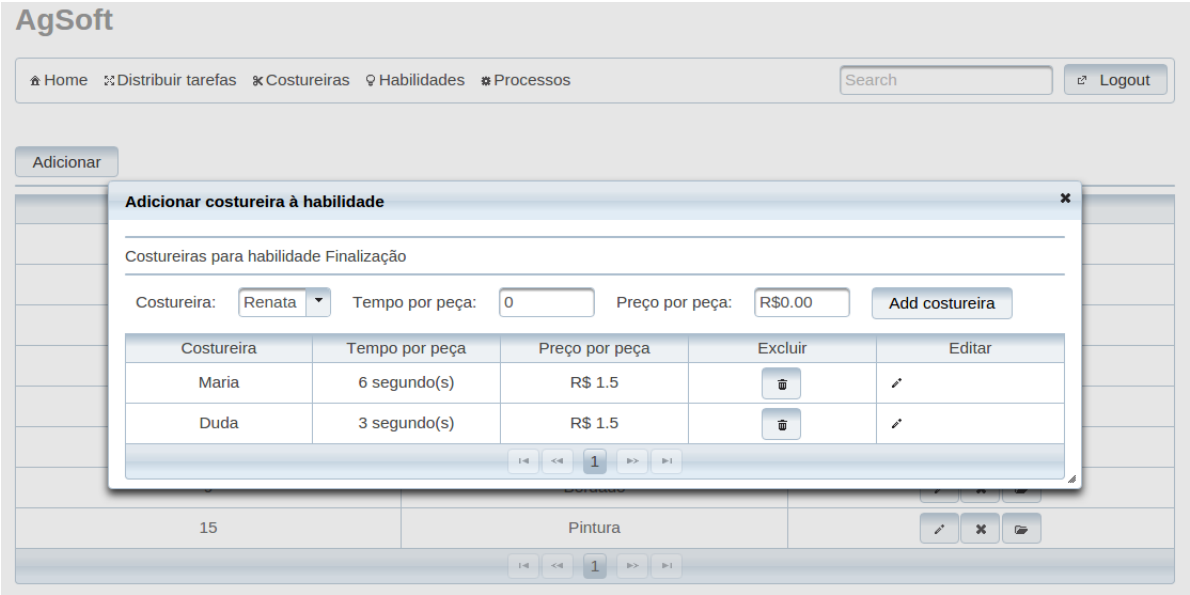


Figura 26 – Inserindo costureiras na habilidade Finalização. **Fonte:** Desenvolvido pelos autores.

de inseridos os dados, ao clicar no botão "Iniciar Distribuição", o sistema executa o algoritmo genético e retorna para o usuário a melhor solução encontrada com base nos dados inseridos, conforme mostra a Figura28.

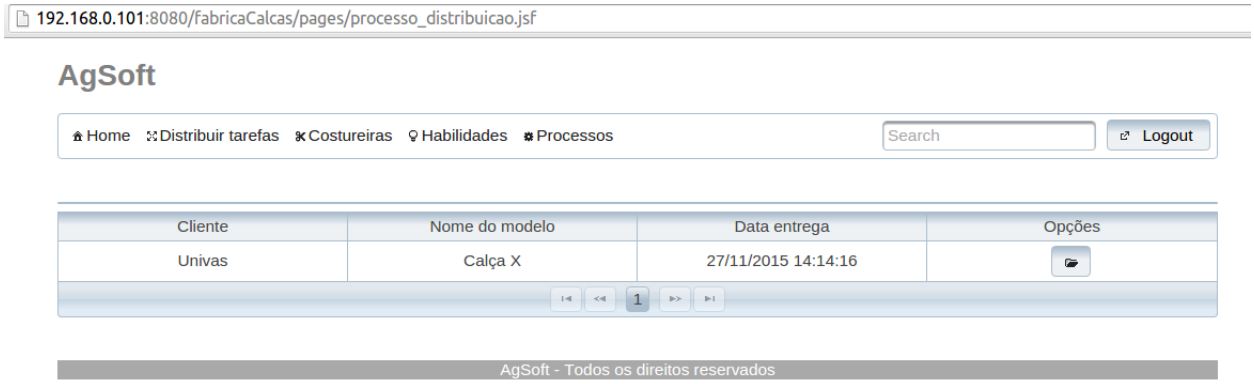


Figura 27 – Tela de dritribuição de tarefas. **Fonte:** Desenvolvido pelos autores.

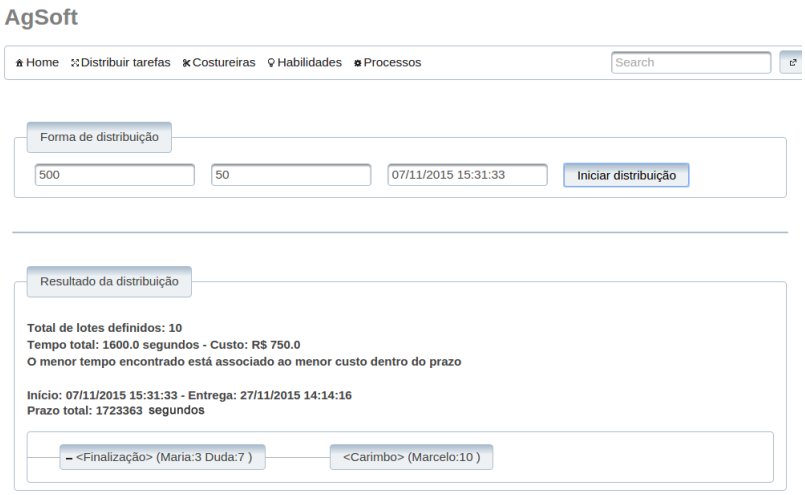


Figura 28 – Resultado da distribuição de lotes considerando o tempo de produção. **Fonte:** Desenvolvido pelos autores.

Os valores então foram definidos, a distribuição foi realizada, e o resultado apresentado. Conforme ilustrado na Figura 26, Maria possui o tempo de produção maior que Duda, por isso ela recebeu um número menor de lotes para produzir conforme ilustrado na Figura 28

Se o tempo de cada costureira for alterado, um novo resultado será retornado levando em consideração as alterações. Para isto, foi definido para Maria o tempo de 4 segundos e para Duda o tempo de 7 segundos conforme ilustra a Figura 29.

Após executar novamente a distribuição foi retornado um novo resultado ilustrado na Figura 30

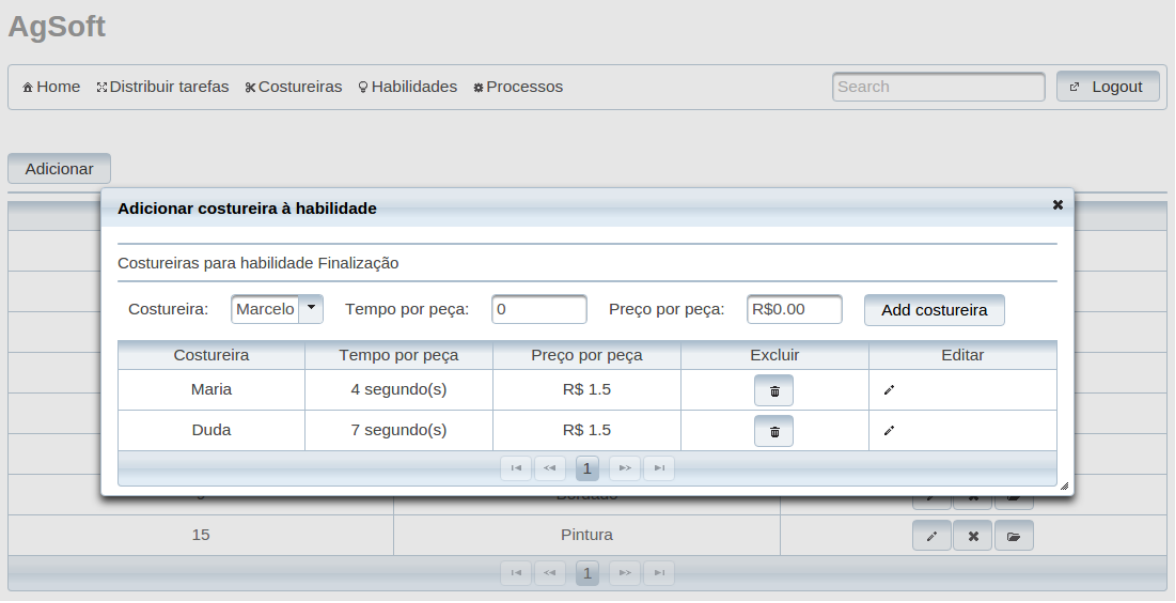


Figura 29 – Alteração no tempo de produção entre as costureiras. **Fonte:** Desenvolvido pelos autores.

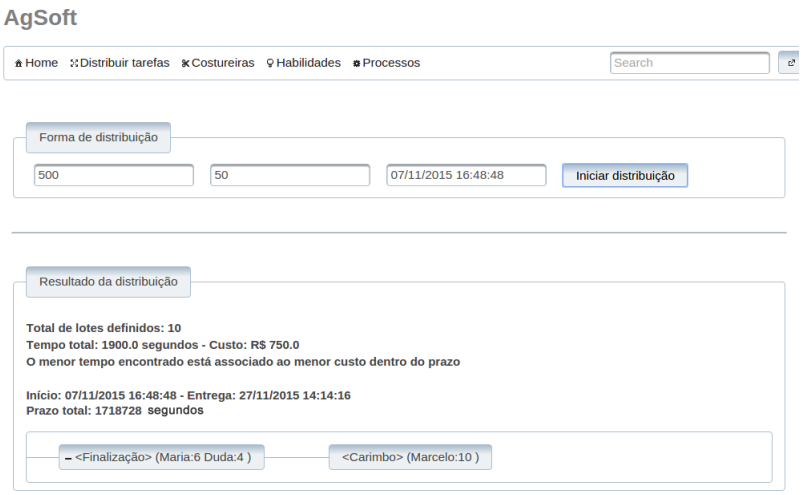


Figura 30 – Resultado da distribuição de lotes após a alteração do tempo de produção entre as costureiras. **Fonte:** Desenvolvido pelos autores.

Neste resultado Maria obteve mais lotes que Duda para produzir pois o seu tempo de produção é menor.

4.2 Teste considerando o custo de produção

Este teste foi feito utilizando-se o mesmo processo do teste realizado na seção anterior, porém foi definido o mesmo tempo de produção para ambas costureiras alterando somente o custo. Foi definido o tempo de 5 segundos para as costureiras Maria e Duda e o preço por peça R$1,50 e R$2,70 respectivamente. Além disso foi definido os mesmos

valores X e Y relativos a posição geográfica para Maria e Duda, de forma a manter o tempo de transporte semelhante para ambas. A Figura 31 mostra a definição de tempo e custo para ambas as costureiras:

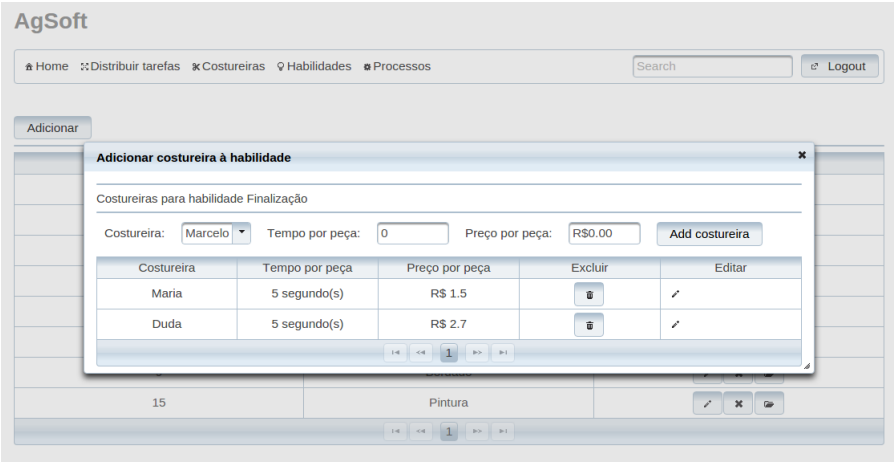


Figura 31 – Custo entre as costureiras da atividade Finalização. **Fonte:** Desenvolvido pelos autores.

Ao executar a distribuição é retornado a seguinte resultado ilustrado na Figura 32:

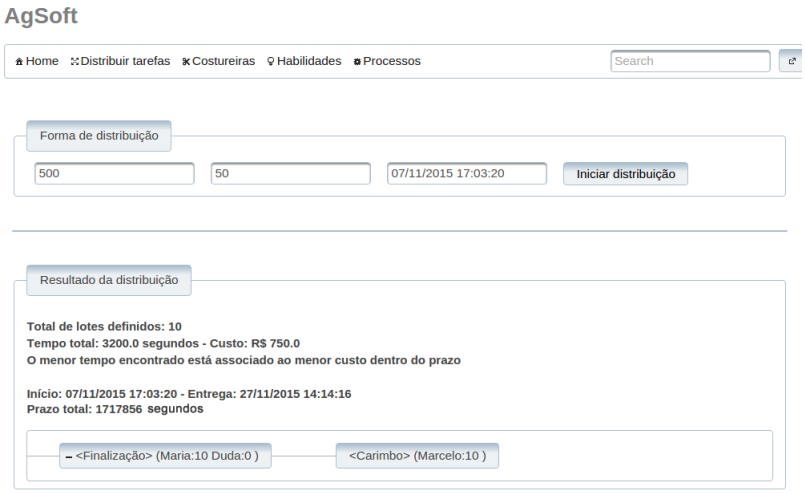


Figura 32 – Resultado da distribuição de lotes considerando o custo. **Fonte:** Desenvolvido pelos autores.

Com base no resultado obtido, o algoritmo decidiu que, pelo fato de Maria e Duda obter o mesmo tempo de produção e Duda ter um preço por peça bem superior que o de Maria, Duda não deve receber nenhum lote para produzir, pois neste caso é possível que Maria produza sozinha os lotes dentro do prazo de entrega do processo. Caso Maria não conseguisse produzir os lotes no prazo estipulado, o algoritmo adequaria a distribuição distribuindo alguns lotes para Duda para conseguir alcançar o prazo de entrega, toda via, fazendo isto, o custo tende a aumentar, este caso será mostrado com mais detalhes na seção 4.3.

Caso Maria e Duda gastassem o mesmo tempo, cobrassem o mesmo preço e o tempo de transporte das peças entre todos os envolvidos no processo fosse o mesmo para ambas, o algoritmo distribuiria lotes para Duda pois, mesmo a costureira Maria conseguindo fabricar as peças dentro do prazo, neste caso, se Duda recebesse alguns lotes para

produzir isto não interferiria no preço final e a produção seria realizada em um tempo menor como mostra a Figura 33.

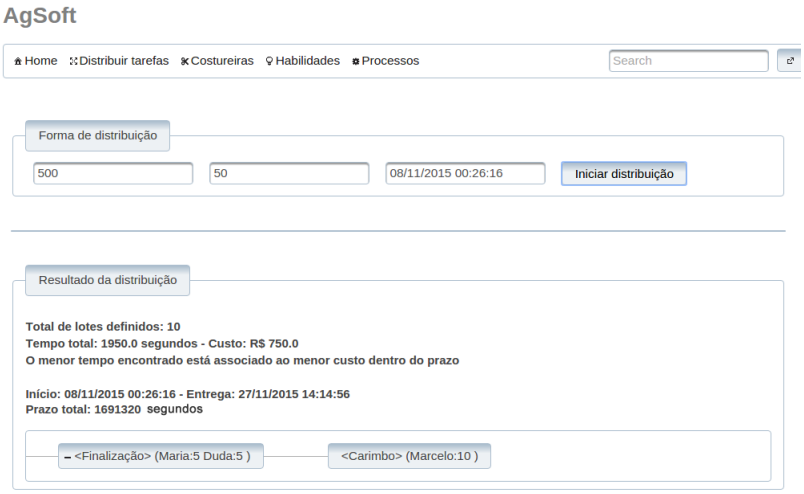


Figura 33 – Resultado da distribuição de lotes com configurações iguais entre as costureiras. **Fonte:** Desenvolvido pelos autores.

4.3 Teste considerando tempo x custo x prazo de entrega

Este teste foi realizado para demonstrar a distribuição considerando o tempo de produção, o custo e o prazo de entrega, assim mesmo que uma costureira for inviável quanto ao custo, ela poderá receber lotes para produzir para que seja cumprido o prazo de entrega. Para isto as costureiras que possuem a habilidade finalização ficarão com as seguintes configurações como ilustra a Figura 34.

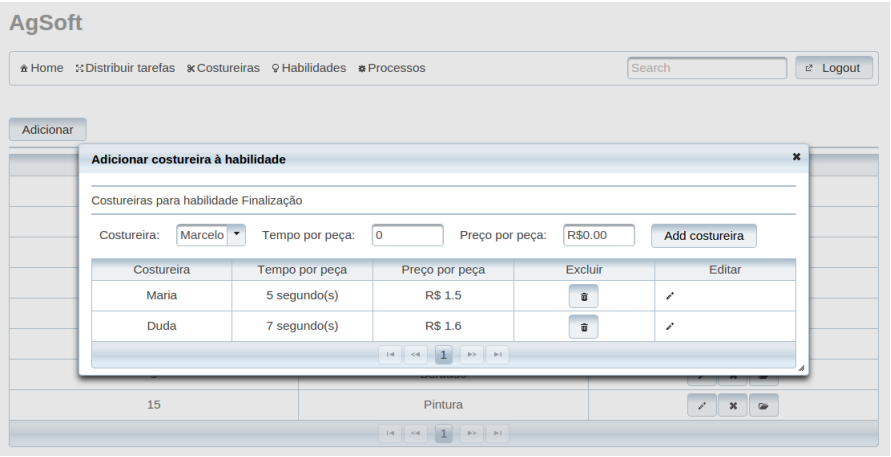


Figura 34 – Configuração das costureiras da atividade Finalização. **Fonte:** Desenvolvido pelos autores.

A Figura 35 mostra o resultado da distribuição com base nas configurações da habilidade Finalização mostrada na Figura 34.

Figura 35 – Resultado da distribuição de lotes com prazo longo. **Fonte:** Desenvolvido pelos autores.

Como ilustrado na Figura 35, a costureira Duda não recebeu nenhum lote pois possui um tempo de produção e custo maior que Maria e esta consegue cumprir o prazo de entrega, porém, caso Maria não conseguisse atender o prazo, Duda poderia receber lotes para produzir mesmo que custo aumente.

A Figura 36 mostra a distribuição de lotes entre as costureiras com a alteração na data de início do processo.

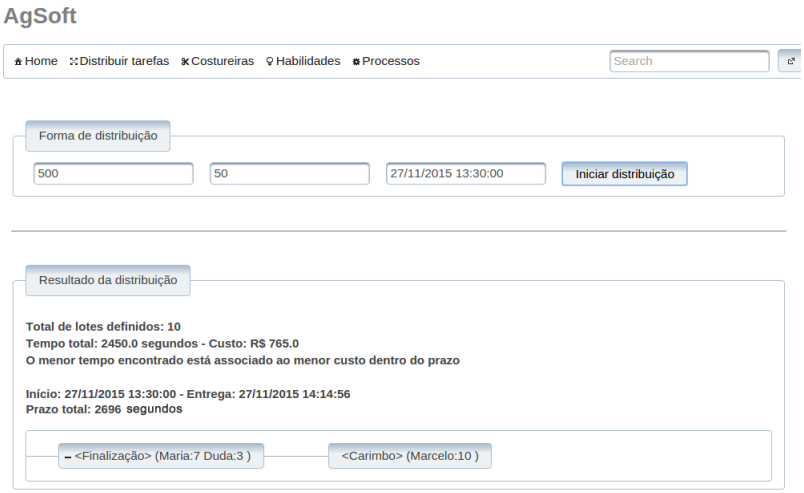


Figura 36 – Resultado da distribuição de lotes com prazo reduzido. **Fonte:** Desenvolvido pelos autores.

Como ilustrado na Figura 36 a costureira Duda recebe 3 lotes para produzir, assim, o custo de produção aumenta com relação ao resultado da Figura 35, porém o tempo de produção diminui sendo possível atender o prazo de entrega.

4.4 Teste considerando o tempo de transporte

Este teste foi realizado para demonstrar que mesmo as costureiras possuindo o tempo e custo iguais, o tempo de transporte entre elas pode influenciar na distribuição dos lotes.

Para isto, foi necessário cadastrar um valor X e Y, correspondente às coordenadas geográficas, para cada costureira, conforme mostra a Figura 37.

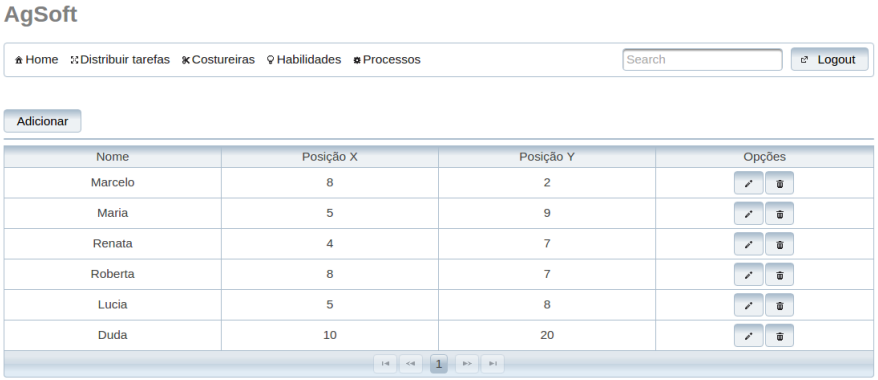


Figura 37 – Tela de cadastro de costureiras. **Fonte:** Desenvolvido pelos autores.

Neste teste foi considerado apenas o tempo de transporte, logo o tempo e o preço

de confecção por peça das costureiras da atividade finalização possuem o mesmo valor, conforme mostra a Figura 38.

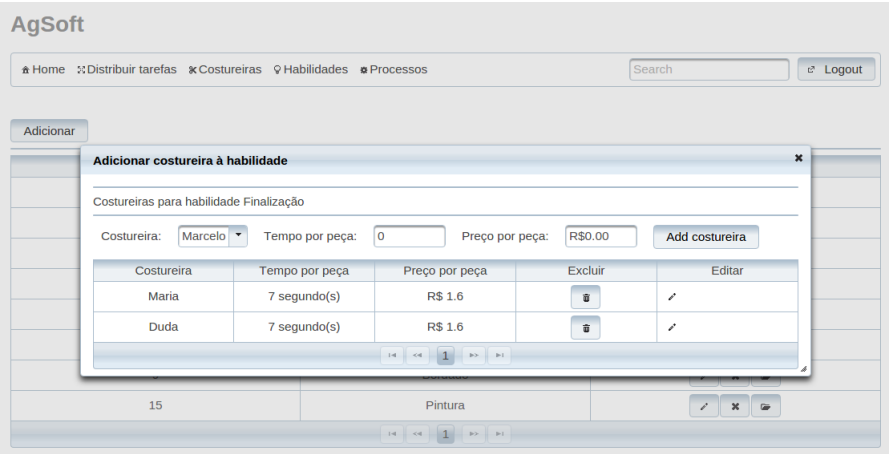


Figura 38 – Configuração das costureiras da habilidade Finalização **Fonte:** Desenvolvido pelos autores.

Feito isto, foi iniciado então o processo de distribuição das atividades, através do menu "Distribuição de Tarefas". A Figura 39 mostra o resultado obtido.

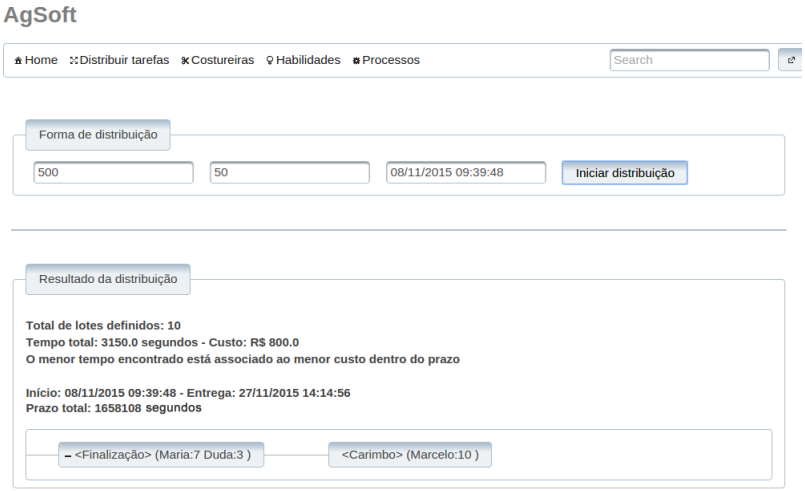


Figura 39 – Resultado da distribuição de lotes considerando o tempo de transporte. **Fonte:** Desenvolvido pelos autores.

Como mostrado na Figura 37, a costureira Duda fica mais distante do Marcelo e por isso recebe menos lotes para produzir em relação à costureira Maria.

Para melhor ilustrar o tempo de envio das peças para as costureiras, na tela de resultados há um detalhamento de todo calculo realizado. A Figura 40 mostra que o tempo de envio das peças para Duda é maior que o tempo de envio para Maria, comprovando o resultado mostrado na Figura 39.

Como mostrado na Figura 40, o tempo de envio das peças para Duda é 1800 segundos e para Maria 700, por isso Duda recebe menos lotes para serem produzidos.

Resultado da distribuição

Total de lotes definidos: 10
Tempo total: 3150.0 segundos - Custo: R$ 800.0
O menor tempo encontrado está associado ao menor custo dentro do prazo

Início: 08/11/2015 10:39:23 - Entrega: 27/11/2015 14:14:56
Prazo total: 1654533 segundos

<Finalização> (Maria:7 Duda:3) — <Carimbo> (Marcelo:10)

Detalhes da distribuição

Atividade/Costureira	Parte	Qtde Lotes	Peç lote	Tot peças	T. peca	T.producao	T. Envio	T. Recebimento	T. total
AT_Finalização	-	-	-	-	-	-	-	-	
Maria	-	7	50	350	7	2450	-	700	3150
Marcelo	Carimbo	-	-	-	-	0	700	-	700
Duda	-	3	50	150	7	1050	-	1800	2850
Marcelo	Carimbo	-	-	-	-	0	1800	-	1800

Figura 40 – Distribuição das atividades. **Fonte:** Desenvolvido pelos autores.

4.5 Teste adicionando mais atividades ao processo

Este teste foi realizado para demonstrar a adição de mais atividades no processo e utiliza o mesmo processo da seção 4.1, porém será adicionado à ele a atividade frente conforme mostra a Figura 41.

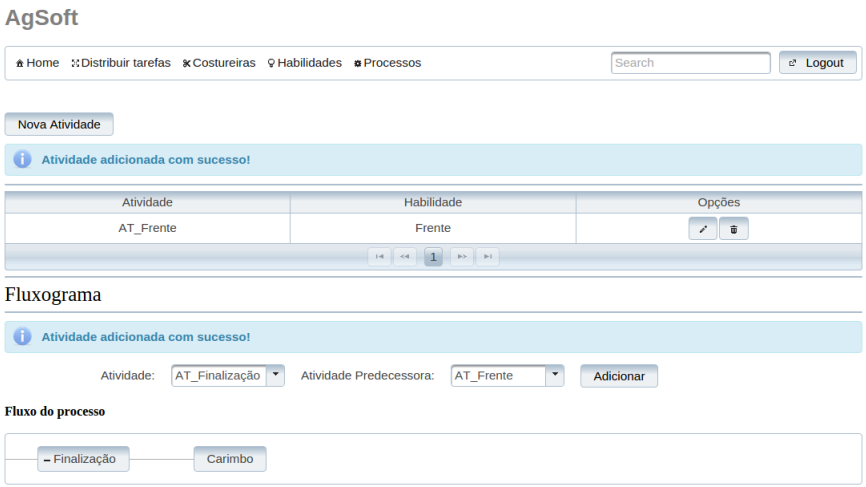


Figura 41 – Adição de uma atividade ao processo. **Fonte:** Desenvolvido pelos autores.

Após a adição da atividade, é preciso adicioná-la ao fluxo do processo como predecessora da atividade finalização. Como a atividade finalização é sempre a última atividade do processo, todas as atividades que forem incluídas sempre serão predecessoras a ela. A Figura 42 mostra a atividade frente adicionada no fluxo do processo.

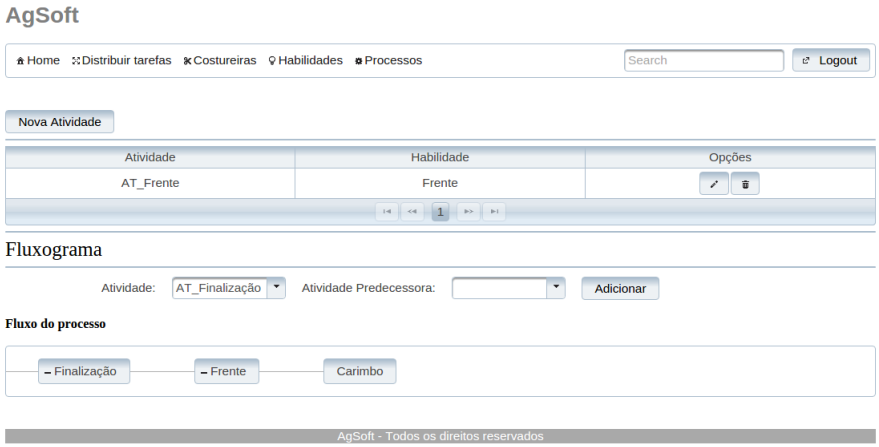


Figura 42 – Adição de uma atividade ao fluxo do processo. **Fonte:** Desenvolvido pelos autores.

Foi adicionado na habilidade frente as costureiras Roberta, Lúcia e Renata para que estas possam realizar a atividade frente do processo.

A Figura 43 mostra as costureiras adicionadas na habilidade frente.

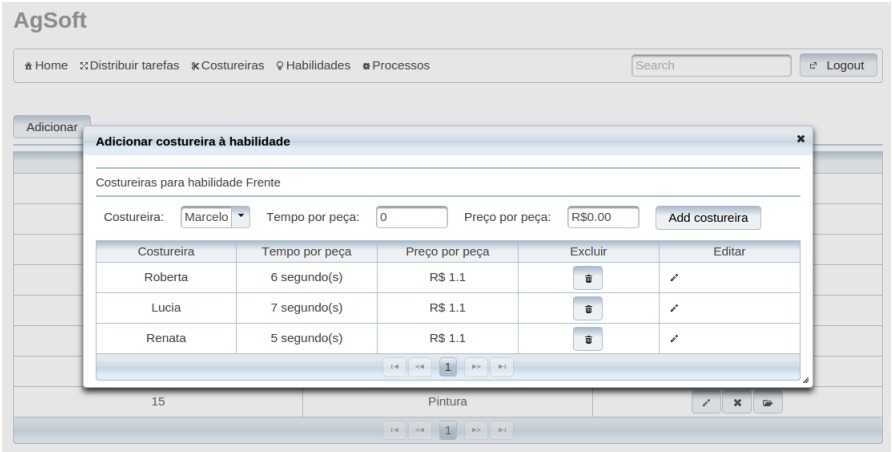


Figura 43 – Adição de costureiras na habilidade Frente. **Fonte:** Desenvolvido pelos autores.

Feito isso foi realizado a distribuição dos lotes com as configurações de tempo e custo mostradas na Figura 43 e com as configurações de posicionamento mostradas na Figura 37.

A Figura 44 mostra o resultado obtido na distribuição.

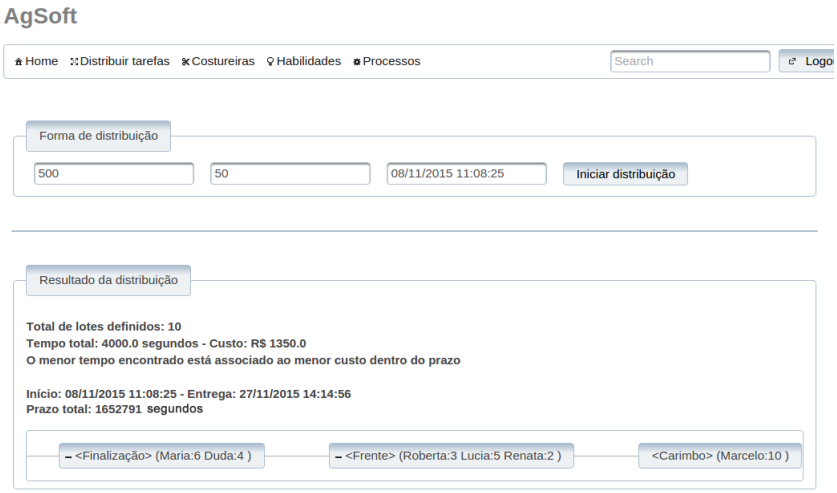
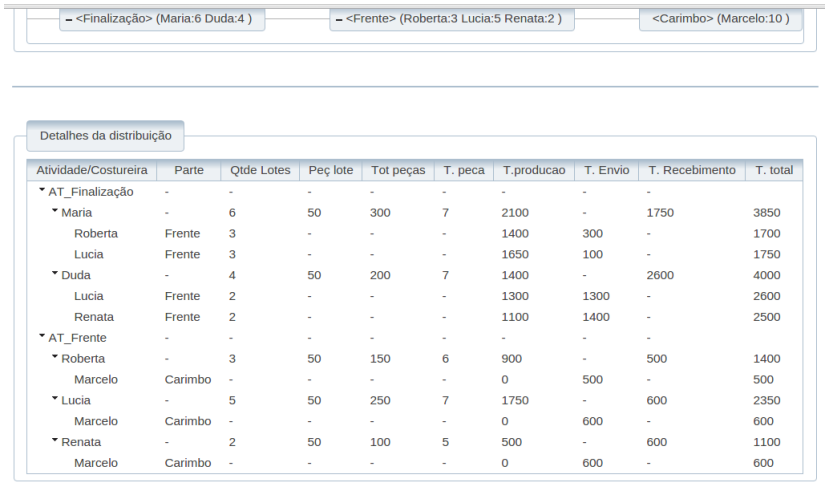


Figura 44 – Resultado da distribuição de lotes com mais uma atividade. **Fonte:** Desenvolvido pelos autores.

Com base na Figura 44 pode-se observar que foi adicionado no resultado final a atividade Frente e suas respectivas costureiras.

A Figura 45 mostra o detalhamento da distribuição da Figura 44.



Atividade/Costureira	Parte	Qtde Lotes	Peç lote	Tot peças	T. peca	T.producao	T. Envio	T. Recebimento	T. total
AT_Finalização	-	-	-	-	-	-	-	-	
Maria	-	6	50	300	7	2100	-	1750	3850
Roberta	Frente	3	-	-	-	1400	300	-	1700
Lucia	Frente	3	-	-	-	1650	100	-	1750
Duda	-	4	50	200	7	1400	-	2600	4000
Lucia	Frente	2	-	-	-	1300	1300	-	2600
Renata	Frente	2	-	-	-	1100	1400	-	2500
AT_Frente	-	-	-	-	-	-	-	-	
Roberta	-	3	50	150	6	900	-	500	1400
Marcelo	Carimbo	-	-	-	-	0	500	-	500
Lucia	-	5	50	250	7	1750	-	600	2350
Marcelo	Carimbo	-	-	-	-	0	600	-	600
Renata	-	2	50	100	5	500	-	600	1100
Marcelo	Carimbo	-	-	-	-	0	600	-	600

Figura 45 – Distribuição das atividades. **Fonte:** Desenvolvido pelos autores.

Vale ressaltar que ao executar a distribuição o algoritmo pode encontrar várias soluções que contém o mesmo resultado e retorna uma delas, se a distribuição for executada novamente mantendo as configurações das costureiras como a posição X e Y, tempo e custo, a distribuição poderá ocorrer de uma outra forma conforme mostra a Figura 46.

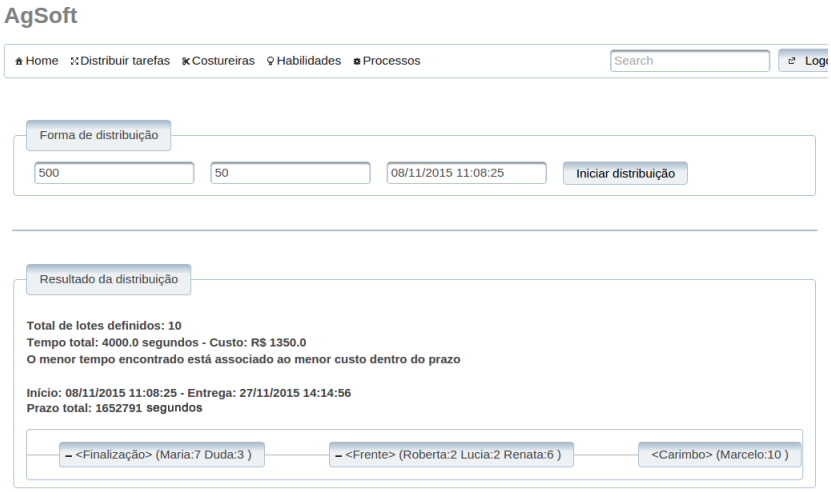
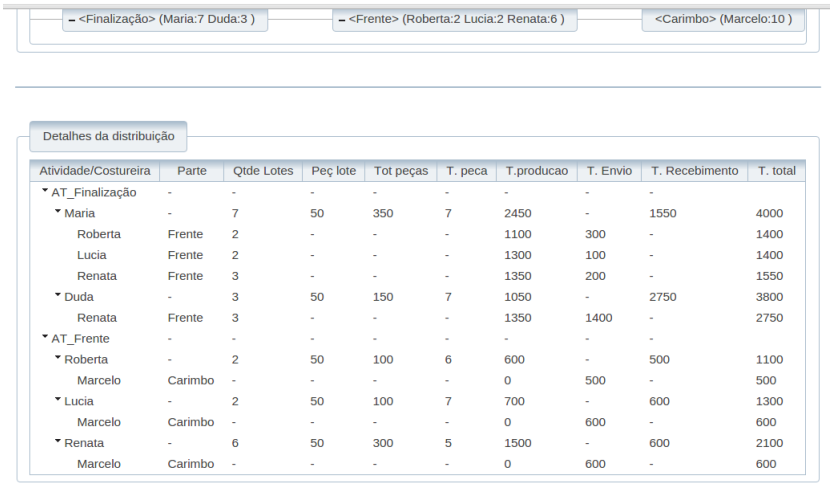


Figura 46 – Resultado da distribuição de lotes após executar novamente a distribuição. **Fonte:** Desenvolvido pelos autores.

A Figura 47 mostra o detalhamento da distribuição da Figura 46.



Atividade/Costureira	Parte	Qtde Lotes	Peç lote	Tot peças	T. peca	T.producao	T. Envio	T. Recebimento	T. total
AT_Finalização	-	-	-	-	-	-	-	-	
Maria	-	7	50	350	7	2450	-	1550	4000
Roberta	Frente	2	-	-	-	1100	300	-	1400
Lucia	Frente	2	-	-	-	1300	100	-	1400
Renata	Frente	3	-	-	-	1350	200	-	1550
Duda	-	3	50	150	7	1050	-	2750	3800
Renata	Frente	3	-	-	-	1350	1400	-	2750
AT_Frente	-	-	-	-	-	-	-	-	
Roberta	-	2	50	100	6	600	-	500	1100
Marcelo	Carimbo	-	-	-	-	0	500	-	500
Lucia	-	2	50	100	7	700	-	600	1300
Marcelo	Carimbo	-	-	-	-	0	600	-	600
Renata	-	6	50	300	5	1500	-	600	2100
Marcelo	Carimbo	-	-	-	-	0	600	-	600

Figura 47 – Distribuição das atividades. **Fonte:** Desenvolvido pelos autores.

Com base nas Figuras 44 e 46 o custo final não teve alterações de uma distribuição para outra, porém pode-se observar que o tempo total de produção e o distribuição dos lotes entre as costureiras é realizado de uma forma diferente, concluindo assim que o algoritmo encontrou outra forma de realizar a distribuição mantendo o melhor resultado.

4.6 Teste quando o prazo não é atendido

Este teste foi realizado para demonstrar quando o prazo de entrega das peças não consegue ser atendido. Este teste segue o mesmo processo da seção 4.1.

A Figura 48 mostra a configuração das costureiras da habilidade finalização.

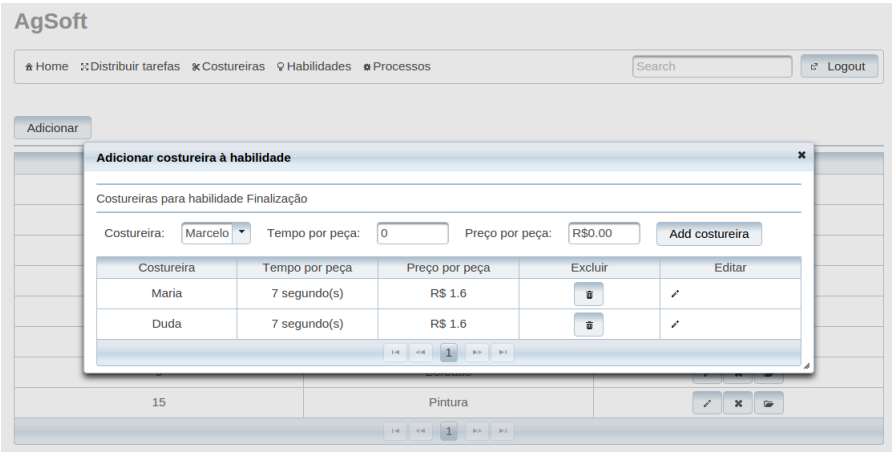


Figura 48 – Configuração das costureiras da atividade Finalização. **Fonte:** Desenvolvido pelos autores.

Considerando as configurações mostradas na Figura 48 foi iniciado o processo de distribuição dos lotes como mostra a Figura 49.

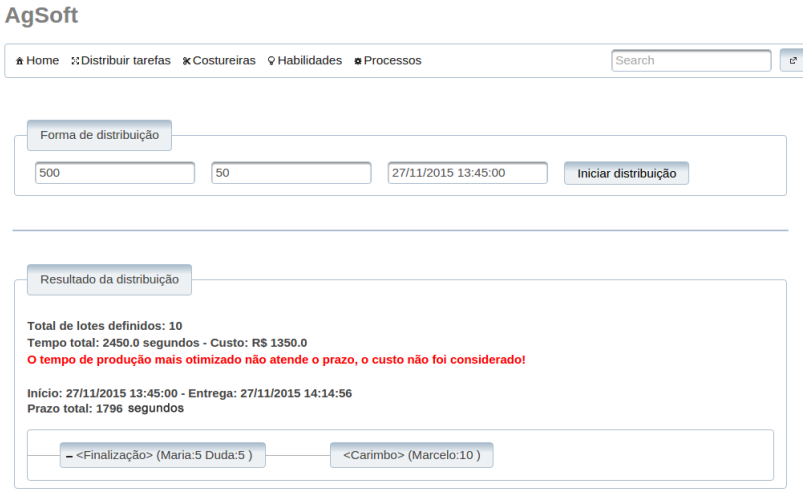


Figura 49 – Resultado da distribuição de lotes com prazo não atendido. **Fonte:** Desenvolvido pelos autores.

Como mostrado na Figura 49, o prazo de produção é de 1796 segundos e, mesmo alocando todas as costureiras não foi possível entregar as peças no prazo determinado pois o tempo mínimo possível de produção é de 2450 segundos. O algoritmo foi desenvolvido para manter sempre o menor custo, procurando manter o tempo dentro do prazo, porém caso nenhum dos indivíduos tenham o tempo total menor que o prazo, então o melhor tempo é retornado e a ordenação por custo é ignorada. Para demonstrar isto, foi alterado o custo da costureira Duda conforme mostra a Figura 50.

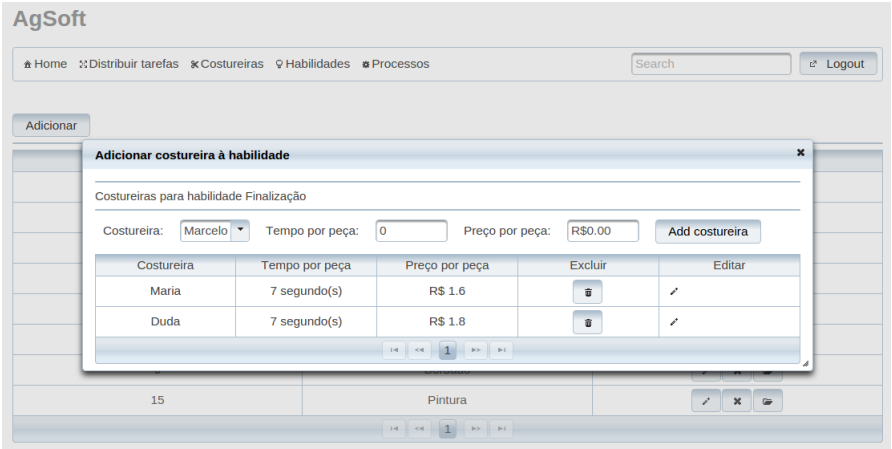


Figura 50 – Resultado da distribuição após alteração no custo da costureira Duda. **Fonte:** Desenvolvido pelos autores.

Foi iniciado novamente a distribuição dos lotes conforme mostra a Figura 51.

Conforme mostrado na Figura 51, mesmo com o aumento no custo por peça da costureira Duda, o algoritmo mesmo assim distribuiu lotes para esta costureira aumentando o custo final porém retornou o menor tempo de produção possível.

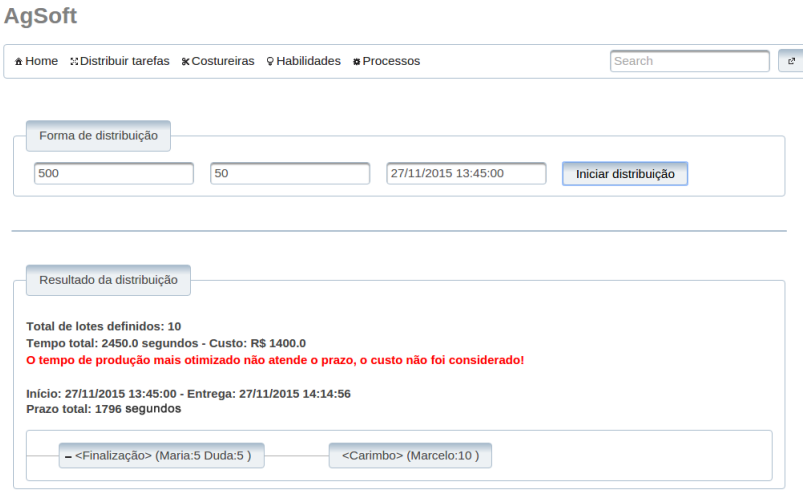


Figura 51 – Resultado da distribuição após alteração no custo da costureira Duda. **Fonte:** Desenvolvido pelos autores.

5 CONCLUSÃO

Nos tempos atuais, os sistemas de informação ocupam um papel crítico no cenário corporativo, pois atuam como uma importante ferramenta no auxílio ao negócio. Softwares de gestão possibilitam o acesso rápido e confiável a informações importantes. Tais características são cruciais em um sistema, uma vez que a informação tornou-se um dos patrimônios mais importantes das empresas.

Além destas características, os softwares são dotados de inteligência artificial e vêm cada vez mais realizando tarefas mais complexas, oferendo um suporte primordial no apoio a tomada de decisões e otimização de processos. Este trabalho visou explorar uma das abordagens da inteligência artificial em softwares, denominada algoritmos genéticos, produzindo uma aplicação capaz de otimizar a distribuição de atividades em uma linha de produção de calças. A escolha de explorar tal assunto se justifica pelo fato de que empresas buscam constantemente se tornar mais competitivas, nesse contexto, a otimização de processos pode ser um dos fatores chave para o aumento da competitividade.

Para demonstrar o conceito de algoritmos genéticos, foi desenvolvida, por meio deste trabalho, uma aplicação *web* que permite que o usuário modele seu processo, adicionando atividades a ele em uma ordem de precedência, além disso o usuário pode definir as costureiras de cada atividade juntamente com o tempo e o preço de produção de cada uma. Assim, com base nessas informações, junto com a quantidade de peças e o tempo de produção, o algoritmo então distribui as atividades de forma a encontrar o menor tempo, aliado ao menor custo de produção dentro do prazo de entrega.

Para a construção do algoritmo genético, foi utilizada uma base desenvolvida pelo professor Artur Barbosa, durante as aulas de sistemas especialistas, que define uma série de regras a serem seguidas durante o desenvolvimento. Tal base é explicada com mais detalhes no quadro metodológico e foi de grande ajuda, pois além de definir as regras, a base já implementava os métodos de seleção, cruzamento e mutação, sendo preciso apenas realizar algumas adaptações neles para que pudessem se adequar à lógica desenvolvida para resolver o problema.

Um dos maiores desafios do trabalho foi a definição da função de avaliação, pois, uma vez que existe uma estrutura de nós contendo as atividades, e o cálculo do tempo de cada atividade depende de nós predecessores, a maior parte da função foi desenvolvida de forma recursiva o que dificultou o *debug*.

Assim, todos os objetivos propostos foram alcançados através dos conceitos e tecnologias descritos no quadro teórico e nos procedimentos descritos no quadro metodológico. O software desenvolvido atendeu todos os requisitos do escopo definidos na seção 3.3.2 do Quadro Metodológico. Porém, além do escopo definido, outras funcionalidades poderiam ser desenvolvidas, mas devido ao tempo disponível para o desenvolvimento do trabalho, não foi possível cobri-las. Uma delas é adição do custo de transporte das peças e materiais entre as costureiras, por enquanto está sendo considerado somente o custo de produção por costureira. Um outro exemplo seria implementar uma restrição de forma que o algoritmo só distribuísse atividades para as costureiras disponíveis. Além disso implementar uma lógica de agendamento indicando quais peças devem ser transportadas em quais horários para quais costureiras.

No âmbito acadêmico, o presente trabalho, agregará à base de conhecimentos da Univás uma pesquisa com conceitos e exemplo de implementação de um algoritmo genético, podendo ser utilizado como base para futuros trabalhos.

Conclui-se então que este trabalho foi de grande relevância, pois resolveu um problema complexo de ser solucionado em um tempo de processamento aceitável, tal problema poderia levar um tempo muito grande de processamento utilizando algoritmos convencionais, além disso o trabalho proporcionou um sólido conhecimento sobre algoritmos genéticos aos pesquisadores.

REFERÊNCIAS

BERGSTEN, H. ***Javaserver Faces***. [S.l.: s.n.], 2004.

CARVALHO, L. *Aprenda algumas técnicas de reunião*. 2012. <http://www.administradores.com.br/artigos/negocios/aprenda-algumas-tecnicas-de-reuniao/62516/>. Acessado em 04 de abril.

DATE, C. J. *Introdução a sistemas de banco de dados*. 8º. ed. São Paulo: Campus, 2004.

FARIA, J. de. *TCC I*. Pouso Alegre: Univás. Notas de Aula 26 de março: [s.n.], 2015.

FARIA, T. *Java EE 7 com JSF, PrimeFaces e CDI*. [S.l.: s.n.], 2013.

FERNANDES, A. M. D. R. *Inteligência Artificial: Noções Gerais.* [S.l.]: Visual Books, 2003.

FILITTO, D. Algoritimos genéticos: Uma visão explanatória. *Saber Acadêmico*, n. 06, p. 136–143, 2008.

FREITAS, C. C. et al. Uma ferramenta baseada em algoritmos genéticos para a geração de tabela de horário escolar. *SÉTIMA ESCOLA REGIONAL DE COMPUTAÇÃO Bahia-Sergipe. Vitória da Conquista:[sn]*, 2007.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.: s.n.], 2009.

GERHARDT, T. E.; SILVEIRA, D. T. *Métodos de Pesquisa.* 1º. ed. Porto Alegre: UFRGS, 2009.

GIL, A. C. *Métodos e técnicas de pesquisa social.* São Paulo: Atlas, 1999.

GOLDBERG, D. E. *Genetic Algorithms in search optimization and machine learning*. 1º. ed. New York: Addison-Wesley publishing company,inc., 1989.

HAGUETTE, T. M. F. *Metodologias qualitativas na Sociologia*. 5º. ed. Petrópolis: Vozes, 1997.

JUNEAU, J. *Primefaces in the Enterprise*. 2014. <http://www.oracle.com/technetwork/articles/java/java-primefaces-2191907.html>. Acesso em: 15 de janeiro de 2015.

JUNIOR, P. J. *P. de JAVA: Guia do Programador.* 1º. ed. [S.l.]: Novatec, 2007.

LACERDA, E. G. M. de; CARVALHO, A. C. P. L. F. de. *Introdução aos Algritmos Genéticos*. 2015. <http://www.leca.ufrn.br/~estefane/metaheuristicas/ag.pdf>. Acessado em 1 de Agosto.

LAUDON, K. C.; LAUDON, J. P. *Sistemas de informação gerenciais.* 7º. ed. [S.l.]: Pearson, 2009.

LINDEN, R. *Algoritmos genéticos*. 1º. ed. Rio de Janeiro: Ciência Moderna, 2012.

LUCKNOW, D. H.; MELO, A. A. de. *Programação Java para Web*. 1º. ed. São Paulo: Novatec, 2010.

LUGER, G.; STUBBLEFIELD, W. A. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 2º. ed. [S.l.]: Palo Alto, 1993.

LUQUE, L.; SILVA, R. Algoritmos genéticos em java, conceitos e aplicação. *Java Magazine*, Rio de Janeiro, v. 82, p. 44–55, 2010.

MANZONI, A. Desenvolvimento de um sistema computacional orientado a objetos para sistemas elÉtricos de potÊncia: AplicaÇÃo a simulaÇÃo rÁpida e anÁlise da estabilidade de tensÃo. In: . [S.l.: s.n.], 2005.

MARCONI, M. A.; LAKATOS, E. M. *técnicas de Pesquisa*. 7º. ed. [S.l.]: Atlas, 2009.

MELANIE, M. *An introduction to genetic algorithms*. 1º. ed. London: Massachusetts Institute of Technology, 1999.

ORACLE. *JavaServer Faces Technology Overview*. 2015. <http://www.oracle.com/technetwork/java/javaee/overview-140548.html>. Acesso em 15 de janeiro de 2015.

ORACLE. *O que é Java?* 2015. <https://www.java.com/pt_BR/download/whatis_java.jsp>. Acessado em 12 de Fevereiro.

PRESSMAN, R. *Engenharia de Software*. McGraw Hill Brasil, 2011. ISBN 9788580550443. Disponível em: <http://books.google.com.br/books?id=y0rH9wuXe68C>.

PRICE, J. *Oracle Database 11g SQL*: Domine sql e pl/sql no banco de dados oracle. 1º. ed. Porto Alegre: bookman, 2008.

ROSS, C. L.; BORSOI, B. T. Uso de primefaces no desenvolvimento de aplicações ricas para web. 2012.

SANTOS, J. C. et al. Seleção de atributos usando algoritmos genéticos para classificação de regiões. In: *XIII Simposio Brasileiro de Sensoriamento Remoto, Florianópolis, Brasil. INPE-Instituto Nacional de Pesquisas Espaciais*. [S.l.: s.n.], 2007. p. 6143–6150.

SANTOS, R. *Introdução à Programação Orientada a Objetos usando Java*. 3º. ed. Rio de Janeiro: Campus, 2003.

SCHILDT, H. *The complete reference Java*. 7º. ed. Nova York: MC Graw Hill Osborne, 2007.

SILVA, E. E. d. *Otimização de estruturas de concreto armado utilizando algoritmos genéticos.* Tese (Doutorado) — Universidade de São Paulo, 2001.

SUEHRING, S. *MySQL Bible*. Nova York: Wiley Publishing, Inc, 2002.

VUKOTIC, A.; GOODWILL, J. *Apache Tomcat 7*. 1º. ed. Nova York: Apress, 2011.

ZANELLA, L. C. H. *Metodologia de Estudo e de Pesquisa em Administração.* 1º. ed. Florianópolis: CAPES, 2009.