

**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.

94 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. evolução. 3. espécies.

**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 01/01/2014 pela banca examinadora constituída pelos professores:

Prof. Me. Roberto Ribeiro Rocha
Orientador

Prof^a. ME. Nome da professora
Avaliadora

Prof. MSc. Nome do professor
Avaliador

*“Complicar é simples,
simplificar que é complicado.
(Paulo Sérgio dos Santos)*

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 uma sistema de produção de calças em um modo de se produzir que permite que os empregados trabalhem em suas casas e que cada parte da calça é 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 permita 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 otimização desse.

Palavras-chave: Algoritmos genéticos. evolução. espécies.

SANTOS, Jonathan Ribeiro dos; Oliveira, Sandro Augusto de. **Algoritmos genéticos aplicados à linha de produção de calças**. 2015. Monografia – Curso de SISTEMAS DE INFORMÁTICA, 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. evolution. species.

LISTA DE FIGURAS

Figura 1 – Demonstração da execução de um AG	16
Figura 2 – Demonstração do método de seleção Roleta	18
Figura 3 – Demonstração do Modelo MVC	23
Figura 4 – Demonstração de um processo de fabricação	34
Figura 5 – Representação do processo de fabricação no banco de dados	35
Figura 6 – Classes Atividade e AtividadeOrdem	36
Figura 7 – Demonstração da execução de um AG	36
Figura 8 – Demonstração da execução de um AG	37
Figura 9 – Exemplo de distribuição aleatória de lotes para as costureiras	41
Figura 10 – Classe ProcessoChromosome	42
Figura 11 – Armazenamento de dados das costureiras	42
Figura 12 – Classe CostureiraHabilidade	43
Figura 13 – Classe ProcessoIndividual	44
Figura 14 – Distribuição em porcentagem	45
Figura 15 – Demonstração de costureiras e habilidades	49
Figura 16 – Estrutura de representação da ordem de precedência	49
Figura 17 – Exemplo de solicitação de lotes predecessores	54
Figura 18 – Distribuição demonstrando o tempo	56
Figura 19 – Seleção dos indivíduos	62
Figura 20 – Cruzamento dos indivíduos	62
Figura 21 – Exemplo de mutação	66
Figura 22 – Menu Distribuir tarefas	68
Figura 23 – Tela de distribuição de tarefas	69
Figura 24 – Tela de resultado de distribuição de tarefas	73
Figura 25 – Tela inicial da aplicação	75
Figura 26 – Criação de um processo	76
Figura 27 – Detalhes do processo cadastrado	76
Figura 28 – Demonstração inserir costureira à habilidade	77
Figura 29 – Demonstração tela de distribuição de tarefas	77
Figura 30 – Resultado da distribuição de lotes	78
Figura 31 – Tempo de produção entre as costureiras	79

Figura 32 – Resultado da distribuição de lotes	79
Figura 33 – Custo entre as costureiras atividade Finalização	80
Figura 34 – Custo entre as costureiras atividade Finalização	81
Figura 35 – Custo entre as costureiras atividade Finalização	82
Figura 36 – Custo entre as costureiras atividade Finalização	83
Figura 37 – Custo entre as costureiras atividade Finalização	83
Figura 38 – Custo entre as costureiras atividade Finalização	84
Figura 39 – Caso de teste com tempo de distribuição	85
Figura 40 – Caso de teste com tempo de distribuição	85
Figura 41 – Distribuição das atividades	86
Figura 42 – Distribuição das atividades	86
Figura 43 – Caso de teste com tempo de distribuição	87
Figura 44 – Caso de teste com tempo de distribuição	88
Figura 45 – Caso de teste com tempo de distribuição	88
Figura 46 – Caso de teste com tempo de distribuição	89
Figura 47 – Caso de teste com tempo de distribuição	89
Figura 48 – Caso de teste com tempo de distribuição	90
Figura 49 – Caso de teste com tempo de distribuição	90

LISTA DE SIGLAS E ABREVIATURAS

ABNT	Associação Brasileira de Normas Técnicas
API	<i>Application Programming Interface</i>
GNU	<i>Gnu is Not Unix</i>
GPL	<i>General Public License</i>
MVC	<i>Model – View – Controller</i>

SUMÁRIO

INTRODUÇÃO	11
2 QUADRO TEÓRICO	13
2.1 Algoritmos Genéticos	13
2.1.1 Fundamentos.....	13
2.1.2 Características dos Algoritmos Genéticos	15
2.2 Tecnologias	20
2.2.1 Linguagem de programação Java	20
2.2.2 Interface Gráfica - JSF	22
2.2.3 Armazenamento de dados	24
3 QUADRO METODOLÓGICO	26
3.1 Tipo de pesquisa	26
3.2 Contexto de pesquisa.....	27
3.3 Instrumentos	27
3.3.1 Entrevistas	28
3.3.2 Reuniões	28
3.4 Procedimentos	30
3.4.1 Framework de desenvolvimento	30
3.4.2 Representação do processo de produção.....	33
3.4.3 População inicial: Distribuição das atividades, Indivíduos e Cromossomos	37
3.4.4 Função de avaliação	48
3.4.5 Classificação dos indivíduos.....	56
3.4.6 Indivíduos estrangeiros e elitismo	58
3.4.7 Seleção de indivíduos e cruzamento	60
3.4.8 Mutação	65
3.4.9 Interface gráfica de distribuição	68
4 DISCUSSÃO DE RESULTADOS	74
4.1 Teste considerando somente o tempo de produção	75
4.2 Teste considerando o custo de produção	80
4.3 Teste considerando tempo x custo x prazo de entrega.....	82
4.4 Teste considerando o tempo de transporte	84
4.5 Teste adicionando mais atividades ao processo	87
5 CONCLUSÃO	91
REFERÊNCIAS.....	94

INTRODUÇÃO

Sabe-se que atualmente existem diversos softwares disponíveis no mercado. 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 a cada dia se tornarem mais competitivas, é necessário que um sistema ofereça suporte para que estas 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 - IA¹ - 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, o 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 no menor tempo possível e com o menor custo. Para a resolução deste problema, pode se fazer uso de IA através de um dos ramos da abordagem evolutiva denominado Algoritmos Genéticos - AGs² - 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.

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.

O presente trabalho têm por objetivo geral, desenvolver uma aplicação utilizando técnicas de inteligência artificial, capazes de realizar a alocação de empregados em uma linha de produção de forma otimizada. Para a realização do mesmo 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 o menor custo e o menor tempo de produção.

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, dentre 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 assim com que estas 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 ter sempre em mãos a solução mais otimizada, irá ter um menor tempo de transporte de materiais, o que irá reduzir 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, que hoje estão presentes em diversos sistemas críticos de apoio a decisão e otimização de processos nas empresas.

2 QUADRO TEÓRICO

Neste capítulo serão listados os conceitos e as tecnologias que serão utilizados no desenvolvimento da proposta de trabalho apontada na seção objetivos. Para tal, serão discutidos 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, dentre 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 que existe 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 infor-

mação 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 necessário 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 estas 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 atinja um limite, que é definido previamente, ou até atingir outra condição definida pelo programador.

A Figura 1 ilustra os passos do algoritmo.

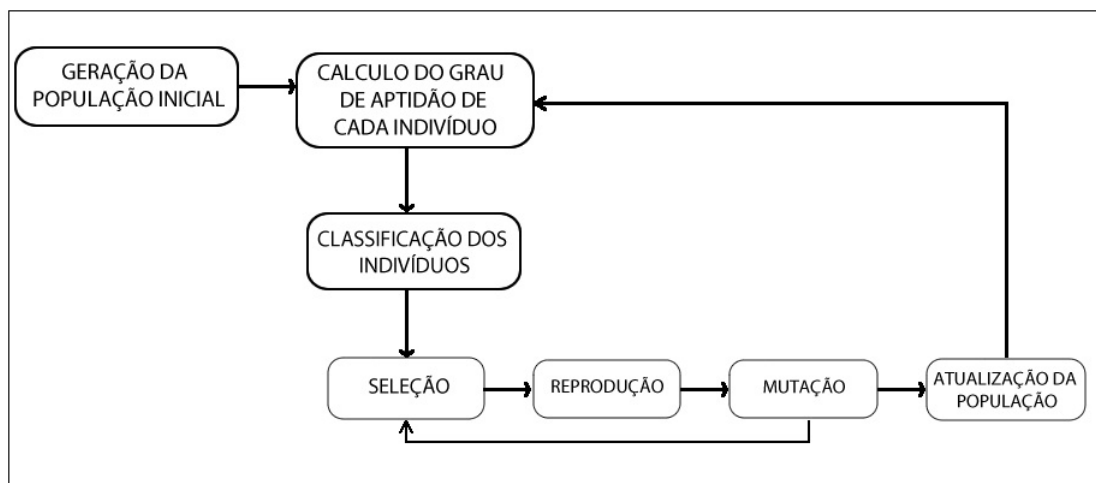


Figura 1 – Demonstração da execução de um AG **Fonte:** Desenvolvido pelos autores.

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 de gerações seja atingido.

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, esta é então avaliada através de uma função de avaliação que mede a qualidade de cada uma de suas soluções e é realizada então uma classificação que ordena as soluções das melhores para as piores, para então 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, que é realizada de acordo com a

qualidade destes, 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 de se reproduzirem.

O autor ressalta que a razão de o mecanismo de seleção não escolher apenas as melhores soluções é devido ao fato de que 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 esta 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 utilizado para este fim é o método roleta, muito utilizado 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 possuem 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).

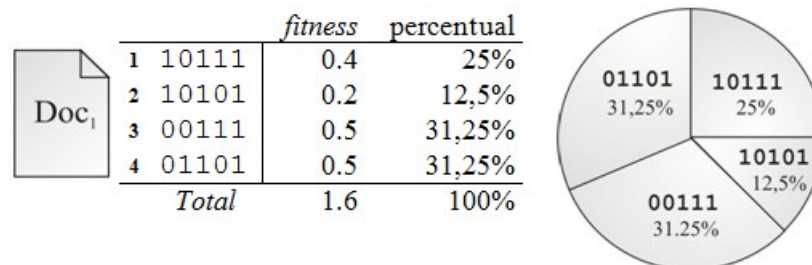


Figura 2 – Demonstração do método de seleção Roleta **Fonte:** "<http://www.dgz.org.br/fev09/Art04.htm>".

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, neste processo dois novos indivíduos, ou seja, 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 utilizado 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, dentre 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 dentre 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 dentre 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* - JVM².

O ponto chave que permite o 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 esta, por sua vez, traduz o mesmo para a arquitetura a qual

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

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 este deve ter e quais operações este terá e então após a criação da classe o programador pode criar um objeto desta. Uma analogia simples seria a planta de uma casa e a construção da mesma em si, 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. Isto 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 da linguagem de programação e podem também ser representados por outros modelos criados pelo programador.

Para o desenvolvimento do sistema de informação deste projeto, 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. Dentre as opções disponíveis, foi escolhido o *Tomcat* pelo fato de este 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 que será explanado na seção. 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 o faz 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. Esta

divisão é possível pois ele utiliza o padrão de projeto *Model-View-Controller* - MVC³, 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.

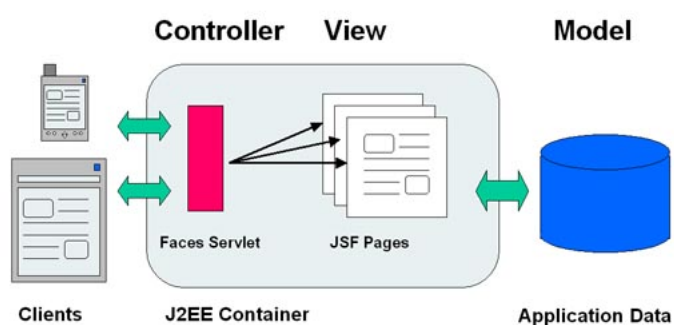


Figura 3 – Demonstração do Modelo MVC. **Fonte:** <http://www.javabeat.net/jsf-2/>.

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 Community Process* - JCP⁴, 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 são 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

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.

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

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. Ele possui uma ampla gama de componentes 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 páginas *web* deste trabalho.

2.2.3 Armazenamento de dados

A aplicação desenvolvida neste trabalho irá demandar o armazenamento de informações das costureiras e da disponibilidade das mesmas. Além disso, as saídas oferecidas pelo algo-

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

ritmo genético deverão ser armazenadas para uso posterior, alimentando, assim, a base de dados do sistema, fazendo-se necessário o uso de um banco de dados.

O gerenciador de banco de dados a ser utilizado nesta aplicação é o PostgreSQL, que foi escolhido por ser uma ferramenta robusta e *open source*. O PostgreSQL é um banco de dados relacional desenvolvido pela universidade da Califórnia por volta de 1970. Na época, o projeto se chamava Ingres e só passou a se chamar Postgres por volta de 1986 quando Michael Stonebraker adicionou o conceito de orientação a objetos ao projeto e decidiu então definir um novo nome para a nova versão (DOUGLAS; DOUGLAS, 2003).

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 especificamente 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, procedimentos, 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 a 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 estes se diferenciem dos demais. Dentre os fatores que viabilizam tais reduções está a otimização de processos que consiste em organizar os procedimentos relacionados à produção de forma que estes se tornem mais eficazes.

O software desenvolvido neste trabalho visa organizar uma linha de produção de forma que esta 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 em um tempo menor 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 estas sejam alocadas de forma a receberem 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 estas informações de forma a se produzir soluções para o problema e, no final, a melhor destas soluções será aquela que ofereça o menor custo e o menor tempo de produção. Porém, sabe-se que, em algoritmos genéticos, não há garantias que a solução encontrada é a melhor para se resolver o problema mas a tendência é que a solução proposta pelo algoritmo seja muito boa e 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 foi utilizado 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, obtêm 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 atenda 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.

A seção a seguir descreve como foi definido o escopo demonstrado acima.

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, este processo de geração de novas populações se repete até que seja atingido um número de gerações definido pelo programador. Este atributo representa esta quantidade;
- *elitism*: atributo do tipo *boolean* que representa se o algoritmo vai ter a função de elitismo. Esta 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. Este

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 Indivíduos Estrangeiros.
- *mutationRate*: como descrito no quadro teórico, a mutação é o fato de realizar pequenas alterações no indivíduo a fim de que este possa se tornar ainda melhor. Este parâmetro define uma porcentagem, geralmente baixa, que define quando o indivíduo sofrerá mutação ou não. Esta questão ficará mais clara logo abaixo, quando será explicado o passo-a-passo da execução do algoritmo.
- *mutationQuantity*: caso a mutação for ocorrer para o indivíduo, a alteração aleatória será feita nos cromossomos. Este parâmetro define quantos cromossomos do indivíduo deve ser alterado 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 escolhida pelo programador ao implementar o seu problema. No *framework* este 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 roleta;
- *crossType*: Assim como a seleção, existe 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 algoritmo deve utilizar, esta aplicação utilizará o método *Permutation*, desta forma somente um método de cruzamento foi implementado conforme mostra a seção Seleção, cruzamento e mutação;
- *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, 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 representa o indivíduo do problema deve herdar desta classe `Individual`. Fazendo isso, tal classe passará a ter uma lista de cromossomos e o atributo que representa o seu valor e a classe 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. Este 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 `IndividualPair` 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 pontos a seguir 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 estendido 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 `doMutation()` 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. Este processo para quando o número de gerações, definido no parâmetro `generationQuantity` do objeto `model` for atingido, neste caso 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 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.

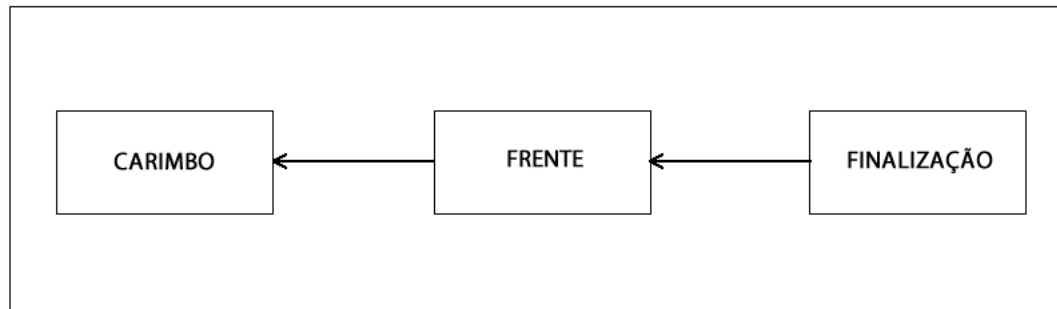


Figura 4 – Demonstração de um processo de fabricação **Fonte:** Desenvolvido pelos autores.

Uma questão importante que foi definida é que, independente da complexidade e tamanho do processo, este 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 e o custo gastos nesta 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 que, 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.

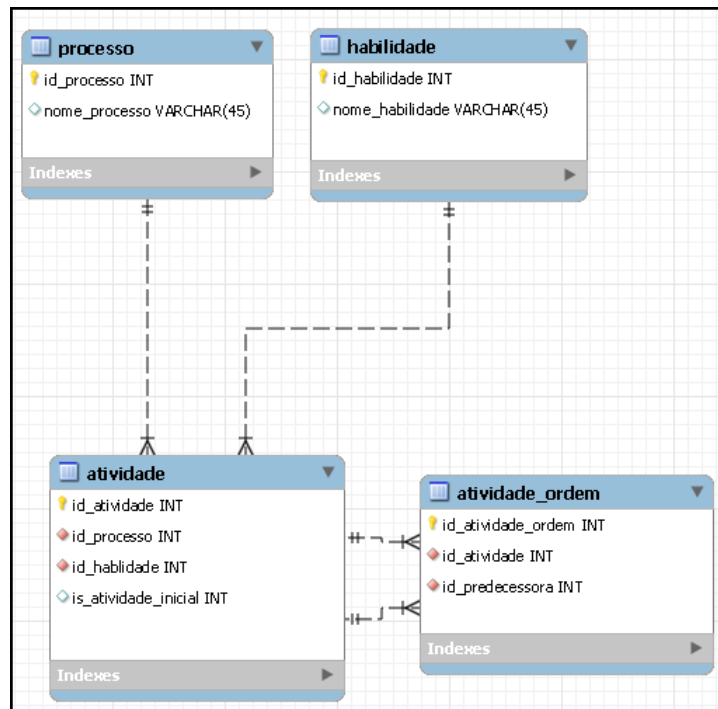


Figura 5 – Representação do processo de fabricação no banco de dados **Fonte:** Desenvolvido pelos autores.

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 diferentes 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_inicial*, 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 e, por fim, a tabela *atividade_ordem* é onde é feita a definição de ordem de execução das atividades.

A Figura 6 demonstra o mapeamento da relação entre a tabela *atividade* e *atividade_ordem* para o Java.

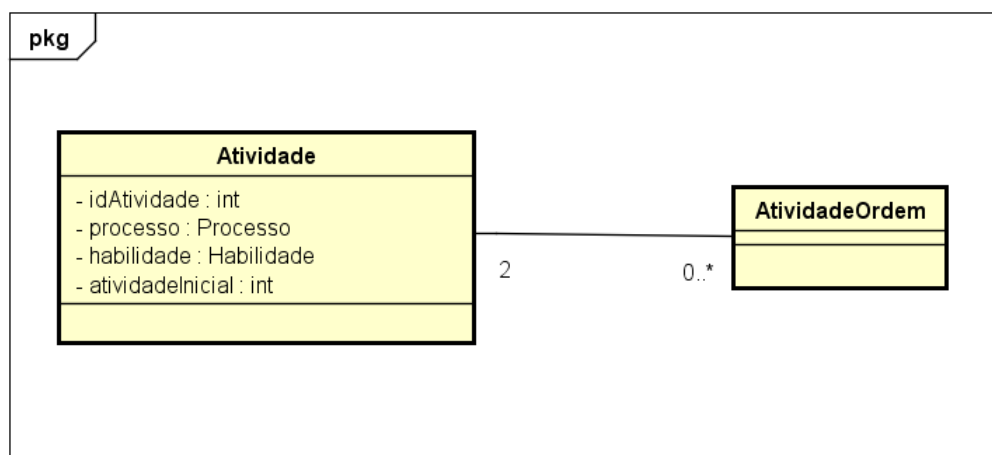


Figura 6 – Classes Atividade e AtividadeOrdem **Fonte:** Desenvolvido pelos autores.

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 acima, foi desenvolvido um algoritmo genético para a solução do problema de otimização. Conforme já demonstrado no Quadro Teórico, um algoritmo genético deve seguir uma ordem de execução conforme mostra a Figura 7.

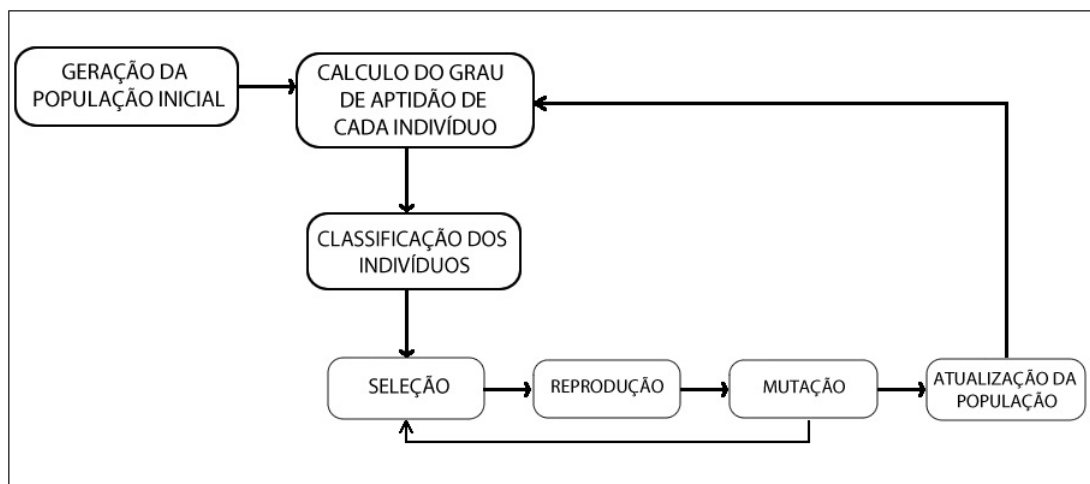


Figura 7 – Demonstração da execução de um AG **Fonte:** Desenvolvido pelos autores.

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

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 que descreve a construção da interface gráfica, o usuário irá iniciar a execução do algoritmo genético através de uma tela de distribuição de atividades, nesta tela 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 a 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 da início a execução deste. A Figura 8 ilustra o fluxo básico de execução da aplicação.

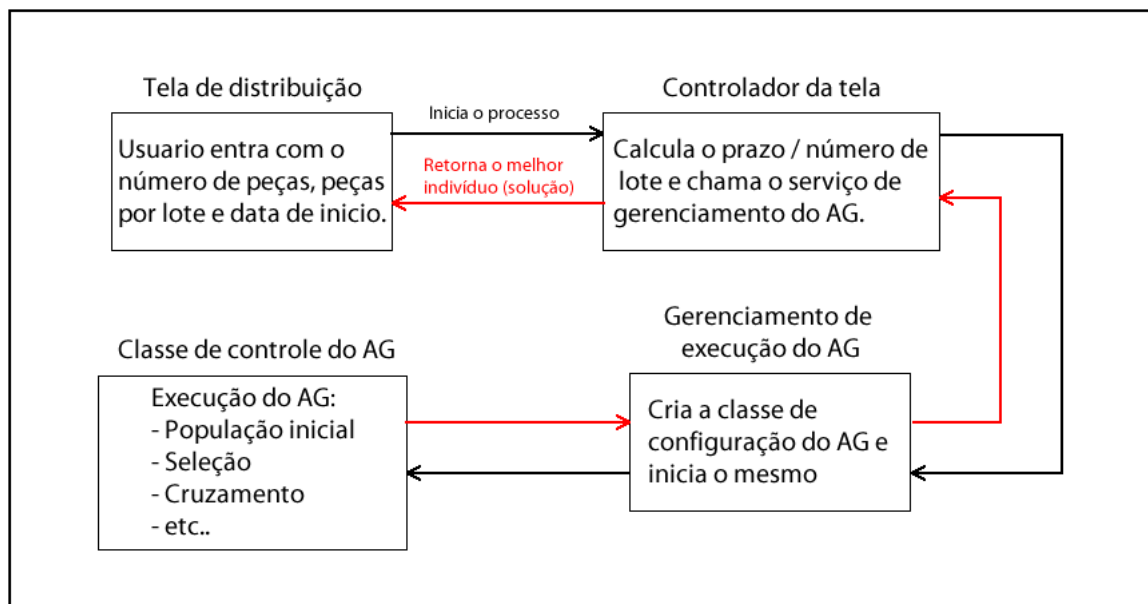


Figura 8 – Fluxo básico de execução **Fonte:** Desenvolvido pelos autores.

Conforme ilustrado na Figura 8, 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 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.

A classe criada para o gerenciamento da execução do algoritmo genético é denominada *GeneticAlgorithmManagement* e, conforme demonstrado na Figura 8, esta é instanciada pelo controlador da tela de distribuição de atividades. Tal classe possui um único método denominado *iniciarDistribuicao()* e recebe neste os dados informados pelo usuário, além disto, neste método, é criado um objeto de *ProcessoModel*, que herda da classe *GAModel* do *framework*, e, neste, 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 AG. O Código 3.1 demonstra o método *iniciarDistribuicao()*.

Código 3.1 – 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* 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 8.

A classe *ProcessoModel*, ao ser instanciada, recebe em seu construtor uma conexão para o banco de dados e o ID do processo a qual será executado o algoritmo. O construtor desta 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

tem a habilidade de fazer cada uma destas 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 3.2.

Código 3.2 – 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, 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 a execução do algoritmo genético. A primeira coisa a ser feita então é criar a população inicial de indivíduos que é criada 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 3.3.

Código 3.3 – 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. A realização desta distribuição foi realizada considerando que o total de peças a ser produzido deveria ser dividido em lotes e então, em cada atividade, este número de lote deveria ser distribuído entre as costureiras que possuísem 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.

Neste 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 estas 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 9.

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

Figura 9 – Exemplo de distribuição aleatória de lotes para as costureiras **Fonte:** Desenvolvido pelos autores.

Uma costureira pode não receber lote (:0), isso permite que a decisão de quem vai participar ou não também fique por conta do algoritmo. O exemplo demonstrado está considerando que se irá produzir 500 peças em lotes de 50, resultando assim em um total de 10 lotes.

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 desta, 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.

Neste 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 estes 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.

Neste caso, cada indivíduo da população irá representar uma forma de distribuir as atividades e cada número de lotes distribuídos a determinada costureira em uma determinada atividade irá representar um cromossomo. Tomando como base o exemplo da Figura 9, o quadro, como um todo, representa o indivíduo e cada distribuição, como por exemplo a Roberta, que recebeu 5 lotes para confeccionar, representa um cromossomo.

Para fazer esta representação em Java, primeiramente foi criada uma classe denominada *ProcessoChromosome* que é representada na Figura 10:

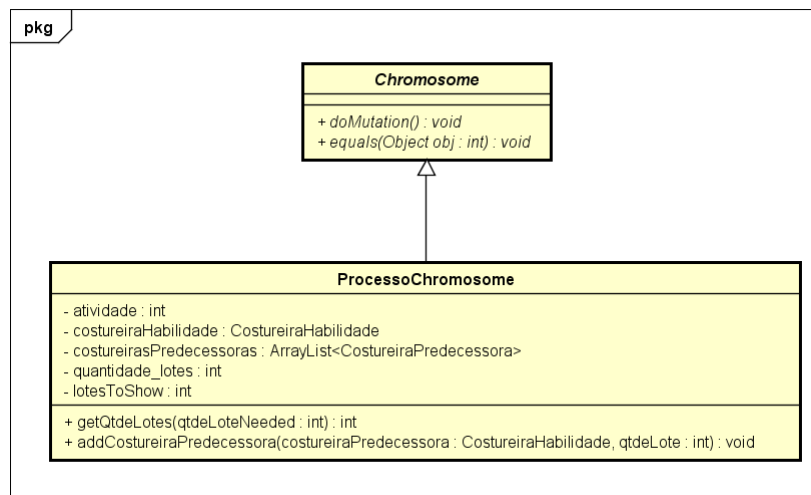


Figura 10 – Classe ProcessoChromosome **Fonte:** Desenvolvido pelos autores.

A classe **ProcessoChromosome** herda de **Chromosome** do *framework* descrito na Figura 10. 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` é do tipo `int` e representa o ID da atividade, no banco de dados, a 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, e faz 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, conforme demonstra a Figura 11.

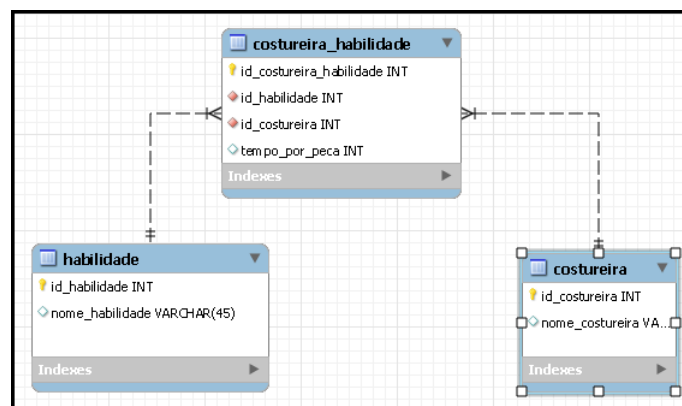


Figura 11 – Armazenamento de dados das costureiras **Fonte:** Desenvolvido pelos autores.

A classe **CostureiraHabilidade**, conforme ilustra a Figura 12, por sua vez, possui o atributo `habilidade`, outro que representa a costureira e um terceiro para representar o tempo

que tal costureira gasta para confeccionar uma peça com certa habilidade. 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, necessita ter o tempo que a costureira gasta para fazer a peça e este tempo pode variar para uma mesma costureira dependendo de suas habilidades.

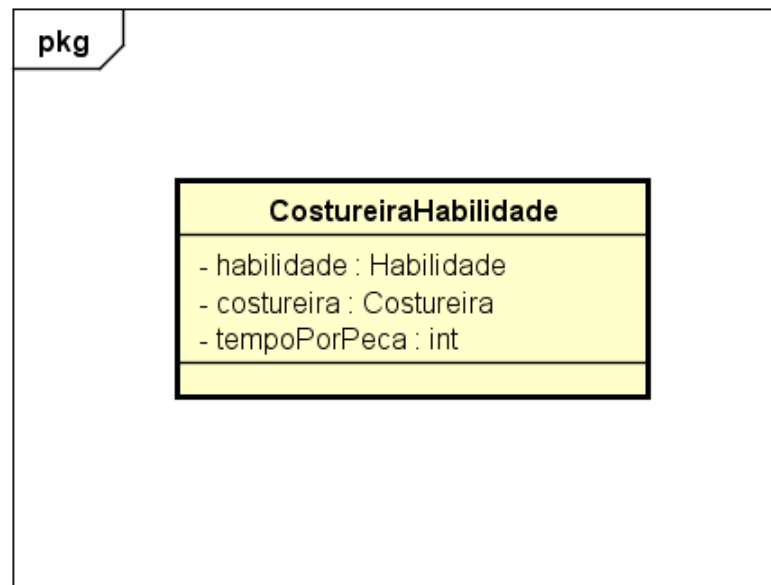


Figura 12 – Classe CostureiraHabilidade Fonte: Desenvolvido pelos autores.

Assim, para representar cada característica da solução, tomando como exemplo a Figura 9, o fato 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 Marta, o atributo habilidade que representa a habilidade em questão e a quantidade de lotes que Marta deverá confeccionar, que seria, neste caso, cinco.

A representação do indivíduo foi feita criando-se a classe ProcessoIndividual como demonstra a Figura 13:

A classe ProcessoIndividual herda da classe Individual do *framework*, e por isso, esta passa a ter um ArrayList com objetos do tipo Chromosome. Neste 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 para cada atividade costureira. O construtor da classe ProcessoIndividual recebe como parâmetro

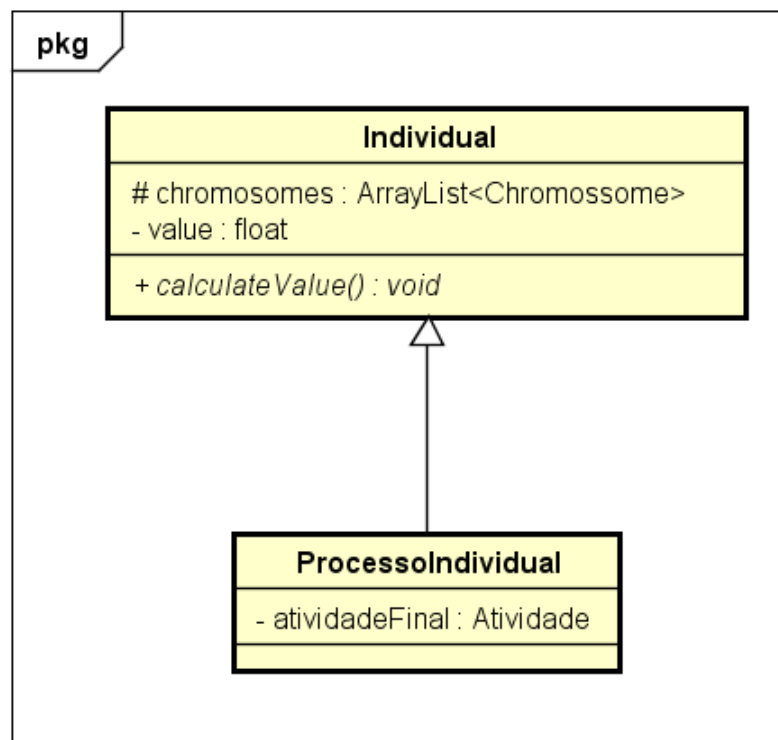


Figura 13 – Classe ProcessoIndividual **Fonte:** Desenvolvido pelos autores.

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 o tempo gasto por cada uma para 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, desta forma a distribuição seria feita da forma demonstrada na 3.4.

Código 3.4 – Criação de cromossomos (Primeira Abordagem). **Fonte:** Elaborado pelos autores.

```

1
2  package edu.univas.edu.tcc.ga_code;
3
4  import java.util.ArrayList;
5
6  public class ProcessoIndividual extends Individual {
7
8      public Atividade atividadeFinal;
9
10     public ProcessoIndividual(Atividade atividadeInicial,
11         Map<Integer, List<CostureiraHabilidade>> atividadesCostureiras){
12
13         chromosomes = new ArrayList<Chromosome>();
14         this.atividadeFinal = atividadeInicial;
  
```

```

15
16     for(Integer key: atividadesCostureiras.keySet()){
17         for(CostureiraHabilidade costureira :
18             atividadesCostureiras.get(key)){
19
20             Float porcentagem = (float) (Math.random() *1);
21             chromosomes.add(new ProcessoChromosome(key,
22                 costureira, porcentagem));
23         }
24     }
25 }
26 }

```

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 14, este cálculo seria feito da seguinte forma:

Finalização
Andrea:0,53%
Dita:0,44%
Cida:0,29%

Figura 14 – Distribuição em porcentagem **Fonte:** Desenvolvido pelos autores

- Primeiramente deveria ser feito 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, neste sentido o cálculo, já fazendo o arredondamento, seria:

$$0,53 / 1,26 = 0,42 \mid 0,44 / 1,26 = 0,35 \mid 0,29 / 1,26 = 0,23$$

Logo, neste 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:

$$(500 * 42) / 100 = 210 \mid (500 * 35) / 100 = 175 \mid (500 * 23) / 100 = 115$$

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

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

$$210 / 50 = 4 \mid 175 / 50 = 4 \mid 115 / 50 = 2$$

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, as 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 a nível de 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 3.5.

Código 3.5 – Distribuição em lotes diretamente. **Fonte:** Elaborado pelos autores.

```

1
2 public ProcessoIndividual(Atividade atividadeFinal,
3     BigDecimal prazoEmSegundos,
4     Map<Integer, List<CostureiraHabilidade>> atividadesCostureiras,
5     int numeroLote, int pecasPorLote){
6
7     chromosomes = new ArrayList<Chromosome>();
8     Map<Integer, ProcessoChromosome> chromossomosMap =
9         new HashMap<Integer, ProcessoChromosome>();
10
11     this.atividadeFinal = atividadeFinal;
12     this.numeroLote = numeroLote;
13     this.pecasPorLote = pecasPorLote;
14     this.prazo = prazoEmSegundos;
15
16     boolean distribuiuPorTodasCostureiras = false;
17
18     int qtdeLote = 0;

```

```

19  int cont = 0;
20
21  for (Integer key : atividadesCostureiras.keySet()) {
22      qtdeLote = numeroLote;
23      cont = 0;
24      int loteCostureira = 0;
25      distribuiuPorTodasCostureiras = false;
26
27      while (true){
28          /* Verificou-se que quando a qtdeLote era 1 o valor
29             sorteado nunca era zero usando o Math.random com
30             CAST para INT */
31          if(qtdeLote == 1){
32              loteCostureira = Math.round((float) Math.random() * 1);
33          }else{
34              loteCostureira = (int) (Math.random() * qtdeLote);
35          }
36
37          CostureiraHabilidade costureiraHabilidade =
38              atividadesCostureiras.get(key).get(cont);
39
40          ProcessoChromosome intermediario =
41              cromossomosMap.get(costureiraHabilidade.
42                  getIdCostureiraHabilidade());
43
44          if(intermediario == null){
45              ProcessoChromosome pc = new ProcessoChromosome
46                  (key, costureiraHabilidade, loteCostureira);
47
48              cromossomosMap.put
49                  (costureiraHabilidade.getIdCostureiraHabilidade(), pc);
50
51              chromosomes.add(pc);
52          }else{
53              int oldValue = intermediario.getQuantidade_lotes();
54              int newValue = oldValue + loteCostureira;
55              intermediario.setQuantidade_lotes(newValue);
56              intermediario.setLotesToShow(newValue);
57          }
58
59          if (cont == atividadesCostureiras.get(key).size() - 1) {
60              cont = -1;
61              distribuiuPorTodasCostureiras = true;
62          }
63
64          qtdeLote -= loteCostureira;
65
66          if(qtdeLote == 0 && distribuiuPorTodasCostureiras){
67              break;
68          }
69          cont++;
70      }
71 }

```

Conforme descrito no Código 3.5, 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 é criado objetos da classe ProcessoChromosome e colocado 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 desta forma, o algoritmo tendia a nunca distribuir zero lotes à última costureira, o que causou resultados ineficazes na distribuição. Alterando para esta forma, a distribuição passou a ser feita de maneira totalmente uniforme deixando o resultado coerente. Como é possível perceber, neste 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 9.

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 seções, ressaltando porém que no processo de cruzamento, os cromossomos do indivíduo é 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 suas localidades e então, através da fórmula

euclidiana, é possível calcular a distância entre elas. A Figura 15 demonstra os 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 15 – Demonstração de costureiras e habilidades **Fonte:** Desenvolvido pelos autores.

Com base nestes dados é realizado um cálculo a fim de se encontrar 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, tais valores serão utilizados 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 16, deveria ter nós representando cada atividade, as costureiras que trabalham em cada atividade e o número de lotes atribuídos a cada uma aleatoriamente pelo algoritmo.

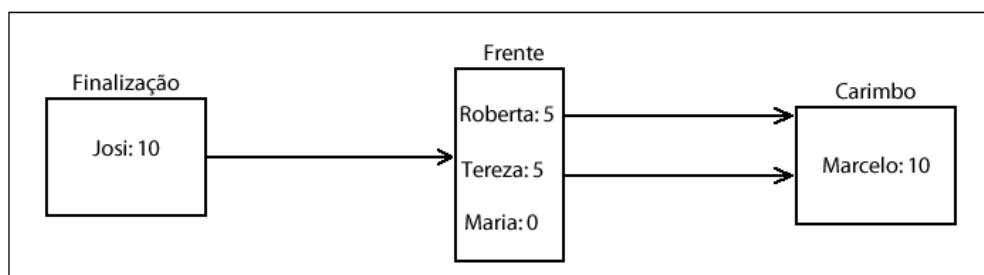


Figura 16 – Estrutura de representação da ordem de precedência **Fonte:** Desenvolvido pelos autores.

Como foi visto na seção 1.4.2, 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 3.6.

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 3.6, 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 esta 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 atividadeFinal e o número de peças por lote recebidos na criação do indivíduo, conforme descrito na seção anterior, e o map atividadeCromossomos conforme mostra a Código 3.6.

Código 3.6 – Método calculateValue(). Fonte: Elaborado pelos autores.

```

1
2  public void calculateValue() {
3      Map<Integer, List<Chromosome>> atividadeCromossomos
4          = new HashMap<Integer, List<Chromosome>>();
5
6      Integer lastAtividade = null;
7      float custoTotal = 0;
8      int totalPecasAProduzir = 0;
9      List<Chromosome> cromossomos = null;
10
11     /*Calcular o custo total*/
12     for(Chromosome chromosomeCusto : chromosomes){
13         totalPecasAProduzir = 0;
14         ProcessoChromosome processoChromossome =
15             (ProcessoChromosome) chromosomeCusto;
16
17         totalPecasAProduzir =
18             processoChromossome.getLotesToShow() * this.pecasPorLote;
19
20         custoTotal += totalPecasAProduzir *
21             processoChromossome.getCostureiraHabilidade()
22                 .getPrecoPorPeca();
23     }
24
25     for (Chromosome chromosome : chromosomes) {
26         ProcessoChromosome processoChromossome = (ProcessoChromosome)
27             chromosome;
28
29         if (lastAtividade == null || lastAtividade !=
30             processoChromossome.getAtividade()) {
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(atividadeFinal, atividadeCromossomos, this.
40         pecasPorLote);
41
42     setCusto(custoTotal);
43 }

```

```

41
42     /*So deve-se calcular o valor do individuo se
43     ele nao foi calculado ainda porque uma vez calculado
44     o valor o numero de lotes no objeto cromossomo foi
45     decrementado */
46     if (getValue() == 0) {
47         setValue(node.getValorTotal());
48         setRootNode(node);
49     }
50 }

```

A estrutura da classe Node foi realizada de forma a produzir objetos de si mesma de forma recursiva, assim cada vez que esta é instanciada, é como se criasse um nó daqueles mostrados na Figura 15. Assim, quando o método calculateValue() instancia um objeto Node, no construtor deste outros nós são criados, e então toda estrutura, como foi ilustrada na Figura 15, será criada. O Código 3.7 mostra a construção de um objeto Node.

Código 3.7 – Construtor da classe Node. Fonte: Elaborado pelos autores.

```

1
2     public Node(Atividade atividade, Map<Integer, List<Chromosome>>
3         atividadeCromossomos, int pecasPorLote){
4
5         this.atividade = atividade;
6         this.pecasPorLote = pecasPorLote;
7         cromossomos = atividadeCromossomos.
8             get(atividade.getIdAtividade());
9
10        Atividade atividadePredecessora = null;
11
12        for(AtividadeOrdem predecessora :
13            atividade.getAtividadeOrdemsForIdAtividade()){
14
15            atividadePredecessora =
16                predecessora.getAtividadePredecessora();
17
18            predecesoras.add(new Node(atividadePredecessora,
19                atividadeCromossomos, pecasPorLote));
20        }
21    }

```

A classe Node recebe em seu construtor um objeto de Atividade, que na primeira vez que o objeto 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 a atividade recebida e, por fim, faz uma iteração na lista de atividades predecessoras da atividade recebida e, recursivamente, cria novos nós, construindo assim, a estrutura demonstrada na Figura 15.

Como se pode ver no método calculateValue() no Código 3.6, após criar a estrutura de nós, é chamado o método getValorTotal() do objeto node criado. 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 pra quem e calculando o tempo de transporte de cada envio, conforme demonstra Código 3.8 .

Código 3.8 – Método getValorTotal(). Fonte: Elaborado pelos autores.

```
1
2 public long getValorTotal(){
3     long valor = 0;
4     long maior = 0;
5
6     for(Chromosome chromosome : cromossomos){
7         ProcessoChromosome processoChromosome =
8             (ProcessoChromosome) chromosome;
9
10        if(processoChromosome.getQuantidade_lotes() > 0){
11            valor = getCromossomeValue(processoChromosome ,
12                processoChromosome.getQuantidade_lotes());
13        }
14
15        if(valor > maior){
16            maior = valor;
17        }
18    }
19    return maior;
20 }
```

O método faz uma iteração na lista de cromossomos do nó da atividade final e irá chamar o método getCromossomeValue passando cada cromossomo e o valor de seus lotes, e irá retornar o valor do maior cromossomo.

Tomando como base a Figura 16, para facilitar o entendimento, o método getCromossomeValue() 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 realizar 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 3.9.

Código 3.9 – 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 3.10.

Código 3.10 – 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 3.11 mostra este método.

Código 3.11 – 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 17, a atividade anterior é a “Frente” e a primeira costureira da lista é a Roberta, neste caso a Josi solicita 10 lotes da parte da frente para a Roberta, porém, neste caso, a Roberta confeccionou somente 5 lotes, neste caso , a Josi irá consumir os 5 lotes confeccionados.

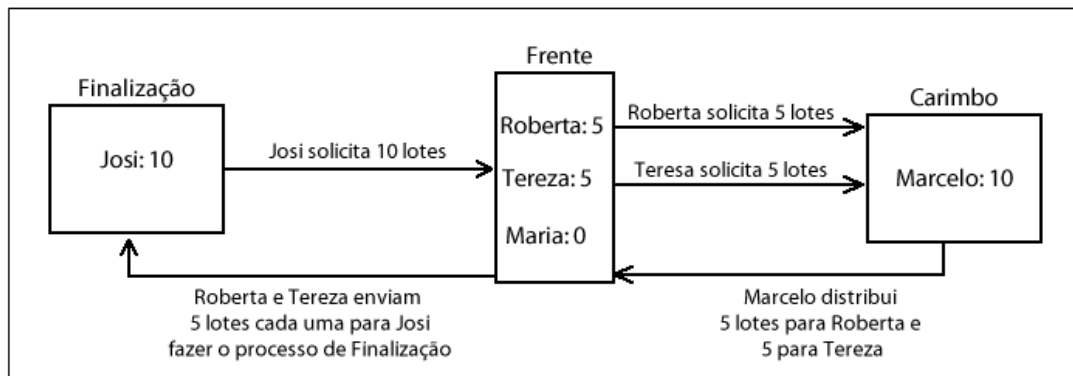


Figura 17 – Exemplo de solicitação de lotes predecessores **Fonte:** Desenvolvido pelos autores.

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 3.12 demonstra este método `calcularTempoEntreCostureiras()`.

Código 3.12 – Método `getValueChromosomosPredecessores()`. **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 }
```

Seguindo as execuções dos métodos já explicados, o fluxo de chamadas então será `getCromossomeValue()`, `getTempoRecebimentoPecas()`, `getValueChromosomosPredecessores()` e neste ponto o cromossomo Roberta irá solicitar para sua atividade anterior, 5 lotes, conforme mostra a Figura 17.

Neste caso, a atividade anterior é o Carimbo, neste ponto existe uma exceção, pois quando se solicita para atividade Carimbo, não são consumidos os lotes desta, 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 como o tempo de produção e o custo por peça do Marcelo é zero, só será considerado apenas o tempo de transporte entre o Marcelo e a Roberta.**

E assim, todo o processo é feito recursivamente, na qual as costureiras da primeira atividade vão consumindo os lotes das costureiras das atividades predecessoras, conforme mostra a Figura 17. Como a Roberta não conseguiu atender a Josi, uma vez calculado o tempo de produção ds 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 neste caso.

Concluindo, o processo é todo feito recursivamente, na qual as costureiras das primeiras atividades vão consumindo os lotes das costureiras das atividades predecessoras **e somado o**

tempo de produção e o tempo de transporte entre elas, conforme mostra a Figura 18.

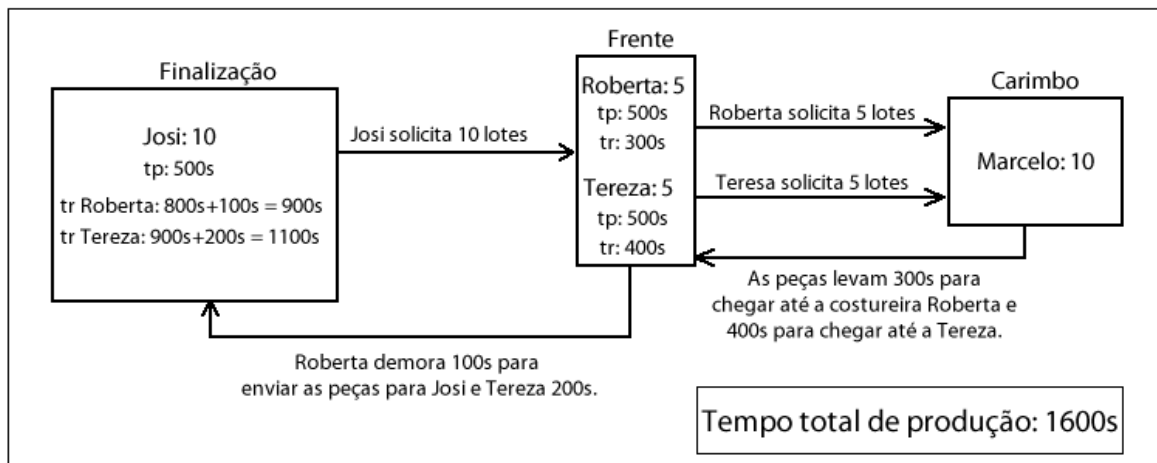


Figura 18 – Distribuição demonstrando o tempo **Fonte:** Desenvolvido pelos autores.

A recursividade para quando se chega na atividade Carimbo, então os valores começam a ser retornados. **No fim, prevalece sempre o maior valor** e assim o método `getValorTotal()` irá retornar o tempo total de produção das partes e o transporte das mesmas entre as costureiras e então este valor é colocado no atributo `value` 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 definido 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 3.13.

Código 3.13 – Método `classify()`. **Fonte:** Elaborado pelos autores.

```

1
2 public void classify(List<Individual> population){
3     List<Individual> populationAux = new ArrayList<Individual>();
4     List<Individual> populationGood = new ArrayList<Individual>();
5
6     for(Individual individual : population){
7         individual.calculateValue();
8     }
9     Collections.sort(population);
10
11

```

```

12  for(Individual individual : population){
13
14      ProcessoIndividual pi = (ProcessoIndividual) individual;
15      if(individual.getValue() <= pi.getPrazo().longValue()){
16          individual.setTipoDeClassificacao(
17              Constants.CLASSIFICACAO_POR_CUSTO);
18
19          populationGood.add(individual);
20      }
21  }
22  Collections.sort(populationGood);
23
24  for(Individual individual : populationGood){
25      individual.setTipoDeClassificacao(
26          Constants.CLASSIFICACAO_POR_TEMPO);
27  }
28
29  //Remove todos os objetos com tempo < prazo da populacao
30  population.removeAll(populationGood);
31
32  //Adiciona-os na populacao auxiliar
33  populationAux.addAll(populationGood);
34
35  //adiciona os demais indivíduos da populacao
36  populationAux.addAll(population);
37
38  //Redefine a populacao
39  population.clear();
40  population.addAll(populationAux);
41
42 }

```

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 `compareTo` da classe `Individual`, implementado na classe `ProcessoIndividual`, por ter sido declarado como abstrato, foi definido uma *flag* de forma a definir se os indivíduos devem ser ordenados pelo tempo total ou pelo custo, conforme mostra o Código 3.14.

Código 3.14 – 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 3.13, na linha 9, 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, este prazo foi passado pelo usuário e atribuído ao indivíduo no construtor da classe `ProcessoIndivíduo`, conforme já explicado na seção anterior.

Desta forma, de acordo com o Código 3.13, 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 disto 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 inconsistência 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.

Desta forma, o algoritmo irá 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.

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 3.15.

```

1
2 public Individual execute(){
3     model.createInitialPopulation();
4     Individual lastBest = null;
5
6     for(int i = 0; ; i++){
7         ArrayList<Individual> population = model.getPopulation();
8         ArrayList<Individual> newGeneration =
9             new ArrayList<Individual>();
10
11         classify(population);
12         if(lastBest == null || lastBest != population.get(0)){
13             lastBest = population.get(0);
14         }
15
16         //verifica o final da execucao
17         if(i == model.getGenerationQuantity()){
18             break;
19         }
20
21         //A partir deste ponto e iniciado a criacao de
22         //uma nova populacao
23
24         //elitismo
25         if(model.isElitism()){
26             doElistim(newGeneration);
27         }
28
29         int foreignQuantity = Math.round(
30             model.getPopulationSize() * model.getForeignIndividualRate());
31
32         foreignQuantity = foreignQuantity % 2 == 0 ?
33             foreignQuantity : foreignQuantity +1;
34
35         for(int j = 0;j < foreignQuantity;j++ ){
36             newGeneration.add(model.createIndividual());
37         }
38
39         while(newGeneration.size() < model.getPopulationSize()){
40             //selecao
41             Individual individual1 = doSelection();
42             Individual individual2;
43
44             do{
45                 individual2 = doSelection();
46             }while(individual1 == individual2);
47
48             //cruzamento
49             IndividualPair pair = doCrossing(individual1, individual2);
50
51             //mutacao
52             doMutation(pair.getIndividual1());
53             doMutation(pair.getIndividual2());
54
55             newGeneration.add(pair.getIndividual1());
56             newGeneration.add(pair.getIndividual2());
57         }
58
59         model.setPopulation(newGeneration);
60     }

```

```
61     return lastBest;
62 }
```

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* de desenvolvimento adotado, 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 36 do Código 3.15, 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.

3.4.7 Seleção de indivíduos e cruzamento

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 3.15. O cruzamento começa com a seleção dos indivíduos que devem participar deste processo, esta seleção é realizada por meio do método `doSelection()` mostrado no Código 3.16.

Código 3.16 – 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 3.17.

Código 3.17 – Método `doSelectionByClassification()`. Fonte: Elaborado pelos autores.

```
1
2  private Individual doSelectionByClassification() {
3      int maxValue = 0;
4
5      for(int i = 0; i < model.getPopulationSize(); i++){
6          maxValue += (i + 1);
7      }
8
9      double index = Math.random() * maxValue;
10     int cursor = 0;
11
12     for(int i = 0; i < model.getPopulationSize(); i++){
13         cursor += model.getPopulationSize() - 1;
14
15         if(index <= cursor){
16             return model.getPopulation().get(i);
17         }
18     }
19
20     return null;
21 }
```

Tal método é responsável pela seleção de um indivíduo para realização do cruzamento. Primeiramente é realizado 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. Feito esta 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 deste 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 o for, o indivíduo da posição `i` é retornado. A figura Figura 19 ilustra este processo.

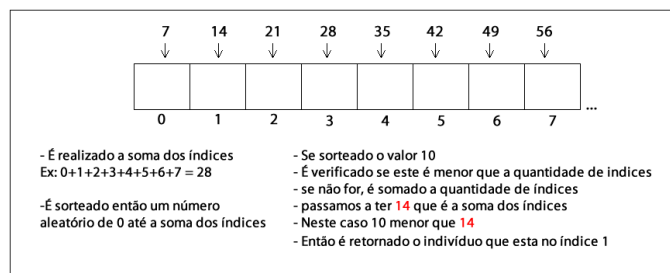


Figura 19 – Representação do processo de seleção dos indivíduos **Fonte:** Desenvolvido pelos autores.

Ao finalizar este processo um indivíduo terá sido escolhido, assim, de acordo com a linha 45 do Código 3.15, um novo 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 a nível de cromossomos, neste caso, 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. Isto foi necessário pois, para cada atividade, 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 a cada atividade e assim a cada cruzamento é criado dois novos indivíduos que irão fazer parte da nova população. A Figura 20 ilustra este processo.

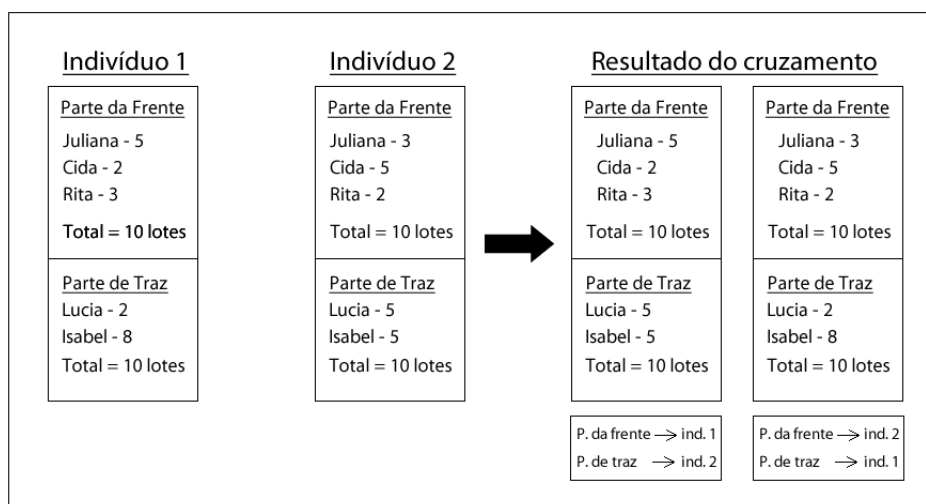


Figura 20 – Representação do processo de cruzamento dos indivíduos **Fonte:** Desenvolvido pelos autores.

A partir da linha 49 do código Código 3.15, 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 3.18.

Código 3.18 – Método doCrossing(). Fonte: Elaborado pelos autores.

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

Neste caso, o método escolhido para o cruzamento foi o de permutação, desta 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 3.19.

Código 3.19 – Método doPermutationCrossing(). Fonte: Elaborado pelos autores.

```
1
2  private IndividualPair doPermutationCrossing(
3      Individual individual1, Individual individual2){
4
5      ArrayList<Chromosome> chromosomes1 =
6          new ArrayList<Chromosome>();
7
8      ArrayList<Chromosome> chromosomes2 =
9          new ArrayList<Chromosome>();
10
11      Integer lastAtividade = null;
12      float chooseIndividual = 0;
13      int i = 0;
14
15      for(Chromosome chromosome : individual1.getChromosomes()){
16          ProcessoChromosome processoChromosome =
17              (ProcessoChromosome) chromosome;
18
19          if(lastAtividade == null ||
20             !lastAtividade.equals(processoChromosome.getAtividade())){
21
22              chooseIndividual = Math.round((float) Math.random() * 1);
23              lastAtividade = processoChromosome.getAtividade();
24          }
25
26          if(chooseIndividual == 1){
27              chromosomes1.add(individual1.getChromosomes().
28                  get(i).clone());
29
30              chromosomes2.add(individual2.getChromosomes().
```



```

31         get(i).clone());
32     }else{
33         chromosomes1.add(individual2.getChromosomes().
34             get(i).clone());
35
36         chromosomes2.add(individual1.getChromosomes().
37             get(i).clone());
38     }
39     i++;
40 }
41
42 Individual newIndividual1 =
43     model.createIndividual(chromosomes1);
44
45 Individual newIndividual2 =
46     model.createIndividual(chromosomes2);
47
48 return new IndividualPair(newIndividual1,newIndividual2);
49 }

```

Conforme visto na Figura 20, os cromossomos estão ordenados na lista de acordo com suas respectivas atividades, considerando isto, 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 3.19, 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 destas 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 a 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 3.20

Código 3.20 – Método `creatIndividual()` utilizando cromossomos. **Fonte:** Elaborado pelos autores.

```

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

```

Neste caso são passados quase todos os parâmetros enviados durante a criação da população inicial, a diferença é que, neste caso, o indivíduo será criado a partir dos cromossomos conforme mostra o Código 3.21.

Código 3.21 – Construtor de `ProcessoIndividual` utilizando cromossomos. **Fonte:** Elaborado pelos autores.

```
1
2 public ProcessoIndividual(Atividade atividadeInicial ,
3                           BigDecimal prazoEmSegundos ,
4                           ArrayList<Chromosome> chromosomes ,
5                           int numeroLote , int pecasPorLote) {
6
7     super(chromosomes);
8     this.atividadeFinal = atividadeInicial;
9     this.numeroLote = numeroLote;
10    this.pecasPorLote = pecasPorLote;
11    this.prazo = prazoEmSegundos;
12 }
```

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 um grupo de cromossomos de cada atividade do indivíduo, conforme mostra a Figura 21.

O processo de mutação inicia-se dentro do método `execute()` da classe `GAController`, e é realizado nos indivíduos criados, através do processo de cruzamento, 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()`, tal método verifica o tipo de mutação definido chamando o método correspondente, além disto, 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 3.22.

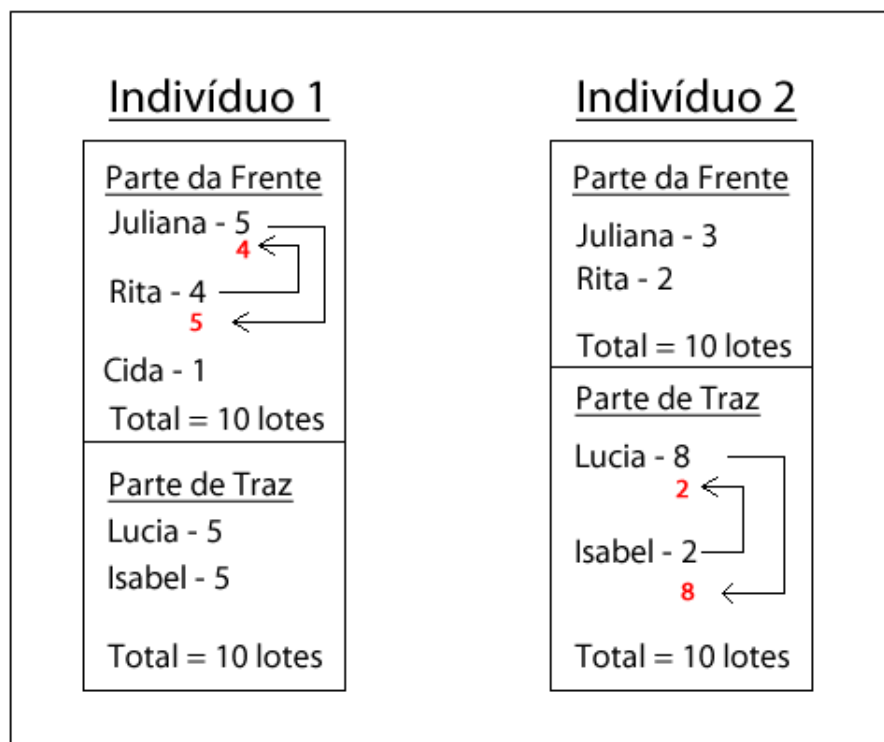


Figura 21 – Exemplo de mutação **Fonte**: Desenvolvido pelos autores.

Código 3.22 – Método doMutation(). **Fonte**: Elaborado pelos autores.

```

1
2 public void doMutation(Individual individual){
3     if(Math.random() < model.getMutationRate()){
4         for(int i = 0; i < model.getMutationQuantity();i++){
5             switch (model.getMutation()) {
6                 case BINARY: doMutationBinary(individual);break;
7                 case NUMERICAL: break;
8                 case PERMUTATION: doMutationPermutation(individual);break;
9             }
10        }
11    }
12 }

```

Neste caso, o método escolhido para a mutação foi o de permutação, desta 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 3.23.

Código 3.23 – Método doMutationPermutation(). **Fonte**: Elaborado pelos autores.

```

1
2 public void doMutationPermutation(Individual individual) {
3
4     Integer lastAtividade = null;
5     ArrayList<ProcessoChromosome> chromossomesToMutate =
6         new ArrayList<ProcessoChromosome>();
7
8     for(Chromosome chromosome : individual.getChromosomes()){
9         ProcessoChromosome processoChromosome =

```

```

10         (ProcessoChromosome) chromosome;
11
12         if(lastAtividade == null ||
13            !lastAtividade.equals(processoChromosome.getAtividade())){
14
15             if(!chromossomesToMutate.isEmpty() &&
16                chromossomesToMutate.size() > 1){
17
18                 doMutationOnChromossome(chromossomesToMutate);
19             }
20             chromossomesToMutate.clear();
21             lastAtividade = processoChromosome.getAtividade();
22         }
23         chromossomesToMutate.add(processoChromosome);
24     }
25     //Para o ultimo grupo
26     if(!chromossomesToMutate.isEmpty() &&
27        chromossomesToMutate.size() > 1){
28
29         doMutationOnChromossome(chromossomesToMutate);
30     }
31 }

```

Para a realização da lógica que realiza a mutação, é realizada uma iteração na lista de cromossomos do indivíduo a qual a mutação será realizada, dentro desta iteração, os cromossomos de uma atividade são adicionados à lista `chromossomesToMutate`, e assim, quando todos os cromossomos de uma atividade são adicionados nesta lista, o método `doMutationOnChromossome` demonstrado no Código 3.25, 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 esta e posteriormente sofram a mutação, logicamente, o grupo de atividades sofrerá mutação somente se possuir mais de um cromossomo.

Código 3.24 – Método `doMutationOnChromossome()`. Fonte: Elaborado pelos autores.

```

1
2
3     private void doMutationOnChromossome(
4         ArrayList<ProcessoChromosome> chromossomesToMutate){
5
6         int position1;
7         int position2;
8         int varAux;
9         int varAux2;
10
11         ProcessoChromosome chromosome1 = null;
12         ProcessoChromosome chromosome2 = null;
13
14         position1 = (int) (Math.random() * (chromossomesToMutate.size()));
15
16         do{
17             position2 = (int) (Math.random() *
18                (chromossomesToMutate.size()));
19         }while(position1 == position2);
20

```

```

21     chromosome1 = chromossomesToMutate.get(position1);
22     chromosome2 = chromossomesToMutate.get(position2);
23
24     varAux    = chromosome2.getQuantidade_lotes();
25     varAux2   = chromosome2.getLotesToShow();
26
27     chromosome2.setQuantidade_lotes(
28         chromosome1.getQuantidade_lotes());
29
30     chromosome2.setLotesToShow(chromosome1.getLotesToShow());
31
32     chromosome1.setQuantidade_lotes(varAux);
33     chromosome1.setLotesToShow(varAux2);
34 }

```

Neste método, é selecionado quais cromossomos terão seus valores trocados entre si. O segundo cromossomo é escolhido dentro de um `doWhile()` 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`, tem seus valores trocados entre si.

3.4.9 Interface gráfica de distribuição

Para a interação do usuário com o algoritmo de distribuição e possibilidade de cadastro de cadastro de fluxos de processos e costureiras assim como suas respectivas habilidades, tempo e preço de produção, foi realizado uma aplicação em plataforma web utilizando *JSF* e *Primefaces*, porém a forma que o mecanismo de cadastro das informações foi realizado não é relevante ser apresentado mas é importante 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 22.

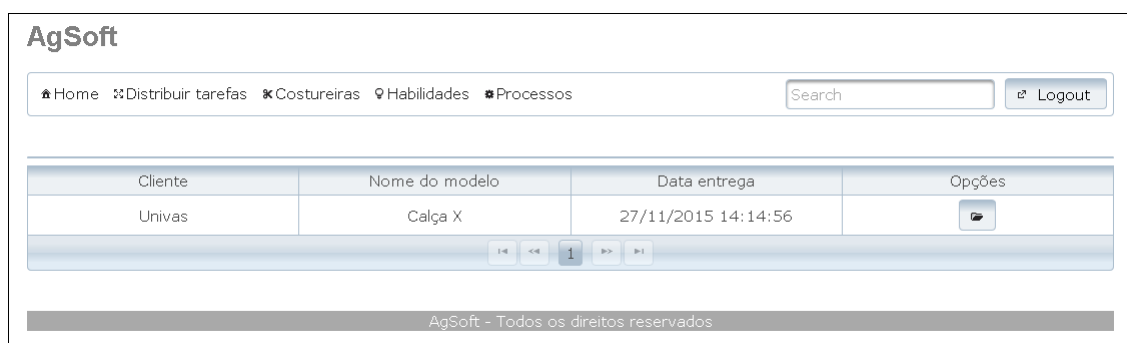


Figura 22 – Menu Distribuir tarefas **Fonte:** Desenvolvido pelos autores.

Ao clicar no ícone "abrir", é aberta uma tela de distribuição para aquele processo, esta tela é responsável por captar alguns dados importantes para a regra de negócio do algoritmo genético, além disto, é nela que as informações do melhor indivíduo (melhor solução) são apresentadas. Os dados informados através desta 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 23.

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

Toda tela do 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 3.25 demonstra o documento XHTML da tela demonstrada acima.

Código 3.25 – Método doMutationOnChromossome(). **Fonte:** Elaborado pelos autores.

```

1
2 <?xml version="1.0" encoding="UTF-8"?>
3 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
4 xmlns:ui="http://java.sun.com/jsf/facelets"
5 xmlns:f="http://java.sun.com/jsf/core"
6 xmlns:h="http://java.sun.com/jsf/html"
7 xmlns:p="http://primefaces.org/ui">
8
9 <h:form id="formAbrirProcessoDis">
10   <p:messages autoUpdate="true"></p:messages>
11
12   <p:panelGrid columns="1">
13
14     <div id="distribuicaoFields">
15       <p:fieldset legend="Forma de distribuicao">
16         <p:panelGrid columns="4">
17           <p:inputText id="nPecas" required="true"
18             value="#{distribuicaoController.totalPecas}"
19             requiredMessage="Por favor informe o total de pecas"
20             maxLength="19" placeholder="Numero de pecas" />
21
22           <p:inputText id="totalPecaPorLote" required="true"
23             value="#{distribuicaoController.totalPecasPorLote}"
24             requiredMessage="Por favor informe o total
25               de pecas por lote"
26             maxLength="19" placeholder="Total de pecas por lote"/>
27

```

```

28
29         <p:calendar id="inicio" required="true"
30             mask="true" effect="fold"
31             value="#{distribuicaoController.dataInicio}"
32             requiredMessage="Por favor informe uma data de inicio"
33             pattern="dd/MM/yyyy HH:mm:ss"
34             mindate="#{processosController.getCurrentDate()}"
35
36             maxdate="#{distribuicaoController.getDataEntrega()}"
37             placeholder="Data de inicio"/>
38
39         <p:commandButton action="#{distribuicaoController.
40             iniciarDistribuicao()}"
41
42             value="Iniciar distribuicao"
43             update="formAbrirProcessoDis"
44             ajax="true" onstart="PF('waitDialog').show();"
45             onsuccess="PF('waitDialog').hide();"/>
46
47     </p:panelGrid>
48 </p:fieldset>
49 </div>
50 .
51 .
52 .

```

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 notação `@ManagedBean(name = "[nome da classe em minusculo]")`. No caso da tela de distribuição, a classe controladora é denominada `DistribuicaoController`, portanto possui a notação `@ManagedBean(name = "[distribuicaoController]")` 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 desta os dados informados pelo usuário serão atribuídos, realizando assim o *bind*, neste, caso os dados informados pelo usuário estarão respectivamente disponíveis nos atributos `totalPecas`, `totalPecasPorLote` e `dataInicio` da classe `DistribuicaoController`.

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 acima. Esta TAG possui o atributo *action* que é responsável por chamar um método da classe controladora para executar uma determinada ação, neste caso, o método chamado é o `iniciarDistribuicao()` responsável por iniciar a distribuição das atividades chamando o gerenciador do algoritmo genético, conforme mostra o Código 3.26.

Código 3.26 – 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 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 disto, 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 ??.

Código 3.27 – 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)
6         .getSeconds());
7 }
```

É importante observar que o prazo é calculado em segundos, devido ao fato 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, neste 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 cálculo do número de lotes e do prazo, é instanciado um objeto da classe `GeneticAlgorithmManagement()`, conforme explicado anteriormente, e então o método `iniciarDistribuição` 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 na tela após a distribuição, tais métodos simplesmente pegam as atividades e suas respectivas costureiras juntamente com o número de lote que cada uma recebeu e apresenta ao usuário, além disto também é apresentado o tempo e o custo total, além de uma tabela de detalhes da distribuição. A Figura 24 mostra a tela com o resultado da execução do algoritmo.

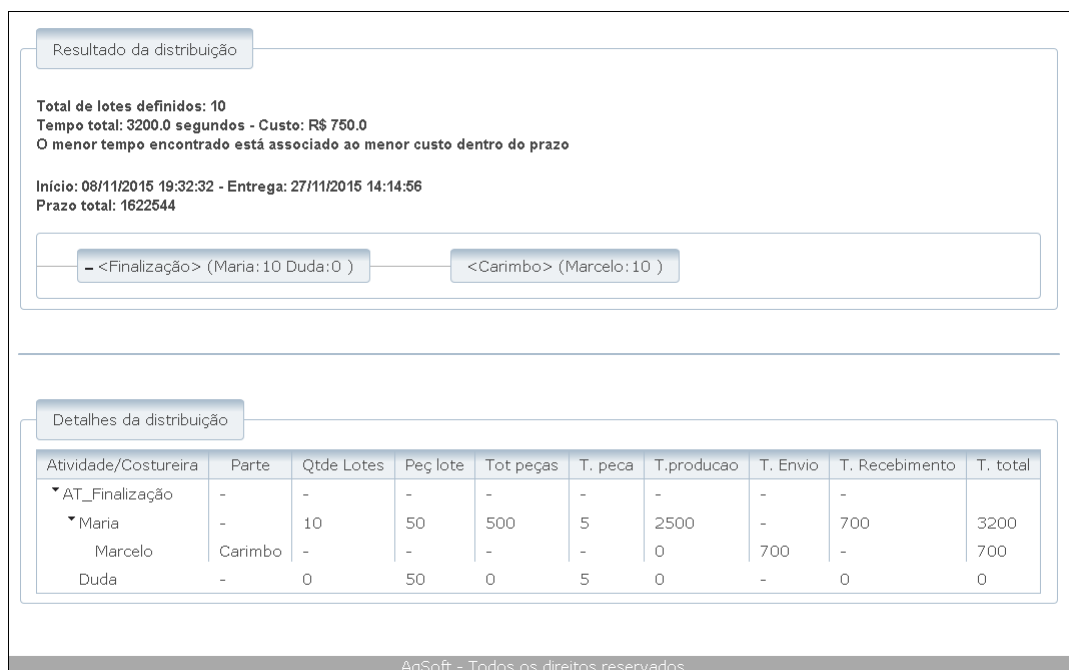


Figura 24 – Tela de resultado de distribuição de tarefas **Fonte:** Desenvolvido pelos autores.

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

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 deste 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, a 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 e constatamos que um sistema Web seria mais cômodo de ser utilizado por não precisar de nenhuma instalação por parte do usuário e este poder acessar o programa de qualquer lugar desde que estivesse conectado à internet.

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

Inicialmente foi tomado como base o caso do microempresário citado acima, todavia logo após iniciarmos o trabalho, o mesmo fez uma reestruturação de processos em sua empresa. Desta forma, sua maneira de trabalhar deixou de ser um cenário o qual algoritmos genéticos pudessem ser aplicados, desta forma, 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.

Após todas as etapas descritas acima, 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 conforme mostra a Figura 25.

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 na distribuição de tarefas, conforme mostra as seções seguintes.



Figura 25 – Tela inicial da aplicação **Fonte:** Desenvolvido pelos autores

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 27:

[Home](#)
[Distribuir tarefas](#)
[Costureiras](#)
[Habilidades](#)
[Processos](#)

[Logout](#)

Novo Processo

Cliente	Nome do modelo	Data entrega	Opções
Univas	Calça X	27/11/2015 14:14:16	<input type="button" value="✎"/> <input type="button" value="✖"/> <input type="button" value="🖨"/>

1

AgSoft - Todos os direitos reservados

Figura 26 – 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 27:

192.168.0.101:8080/fabricaCalcas/pages/abrirProcesso.jsf?idProcesso=97

AgSoft

[Home](#)
[Distribuir tarefas](#)
[Costureiras](#)
[Habilidades](#)
[Processos](#)

[Logout](#)

Nova Atividade

Atividade	Habilidade	Opções
No records found.		

1

Fluxograma

Atividade: Atividade Predecessora:

Fluxo do processo

```

graph LR
    Finalização[Finalização] --> Carimbo[Carimbo]
  
```

AgSoft - Todos os direitos reservados

Figura 27 – Detalhes do processo cadastrado **Fonte:** Desenvolvido pelos autores

Após a criação do processo foram definidas quais costureiras possuem a habilidade para realizar as atividades que compõe o processo, definido 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 28. 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.

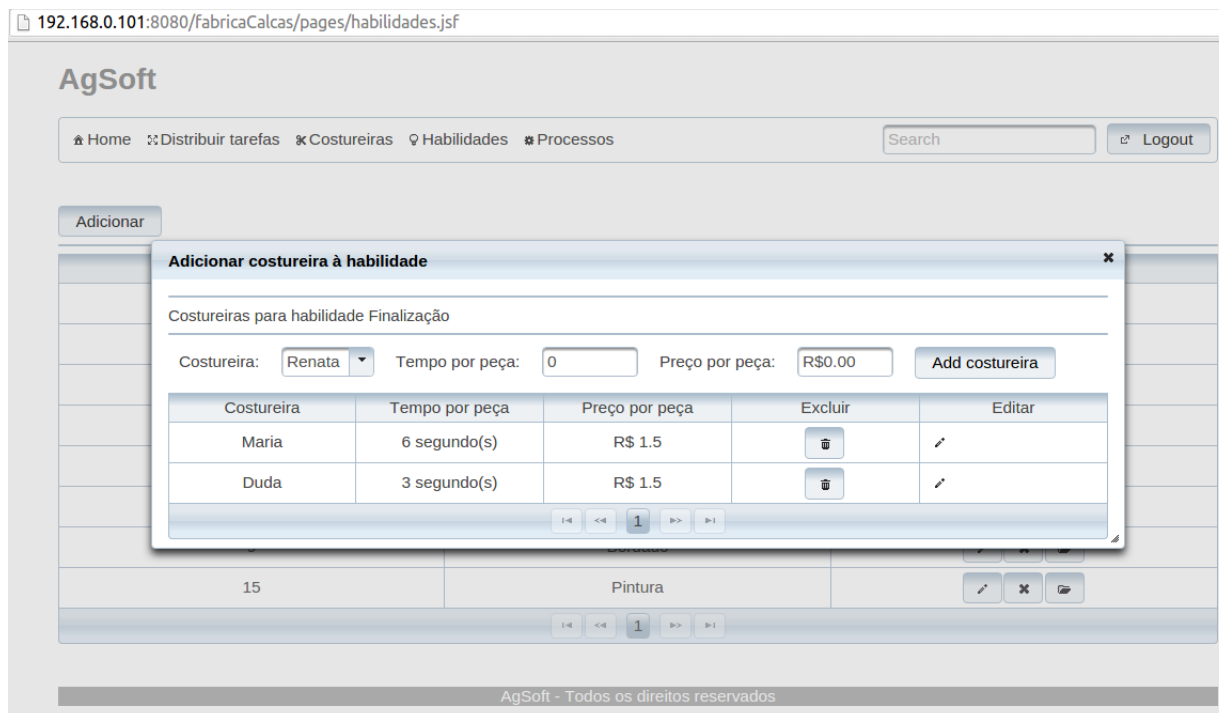


Figura 28 – Demonstração inserir costureira à habilidade **Fonte:** Desenvolvido pelos autores

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 29:

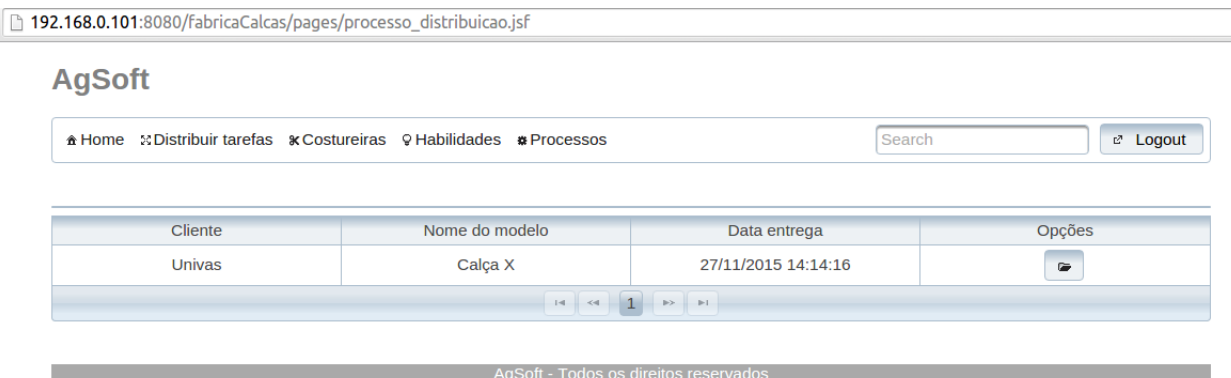


Figura 29 – Demonstração tela de distribuição de tarefas **Fonte:** Desenvolvido pelos autores

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 início. Depois de inseridos os dados ao clicar no botão Iniciar distribuição, o sistema executa toda a parte de algoritmos genéticos e retorna para o usuário a melhor solução encontrada com base nos dados inseridos conforme mostra a Figura 30:

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos

Search

Logout

Forma de distribuição

500

50

07/11/2015 15:31:33

Iniciar distribuição

Resultado da distribuição

Total de lotes definidos: 10

Tempo total: 1600.0 segundos - Custo: R\$ 750.0

O menor tempo encontrado está associado ao menor custo dentro do prazo

Início: 07/11/2015 15:31:33 - Entrega: 27/11/2015 14:14:16

Prazo total: 1723363

<Finalização> (Maria:3 Duda:7)

<Carimbo> (Marcelo:10)

Figura 30 – Resultado da distribuição de lotes **Fonte:** Desenvolvido pelos autores

Os valores então foram definidos, a distribuição foi realizada, e o resultado apresentado . Conforme ilustrado na Figura 28, Maria possui o tempo de produção maior que Duda, por isso ela recebeu um número menor de lotes para produzir conforme ilustrado na Figura 30

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 31.

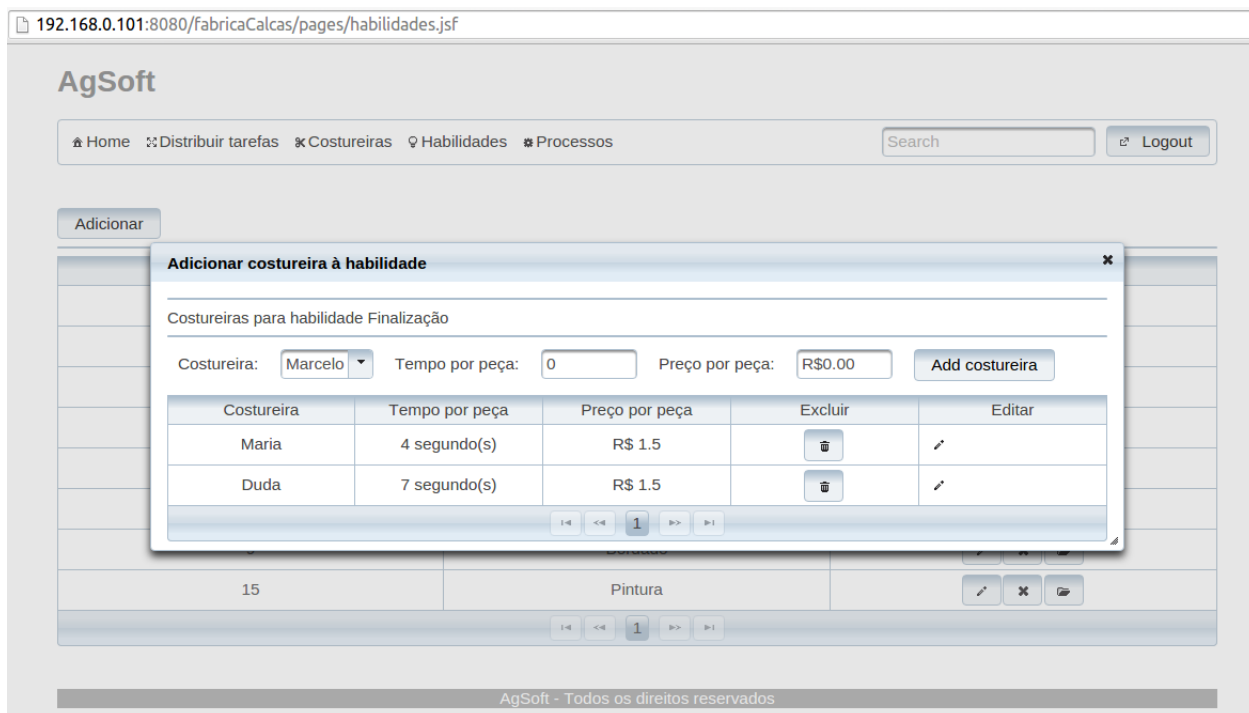


Figura 31 – Tempo de produção entre as costureiras **Fonte:** Desenvolvido pelos autores

Após executar novamente a distribuição foi retornado um novo resultado ilustrado na Figura 32:

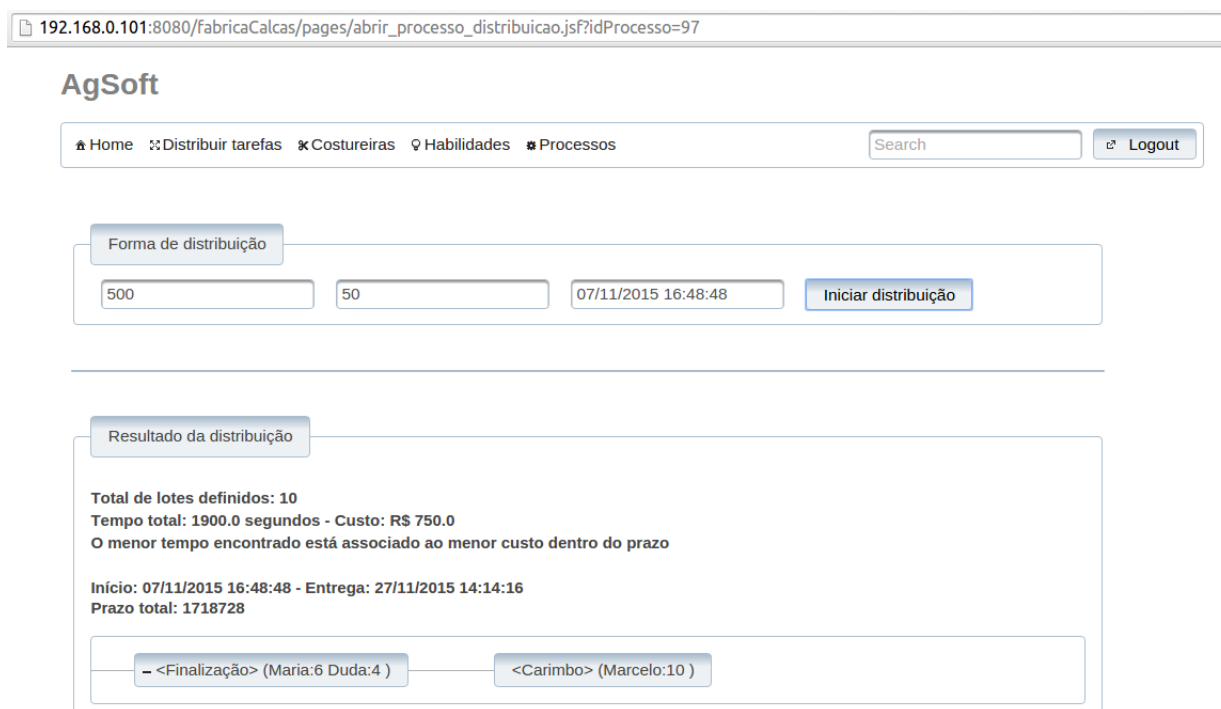


Figura 32 – Resultado da distribuição de lotes **Fonte:** Desenvolvido pelos autores

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 realizado 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 33 mostra a definição de tempo e custo para ambas as costureiras:

192.168.0.101:8080/fabricaCalcas/pages/habilidades.jsf

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos Search Logout

Adicionar

Adicionar costureira à habilidade

Costureiras para habilidade Finalização

Costureira: Marcelo Tempo por peça: 0 Preço por peça: R\$0.00 Add costureira

Costureira	Tempo por peça	Preço por peça	Excluir	Editar
Maria	5 segundo(s)	R\$ 1.5		
Duda	5 segundo(s)	R\$ 2.7		

15 Pintura

AgSoft - Todos os direitos reservados

Figura 33 – Custo entre as costureiras atividade Finalização **Fonte:** Desenvolvido pelos autores

Ao executar a distribuição é retornado a seguinte resultado ilustrado na Figura 34:

AgSoft

[Home](#)
[Distribuir tarefas](#)
[Costureiras](#)
[Habilidades](#)
[Processos](#)

Search

Logout

Forma de distribuição

500

50

07/11/2015 17:03:20

Iniciar distribuição

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: 07/11/2015 17:03:20 - Entrega: 27/11/2015 14:14:16

Prazo total: 1717856

<Finalização> (Maria:10 Duda:0)

<Carimbo> (Marcelo:10)

Figura 34 – Custo entre as costureiras atividade Finalização **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 35.

localhost:8080/fabricaCalcas/pages/abrir_processo_distribuicao.jsf?idProcesso=98

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos
Search
Logout

Forma de distribuição

500
50
08/11/2015 00:26:16
Iniciar distribuição

Resultado da distribuição

Total de lotes definidos: 10
Tempo total: 1950.0 segundos - Custo: R\$ 750.0
O menor tempo encontrado está associado ao menor custo dentro do prazo

Início: 08/11/2015 00:26:16 - Entrega: 27/11/2015 14:14:56
Prazo total: 1691320

- <Finalização> (Maria:5 Duda:5)
<Carimbo> (Marcelo:10)

Figura 35 – Custo entre as costureiras atividade Finalização **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 36

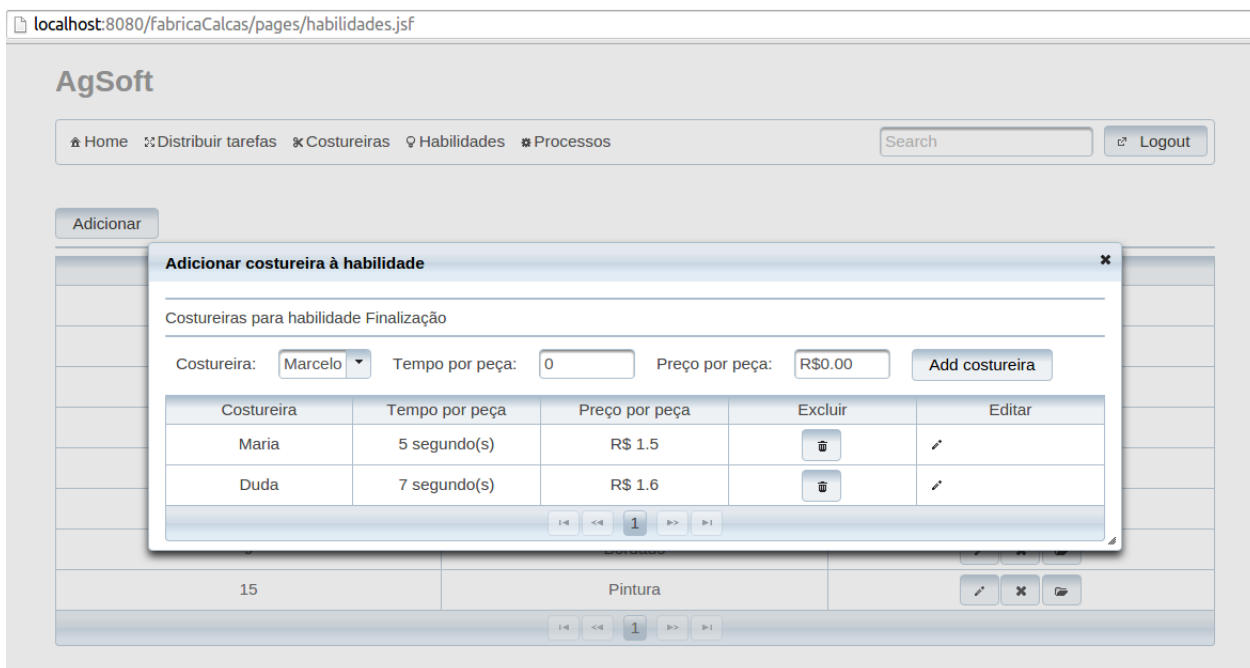


Figura 36 – Custo entre as costureiras atividade Finalização **Fonte:** Desenvolvido pelos autores

A Figura 37 mostra o resultado da distribuição com base nas configurações da habilidade Finalização mostrada na Figura 36.

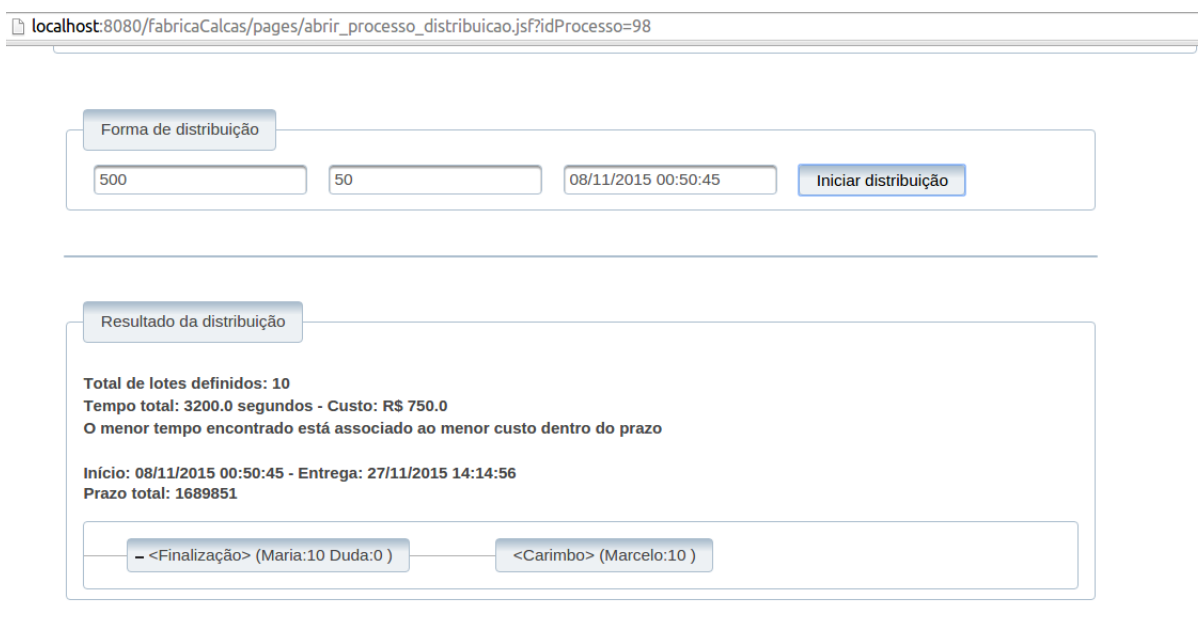


Figura 37 – Custo entre as costureiras atividade Finalização **Fonte:** Desenvolvido pelos autores

Como ilustrado na Figura 37, 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 38 mostra a distribuição de lotes entre as costureiras com a alteração na data de início do processo.

The screenshot displays the AgSoft web application interface. At the top, the browser address bar shows 'localhost:8080/fabricaCalcas/pages/abrir_processo_distribuicao.jsf?idProcesso=98'. The application header includes the 'AgSoft' logo, a navigation menu with links for Home, Distribuir tarefas, Costureiras, Habilidades, and Processos, a search bar, and a Logout button. The main content area is divided into two sections. The first section, titled 'Forma de distribuição', contains input fields for '500' and '50', a date and time field set to '27/11/2015 13:30:00', and an 'Iniciar distribuição' button. The second section, titled 'Resultado da distribuição', displays the following information: 'Total de lotes definidos: 10', 'Tempo total: 2450.0 segundos - Custo: R\$ 765.0', and a message stating 'O menor tempo encontrado está associado ao menor custo dentro do prazo'. It also shows the start and end times: 'Início: 27/11/2015 13:30:00 - Entrega: 27/11/2015 14:14:56' and 'Prazo total: 2696'. At the bottom of this section, there are two buttons: '<Finalização> (Maria:7 Duda:3)' and '<Carimbo> (Marcelo:10)'.

Figura 38 – Custo entre as costureiras atividade Finalização **Fonte:** Desenvolvido pelos autores

Como ilustrado na Figura 38 a costureira Duda recebe 3 lotes para produzir, assim, o custo de produção aumenta com relação ao resultado da Figura 37, 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, a distancia entre a fábrica pode determinar a distribuição dos lotes.













Para isto, foi necessário cadastrar um valor X e Y para cada costureira, conforme mostra a Figura 39.

localhost:8080/fabricaCalcas/pages/costureiras.jsf

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos Search Logout

Adicionar

Nome	Posição X	Posição Y	Opções
Marcelo	8	2	 
Maria	5	9	 
Renata	4	7	 
Roberta	8	7	 
Lucia	5	8	 
Duda	10	20	 

AgSoft - Todos os direitos reservados

Figura 39 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

Neste teste foi considerado apenas o tempo de transporte, logo o tempo de confecção por peça das costureiras da atividade Finalização possuem o mesmo valor de tempo e custo, conforme mostra a Figura 40.

localhost:8080/fabricaCalcas/pages/habilidades.jsf

AgSoft





Home Distribuir tarefas Costureiras Habilidades Processos Search Logout

Adicionar

Adicionar costureira à habilidade

Costureiras para habilidade Finalização

Costureira: Marcelo Tempo por peça: 0 Preço por peça: R\$0.00 Add costureira

Costureira	Tempo por peça	Preço por peça	Excluir	Editar
Maria	7 segundo(s)	R\$ 1.6		
Duda	7 segundo(s)	R\$ 1.6		

15 Pintura

AgSoft - Todos os direitos reservados

Figura 40 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

Feito isto, foi iniciado então o processo de distribuição das atividades, através do menu Distribuição de Tarefas, a Figura 41 mostra o resultado obtido.

The screenshot shows a web application interface for 'AgSoft'. At the top, there is a navigation bar with links: Home, Distribuir tarefas, Costureiras, Habilidades, and Processos. A search bar and a 'Logout' button are also present. Below the navigation bar, there is a section titled 'Forma de distribuição' (Distribution Form) with input fields for '500', '50', and a date/time '08/11/2015 09:39:48'. A button labeled 'Iniciar distribuição' (Start distribution) is next to these fields. Below this, there is a section titled 'Resultado da distribuição' (Distribution Result) which displays the following information: 'Total de lotes definidos: 10', 'Tempo total: 3150.0 segundos - Custo: R\$ 800.0', and 'O menor tempo encontrado está associado ao menor custo dentro do prazo'. It also shows the start time 'Início: 08/11/2015 09:39:48 - Entrega: 27/11/2015 14:14:56' and the total time 'Prazo total: 1658108'. At the bottom of this section, there are two buttons: '<Finalização> (Maria:7 Duda:3)' and '<Carimbo> (Marcelo:10)'.

Figura 41 – Distribuição das atividades Fonte: Desenvolvido pelos autores

Como mostrado na figura 39, a costureira Duda fica mais distante da fábrica por isso recebe menos lotes para produzir em relação a 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 42 mostra que o tempo de envio da peças para Duda é maior que o tempo de envio para Maria, comprovando o resultado mostrado na Figura 41.

The screenshot shows the same web application interface as Figure 41, but with the 'Detalhes da distribuição' (Distribution Details) section expanded. This section contains a table with the following data:

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	700
▼ Duda	-	3	50	150	7	1050	-	1800	2850
Marcelo	Carimbo	-	-	-	0	1800	-	1800	1800

Figura 42 – Distribuição das atividades Fonte: Desenvolvido pelos autores

Como mostrado na Figura 42, o tempo de envio das peças para Duda é 1800 segundos e para Maria 700, por isso Duda recebe menos lotes para ser produzidos.

4.5 Teste adicionando mais atividades ao processo

Este teste foi realizado para demonstrar a adição de mais atividades no processo. Este teste segue o mesmo processo da seção 4.1.

Neste caso deve-se considerar o tempo de transporte entre as costureiras da atividade Frente e a fábrica para a retirada dos materiais e o tempo entre tais costureiras e a costureira responsável pela Finalização, para isso foi adicionado no processo a atividade Frente conforme mostra a Figura 43.

localhost:8080/fabricaCalcas/pages/abrirProcesso.jsf?idProcesso=98

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos Search Logout

Nova Atividade

Atividade adicionada com sucesso!

Atividade	Habilidade	Opções
AT_Frente	Frente	

Fluxograma

Atividade adicionada com sucesso!

Atividade: AT_Finalização Atividade Predecessora: AT_Frente Adicionar

Fluxo do processo

Finalização Carimbo

Figura 43 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

Após a adição da atividade, é preciso adicioná-la ao fluxo do processo como predecessora da atividade Finalização. Como atividade Finalização é sempre a última atividade do processo, todas as atividades que forem incluídas sempre serão predecessoras a ela.

A Figura 44 mostra a atividade Frente adicionada no fluxo do processo.

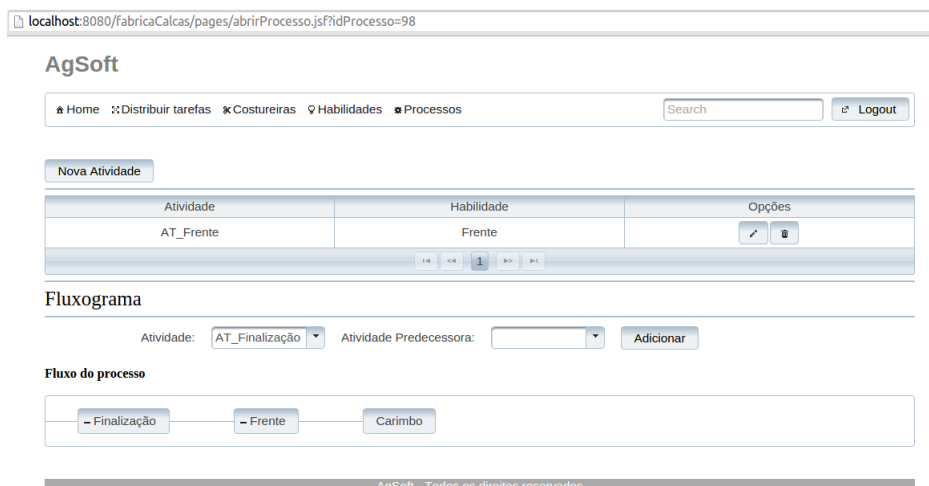


Figura 44 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

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

A Figura 45 mostra as costureiras adicionadas na habilidade Frente.

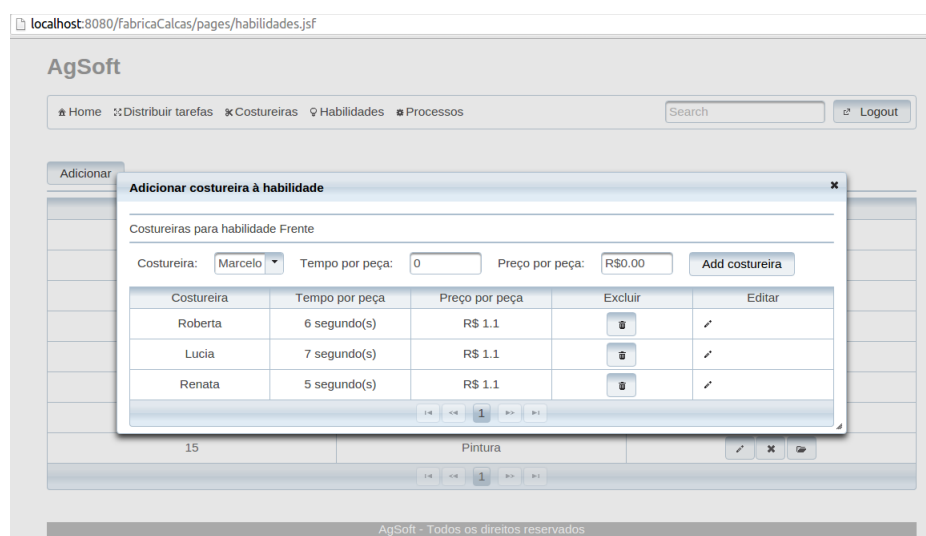


Figura 45 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

Feito isso foi realizado a distribuição dos lotes com as configurações de tempo e custo mostradas na Figura 45 e com as configurações de posicionamento mostradas na Figura 39.

A Figura 46 mostra o resultado obtido na distribuição.

localhost:8080/fabricaCalcas/pages/abrir_processo_distribuicao.jsf?idProcesso=98

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos Search Logout

Forma de distribuição

500 50 08/11/2015 11:08:25 Iniciar distribuição

Resultado da distribuição

Total de lotes definidos: 10
 Tempo total: 4000.0 segundos - Custo: R\$ 1350.0
 O menor tempo encontrado está associado ao menor custo dentro do prazo

Início: 08/11/2015 11:08:25 - Entrega: 27/11/2015 14:14:56
 Prazo total: 1652791

<Finalização> (Maria:6 Duda:4) <Frente> (Roberta:3 Lucia:5 Renata:2) <Carimbo> (Marcelo:10)

Figura 46 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

Com base na Figura 46 pode-se observar que foi adicionado no resultado final a atividade Frente e suas respectivas costureiras.

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

localhost:8080/fabricaCalcas/pages/abrir_processo_distribuicao.jsf?idProcesso=98

<Finalização> (Maria:6 Duda:4) <Frente> (Roberta:3 Lucia:5 Renata:2) <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	-	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 47 – Caso de teste com tempo de distribuição **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 48.

localhost:8080/fabricaCalcas/pages/abrir_processo_distribuicao.jsf?idProcesso=98

AgSoft

Home Distribuir tarefas Costureiras Habilidades Processos Search Logout

Forma de distribuição

500 50 08/11/2015 11:08:25 Iniciar distribuição

Resultado da distribuição

Total de lotes definidos: 10
Tempo total: 4000.0 segundos - Custo: R\$ 1350.0
O menor tempo encontrado está associado ao menor custo dentro do prazo

Início: 08/11/2015 11:08:25 - Entrega: 27/11/2015 14:14:56
Prazo total: 1652791

- <Finalização> (Maria:7 Duda:3) - <Frente> (Roberta:2 Lucia:2 Renata:6) - <Carimbo> (Marcelo:10)

Figura 48 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

A Figura 49 mostra o detalhamento da distribuição da Figura 48.

localhost:8080/fabricaCalcas/pages/abrir_processo_distribuicao.jsf?idProcesso=98

- <Finalização> (Maria:7 Duda:3) - <Frente> (Roberta:2 Lucia:2 Renata:6) - <Carimbo> (Marcelo:10)

Detalhes da distribuição

Atividade/Costureira	Parte	Qtde Lotes	Pçq 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 49 – Caso de teste com tempo de distribuição **Fonte:** Desenvolvido pelos autores

Com base nas Figuras 46 e 48 pode-se observar que o tempo total de produção e o custo final não tiveram alterações de uma distribuição para outra, porém a 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.

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 à informações importantes, sendo estas características cruciais, uma vez que a informação tornou-se um dos patrimônios mais importantes das empresas.

Além destas características, atualmente os softwares são dotados de inteligência artificial e vem cada vez mais realizando tarefas mais complexas oferecendo um suporte primordial no apoio à tomadas 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 a cada dia se tornarem mais competitivas, neste contexto, a otimização de processos pode ser um dos fatores chaves 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 desenhe seu processo, adicionando atividades a este 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 nestas 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 utilizado uma base desenvolvida pelo professor Artur Barbosa durante as aulas de sistemas especialistas, que define uma série de regras a ser seguida 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, neste sentido só foi preciso realizar algumas adaptações nestes 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 pensada para ser desenvolvida de forma recursiva o que dificultou o *debug*.

Assim, todos os objetivos propostos foram alcançados através dos conceitos e tecno-

logias descritos no quadro teórico e dos procedimentos descritos no quadro metodológico. O software desenvolvido atendeu todos os requisitos do escopo definidos na seção de Reuniões do Quadro Metodológico. Porém, além do escopo definido, outras funcionalidades poderiam ser desenvolvidas, porém 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 só está sendo considerado 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 e 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 Univas 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 disto o trabalho proporcionou um sólido conhecimento sobre algoritmos genéticos aos pesquisadores.

REFERÊNCIAS

BERGSTEN, H. *Javascript 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.

DOUGLAS, K.; DOUGLAS, S. *PostgreSQL: A comprehensive guide to building, programming, and administering postgresql databases*. 1º. ed. EUA: Sams Publishing, 2003.

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. Algoritmos 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 Algoritmos 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.

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.