Projeto Prático de Inteligência Artificial
"*StockingProblem*"

João Russo
2181085
PL4
2181085@my.ipleiria.pt

Alberto Anastácio
2171083
PL4
2171083@my.ipleiria.pt

Objetivo

Este projeto teve como objetivo desenvolver uma aplicação que permitisse otimizar a maneira como um conjunto de peças eram dispostas numa superfície de matéria-prima de modo a desperdiçar a menor quantidade de matéria-prima e assim melhorar o processo de fabrico das peças.

1. INTRODUÇÃO

Neste projeto da unidade curricular de Inteligência Artificial, desenvolvemos uma aplicação com recurso à linguagem de programação orientada a objetos Java e foi utilizado o IntelliJ IDEA como editor, compilador e *debugger* de código.

Também foi desenvolvido para o projeto um algoritmo genético e um algoritmo *random* para encontrar a melhor solução para o conjunto de peças dado.

Neste relatório vão ser abordados os seguintes tópicos:

- **Implementação (cap. 2)**, nesta secção vai ser abordado todo o código desenvolvido para este projeto.

- **Experiências (cap. 3)**, nesta secção será descrito como foram feitas todas as experiências bem como os seus resultados.

- **Conclusão (cap. 4)**, nesta secção vai ser apresentada uma breve conclusão de todas as experiências feitas.

- **Referências (cap. 5)**, nesta secção vão estar presentes todas as referências por nós utilizadas durante o desenvolvimento do projeto.

2. IMPLEMENTAÇÃO

Nesta secção será descrito como funciona o algoritmo *random* e o algoritmo genético, todos os operadores de mutação e recombinação implementados tal como as alterações feitas aos operados já existentes. Irão também ser apresentados exemplos dos operadores implementados aplicados a um genoma aleatório.

2.1 Algoritmo *Random*

O algoritmo *Random* implementado funciona de uma maneira bastante simples, é criado um individuo inicial, `globalBest = problem.getNewIndividual()`, e não uma população como no algoritmo genético (este algoritmo irá ser descrito posteriormente), sendo este individuo considerado o melhor encontrado. Após a sua criação, é avaliado e é calculada a sua *fitness*, `globalBest.computeFitness()`, este é também apresentado ao utilizador.

Dentro do ciclo `while`, que itera o número de gerações restantes, é criado outro novo individuo, `I new_individual = problem.getNewIndividual()`, este é avaliado e caso o seu valor de *fitness* seja melhor que o valor de *fitness* do individuo inicial (os problemas que o projeto pretende resolver são de minimização e não maximização, logo um valor de *fitness* melhor será o valor mais pequeno possível), estes são trocados e o `globalBest` passa a ser o novo individuo.

Este ciclo de criação de um novo individuo, a sua avaliação e verificação se é ou não melhor que o melhor individuo até aí encontrado repete-se por `t` gerações.

No final o melhor individuo encontrado de todas as `t` gerações é *returned* e apresentado ao utilizador, esta será a melhor solução encontrada pelo algoritmo para o problema.

2.2 Algoritmo Genético

O algoritmo genético começa por criar uma população de `populationSize` indivíduos cada um com um genoma aleatório, `new Population<>(populationSize,problem)`. O algoritmo avalia cada um dos indivíduos da população e vai á procura do melhor para o problema em questão sendo este guardado, `globalBest = population.evaluate()`, e apresentado ao utilizador.

Posteriormente, dentro do ciclo `while` (que itera o número de gerações restantes), será criada uma nova população constituída por alguns dos melhores indivíduos da população anterior, usando para isso um método de seleção, e novos indivíduos com genomas aleatórios, `Population<I, P> populationAux = selection.run(population)`. De seguida os indivíduos da nova população têm uma probabilidade de sofrer recombinações e mutações de modo a gerar indivíduos diferentes, (baseados nas características dos indivíduos anteriores), que possam trazer uma melhor solução para o problema, `recombination.run(populationAux)` e `mutation.run(populationAux)`. Esta nova população é avaliada da mesma maneira que a população inicial e o melhor individuo dessa geração é guardado, `I bestInGen = population.evaluate()`.

Após este ser guardado este é comparado, e caso o seu valor de *fitness* seja melhor que o valor de *fitness* do individuo inicial, o `globalBest` passa a ser o melhor individuo da nova população criada nesta geração, `globalBest = (I) bestInGen.clone()`.

Este ciclo de seleção de indivíduos, criação de novos indivíduos para a nova população, recombinação, mutação e avaliação é repetida por `t` gerações.

No final o melhor individuo encontrado de entre todas as gerações é apresentado ao utilizador sendo que esta será a melhor solução que o algoritmo genético encontrou, `return globalBest`.

2.3 StockingProblem

A classe *StockingProblem* representa o problema a ser tratado.

Esta classe tem as seguintes propriedades:

```
10 private int materialHeight;
11 private double NumColsPer;
12 private double MaxSizePer;
13 private ArrayList<Item> items;
```

Quando a classe é instanciada, a função `StockingProblem()` recebe a altura máxima da matéria-prima, `int materialHeight`, e a lista de items do problema, `ArrayList<Item> items`. Esta lista é criada dentro da função `buildProblem(File file)` e é também dentro desta função que é lido o tamanho máximo da matéria-prima.

A função `buildProblem` recebe um ficheiro `File file` e lê o ficheiro da seguinte forma:

- Lê um número inteiro, `f.nextInt()`, que representa a altura da matéria-prima, `int materialHeight`.
- Lê outro número inteiro, `f.nextInt()`, que representa a quantidade de items que o problema contém, `int numberOfItems`.
- Ciclo `for` que lê as próximas `numberOfItems` linhas do ficheiro. Cada linha representa parte das características que cada item ou peça, `Item item`, irá ter após a sua instanciação. Em cada uma das próximas linhas irá:
    - Ser lido um número inteiro, `f.nextInt()`, que representa o número de *rows* ou linhas que o item tem, `int itemLines`. Este valor é posteriormente comparado com a altura máxima da matéria-prima (que foi anteriormente lida) e caso esta altura seja maior que a altura máxima da matéria-prima o item considera-se inválido e o programa para.
    - Ser lido um número inteiro, `f.nextInt()`, que representa o número de *columns* ou colunas que o item tem, `int itemColumns`.
    - Ser criada uma matriz, `int[][] matrix = new int[itemLines][itemColumns]`, com a representação física do item. Esta matriz é preenchida por um ciclo `for` que lê `itemColumns` números inteiros por cada uma das `itemLines` linhas da matriz, preenchendo-a com esses números lidos, `matrix[j][k] = f.nextInt()`.
    - No final de cada linha ser toda lida, será criado um novo item, `Item item`, sendo adicionado a lista de items do problema. O `id` deste item será o número da iteração atual.

As propriedades `materialHeight` e `items` são inicializadas quando a classe é instanciada, contudo, as outras duas propriedades desta classe, `NumColsPer` e `MaxSizePer`, usadas na função de *fitness*, só serão inicializadas depois da classe ser instanciada. Caso se utilize um ficheiro de configuração, estas são inicializadas nas linhas 80 e 81 da classe *StockingProblemExperimentsFactory*, classe esta que já foi instanciada na linha 78:

```
//PROBLEM
78 problem = StockingProblem.buildProblem(new
File(getParameterValue("Problem_file")));
79
80 problem.setMaxSizePer(
Double.parseDouble(getParameterValue("MaxSizePer")
));
81 problem.setNumColsPer(
Double.parseDouble(getParameterValue("NumColsPer")
));
```

E caso não seja usado um ficheiro de configuração estas serão inicializadas na classe *MainFrame*, linhas 170 a 171.

```
170 warehouse.setNumColsPer( Double.parseDouble(
panelParameters.jTextFieldNumCutsPer.getText()));

171 warehouse.setMaxSizePer( Double.parseDouble(
panelParameters.jTextFieldMaxSizePer.getText()));
```

Neste caso, a classe já tinha sido instanciada quando foi introduzido o ficheiro de texto que continha o problema a ser tratado, classe *MainFrame*, linha 150:

```
150 warehouse = StockingProblem.buildProblem(
dataSet);
```

Estas duas propriedades podem ser alteradas tanto nos ficheiros de configuração das experiências como nos dois campos de texto do *GUI* para valores que o utilizador considere serem adequados ao problema.

2.4 Item

Cada `Item item` representa uma das peças do problema.

As propriedades da classe *Item* são as seguintes:

```
9  private int id;
10 private char representation;
11 private int lines;
12 private int columns;
13 private int[][] matrix;
14 private HashMap<Integer, int[][]>
computedRotations;
15 private int[] possibleRotations;
```

O `int id`, como já foi referido, é o número da iteração atual que lê o ficheiro onde estão contidos os items do problema. A `char representation` é a representação do item como caracter *ASCII*. Se o `id` do item for menor que `26` então a `representation` será um carácter entre **A** e **Z**, caso seja superior ou igual a `26` então será representado por um carácter entre **a** e **z**, linha 19 da classe *Item*:

```
19 this.representation = (id < 26) ? (char) ('A' +
id) : (char) ('A' + id + 6);
```

A adição do numéro `6` quando o `id` é superior ou igual a `26`, deve-se ao facto de na tabela *ASCII*, os caracteres que representam letras maiúsculas e os caracteres que representam letras minúsculas estarem afastados por `6` outros caracteres que não representam letras, e a `representation` tem que ser uma letra.

Decimal	Carácter
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`
97	a
97	b

Tabela 1 - Tabela ASCII desde a posição 89 até a 97 em decimal

A `int[][] matrix` é a matriz que representa fisicamente o `Item`. O `int lines` e o `int columns` serão, respetivamente, a quantidade de linhas e de colunas desta matriz. Esta matriz caberá sempre na matéria-prima, pois já foi verificada se a sua altura, `lines`, é maior que a altura da matéria-prima.

Foi também implementada a rotação das matrizes de cada `Item` para uma melhor disposição dos items na matéria-prima, de modo a haver um menor desperdício. Para isto tivemos de calcular todas as rotações possíveis e diferentes de cada `matrix` de cada `Item`.

Com esta finalidade criamos a função:

```
28 private void computeItemRotations(int
problemHeight)
```

Esta função recebe como parâmetro a altura da matéria-prima, pois esta é necessária caso alguma das rotações da matriz do `Item` ultrapasse o limite de altura. Dentro desta função, é primeiramente adicionado a um `HashMap<Integer, int[][]>` a matriz inicial do `Item`, visto que esta é uma das rotações possíveis e diferentes da matriz do próprio `Item`. Este `HashMap<Integer, int[][]> computedRotations` guarda como chave um `Integer`, que representa a quantidade de rotações da matriz do `Item` 90 graus no sentido dos ponteiros do relógio, este guarda como valor um `int[][]`, que representa a matriz do `Item` rodada 90 graus `Integer` vezes no sentido dos ponteiros do relógio. Posteriormente é verificada a existência de zeros dentro da `matrix`. Isto é feito porque por exemplo, uma matriz sem zeros só tem 1 ou 2 rotações diferentes e não 4.

Para a verificação das rotações possíveis e diferentes desenvolvemos os seguintes blocos de código.

Para as matrizes quadradas, em que o número de linhas da matriz é igual ao número de colunas, que não contenham zeros, linhas 49 a 53 da classe *Item*:

```
49 if(columns == lines && !has_zeros)
50 {
51     possibleRotations = new int[]{0};
52     return;
53 }
```

Todos os seguintes blocos de código são relativos a matrizes não quadradas, em que o número de linhas da matriz é diferente do número de colunas. Os 2 seguintes blocos dizem respeito a matrizes sem zeros.

Para estas matrizes caso o número de colunas não ultrapasse o limite de altura, linhas 57 a 62 da classe *Item*:

```
57 if(!has_zeros && columns <= problemHeight)
58 {
59     possibleRotations = new int[]{0, 1};
60     computedRotations.put(1,
rotate90DegreesClockwise(matrix));
61     return;
62 }
```

Para estas matrizes, caso o número de colunas ultrapasse o limite de altura, linhas 67 a 71 da classe *Item*:

```
67 if(!has_zeros && columns > problemHeight)
68 {
69     possibleRotations = new int[]{0};
70     return;
72 }
```

Os seguintes 2 blocos são relativos a matrizes com zeros.

Para estas matrizes caso o número de colunas ultrapasse o limite de altura, linhas 75 a 80 da classe *Item*:

```
75 if(columns > problemHeight)
76 {
77     possibleRotations = new int[]{0, 2};
78     computedRotations.put(2,
rotate90DegreesClockwise(rotate90DegreesClockwise(
matrix)));
79     return;
80 }
```

Para estas matrizes que não ultrapassem o limite de altura, linhas 83 a 88 da classe *Item*:

```
83 possibleRotations = new int[]{0, 1, 2, 3};
84
85 for(int i = 0; i < 3; i++)
86 {
87     computedRotations.put(i + 1,
rotate90DegreesClockwise(computedRotations.get(i))
);
88 }
```

É necessário em alguns blocos de código verificar se o número de colunas da matriz que contém a posição inicial, é maior que a altura máxima da matéria-prima, pois uma rotação de 90 graus ou 270 graus no sentido dos ponteiros do relógio de uma matriz deste tipo, daria origem a matrizes que nunca poderiam ser encaixadas na matéria-prima devido a sua altura.

A propriedade `int[] possibleRotations` é usada para guardar todas as rotações possíveis e diferentes, por exemplo, uma matriz de um item que não contenha zeros e que não seja quadrada:

1	1	1
1	1	1

Neste caso o vetor `possibleRotations` irá guardar os valores `[0, 1]`, o que significa que as rotações possíveis e diferentes serão a posição inicial, `0`, e a matriz do item rodada `1` vez 90 graus no sentido dos ponteiros do relógio.

Outro exemplo poderá ser uma matriz não quadrada que contenha zeros, em que o número de colunas ultrapassa o limite de altura da matéria-prima (como limite de altura podemos considerar o valor 2):

1	0	1
1	1	1

Neste caso, se a matriz for rodada 90 graus para a direita, está terá uma altura de 3 (que é o número de colunas da matriz sem rotações), e esta rotação nunca caberá em nenhuma posição da matéria-prima, logo, não é calculada tal como a rotação de 270 graus no sentido dos ponteiros do relógio, logo o vetor `possibleRotations` irá guardar os seguintes valores, `[0, 2]`, que são as únicas rotações possíveis.

Outro exemplo será uma matriz quadrada sem zeros:

1	1
1	1

Esta matriz só tem 1 rotação diferente, que é a posição inicial, neste caso, esta posição é a única adicionada ao vetor `possibleRotations`, que irá somente guardar o valor `[0]`. Neste caso nenhuma rotação precisa de ser calculada.

Para rodar as matrizes dos exemplos anteriores, fomos pesquisar, e encontramos um bloco de código em C# (C Sharp) que depois foi transformado em código Java [4], classe *Item*, linhas 91 a 110:

```
91 private int[][]
rotate90DegreesClockwise(int[][] oldMatrix)
92  {
93      int cols = oldMatrix[0].length, rows =
oldMatrix.length;
94      int[][] rotated = new int[cols][rows];
95      int newCol, newRow = cols - 1;
96
97      for (int oldCol = cols - 1; oldCol >= 0;
oldCol--)
98      {
99          newCol = rows - 1;
100
101         for (int oldRow = 0; oldRow < rows;
oldRow++)
102         {
103             rotated[newRow][newCol] =
oldMatrix[oldRow][oldCol];
104             newCol--;
105         }
106         newRow--;
107     }
108
109     return rotated;
110 }
```

Este bloco de código cria uma nova matriz que vai guardar a matriz rodada, e percorre nos seguintes ciclos *for* a matriz sem ser rodada, no primeiro ciclo percorre-a desde a última coluna até a primeira, e no segundo ciclo *for* percorre-a desde a primeira linha até a última. Cada uma destas posições é então colocada na matriz rodada, começando na última fila da coluna mais a direita, e seguindo para a coluna mais a esquerda, subindo pelas linhas.

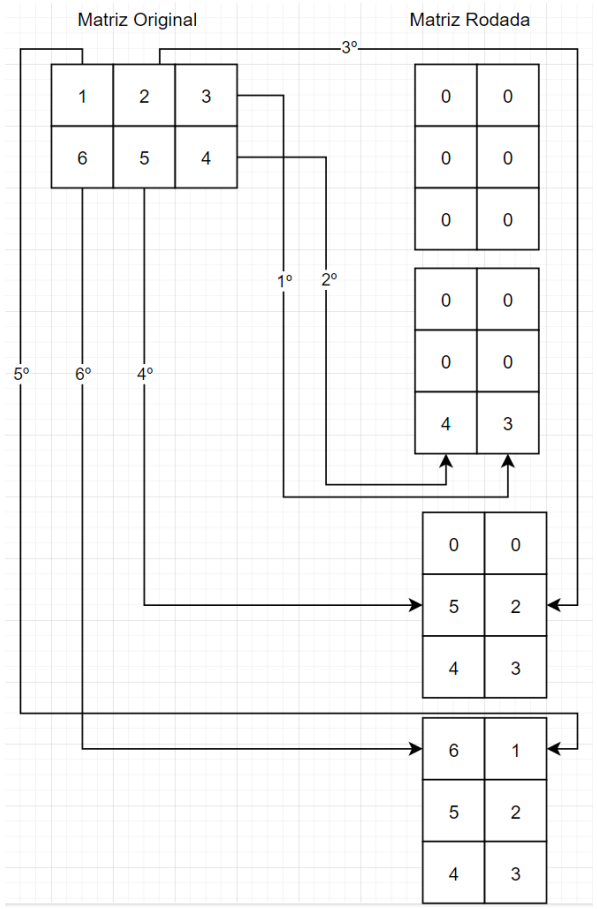

Figure 1 - Ilustração de como a matriz é rodada

2.5 StockingProblemIndividual

Esta classe representa um indivíduo (no caso do algoritmo genérico, um individuo da população).

As suas propriedades são:

```
10 private ArrayList<Integer>[] material;
11 private int num_cuts, materialMaxSize;
```

Esta classe herda também propriedades da *IntVectorIndividual*.

```
7 protected int[] genome;
8 protected int[] rotations;
```

Quando um destes indivíduos é criado, na função `StockingProblemIndividual(StockingProblem problem, int size)`, é-lhe atribuído um genoma aleatório. Para este efeito foi utilizado um ciclo *for* para preencher o genoma de 0 a `genome.length`, este ciclo preenche o genoma com o `id` de cada

uma das peças do problema. Para o efeito de aleatoriedade criamos a função `shuffleGenome()`, que vai a cada uma das posições do genoma, `i`, e troca o valor dessa posição com o valor de uma posição aleatória. Esta posição aleatória é obtida através da seguinte linha de código:

```
198 random_pos = Algorithm.random.nextInt(length);
```

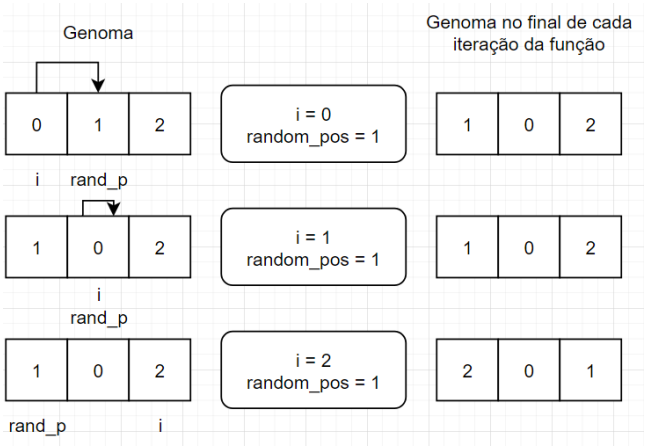

Figure 2 - Exemplo da função `shuffleGenome()` a correr

Nota: `rand_p` e `random_pos` são a mesma variável.

É também atribuído a cada individuo uma rotação da matriz de cada item presente no genoma aleatória, para este efeito criamos a função `fillWithRandomRotations()`.

A função `fillWithRandomRotations()` faz uso de um ciclo *`for`* que vai a cada posição do vetor `rotations` (este vetor guarda a rotação da matriz do item), e vai atribuir a essa posição uma das rotações (possíveis e diferentes) da matriz do item que está na mesma posição mas no vetor `genome`, usando para isso a função `setRandomRotation(int pos)`. Esta função recebe a posição no vetor `genome` onde está o `id` do item para o qual se quer obter rotação aleatória possível, e caso haja mais que uma rotação possível, é lhe atribuída uma das rotações possíveis de maneira aleatória, usando para isso a seguinte linha de código, linha 313 da classe *StockingProblemIndividual*:

```
313 this.rotations[pos] = item.possibleRotations(
Algorithm.random.nextInt(possibleRotationsSize));
```

Esta linha vai buscar um valor inteiro aleatório entre 0(inclusivo) e `possibleRotationsSize`(exclusivo), que representa a posição no vetor `possibleRotations`, e de seguida vai buscar ao vetor `possibleRotations` (presente na classe *Item*), o valor da rotação que estiver presente nessa posição.

Caso a matriz do item só tiver uma rotação possível e diferente, esta única rotação é a posição inicial que corresponde ao valor 0.

2.5.1 Função de *Fitness*

A função de *fitness* implementada por nós no projeto, encontra-se presenta na linha 36 da classe *StockingProblemIndividual*:

```
36 public double computeFitness()
```

Esta função começa por inicializar a propriedade da classe `ArrayList<Integer>[] material`, que representa a matéria-prima, usando o ciclo *`for`*, este ciclo percorre o vetor `material` e para cada uma das posições inicializa-a com um *`new`* `ArrayList()`.

Seguidamente são colocadas as matrizes dos items dentro do vetor `material`. Para isto é percorrido o `genome` do individuo usando um ciclo *`for`*, que vai buscar cada item a cada posição do `genome`, linha 58 da classe *StockingProblemIndividual*:

```
58 item_to_place = items.get(genome[l]);
```

Depois de obter o item daquela posição, é obtida a matriz desse item com certa rotação, esta rotação está guardada no vetor `rotations` na mesma posição que no vetor `genome`, linha 59 da classe *StockingProblemIndividual*:

```
59 rotation = item_to_place.getRotation(rotations
[l]);
```

De seguida tenta-se colocar a matriz do item no `material`, para isto é usado um ciclo *`while`* para percorrer cada posição de cada `ArrayList` do vetor `material`, essa posição é guardada na variável `i`, e de seguida é percorrido cada `ArrayList j` nessa posição `i`, isto assegura que todas as peças são colocadas o mais a esquerda e em cima possível.

Para verificar que a matriz do item "cabe" no `material` com início na posição `i` do `ArrayList j`, é usada a função `checkValidPlacement(rotation, i, j)`, que recebe a rotação da matriz do item que se está a tentar colocar, a posição `i` e o `j`, que representa o `ArrayList` a partir do qual se quer começar a colocar a matriz do item.

Antes da matriz do item tentar ser colocada no `material`, é verificado se a altura da matriz mais a posição da qual esta seria colocada é maior que a altura do material, caso verdade, é impossível colocar a matriz tanto nessa posição tal como nas seguintes posições da mesma coluna `i`, neste caso ocorre um *`break`* e "salta-se" para a próxima coluna.

A função `checkValidPlacement(rotation, i, j)` funciona da seguinte maneira:

1. É percorrida a matriz do item por um ciclo *`for`*.
    a. Caso o tamanho do `ArrayList` seja 0 significa que essa linha está vazia então toda a linha da matriz do item pode ser colocada.
    b. Caso contrário, o `ArrayList` já contém posições preenchidas, é calculada a última posição preenchida do `ArrayList`, caso essa posição seja inferior a posição de onde nós queremos começar a preencher a linha podemos colocar a linha total da matriz do item sem problema.

       Caso a última posição preenchida do `ArrayList` seja maior que a posição de onde queremos começar a preencher o `ArrayList` então aí é criado um ciclo *`for`*.

       Este ciclo percorre a linha da matriz do item e verifica se o índice da linha mais a posição de onde queremos começar a preencher já passou pela última posição preenchida do `ArrayList`, caso seja verdade significa que as posições para a frente estarão todas vazias

no `ArrayList` e não precisamos de verificar mais.

Caso o `ArrayList` ainda tenha posições para a frente preenchidas então temos de verificar se a matriz do item contém ou não um zero nessa posição, caso seja verdade não é necessário mais verificações, caso não se verifique significa então que temos de verificar se a posição do material é 0 ou não, caso seja 0 significa que a nossa posição da matriz do item cabe naquela posição caso o material nessa posição esteja preenchido com outro valor sem ser 0 isto significa que já está la colocado parte de outro item o que faz com que a nosso item atual não consiga ser colocado a partir da posição `i j`.

c. Caso nenhum return de dentro do ciclo tenho sido "ativado" significa que a matriz do item pode ser colocada nessa posição `i j` e o return final é "ativado", linha 127 da classe *StockingProblemIndividual*.

```
127 return true;
```

Caso a função `checkValidPlacement()` dê *return* a *true*, então a matriz do item é colocada a partir da posição `i j` pela função `place_item_in_position()`.

A função `place_item_in_position()` atua da seguinte forma:

1. É verificado se a soma da coluna desde onde a matriz do item vai começar a ser colocada, `i`, até á última coluna que esta matriz do item vai preencher, `rotation[0].length`, é maior que o tamanho do matéria-prima em termos de largura, esta largura está guardada na variável `materialMaxSize`, caso seja verdade então a variável `materialMaxSize` fica com o valor desta soma.
2. Ciclo *for* que itera as colunas da matriz do item e outro ciclo *for* que itera as linhas da matriz do item. Dentro destes ciclos *for* é:
    a. Calculada a última posição do `ArrayList` que está a ser preenchido atualmente.
    b. Verificado se a matriz do item na posição `[k][l]` um valor diferente de 0.
        i. Caso se verifique é necessário preencher essa posição no `ArrayList`, para isso é verificado se a última posição do `ArrayList` preenchida, calculada anteriormente, é a posição imediatamente antes de onde se quer começar a preencher a matriz, caso isto seja verdade basta adicionar ao `ArrayList` a representação do item, letra que é atribuída dependendo do `id` do item quando este item é criado, caso isto não seja verdade é necessário preencher as posições anteriores até chegar a posição imediatamente antes de onde queremos começar a preencher, para isso outro ciclo *for* é criado. Este ciclo vai desde a última posição preenchida do `ArrayList` até a posição imediatamente antes de onde se quer começar a preencher o `ArrayList`, e essas posições são preenchidas cada uma com um zero.

Depois de todas as matrizes de todos os items do `genome` serem colocados no `material` com a rotação indicada no vetor `rotations`, então é feito o *padding* do material.

O *padding* ocorre quando se chama a função `zeroPaddings()`, esta função percorre cada `ArrayList` do `material` desde a última posição preenchida de cada linha e adiciona zeros até que chegue a coluna mais a direita, guardada na variável `materialMaxSize`.

Depois dos *padding* feitos, são contados os cortes necessários para retirar as peças da matéria-prima, esta contagem é feita pela função `countCuts()` que funciona da seguinte forma:

1. Ciclo *for* que itera as linhas do `material` até `material.length - 1`(exclusivo), seguido de um ciclo *for* que itera as colunas do `material` até `materialMaxSize - 1`(exclusivo), as subtrações por `1` devem-se ao facto de não se poder comprar os valores das posições da última coluna, `materialMaxSize - 1`, do `material` com posições a direita não existentes e também não se poderem comparar os valores das posições da última linha, `material.length - 1`, do `material` com posições em baixo não existentes.
    a. Dentro do ciclo são comparadas a coluna `j` do `ArrayList i` com, a coluna `j + 1` do `ArrayList i` e a mesma coluna `j` mas do `ArrayList i + 1`, desta maneira são comparadas todas as posições do `material` com a posição a direita e em baixo. As únicas comparações que faltam fazer são entre os valores das posições da última linha do `material` com o valor da posição a direita e o valor das posições da última coluna do `material` com o valor da posição abaixo. Para resolver um destes problemas dentro do primeiro ciclo *for* são comparados os valores da última coluna do `material`, `materialMaxSize - 1`, com os valores abaixo, o valor da última coluna do `ArrayList i` é comparado com o valor da última coluna mas do `ArrayList i + 1`.
2. Para resolver o problema dos valores da última linha não serem comparados com os valores a direita um ciclo *for* foi criado. Este ciclo vai as posições da utlima linha até a posição `materialMaxSize - 1` (exclusivo), e compara o valor dessa posição com o valor da posição a direita.

Dentro de cada comparação, *if*, caso os valores sejam distintos é adicionado 1 á quantidade de cortes necessário.

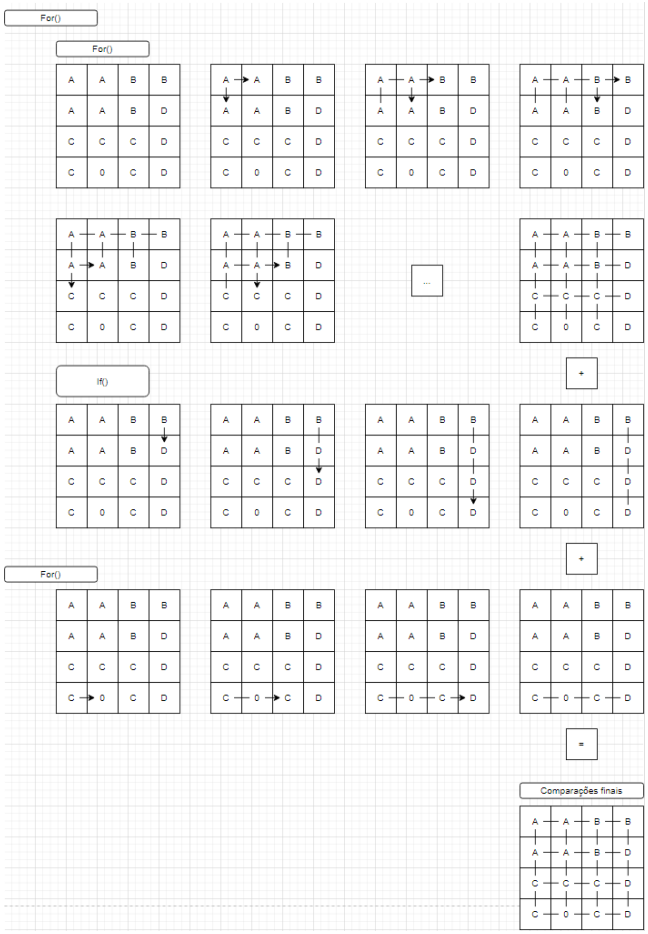

Figure 3 - Exemplo das comparações feitas pela função `countCuts()`

Nota: Cada linha representa um `ArrayList`, cada quadrado representa uma posição desse `ArrayList` e cada traço representa um comparação que foi feita entre os valores dessas 2 posições.

2.6 Mutações

Neste projeto implementamos duas mutações, a mutação *Swap* [1] e a mutação *Inversion*. [1]. Estas mutações nem sempre são "executadas", só quando a probabilidade de mutação é maior que uma percentagem aleatória, linha 19 da classe abstrata *Mutation*:

```
19 if (random.nextDouble() < getProbability())
```

2.6.1 *Swap*

Nesta mutação o objetivo é fazer a troca de valores entre duas posições aleatórias do genoma do individuo.

O código desenvolvido para esta mutação foi o seguinte, linhas 17 a 40 da classe *MutationSwap*:

```
17 // SWAP MUTATION
18 int size = ind.getNumGenes();
19
20 // Get 2 random different genome positions
21 int pos1 = GeneticAlgorithm.random.
nextInt(size);
22 int pos2 = GeneticAlgorithm.random.
nextInt(size);
23
24 while(pos2 == pos1)
25 {
26     pos2 = GeneticAlgorithm.random.
nextInt(size);
27 }
28
29 // Swap allele and rotation values
30 int aux = ind.getGene(pos1);
31 int auxRotation = ind.getRotation(pos1);
32
33 ind.setGene(pos1, ind.getGene(pos2));
34 ind.setRotation(pos1, ind.getRotation(pos2));
35 ind.setGene(pos2, aux);
36 ind.setRotation(pos2, auxRotation);
37
38 // Mutate Genome Item Rotation
39 mutateRotation(pos1, ind);
40 mutateRotation(pos2, ind);
```

Primeiro são obtidas duas posições aleatórias e diferentes entre 0 e `size` (exclusivo) (o `size` representa o tamanho do genoma do individuo), a `pos1` e a `pos2`. Para garantir posições diferentes foi implementado o código das linhas 24 a 27, que, enquanto a `pos2` não for diferente da `pos1`, um novo valor vai ser atribuído à `pos2` entre 0 e `size` (exclusivo). Depois de obtidas as posições para a troca dos valores, estes são trocados tal como as suas rotações linhas 29 a 36 (o item e a sua rotação andam sempre "pegados").

No final há ainda a possibilidade das rotações das matrizes dos items de cada uma das posições trocadas serem alteradas para outra posição, para isto implementou-se a função `mutateRotation(int pos, I ind)`, linhas 25 a 31 da classe abstrata *Mutation*:

```
25 public void mutateRotation(int pos, I ind)
26 {
27     if(probability < Algorithm.random.
nextDouble())
28     {
29         ind.setRandomRotation(pos);
30     }
31 }
```

Esta função recebe o individuo para o qual a alteração da rotação poderá ou não acontecer e a posição na qual a alteração ocorrerá, esta alteração só ocorre se a nova probabilidade obtida, `Algorithm.random.nextDouble()`, for maior que a probabilidade de mutação, `probability`. Caso isto aconteça esse individuo na posição `pos` vai sofrer uma mutação ao nível da rotação da matriz da peça. (o funcionamento da função `setRandomRotation()` já foi explicado na secção 2.4).

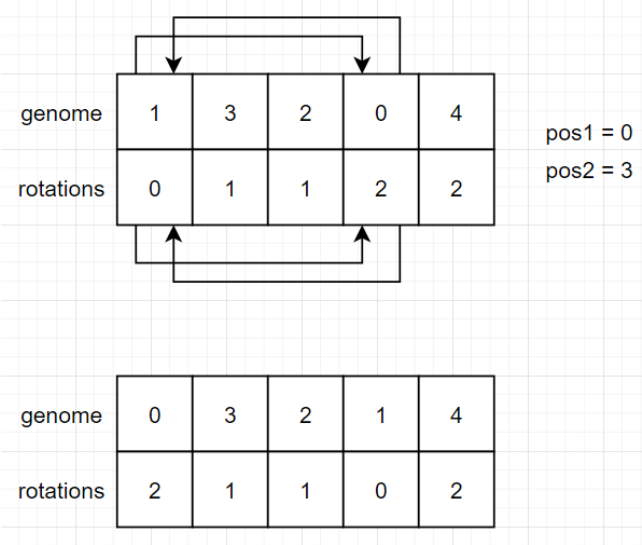

Figure 4 - exemplo da execução da mutação *Swap*

2.6.2 *Inversion*

Nesta mutação o objetivo é obter duas posições do genoma aleatórias e inverter a ordem dos valores do genoma entre essas duas posições.

O código desenvolvido para esta mutação foi o seguinte, linhas 17 a 57 da classe *MutationInversion*:

```
16 // INVERSION MUTATION
17 int size = ind.getNumGenes();
18
19 // Get 2 random different genome positions
20 int pos1 = GeneticAlgorithm.random.
nextInt(size);
21 int pos2 = GeneticAlgorithm.random.
nextInt(size);
22
23 while(pos2 == pos1)
24 {
25     pos2 = GeneticAlgorithm.random.
nextInt(size);
26 }
27
28 // Swap values for loop if necessary
29 if(pos2 < pos1)
30 {
31     int aux = pos2;
32     pos2 = pos1;
33     pos1 = aux;
34 }
35
36 int length = pos2 - pos1 + 1;
37
38 // Swap allele values
39 for(int i =  0; i < length/2; i++)
40 {
41     int aux = ind.getGene(pos1 + i);
42     int auxRotation = ind.getRotation(pos1 +
i);
43
44     ind.setGene(pos1 + i, ind.getGene(pos2 -
i));
45     ind.setRotation(pos1 + i, ind.getRotation
(pos2 - i));
46
47     ind.setGene(pos2 - i, aux);
48     ind.setRotation(pos2 - i, auxRotation);
49
50     mutateRotation(pos1 + i, ind);
51     mutateRotation(pos2 - i, ind);
52 }
53
54 if(length % 2 == 1)
55 {
56     mutateRotation(pos1 + length/2, ind);
57 }
```

Tal como na mutação *Swap*, são inicialmente obtidas duas posições aleatórias e diferentes. Depois de obtidas é preciso que `pos2` seja maior que `pos1` devido ao ciclo *for*. Após trocados os valores caso `pos2` seja maior que `pos1`, é obtido o numéro de iterações do ciclo, `pos2 - pos1 + 1`, isto significa que o ciclo só chegará até metade do comprimento entre as 2 posições caso estas sejam pares e caso sejam ímpares chegará até metade menos 1 posição devido á conversão de float para int na linha 39 na operação `length/2`.

Já dentro do ciclo *for*, há a troca do valor do genoma tanto como o da rotação da matriz do item entre as posições `pos1 + i` e `pos2 – i` (o item e a sua rotação andam sempre “pegados”). No final de cada iteração ainda há a possibilidade de haver uma mutação ao nivél da rotação da matriz do item, tal como na mutação *Swap*. Todos os valores entre `pos1` e `pos2` (inclusivo) podem sofrer mutação ao nivél da rotação da matriz do item, deste modo, se o comprimento entre posições for ímpar esta mutação nunca ocorrerá pois o ciclo só vai até `length`, para que haja a possibilidade desta mutação ocorrer, o bloco de código entre as linhas 54 e 57 foi desenvolvido, onde, caso o comprimento seja ímpar, a função `mutateRotation()` é invocada e é lhe passada a posição do meio, `pos1 + length/2`.

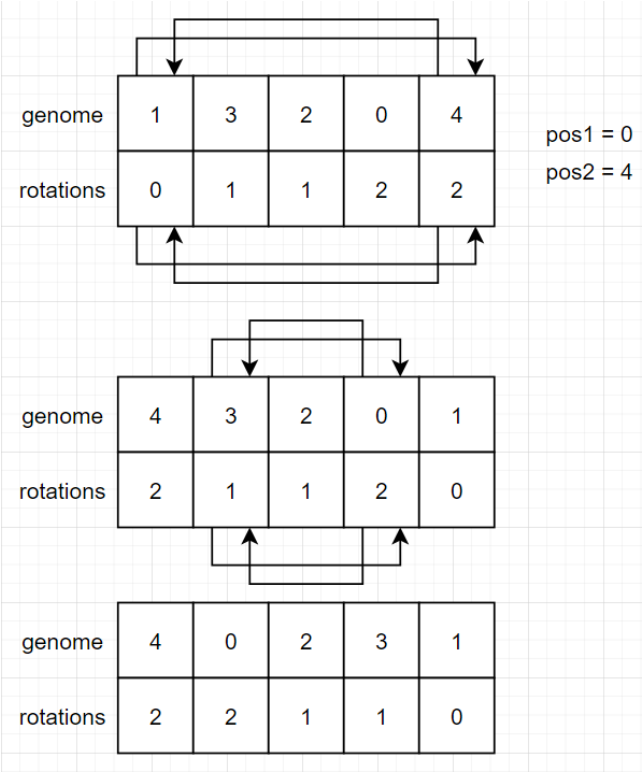

Figure 5 - exemplo da execução da mutação *Inversion*

2.6.3 *Insert*

Na mutação já implementada no projeto, mutação *Insert*, foi necessário alterar parte da sua implementação para que funcionasse com as rotações, para isso foram adicionadas as seguintes linhas á classe *MutationInsert*:

```
34 int auxRotation = ind.getRotation(i + 1);
37 ind.setRotation(i + 1, ind.getRotation(i));
40 ind.setRotation(i, auxRotation);
42 mutateRotation(i + 1, ind);
45 mutateRotation(cut1 + 1, ind);
```

A primeiras três linhas servem para guardar a rotação da posição `i + 1` do vetor rotações, dar *set* da rotação presente na posição `i` para a posição `i + 1` e dar *set* do valor da rotação guardado no `auxRotation`, valor da rotação da posição `i + 1` antes da sua troca de valor, para a posição `i`, isto assegura que a rotação anda sempre "pegada" ao item.

2.7 Recombinações

Neste projeto implementamos duas recombinações, a recombinação *Cycle* [1] [2] e a recombinação *Order1* [1] [3].

Estas recombinações nem sempre são "executadas", só quando a probabilidade de recombinação é maior que uma percentagem aleatória, linha 18 da classe abstrata *Recombination*:

```
18 if (random.nextDouble() < getProbability())
```

2.7.1 *Order1*

Esta recombinação funciona do seguinte modo:

1. Escolher uma parte aleatória do pai para copiar para o filho, nas linhas 27 a 40 da classe *RecombinationOrder1*, são determinados os 2 pontos de corte, todos os valores entre estes dois pontos são copiados para o filho no primeiro ciclo `for` dentro da função `getChild()`. Juntamente com os valores dos items são também copiados os valores das rotações de cada item, estes andam sempre "pegados".
2. Copiar todos os valores ainda não presentes no filho começando no segundo corte, pela ordem como aparecem no segundo pai. Isto é feito no segundo ciclo `for` da função `getChild()`, onde antes é criado um vetor que guarda, pela ordem como aparecem, os valores dos items e das rotações como aparecem no segundo pai, começando no `cut2 + 1` até fazer um *loop* pelo genoma todo do segundo pai.
3. Estes valores são posteriormente copiados para o filho no terceiro ciclo `for`, começando no `cut2 + 1` até chegar a posição do `cut1`.

Isto é repetido uma segunda vez com as mesmas posições de corte mas com os papeis dos pais invertidos para obter um segundo filho.

No final, linhas 46 a 52 da classe *RecombinationOrder1*, é feito um ciclo `for` para colocar o genoma e as rotações dos indivíduos pais com os valores do genome e rotações dos filhos.

Nota: O genoma e rotações dos filhos são a linha no índice 0 e índice 1 respetivamente, da variável `int[][] childX`.

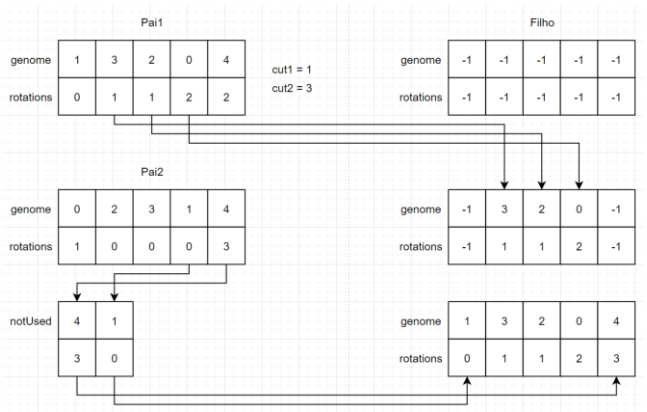

Figure 6 - exemplo da execução da recombinação *Order1*

2.7.2 *Cycle*

Esta recombinação funciona do seguinte modo:

1. Identificar um ciclo, para isto começamos na 1ª posição do primeiro pai, segundo argumento passado para a função `cycle()`, o valor que aqui se encontrar será o nosso início e fim de ciclo, variável `allele`.
2. De seguida vamos a mesma posição mas no segundo pai, variável `value2` que guarda o valor do genoma do segundo pai nesta posição e variável `value2Rotation` que guarda a rotação dessa posição, estas andam sempre juntas, e procuramos o valor que aqui se encontra mas no primeiro pai,

variável `value1` e `value1Rotation`, estas têm o mesmo comportamento que as duas variáveis anteriores mas para os valores do primeiro pai.

3. Com o valor da 1ª posição do segundo pai encontrado no primeiro pai, primeiro verificamos se o ciclo já acabou, caso não se confirme continuamos o ciclo.
4. Este ciclo de, ir a uma posição do primeiro pai, ir a mesma posição mas no segundo pai e de seguida encontrar este valor mas no primeiro pai é repetida até que o valor que encontramos no primeiro pai é o valor inicial (valor da primeira posição do primeiro pai), linha 70 da classe *RecombinationCycle*:

```
70 while(value1 != allele);
```

5. Este ciclo é repetido mais uma vez mas para a segunda posição do primeiro pai, linha 27 da classe RecombinationCycle:

```
27 cycle(ind1, ind2, 1, child2,
child1);
```

6. Os valores encontrados no primeiro pai no primeiro ciclo são colocados no primeiro filho nas mesmas posições que aparecem no pai, e os valores encontrados no segundo pai no primeiro ciclo são colocados no segundo filho também nas mesmas posições que aparecem no segundo pai, linhas 61 a 64 da classe *RecombinationCycle*:

```
61 child1[0][index] = value1;
62 child1[1][index] = value1Rotation;
63 child2[0][index] = value2;
64 child2[1][index] = value2Rotation;
```

7. O segundo ciclo segue o mesmo princípio de colocação de valores mas para este são colocados os valores do segundo pai no primeiro filho e os valores do primeiro pai do segundo filho, linha 27 da classe *RecombinationCycle*, onde os filhos estão trocados relativamente á linha de cima da mesma classe:

```
26 cycle(ind1, ind2, 0, child1,
child2);
```

```
27 cycle(ind1, ind2, 1, child2,
child1);
```

8. Caso os filhos não estejam completamente preenchidos, um ciclo `for` percorre um filho e quando encontrar o valor `-1` vai a essa posição nos pais respetivos de cada filho, primeiro pai primeiro filho, segundo pai segundo filho, e preenche os filhos com o valor que aí se encontrar.
9. No final os valores do genoma e rotações dos filhos são colocados nos respetivos pais tal como na recombinação *Order1*.

Nota: Tal como nas mutações e recombinações anteriormente faladas o valor do item e da rotação da sua matriz andam sempre "pegados" e tal como acontece na recombinação anteriormente falada, *Order1*, o filho é do tipo `int[][] childX`, no que significa que a linha de índice 0 e a linha de índice 1 são respetivamente o genoma e as rotações deste filho.

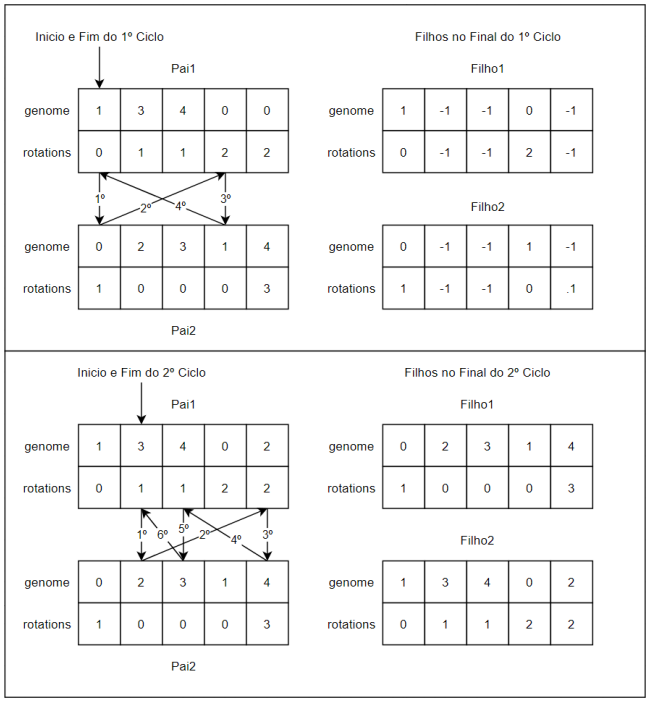

Figure 7 - exemplo da execução da recombinação *Cycle*

2.7.3 *Partial Mapped*

Na recombinação já implementada no projeto, recombinação *Partial Mapped* ou *PMX*, foi necessário alterar parte da sua implementação para que funcionasse com as rotações, para isso foram adicionadas ou alteradas as seguintes linhas da classe *RecombinationPartialMapped*:

```
20 child1 = new int[2][ind1. getNumGenes()];
21 child2 = new int[2][ind2. getNumGenes()];
```

Para que fosse possível guardar também o valor da rotação do item.

Também foram alteradas as linhas 40 e 43 e adicionadas as linhas 41 e 44:

```
40 ind1.setGene(i, child1[0][i]);
41 ind1.setRotation(i, child1[1][i]);
43 ind2.setGene(i, child2[0][i]);
44 ind2.setRotation(i, child2[1][i]);
```

Adicionadas pois é também preciso copiar as rotações de cada peça para o pai e não só o genoma dos filhos e alteradas pois o genoma está na linha de índice 0 de cada um dos filhos.

As restantes linhas alteradas dentro da classe foram na sua maioria as condições de paragem dos ciclos `for` que passou a `offspring[0].length` devido ao facto de o genoma estar no índice `0`, caso a condição anterior não fosse alterada, continuasse `offspring.length`, o ciclo só iria até 2 devido as alterações no tipo de dados dos dois filhos nas linhas 20 e 21.

Outra alteração feita foi ao tipo de dados dos segmentos que passaram a também ter de guardar a rotação do item, estas alterações ocorreram na função `crossOver()` e na função

`create_Segments()` da classe *RecombinationPartialMapped*, linhas 107 e 111, e linhas 74 e 75 das classes acima referidas respetivamente:

```
107 int[][] segment = segment2;
111 int[][] segment = segment1;
```

```
74 segment1 = new int[2][capacity_ofSegments];
75 segment2 = new int[2][capacity_ofSegments];
```

No coluna índice 0 o genoma é guardado e no índice 1 as rotações das peças do genoma são guardadas. Isto levou a que no ciclo *for* imediatamente abaixo tivessem que ser alteradas algumas linhas e outras acrescentadas:

```
80 int xRotation = ind1.getRotation(index);
82 int yRotation = ind2.getRotation(index);
84 segment1[0][segment1and2Index] = x;
85 segment1[1][segment1and2Index] = xRotation;
86 segment2[0][segment1and2Index] = y;
87 segment2[1][segment1and2Index] = yRotation;
```

As linhas alteradas, 84 e 86 devem-se ao facto de o tipo de dados do segmento ter sido alterado, as restantes linhas 80 e 82 guardam a rotação do item da posição `index` para serem posteriormente copiadas para os segmentos, isto é feito nas linhas 85 e 87.

Outra alteração feita foi quando estavam a ser inseridos valores nos filhos, estas alterações ocorreram na função `sort_duplicates()` e na função `insert_segments()` da classe *RecombinationPartialMapped*, linhas 62 a 67 e linhas 97 e 98 das classes acima referidas, respetivamente:

```
62 if (segment1[0][index] == offspring[0]
[indexOfElement]) {
63     offspring[0][indexOfElement] = segment2[0]
[index];
64     offspring[1][indexOfElement] = segment2[1]
[index];
65 } else if (segment2[0][index] == offspring[0]
[indexOfElement]) {
66     offspring[0][indexOfElement] = segment1[0]
[index];
67     offspring[1][indexOfElement] = segment1[1]
[index];
```

```
97 offspring[0][index] = segment[0][segmentIndex];
98 offspring[1][index] = segment[1][segmentIndex];
```

Estas alterações eram necessárias devido ao facto já referido anteriormente, quando um valor, id do item, é copiado para os filhos, a rotação da matriz do item tem de vir sempre "pegada".

3. EXPERIÊNCIAS

Nesta secção irá ser descrito o modo como realizamos as experiências, desde a estrutura das pastas ao computador utilizado e aos resultados dos testes gerais e especificos.

3.1 Estrutura de Pastas

A estrutura de pastas usada nas experiências foi bastante simples, em baixo, encontra-se uma imagem que ilustra as pastas criadas, onde foram executadas as experiências e onde acabaram por ficar os resultados destas, ficheiros excel e txt.

```
Experiências/
├── Problema1/
│   ├── TestesGerais/
│   │   ├── config.txt
│   │   ├── Problem1.txt
│   │   ├── statistic_average_fitness.xls
│   │   ├── statistic_average_fitness_final.xls
│   │   ├── statistic_best_per_experiment.txt
│   │   ├── statistic_best_per_experiment.xls
│   │   └── StockingProblemProjeto.jar
│   ├── Mutacao/
│   ├── Recombinacao/
│   ├── Torneio/
│   └── Populacao/
├── Problema2/
├── Problema3/
├── Problema4/
└── Problema5/
```

Figure 8 - Estrutura de pastas usada nas experiências

3.2 Ficheiro de configuração

O ficheiro de configuração é um simples ficheiro de texto que tem presente os parâmetros a testar, este ficheiro é escolhido, para depois ser "corrido", pelo utilizador depois de carregar no botão **Experiments** no *GUI* da aplicação, e é lido na classe *ExperimentsFactory* pela função `readParametersFile()`. Depois de o utilizador carregar no botão **Run Experiments** no *GUI* da aplicação, são "corridas" todas as combinações de parâmetros possíveis x numéro de vezes (parâmetro *runs* do config.txt).

Enquanto as combinações "correm" são escritos para os ficheiros excel e txt algumas métricas como por exemplo, a média do *fitness* dos melhores indivíduos das x *runs* de uma combinação de parâmetros.

```
Runs: 50

Population_size: 50, 100, 200

Max_generations: 50

//---------------------------------

Selection: tournament

Tournament_size: 2, 4, 6, 8

//---------------------------------

Recombination: pmx, cycle, order1

Recombination_probability: 0.8, 0.7, 0.6

//---------------------------------

Mutation: insert, swap, inversion

Mutation_probability: 0.1, 0.15, 0.2

//---------------------------------

MaxSizePer: 0.4

NumColsPer: 0.6

//---------------------------------

Problem_file: ./Problema1.txt

//---------------------------------

Statistic: BestIndividual
Statistic: BestAverage
```

Figure 9 - Exemplo de um ficheiro de configuração

3.3 Computador Usado

Em baixo está presente a tabela com as especificações do computador usado para correr as experiências.

Sistema Operativo	Windows 10 Home 64 bits Versão 20H2
Processador	Intel Core i7-7700k 4 Cores (HyperThread) 4.20 Ghz
Memória RAM	2x8GB DDR4 2400Mhz
Armazenamento	Disco SSD 500GB

Tabela 2 - Especificações do computador usado

3.4 Testes Gerais

Para os testes gerais foram corridas cada combinação de parâmetros 50 vezes (parâmetro *runs* do config.txt).

3.4.1 Problema1

Como estes eram problema é um problema "simples", a seguinte combinação de parâmetros foi a usada:

Parâmetro	Valor
Runs	50
Population_size	50, 100, 200
Max_generations	50
Selection	tournament
Tournament_size	2, 4, 6, 8
Recombination	pmx, cycle, order1
Recombination_probability	0.8, 0.7, 0.6
Mutation:	insert, swap, inversion
Mutation_probability:	0.1, 0.15, 0.2

Tabela 3 - Ficheiro de configuração para o problema 1

Relativamente ao problema 1 a melhor combinação de parâmetros encontrada, que gerou o melhor *fitness* médio, foi a seguinte:

Parâmetro	Valor
Runs	50
Population_size	200
Max_generations	50
Selection	tournament
Tournament_size	2
Recombination	cycle
Recombination_probability	0.8
Mutation:	inversion
Mutation_probability:	0.2

Tabela 4 - Melhor configuração para o problema 1

O *fitness* médio obtido por esta combinação de parâmetros foi de 158.4.

É de notar que há muitas combinações de parâmetros que chegam a valores de *fitness* médio muito próximos deste.

O melhor *fitness* obtido por um só individuo foi de 158.4. Pode ser observado no ficheiro excel que todas as combinações de parâmetros tinham no mínimo um individuo que chegava ao melhor *fitness* obtido para este problema, isto deve-se ao facto de ser um problema simples.

3.4.2 Problema2

Para este problema a configuração de parâmetros utilizada para os testes foi a seguinte:

Parâmetro	Valor
Runs	50
Population_size	50, 100, 200
Max_generations	50, 100
Selection	tournament
Tournament_size	2, 4, 6, 8
Recombination	pmx, cycle, order1
Recombination_probability	0.8, 0.7, 0.6
Mutation:	insert, swap, inversion
Mutation_probability:	0.1, 0.15, 0.2

Tabela 5 - Ficheiro de configuração para o problema 2

Relativamente ao problema 2 a melhor combinação de parâmetros encontrada, que gerou o melhor *fitness* médio, foi a seguinte:

Parâmetro	Valor
Runs	50
Population_size	200
Max_generations	100
Selection	tournament
Tournament_size	8
Recombination	pmx
Recombination_probability	0.8
Mutation:	swap
Mutation_probability:	0.2

Tabela 6 - Melhor configuração para o problema 2

O *fitness* médio obtido por esta combinação de parâmetros foi de 176.004.

É de notar que só há poucas combinações de parâmetros que chegam a valores de *fitness* médio muito próximos deste, 176. A combinação da mutação *Swap* com probabilidade de 0.2 com a recombinação *PMX* aparece várias vezes nas primeiras 8 posições.

O melhor *fitness* obtido por um só individuo foi de 166.6.

3.4.3 Problema3

Para este problema a configuração de parâmetros utilizada para os testes foi a seguinte:

Parâmetro	Valor
Runs	50
Population_size	50, 100
Max_generations	100, 200
Selection	tournament
Tournament_size	2, 4, 6, 8
Recombination	pmx, cycle, order1
Recombination_probability	0.8, 0.7, 0.6
Mutation:	insert, swap, inversion
Mutation_probability:	0.1, 0.15, 0.2

Tabela 7 - Ficheiro de configuração para o problema 3

Relativamente ao problema 3 a melhor combinação de parâmetros encontrada, que gerou o melhor *fitness* médio, foi a seguinte:

Parâmetro	Valor
Runs	50
Population_size	100
Max_generations	200
Selection	tournament
Tournament_size	2
Recombination	pmx
Recombination_probability	0.6
Mutation:	swap
Mutation_probability:	0.2

Tabela 8 - Melhor configuração para o problema 3

O *fitness* médio obtido por esta combinação de parâmetros foi de 193.476.

É de notar que a mutação *Swap* aparece várias vezes nas primeiras posições do ficheiro excel.

O melhor *fitness* obtido por um só individuo foi de 181.8.

3.4.4 Problema4

Para este problema foi usada uma configuração diferente para "correr" os testes gerais visto ser um problema mais "complexo", a seguinte combinação de parâmetros foi a usada:

Parâmetro	Valor
Runs	50
Population_size	50, 100, 200
Max_generations	100, 200
Selection	tournament
Tournament_size	2, 4, 6, 8
Recombination	pmx, cycle, order1
Recombination_probability	0.8, 0.7, 0.6
Mutation:	insert, swap, inversion
Mutation_probability:	0.1, 0.15, 0.2

Tabela 9 - Ficheiro de configuração para o problema 4

Relativamente a este problema a melhor combinação de parâmetros encontrada, que gerou o melhor *fitness* médio, foi a seguinte:

Parâmetro	Valor
Runs	50
Population_size	200
Max_generations	200
Selection	tournament
Tournament_size	2
Recombination	pmx
Recombination_probability	0.6

Mutation:	swap
Mutation_probability:	0.2

Tabela 10 - Melhor configuração para o problema 4

O *fitness* médio obtido por esta combinação de parâmetros foi de 132.672.

É de realçar a recombinação aparece várias vezes seguidas nas primeiras 22 posições do ficheiro excel.

O melhor *fitness* obtido por um só individuo foi de 123.4. Pode ser observado no ficheiro excel que este valor foi obtido por 3 configurações diferentes mas as três usavam a recombinação *PMX* e 2 delas o operador de mutação *Swap*.

3.5 Testes Específicos

3.5.1 Torneio

3.5.1.1 Problema1

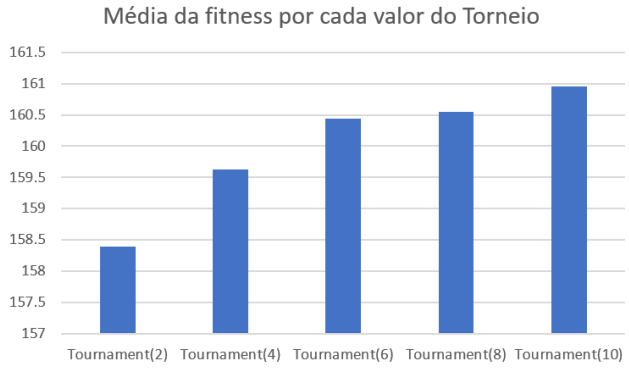

Figure 10 - Gráfico do Torneio para o problema 1

Podemos verificar pelo gráfico que para o problema 1 á medida que aumentamos o valor do torneio a *fitness* piora, valores menores irão gerar soluções melhores.

3.5.1.2 Problema2

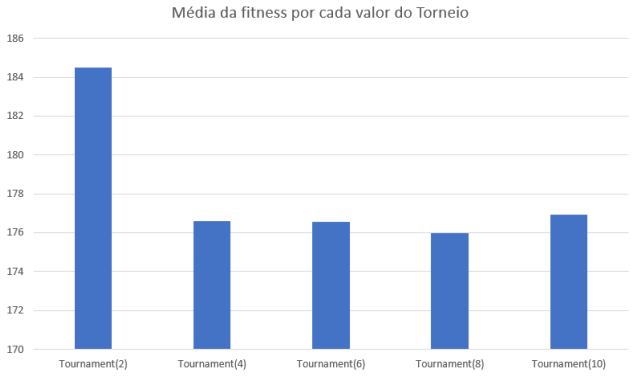

Figure 11 - Gráfico do Torneio para o problema 2

Podemos observar pelo gráfico que valores para o torneio ideais para este problema serão os mais elevados. O valor do torneio de 8 gera os melhores resultados.

3.5.1.3 Problema3

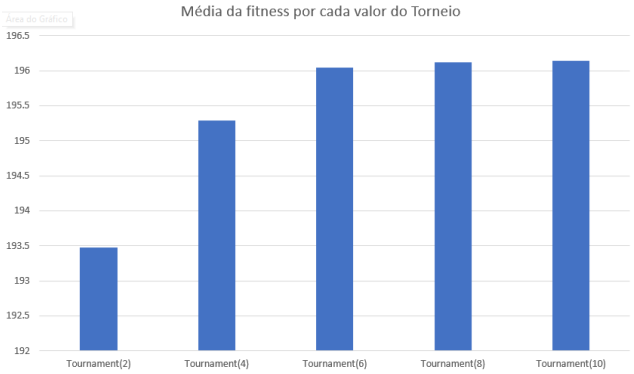

Figure 12 - Gráfico do Torneio para o problema 3

Podemos observar pelo gráfico que ao contrário das conclusões retiradas do gráfico do problema anterior, este problema, tal como o problema 1 obtem valores de fitness muito bons quando o valor do torneio é o menor, ou seja 2.

3.5.1.4 Problema4

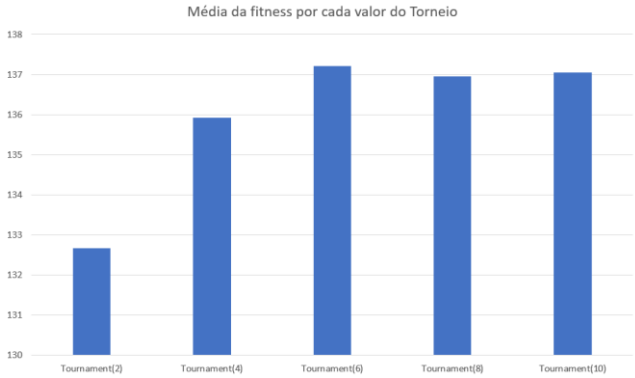

Figure 13 - Gráfico do Torneio para o problema 4

Tal como os problemas 1 e 3, este problema obtem os melhores valores de fitness para valores do torneio baixos.

3.5.2 Recombinação

3.5.2.1 Problema1

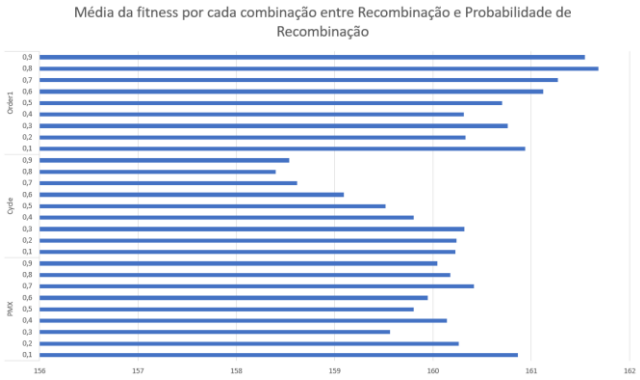

Figure 14 - Gráfico da Recombinação para o problema 1

Neste gráfico podemos observar que a recombinação *Cycle* é a que, na maioria dos casos, obtem os melhores resultados em

termos de fitness, sendo que a recombinação *Order1* é uma das piores recombinações para este problema.

3.5.2.2 Problema2

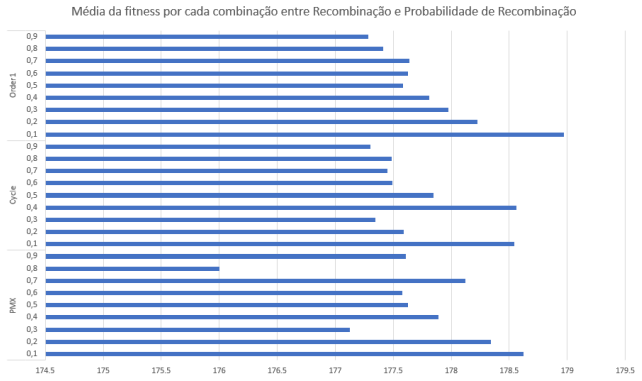

Figure 15 - Gráfico da Recombinação para o problema 2

Podemos ver pelo gráfico que a recombinação *PMX* é uma das melhores recombinações para este problema, sendo que a recombinação *Order1* é, novamente, a pior das três implementadas.

3.5.2.3 Problema3

Figure 16 - Gráfico da Recombinação para o problema 3

Podemos ver pelo gráfico que a recombinação *PMX* é de novo das melhores recombinações, tanto para os problemas anteriores como para este problema, sendo que a recombinação *Order1* continua a ser das piores das três implementadas.

3.5.2.4 Problema4

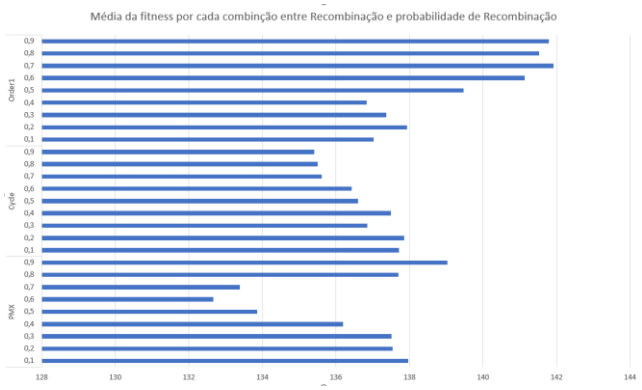

Figure 17 - Gráfico da Recombinação para o problema 4

De novo a recombinação *PMX* é a melhor como visto nos problemas anteriores e a recombinação *Cycle* está de novo em ultimo.

3.5.3 Mutação

3.5.3.1 Problema1

Figure 18 - Gráfico da Mutação para o problema 1

Por este gráfico podemos observar que as 3 mutações, Insert, Swap e Inversion chegam ao melhor valor de fitness possivél. 158.4. A mutação insert é a que chega a este valor o maior numéro de vezes.

3.5.3.2 Problema2

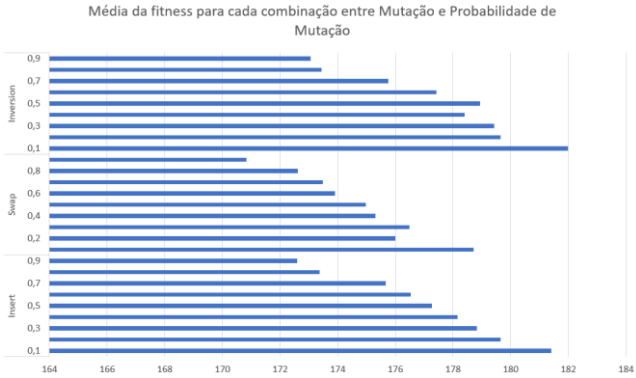

Figure 19 - Gráfico da Mutação para o problema 2

A partir do gráfico a mutação Swap é a que obtem os melhores valores para o fitness com as outras 2 mutações com valores muito proximos uma da outra para cada probabilidade de mutação.

3.5.3.3 Problema3

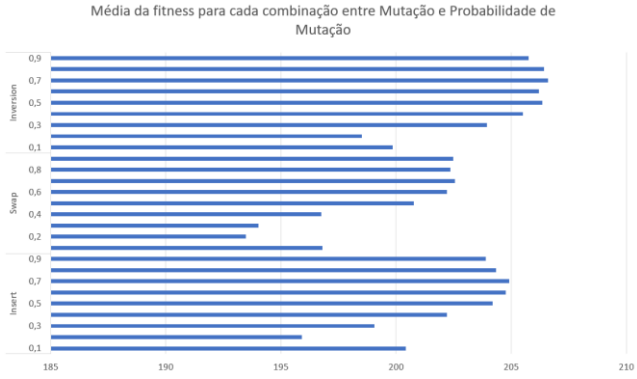

Figure 20 - Gráfico da Mutação para o problema 3

Observa-se que no gráfico, a mutação *Swap* é das que consegue obter dos melhores valores de fitness, sendo a mutação *Inversion* a pior das 3.

3.5.3.4 Problema4

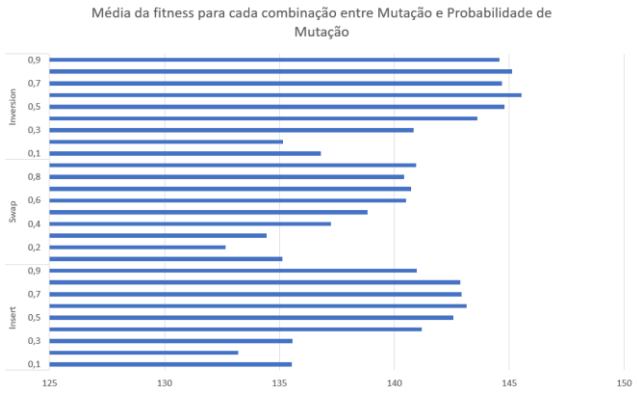

Figure 21 - Gráfico da Mutação para o problema 4

De novo a mutação Swap é das que obtem os melhores resultados para o fitness com a mutação Insert em segundo lugar. A mutação Inversion é de novo a pior entre as 3.

3.5.4 População

3.5.4.1 Problema1

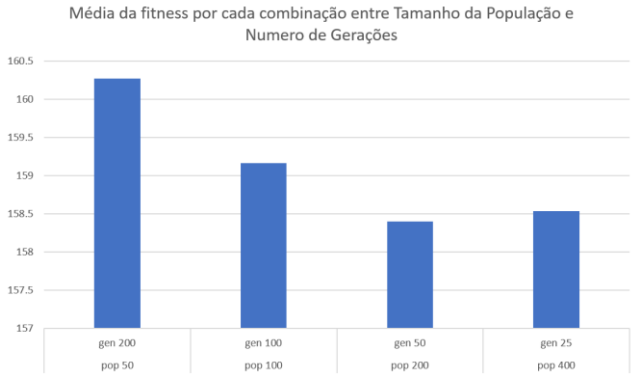

Figure 22 - Gráfico da População para o problema 1

Como podemos observar no gráfico, para este problema um maior numero de população é melhor que ter muitas gerações. Os valores de 25 para a geração e de 400 para a população e de 50 para a geração e 200 para a população deram os melhores resultados.

3.5.4.2 Problema2

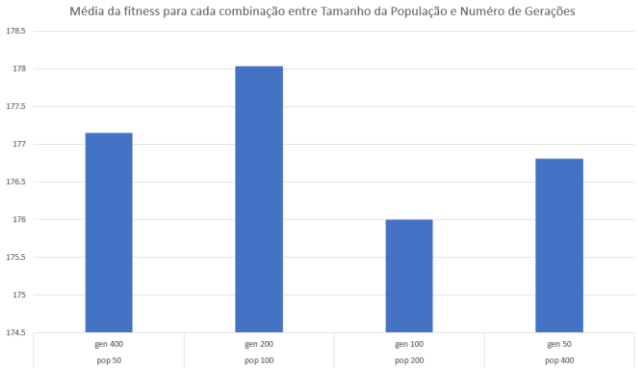

Figure 23 - Gráfico da População para o problema 2

Tal como no problema anterior maior numero de gerações faz com que obtenhamos valores de fitness piores.

3.5.4.3 Problema3

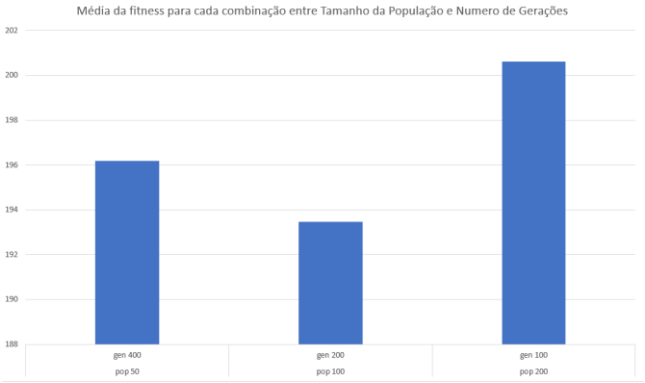

Figure 24 - Gráfico da População para o problema 3

Neste problema, os melhores resultados obtidos foram com uma população com valor médio, de 100, e um valor igualmente médio para as gerações, de 200.

3.5.4.4 Problema4

Figure 25 - Gráfico da População para o problema 4

Tal como no problema anterior, o melhor resultado obtido foi para valores médios de geração e população, de 200 e 200 respetivamente.

4. Conclusões

De todos os testes corridos, a recombinação PMX e a mutação Swap foram das que deram os melhores valores de fitness médios.

A mutação Inversion e a recombinação Order1 foram dos piores operadores genéticos em relação aos 4 problemas testados.

Em termos de valores para o torneio, o valor 2 deu dos melhores resultados em 3 dos 4 problemas.

Já para os valores da população e gerações, em problemas mais "simples" (problema 1 e problema 2), ter um valor de população superior ao valor das gerações traz melhores resultados. Já em problemas mais "complexos", (problema 3 e 4) é mais vantajoso ter um valor de gerações superior ao da população.

Achamos os dois que o projeto desenvolvido ajudou a consolidar a matéria lecionada sobre Algoritmos Genéticos, os seus operadores e a maneira como funcionam.

5. Referências

[1] "Genetic Algorithms," [Online]. Available: http://cobweb.cs.uga.edu/~khaled/ECcourse/chapter03.ppt

[2] "Rubicite Interactive Cycle Crossover," [Online]. Available: https://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/CycleCrossoverOperator.aspx

[3] "Rubicite Interactive Order1 Crossover," [Online]. Available: https://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/Order1CrossoverOperator.aspx

[4] "Stack Overflow Rotate N*M Matrix 90 degrees," [Online]. Available: https://stackoverflow.com/questions/18034805/rotate-mn-matrix-90-degrees