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

SUMÁRIO

1 QUADRO METODOLÓGICO

O quadro metodológico é a descrição dos passos realizados para a execução do trabalho. Serão listados, nos tópicos a seguir, 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 utilizado que será apresentado no tópico a seguir.

1.1 Tipo de pesquisa

Será explicado, nesta sessão, o tipo de pesquisa escolhido para nortear o desenvolvimento deste trabalho, justificando também como ele se enquadra no tipo escolhido.

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 no tópico a seguir.

1.2 Contexto de pesquisa

Sabe-se que com a alta competitividade no mercado, as empresas, cada vez mais, buscam diferenciais competitivos 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. Dentre os fatores que viabilizam tais reduções está a otimização de processos que consistem 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 da fábrica citada trabalha em sua casa, é preciso ter uma boa forma de distribuir a produção para que o transporte da matéria-prima seja eficaz, trazendo para a empresa uma redução de custos e um tempo de produção menor. Para isso, é necessário que o software conheça os procedimentos de produção da fábrica, como seus funcionários trabalham e se estão sendo alocados de forma correta. A aplicação cruza todas essas informações gerando para o usuário uma relação de como ele deve distribuir a matéria-prima para produção das peças.

Como cada funcionário trabalha em sua casa é preciso saber qual a melhor rota para se entregar a matéria-prima ou recolher o que foi produzido para que seja passado para a próxima etapa da produção. O software avalia a melhor rota levando em consideração as peças que precisam ser produzidas, as habilidades de cada funcionário e se os mesmos não estão alocados em outros processos de produção. Analisando esses fatos, o software sabe a melhor forma para distribuição da produção e os funcionários tem suas tarefas distribuídas de forma a se obter o menor tempo e o menor custo de produção.

1.3 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.

1.3.1 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 que eram controladas e alimentadas pelo proprietário. Tais planilhas contemplavam as estimativas de produção, as datas de entrega de lotes de peças encomendado, levando em consideração a quantidade de peças, o corte e também o tempo em que cada peça levaria para ser entregue.

1.3.2 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 saneamento de 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 foi 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

somente 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 feita 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 que foi pesquisado. Assim, com ajuda do professor Roberto, nosso orientador, 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;
- O usuário deveria ser capaz de cadastrar um novo processo, costureiras e habilidades;
- O total de peças deveria ser dividido em lotes e cada costureira deveria receber uma quantidade de lote distribuída aleatoriamente;
- O software deveria então oferecer como saída a melhor distribuição de forma a se produzir no menor tempo, considerando o tempo de produção de cada costureira e o transporte das peças entre elas.

A sessão a seguir descreve como foi desenvolvido o escopo demonstrado acima.

1.4 Procedimentos

Esta sessão descreve os procedimentos realizados na execução do trabalho.

1.4.0.1 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 especia-

listas, 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, ela 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 anteriormente, 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 explicado anteriormente, 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 sessão 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 claras mais 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 representa o método roleta e o *CLASSIFICATION*, que representa o método de classificação;
- 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*;
- mutationType: Segue a mesma forma que o selectionType e o crossType e pode assumir os valores *Permutation*, *Binary* e *Numerical*.

- Classe `Individual`:

  A classe abstrata `Individual` representa a estrutura básica de um indivíduo. A classe contém uma `lista` do tipo `Cromossomo`, que será descrito mais abaixo, 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()`, esta operação é quem realiza a função de avaliação, explicada anteriormente, 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()`, já que esta é abstrata na classe mãe, 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`: Este é o método que realiza a mutação. Este deve ser implementado pela classe que representa o cromossomo, pois a mutação é feita a nível deste.
  - `clone`: Este método devolve um objeto exatamente com os mesmos atributos do objeto, porém com instâncias diferentes.

- Classe `IndividualPair`:

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

- Classe `GAController`:

  A classe `GAController`, como o próprio nome já diz, é o controlador de todo processo de execução do algoritmo genético. Ela recebe no seu construtor o modelo que é do tipo `GAModel`, que como já explicado anteriormente, armazena os parâmetros a serem seguidos na execução do algoritmo. Além disso, através do seu principal método denominado `execute`, ela é responsável por criar novas populações, a partir de cruzamentos, mutações, elitismo e 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`. Mais detalhes serão vistos mais a frente quando será descrito 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 da classe model;
  - Classificação e avaliação da população inicial através do método `classify`;
  - Realização do processo de elitismo, através do método `doElitism`;
  - Inserção de indivíduos estrangeiros na população;

- Realização do processo de seleção de indivíduos, através do método `doSelection`;
- Execução do processo de cruzamento e mutação, através do método `doCrossing` e `doMutaion` respectivamente.

Após os passos citados acima, 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`, o algoritmo volta ao início desta e recomeça 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.

Posterior a definição das regras de desenvolvimento, os próximos tópicos apresentam a implementação dos requisitos da aplicação e dos elementos do algoritmo genético, seguindo as definições do *framework*.

1.4.1 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 1 demonstra basicamente um exemplo de ordem de execução do processo de confecção.

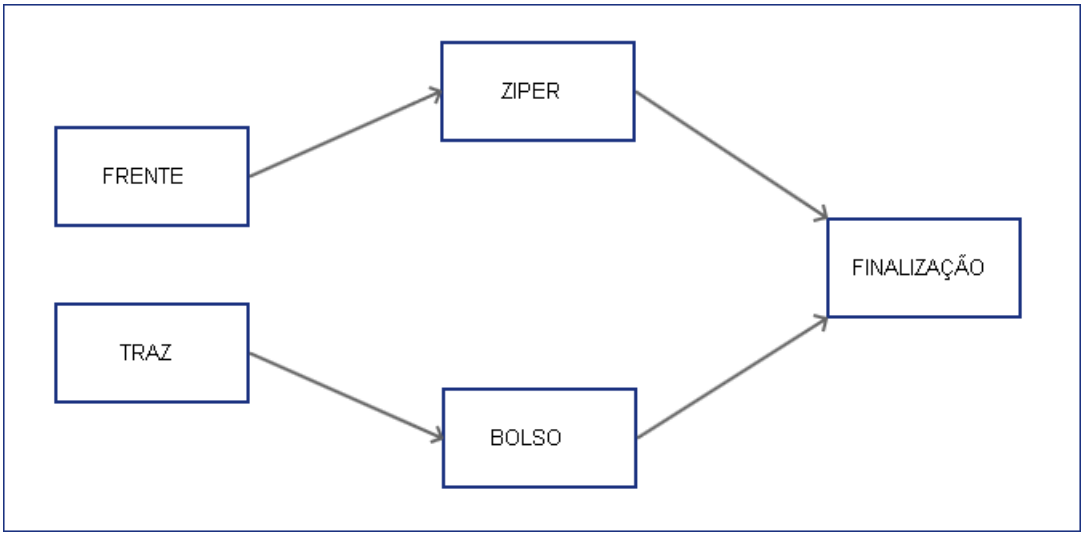

Figura 1 – Demonstração de um processo de fabricação **Fonte:** Desenvolvido pelos autores

Para representar este processo e suas atividades no software, foram utilizadas tabelas do banco de dados, conforme mostra a Figura 2.

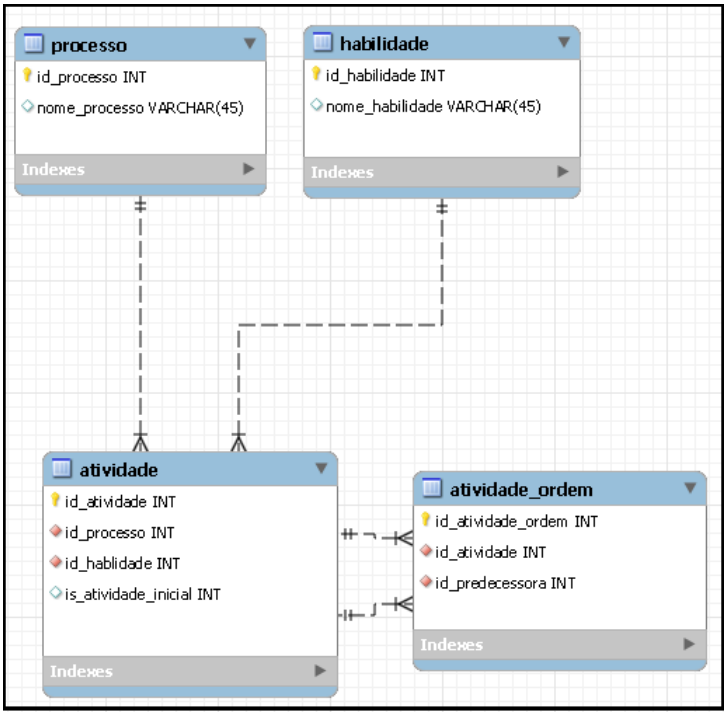

Figura 2 – Representação do processo de fabricação no banco de dados **Fonte:** Desenvolvido pelos autores

A tabela `processo` tem como finalidade gerar um código único para representar cada processo, pois 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 última do processo, tomando como base a Figura 1, ela seria a atividade de Finalização. Esta *flag* é importante no momento de calcular o tempo total de execução do processo e será vista 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 3 demonstra o mapeamento da relação entre a tabela `atividade` e `atividade-ordem` para o Java.

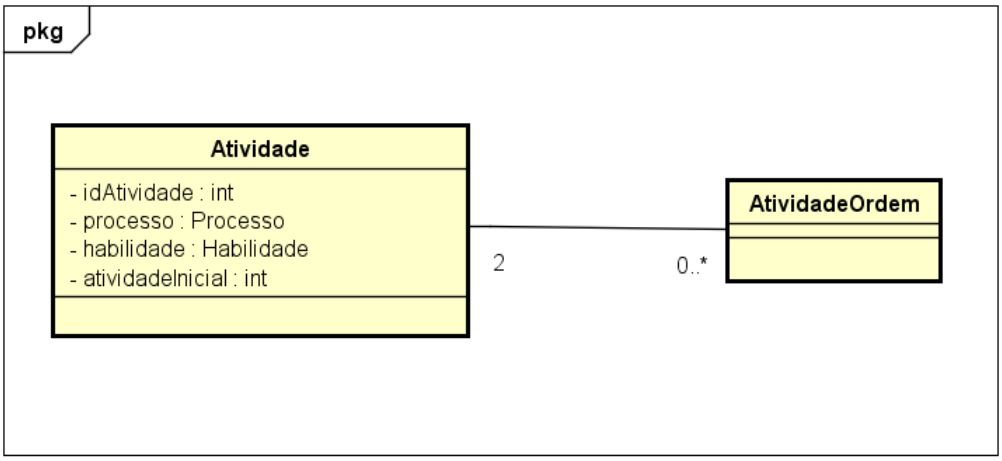

Figura 3 – Classes Atividade e AtividadeOrdem **Fonte:** Desenvolvido pelos autores

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

1.4.2 Distribuição das atividades (Indivíduos e Cromossomos)

Com relação a distribuição das atividades, foi definido 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 possuíssem a habilidade em questão. Por exemplo: se a quantidade total de peças de uma ordem de produção for 500, primeiramente deve-se definir qual será o número de peças por lote, neste caso, se for definido que cada lote deverá ter 50 peças, então o resultado final será 500/50 ou seja 10 lotes contendo 50 calças cada um. Neste sentido, seguindo o exemplo apresentado na Figura 1, a distribuição deverá ser feita de forma que para cada atividade do processo seja distribuído o trabalho de 10 lotes, ou seja, 10 lotes da parte da frente deve ser confeccionado e enviados para as costureiras que sabem colocar o

zíper e posteriormente estas enviam os 10 lotes para as costureiras que fazem finalização. Estas últimas irão depender também de 10 lotes da parte de trás que também devem passar pelas costureiras que fazem a confecção dos bolsos.

Com base nesses requisitos, um dos papeis desempenhados pelo algoritmo genético está na distribuição de trabalho descrita acima. O algoritmo irá distribuir, de forma aleatória, o número de lotes definido entre as costureiras de cada atividade, conforme mostra a Figura 4.

Finalização	Ziper	Frente	Bolso	Carimbo	Traz
Josi:2	Joana:6	Marta:5	Valdene:0	Bete:1	Joana:0
Ana:0	Lu:0	Cida:1	Marta:2	Dita:6	Marta:8
Luana:4	Fia:3	Roberta:1	Dita:8	Fia:1	Geralda:0
Bete:0	Valdene:1	Maria:0		-	Cida:2
Maria:4	-	Joana:0	-	-	Tereza:0
-	-	Tereza:3	-	-	Maria:2
Total:10	Total:10	Total:10	Total:10	Total:10	Total:10

Figura 4 – Exemplo de distribuição de lotes para as costureiras **Fonte:** Desenvolvido pelos autores

Uma costureira pode não receber nenhum 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 gerado uma população inicial de indivíduos e, a partir desta, um processo de cruzamento e mutação é iniciado a fim de que possa ser gerados novos indivíduos que representem soluções ainda melhores que seus antecessores.

Neste sentido, para a realização do algoritmo de distribuição de lotes, o processo de definição de indivíduo e cromossomo foi o primeiro passo do desenvolvimento da aplicação. Isso se deu devido ao fato de que estes elementos são 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 distribuído a determinada costureira em uma determinada atividade irá representar um cromossomo. Tomando como base o exemplo da Figura 4, o quadro, como um todo, representa o indivíduo e cada distribuição, como por exemplo Frente= Marta: 5, representa um cromossomo.

Para fazer esta representação em Java, primeiramente foi criado uma classe denominada `ProcessoChromosome` que é representada no diagrama abaixo:

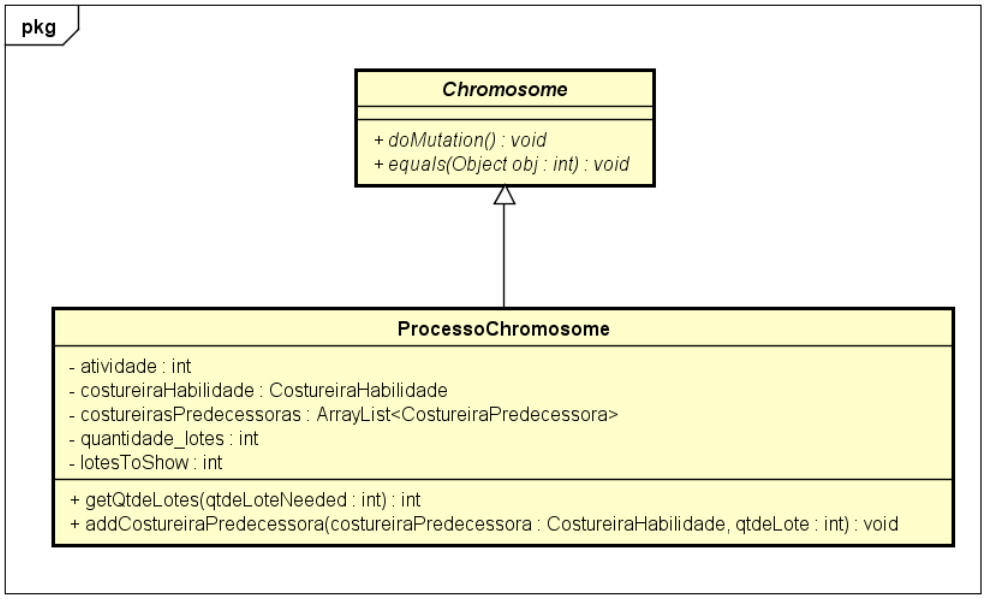

Figura 5 – Classe ProcessoChromosome **Fonte:** Desenvolvido pelos autores

A classe `ProcessoChromosome` herda de `Chromosome` do *framework* descrito acima. Por enquanto é necessário compreender apenas os atributos `atividade`, `costureiraHabilidade` e `quantidade-lotes`, que recebem seus valores por um 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 a qual se está atribuindo a costureira e a quantidade de lotes, este identificador vem do banco de dados e será passado na criação de cada cromossomo sempre que for necessário se criar um novo indivíduo. O atributo costureiraHabilidade é do tipo CostureiraHabilidade. Esta classe é o mapeamento da tabela costureira-habilidade do banco de dados, e faz a relação entre quais habilidades cada costureira possui e quanto tempo cada uma gasta para fazer uma peça de uma determinada parte da calça, conforme demonstra a Figura 6.

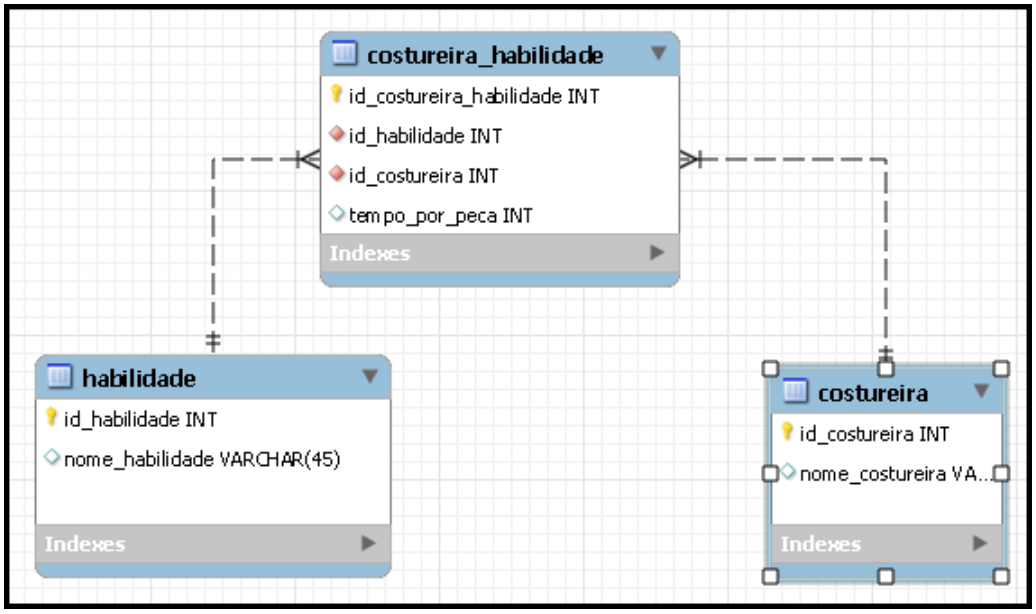

Figura 6 – Armazenamento de dados das costureiras **Fonte:** Desenvolvido pelos autores

A classe `CostureiraHabilidade`, conforme demonstrada na Figura 7, por sua vez, possui um atributo habilidade do mesmo tipo do nome, outro que representa a costureira e um terceiro para representar o tempo que tal costureira gasta para confeccionar uma peça de tal 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, é necessário se ter o tempo que a costureira gasta para fazer a peça e este tempo pode variar para uma mesma costureira dependendo de suas habilidades.

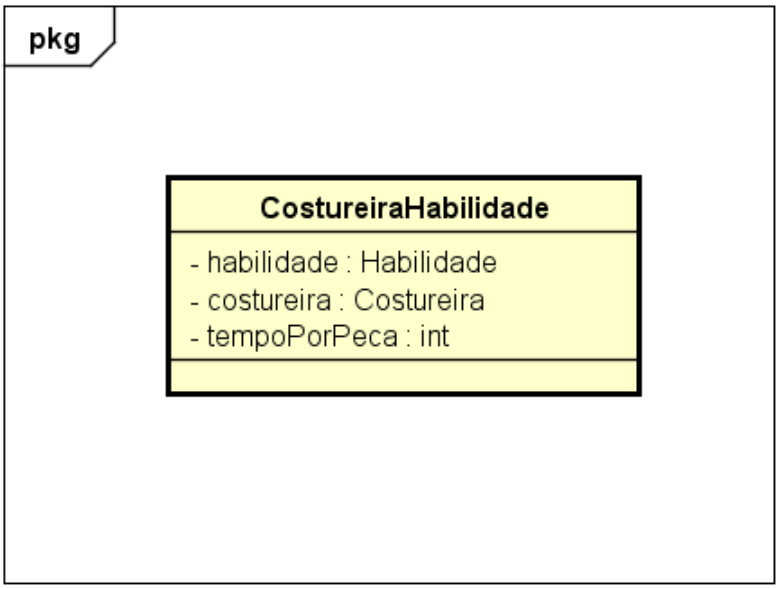

Figura 7 – Classe CostureiraHabilidade **Fonte:** Desenvolvido pelos autores

Assim, para representar cada característica da solução, tomando como exemplo a Figura

4, o fato de Marta fazer 5 lotes da parte da frente é representado no Java 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 lote que Marta deverá confeccionar, que seria, neste caso, 5.

A representação do indivíduo foi feita criando-se a classe `ProcessoIndividual` como demonstra a figura abaixo:

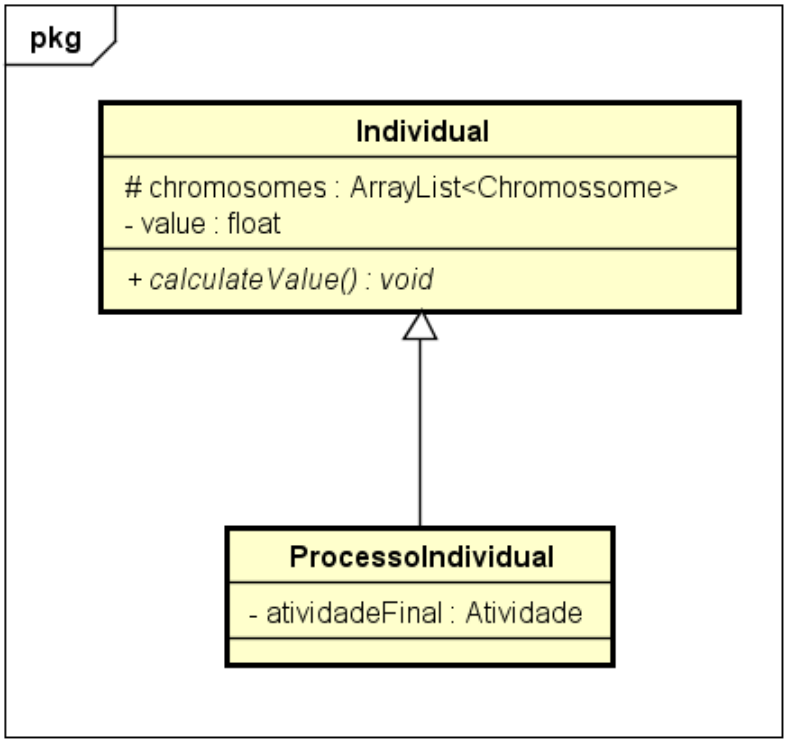

Figura 8 – Classe ProcessoIndividual **Fonte:** Desenvolvido pelos autores

A classe 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` recebe como parâmetro um objeto representando a atividade final, que será utilizado pela função de avaliação mais adiante, e um `HashMap` que possui como chave o `ID` de uma atividade e uma lista do tipo `CostureiraHabilidade` contendo as costureiras e 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 Figura 9.

```
package edu.univas.edu.tcc.ga_code;

import java.util.ArrayList;

public class ProcessoIndividual extends Individual {

    Atividade atividadeFinal;



    public ProcessoIndividual(Atividade atividadeInicial,
                              Map<Integer,List<CostureiraHabilidade>> atividadesCostureiras ) {
        chromosomes = new ArrayList<Chromosome>();
        this.atividadeFinal = atividadeInicial;


        for(Integer key : atividadesCostureiras.keySet()){
               for(CostureiraHabilidade costureira : atividadesCostureiras.get(key)){
                   Float porcentagem = (float) (Math.random() * 1);
                   chromosomes.add(new ProcessoChromosome(key, costureira, porcentagem));
               }
        }
    }
```

Figura 9 – Criação de cromossomos (Primeira Abordagem) **Fonte:** Desenvolvido pelos autores

Feita a distribuição da porcentagem, antes de fazer o cálculo do indivíduo, seria então realizado um cálculo de normalização posteriormente para que se pudesse encontrar o número de lote a ser produzido por cada costureira em cada atividade. Tomando como exemplo a Figura 10, este cálculo seria feito da seguinte forma:

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

Figura 10 – 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 | 0,44 / 1,26 = 0,35 | 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 | (500 * 35) / 100 = 175 | (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 arrendondamento:

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

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

Os passos acima para distribuição de atividades seria então repetido para cada atividade chave do `HashMap` realizando tal distribuição para cada costureira da lista de costureiras de cada uma. No momento de fazer o último cálculo era 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 a Figura 11.

```java
package edu.univas.edu.tcc.ga_code;

import java.util.ArrayList;

public class ProcessoIndividual extends Individual {

    Atividade atividadeInicial;

    public ProcessoIndividual(Atividade atividadeInicial,
                Map<Integer,List<CostureiraHabilidade>> atividadesCostureiras ) {
         chromosomes = new ArrayList<Chromosome>();
         this.atividadeInicial = atividadeInicial;

        int qtdeLote = 10;
        int cont = 0;

        for(Integer key : atividadesCostureiras.keySet()){
            qtdeLote = 10;
            cont = 0;

            for(CostureiraHabilidade costureiraHabilidade : atividadesCostureiras.get(key)){
                int loteCostureira = (int) (Math.random() * qtdeLote);
                if(cont != atividadesCostureiras.get(key).size() - 1){
                    chromosomes.add(new ProcessoChromosome(key, costureiraHabilidade, loteCostureira));
                }else{
                    chromosomes.add(new ProcessoChromosome(key, costureiraHabilidade, qtdeLote));
                }
                qtdeLote-=loteCostureira;
                cont++;
            }
    }
```

Figura 11 – Distribuição em lotes diretamente **Fonte:** Desenvolvido pelos autores

Conforme descrito na imagem acima, é feita uma iteração no `HashMap` e, para cada atividade, é feita a distribuição dos lotes para as costureiras desta. 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 lote é subtraída pelo valor atribuído ao cromossomo recém criado. 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 4.

Concluindo, a distribuição das atividades ocorre todas as vezes que se cria um novo indivíduo. Como será explicado no próximo tópico, indivíduos podem ser criados no processo de criação da `população inicial`, na criação de `indivíduos estrangeiros` e no processo de `cruzamento`, 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 sessão que descreve o cruzamento.

1.4.3 Classe Model e População inicial

Conforme visto no tópico que descreve a base de desenvolvimento, a classe `GAController` é onde se inicia a execução do algoritmo e esta espera em seu construtor um objeto do tipo `GAModel`, porém, como esta classe é abstrata, foi criado a classe `ProcessoModel` que herda de `GAModel`, passando a ter todos os atributos da classe mãe explicado no tópico que define o `framework` de desenvolvimento.

A classe `ProcessoModel` então é a primeira a ser instanciada pela classe principal e recebe em seu construtor uma conexão para o banco de dados e o `ID` do processo a qual será executado o algoritmo, o construtor então chama o método `getInformacoesCostureiras` que tem por finalidade buscar no banco de dados todas 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 define qual é a atividade final do processo, conforme demonstra a Figura 12.

```
@SuppressWarnings("unchecked")
public void getInformacoesCostureiras() {
    if (atividadesCostureiras != null && atividadesProcesso != null) {
        atividadesCostureiras.clear();
        atividadesProcesso.clear();
    }

    Query query = manager
            .createQuery("select a from Atividade as a where a.processo.idProcesso = :idProcesso");
    query.setParameter("idProcesso", 1);

    CostureirasDAO cdao = new CostureirasDAO(manager);

    atividadesCostureiras = new HashMap<Integer, List<CostureiraHabilidade>>();
    List<CostureiraHabilidade> costureirasHabilidades;

    // Montar um MAP tendo como chave cada atividade do processo e a lista
    // de costureiras que tenham
    // a habilidade relacionada.
    atividadesProcesso = query.getResultList();

    for (Atividade atividade : atividadesProcesso) {
        costureirasHabilidades = cdao.getCostureirasByHabilidade(atividade
                .getHabilidade().getIdHabilidade());
        atividadesCostureiras.put(atividade.getIdAtividade(),costureirasHabilidades);
        atividadeFinal = atividade.isAtividadeInicial() == 1 ? atividade: null;
    }
}
```

Figura 12 – Método getInformacoesCostureiras() **Fonte:** Desenvolvido pelos autores

A classe principal então cria um novo objeto de `GAController` passando o objeto da classe `ProcessoModel` e logo chama o método `execute`, dando início então a execução do algoritmo. 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 e o MAP que contém as atividades e suas costureiras (atividadesCostureiras) criados pelo método getInformacoesCostureiras().

1.4.4 Função de avaliação

Após a criação da população inicial, esta é então submetida a um processo de avaliação. Assim é feito 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, conforme mostra a Figura 8.

Conforme demonstrado na Figura 6, cada costureira sabe fazer uma ou mais partes da calça e gasta um determinado tempo, medido em segundos, que varia de acordo com a habilidade, uma demonstração pode ser vista na Figura 13. Além disto, existe um tempo de transporte entre cada costureira, porém, até a entrega do quadro metodológico, o armazenamento do tempo real de transporte entre as empregadas ainda não estava sendo armazenado no banco de dados e foi portanto gerado de forma aleatória, como será visto mais adiante.

Habilidade	Costureira	Tempo/Peca
Finalização	Maria	5s
Finalização	Bete	4s
Finalização	Luana	3s
Finalização	Ana	2s
Finalização	Josi	1s
Bolso	Dita	3s
Bolso	Marta	2s
Bolso	Valdene	1s
Carimbo	Cida	8s
Carimbo	Fia	7s
Carimbo	Dita	6s
Carimbo	Bete	5s
Ziper	Valdene	4s
Ziper	Fia	3s
Ziper	Lu	2s
Ziper	Joana	1s
Traz	Maria	6s
Traz	Tereza	5s
Traz	Silvia	4s
Traz	Geralda	3s
Traz	Joana	2s
Traz	Marta	1s
Frente	Tereza	2s
Frente	Joana	4s
Frente	Maria	5s
Frente	Roberta	1s
Frente	Cida	3s
Frente	Marta	2s

Figura 13 – Demonstração de costureiras e habilidades **Fonte:** Desenvolvido pelos autores

Com base nestas informações é realizado um cálculo a fim de se encontrar o tempo total de fabricação do número de peças desejado levando em consideração o número de lotes atribuído a cada costureira, o tempo gasto por cada uma para se produzir e o tempo gasto com o transporte das partes entre elas procurando assim encontrar uma forma de distribuir o trabalho de forma a se produzir as peças desejadas no menor tempo possível.

Para o desenvolvimento desta função, foi necessário construir uma estrutura para representar a questão da ordem de precedência entre as atividades, tal estrutura, conforme é demonstrado na Figura 14, 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.

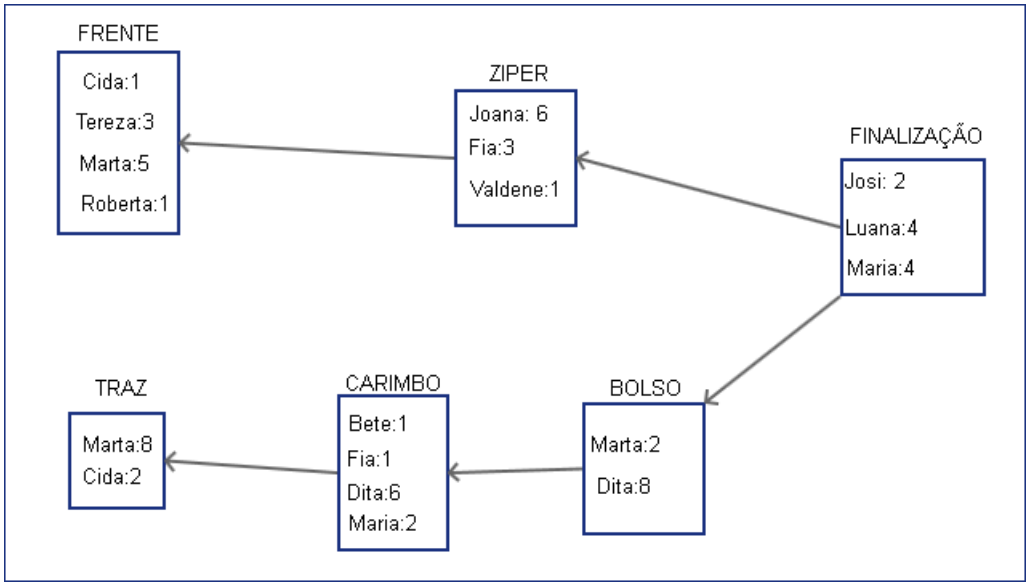

Figura 14 – Estrutura de representação da ordem de precedência **Fonte:** Desenvolvido pelos autores

Como foi visto no tópico que descreve a distribuição das atividades, cada cromossomo deles representa a alocação de uma costureira, contendo os lote que esta deve produzir para cada atividade, o indivíduo, por sua vez, tem uma lista de cromossomos. 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 14 para que por fim o cálculo pudesse ser realizado.

Para isso, 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 criado uma Classe denominada `Node`, sendo esta a responsável por criar a estrutura mostrada na Figura 14.

O Construtor da classe `Node` recebe um `MAP` e um objeto de atividade. Primeiramente o método `calculateValue` instancia um objeto da classe `Node` passando a `atividadeFinal` recebida pelo `ProcessoModel`, conforme descrito na sessão anterior, e o Map atividadeCromossomos conforme mostra a Figura 15.

```java
public void calculateValue() {
    Map<Integer, List<Chromosome>> atividadeCromossomos = new HashMap<Integer, List<Chromosome>>();
    Integer lastAtividade = null;
    List<Chromosome> cromossomos = null;

    for (Chromosome chromosome : chromosomes) {
        ProcessoChromosome processoChromossome = (ProcessoChromosome) chromosome;

        if (lastAtividade == null
                || lastAtividade != processoChromossome.getAtividade()) {
            cromossomos = new ArrayList<Chromosome>();
            atividadeCromossomos.put(processoChromossome.getAtividade(),
                    cromossomos);
            lastAtividade = processoChromossome.getAtividade();
        }
        cromossomos.add(processoChromossome);
    }
    node = new Node(atividadeFinal, atividadeCromossomos);

    /*
     * Só deve-se calcular o valor do indivíduo se ele nao foi calculado
     * ainda porque o valor dos lotes no objeto cromossomo foi decrementado
     */
    if (getValue() == 0) {
        setValue(node.getValorTotal());
    }
    //System.out.println(getValue());
}
```

Figura 15 – Método calculateValue **Fonte:** Desenvolvido pelos autores

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 quadrado (nó) daqueles mostrados na Figura 14. Assim, quando o método `calculateValue` instancia um objeto de `Node`, toda estrutura já é criada. A Figura 16 mostra como isto é realizado.

```java
package br.univas.edu.tcc.model;

import java.util.ArrayList;

public class Node {

    private List<Node> predecesoras = new ArrayList<Node>();
    private List<Chromosome> cromossomos;
    private Atividade atividade;

    public Node(Atividade atividade,Map<Integer,List<Chromosome>> atividadeCromossomos){
        this.atividade = atividade;
        cromossomos = atividadeCromossomos.get(atividade.getIdAtividade());
        Atividade atividadePredecessora = null;
        for(AtividadeOrdem predecessora : atividade.getAtividadeOrdemsForIdAtividade()){
            atividadePredecessora = predecessora.getAtividadePredecessora();
            predecesoras.add(new Node(atividadePredecessora,atividadeCromossomos));
        }
    }
```

Figura 16 – Classe Node (construtor) **Fonte:** Desenvolvido pelos autores

Como se pode ver no método calculateValue na Figura 15, 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 a Figura 17.

```
public long getValorTotal(){
    long valor =  0;
    long maior = 0;
    for(Chromosome chromosome : cromossomos){
        ProcessoChromosome processoChromosome = (ProcessoChromosome) chromosome;
        if(processoChromosome.getQuantidade_lotes() > 0){
            valor = getCromossomeValue(processoChromosome,processoChromosome.getQuantidade_lotes());
        }
        if(valor > maior){
            maior = valor;
        }
    }
    return maior;
}
```

Figura 17 – Método getValorTotal **Fonte:** Desenvolvido pelos autores

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

Tomando como base a Figura 14, para facilitar o entendimento, o método `getChromosomeValue` será chamado passando o cromossomo "Josi"e o inteiro 2 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` em que `NLC` é o número de lotes atribuído 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 a Figura 18.

```
public long getCromossomeValue(ProcessoChromosome processoChromosome,int qtdeLote){
    long valor = 0;
    valor = qtdeLote * 50 * processoChromosome.getCostureiraHabilidade().getTempoPorPeca();
    valor += getTempoRecebimentoPecas(processoChromosome,qtdeLote);
    return valor;
}
```

Figura 18 – Método getChromossomeValue **Fonte:** Desenvolvido pelos autores

O método `getChromossomeValue` chama então o método `getTempoRecebimentoPecas`, passando o cromossomo Josi e o inteiro 2 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 a Figura 19.

```
public long getTempoRecebimentoPecas(ProcessoChromosome processoChromosome, int qtdeLote){
    long valor = 0;
    long maior = 0;
    for(Node node : predecesoras){
        valor = node.getValueChromosomosPredecessores(processoChromosome, qtdeLote);
        if(valor > maior){
            maior = valor;
        }
    }
    return maior;
}
```

Figura 19 – Método getTempoRecebimentoPecas **Fonte:** Desenvolvido pelos autores

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 dois como número de lotes. A Figura 20 mostra este método.

```
public long getValueChromosomosPredecessores(ProcessoChromosome processoChromosome,int qtdeLote){
    int qtdeEachCromossome = 0;
    long valor = 0;
    long maior = 0;
    for(Chromosome chromosome : cromossomos){
        ProcessoChromosome processoChromosomeBefore = (ProcessoChromosome) chromosome;
        qtdeEachCromossome = processoChromosomeBefore.getQtdeLotes(qtdeLote);
        if(qtdeEachCromossome > 0){
            processoChromosome.addCostureiraPredecessora(processoChromosomeBefore.getCostureiraHabilidade(),
                qtdeEachCromossome);
            qtdeLote -= qtdeEachCromossome;
            valor = getCromossomeValue(processoChromosomeBefore, qtdeEachCromossome);
            valor += calcularTempoEntreCostureiras(processoChromosome,processoChromosomeBefore);
        }
        if(valor > maior){
            maior = valor;
        }
        if(qtdeLote == 0){
            break;
        }
    }
    return maior;
}
```

Figura 20 – Método getValueCrhomosomosPredecessores **Fonte:** Desenvolvido pelos autores

Tal método é responsável por iterar sobre a lista de cromossomos do nó anterior buscando de qual ou quais costureiras pode ser pego os lotes retornando o maior valor. No exemplo da Figura 21 uma das atividades anteriores é o Ziper e a primeira costureira da lista é a Joana, neste caso será consumido dois lotes da Joana.

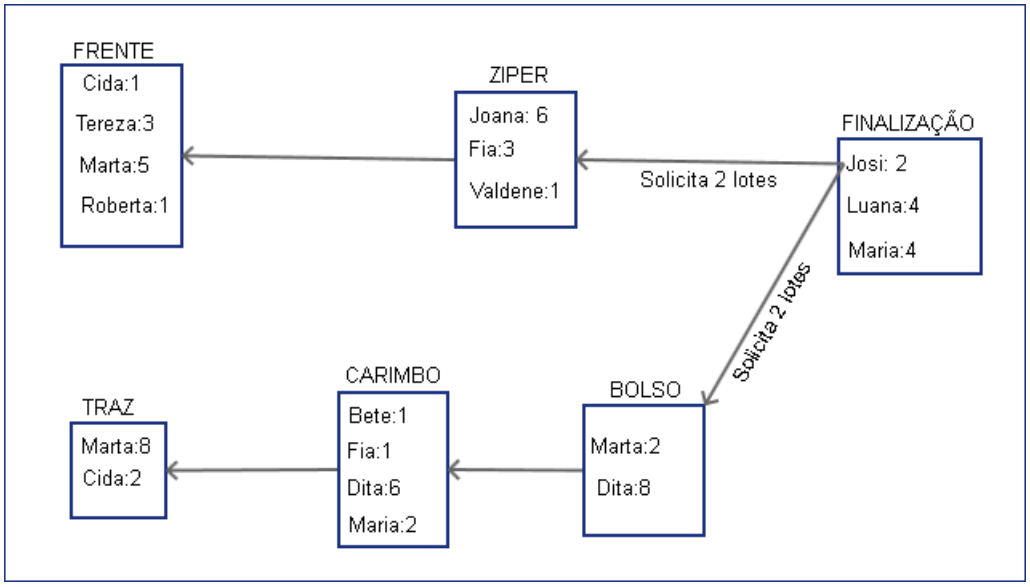

Figura 21 – Costureira Josi solicita dois lotes de atividades predecessoras **Fonte:** Desenvolvido pelos autores

Assim a costureira Joana é adicionada ao cromossomo Josi como costureira predecessora e novamente é chamado o método `getChromosomeValue`, só que agora passando o cromossomo Joana 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 Joana. Conforme explicado anteriormente, por enquanto este tempo está sendo gerado aleatoriamente, e será somado ao valor retornado de `getChromosomeValue` que foi chamado passando o cromossomo Joana. Seguindo as execuções dos métodos já explicados o fluxo então será `getCromossomeValue`, `getTempoRecebimentoPecas`, `getValueChromosomosPredecessores` e neste ponto o cromossomo Joana irá solicitar para sua atividade anterior 2 lotes, conforme mostra a Figura 22.

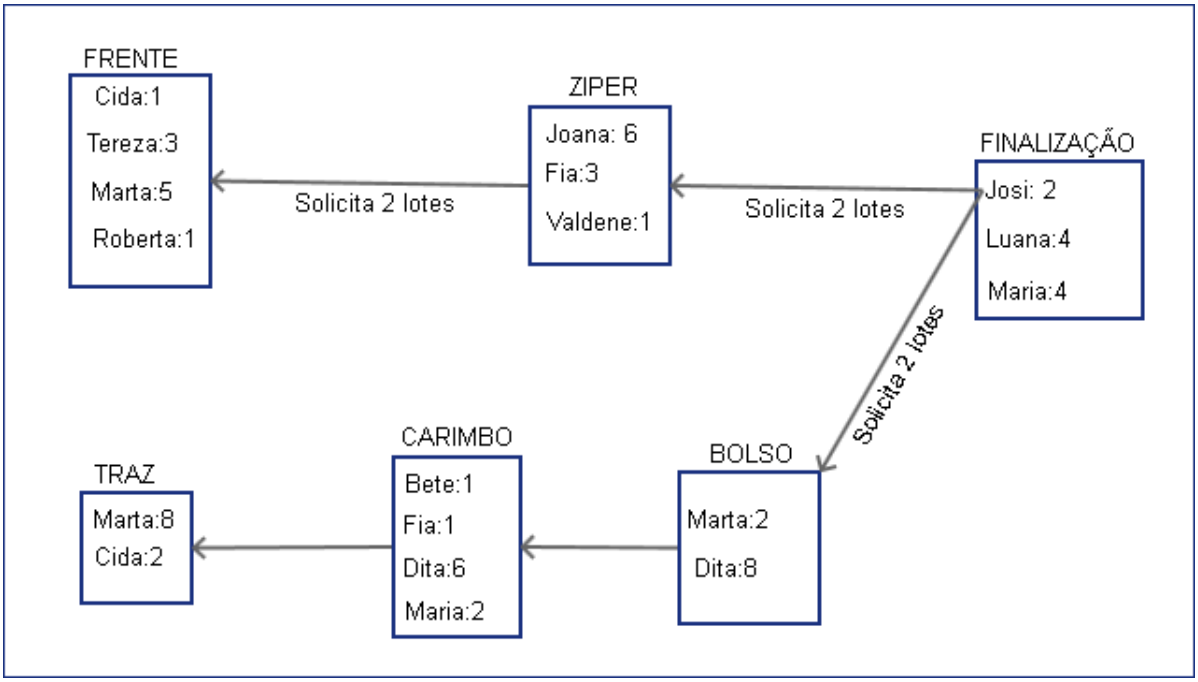

Figura 22 – Costureira Joana solicita dois lotes de atividades predecessoras **Fonte:** Desenvolvido pelos autores

Neste caso será pega uma peça da Cida e uma da Tereza, e será calculado o tempo de produção mais o tempo de transporte entre ambas e a Josi, prevalecendo o maior valor, além disso ambas são adicionadas ao cromossomo Joana como costureiras predecessoras. Isto é feito para futuros relatórios e testes.

Concluindo, o processo é todo feito recursivamente, as costureiras da primeira atividade vão consumindo os lotes das costureiras das atividades predecessoras, conforme mostra a Figura 23.

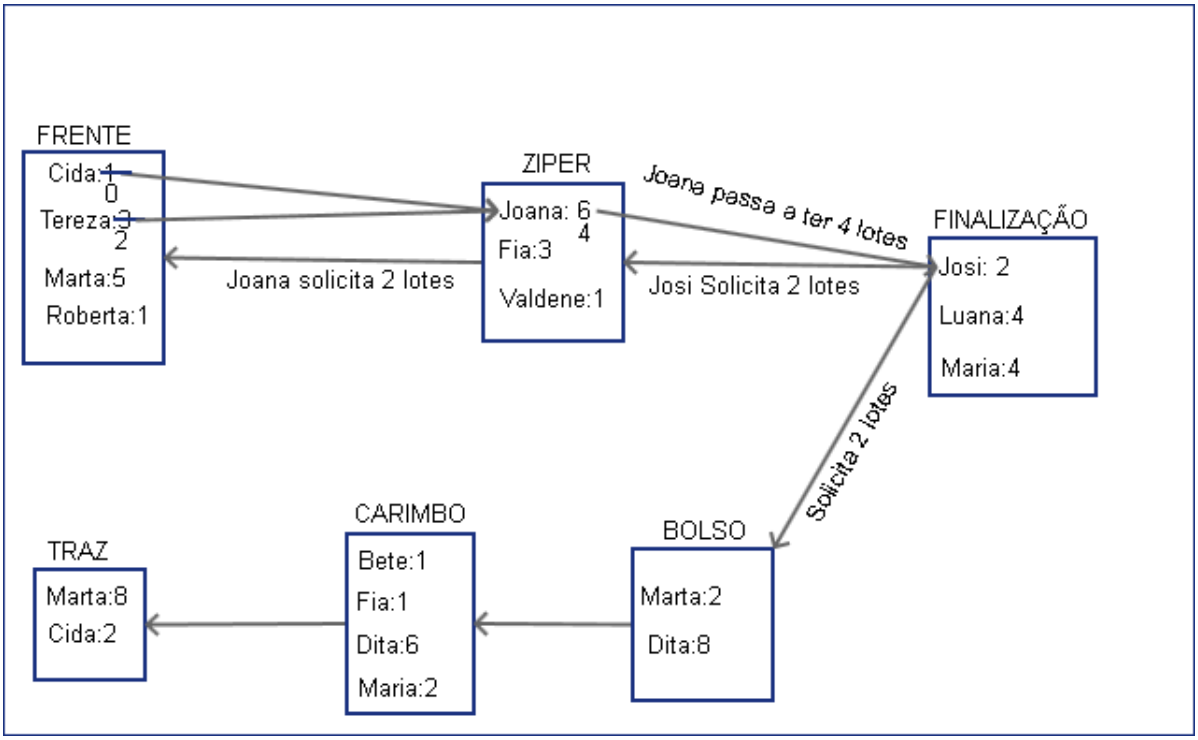

Figura 23 – Costureiras consomem lotes de suas predecessoras **Fonte:** Desenvolvido pelos autores

A recursividade para quando se chega em uma atividade que não tem nós predecessores, 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.

1.4.5 Indivíduos estrangeiros

Até a entrega do quadro metodológico, este item ainda não tinha sido implementado.

1.4.6 Seleção, Cruzamento e mutação

Até a entrega do quadro metodológico, este item ainda não tinha sido implementado.

1.4.6.1 Construção da interface gráfica, `CRUD` e apresentação dos dados

Até a entrega do quadro metodológico, este item ainda não tinha sido implementado. Porém já está definido que será uma aplicação `JSF` com `Primefaces` e basicamente permitirá

que o usuário cadastre novos processos, costureiras e habilidades, além de iniciar o algoritmo e obter informações sobre melhores distribuições encontradas e custo.

2 DISCUSSÃO DE RESULTADOS

2.1 Teste com somente uma atividade

Para verificar a capacidade de otimização do algoritmo, foi realizado um teste com apenas uma atividade, de forma que se pudesse prever a melhor solução, para poder verificar a eficiência do algoritmo. Para tal foi cadastrado então um processo, criando-se apenas uma atividade para este, conforme mostra a Figura abaixo:

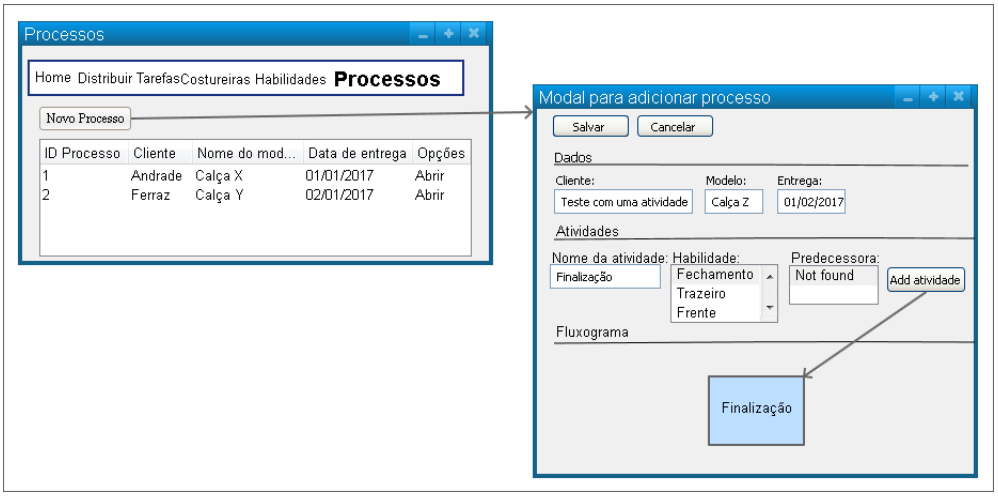

Figura 24 – Caso de teste com uma atividade **Fonte:** Desenvolvido pelos autores

Para a atividade de finalização, atribuída ao processo na Figura acima, foram cadastradas 5 costureiras e seus respectivos tempos para fazer tal atividade, conforme mostra a Figura abaixo:

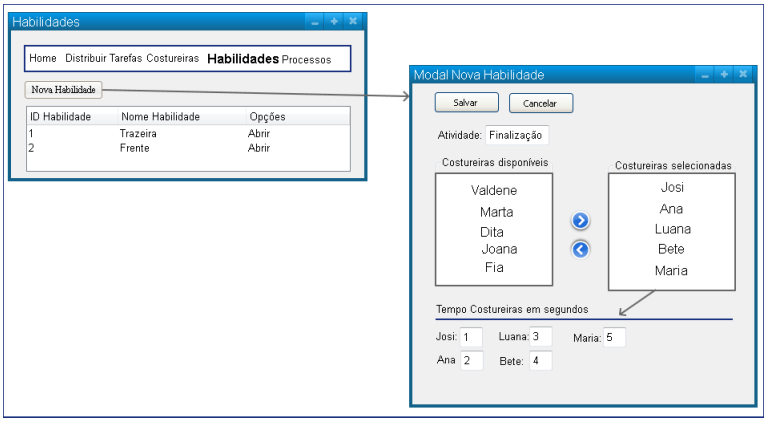

Figura 25 – Cadastro da habilidade e associação das costureiras **Fonte:** Desenvolvido pelos autores

Feito isso foi iniciado então o processo de distribuição das atividades, através do menu Distribuição de Tarefas.

REFERÊNCIAS

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.

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

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.

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

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

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

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