Projeto IA – Otimização de espaço

Rafael Mendes Pereira
2191266
2191266@my.ipleiria.pt

Bruna Alexandra Marques Leitão
2191182
2191182@my.ipleiria.pt

ABSTRACT

Nós, autores, alunos do curso de Engenharia Informática do Instituto Politécnico de Leiria escrevemos o presente relatório para descrever como nós pensámos, desenvolvemos, e testámos a solução para o problema de otimização de espaço, maior número de peças no menor tamanho de material e menor número de cortes.
Como solução desenvolvemos uma aplicação (java) utilizando um algoritmo genético e aleatório. Neste documento explicamos o desenvolvimento da mesma e discutimos os resultados obtidos por meio de testes.

Categorias e Sub-Descrições

[**Linguagem de Programação**]: O código foi escrito em java, esta usada para o leccionamento nas aulas. É uma linguagem orientada a objetos (POO) sendo formada por Classes, Atributos, Métodos, Construtores.

[**Experiências**]: Testes fundamentados usando ficheiros de texto externos devidamente formatados.

Termos Gerais

[Algoritmos]: Procedimentos estruturados e pensados.

[**Inteligência Artificial**]: Capacidade de máquinas pensarem como seres humanos. Aprender, perceber, e decidir racionalmente.

Palavras-chave

[IDE]: Integrated Development Environment ou Ambiente de Desenvolvimento Integrado.

[UC]: Unidade Curricular.

[Genome]: Variável que simula o cromossoma de um ser vivo.

1. Introdução

Fundamentando e contextualizando este relatório.

Hoje em dia as empresas que efetuam cortes sobre um material encaram um problema. Este passa por não conseguirem organizar as peças que vão extrair do material de maneria a aproveitarem o máximo de material relativamente ao número de peças extraídas e custos de energia relativamente aos cortes.

Com base no que foi referido acima e no âmbito da cadeira de “Inteligência Artificial” com o objetivo de conseguirmos aprovação na mesma, foi-nos proposto solucionar este problema recorrendo aos conteúdos lecionados da cadeira.

A aplicação que soluciona este problema foi escrita em java, recorrendo à IDE “IntelliJ”. A mesma teve código base fornecido pelos docentes da UC focada na utilização de um algoritmo genético, e seus operadores e um algoritmo aleatório.

O presente documento está dividido por capítulos. Começando pelo capítulo 2 onde exemplificamos o objetivo da aplicação. Nomeadamente o capítulo 3 onde é descrito as representações das soluções implementadas para este projeto. Seguido pelo capítulo 4 explicando a função de avaliação usada. No capítulo 5 descrevemos todos os operados genéticos desenvolvidos (recombinação e mutação). No capítulo 6 descrevemos detalhadamente a implementação do algoritmo aleatório. Posteriormente no capítulo 7 é discutido e apresentado os resultados de testes feitos à aplicação menciona anteriormente. Para concluir no capítulo 8, descrevemos e explicamos funcionalidades extras implementadas.

Índice

Índice de Figuras

Índice de Tabelas

Índice de Gráficos

2. Exemplo e demostração real

A empresa X quer cortar uma chapa de metal com as seguintes peças:

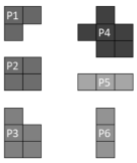

Figura 1 - Peças exemplo

Por razões económicas a empresa quer gastar a menor quantidade de material possível gastando a menor energia possível, gerindo então o número de cortes e o tamanho em largura usado do material.

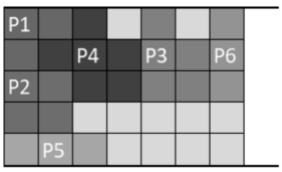

Figura 2 - Resultado esperado exemplo

Frisando e concluindo este exemplo que quanto menor for o gasto de material e menor o número de cortes melhor. Neste exemplo não foi efetuada qualquer rotação de peças, no entanto ao longo deste relatório será mencionada (Maioritariamente no capítulo 9).

3. Representação de Soluções

As soluções faladas neste capítulo são implementadas sobre o código base fornecido pelos docentes da UC.

A classe *StockingProblemIndividual* representativa de um individuo (solução) contem as seguintes funcionalidades e atributos.

Contém uma matriz bidimensional de caracteres (*materialCut*) que expressa o material a ser cortado. (Figura 2). Variáveis que guardam contagem de cortes verticais e horizontais (*verticalCuts*, *horizontalCuts*) (Figura 3), um corte é definido quando existe uma quebra de peça, seja esta para outra peça ou para o próprio material. Ainda é guardado para fins de penalização o comprimento máximo que o material pode ter, juntando todas as peças horizontalmente (*widthMaterialUsed*).

A matriz mencionada tem como tamanho a altura fornecida pelo problema e o comprimento *widthMaterialUsed*.

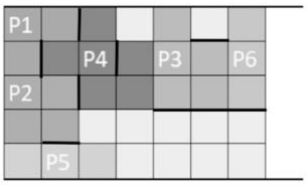

Figura 3 - Representação de Cortes

Por meio de extensão da classe "*IntVectorIndividual*" introduzimos o conceito de *genome*. Este simula um cromossoma de um ser vivo, contendo todos os genes (neste caso equivale a peças).

Genome é um vetor de inteiros que guarda um inteiro representante de uma peça, assim define a ordem a que as mesmas são colocadas, este é gerado aleatoriamente usando o método estático *ramdom* da classe "*Algorithm*", tendo tantas posições como o número de peças existentes no problema, não havendo peças repetidas o *genome* assim é gerado, sem repetição de valores dentro de si.

As peças são colocadas no material de cima para baixo, da esquerda para a direita.

Para calculo do fitness do individuo o algoritmo usa a função "*computeFitness*" e toma os dentro da mesma, a matriz *materialCut* é definida no método "*processMaterialCut",* é criado uma lista de peças seguindo os padrões do *genome,* esta ao invés de ter um valor que presenta uma peça, tem a própria peça. Iterando essa lista, validando e procurando a melhor posição para inserção da peça com auxílio da função "*checkValidPlacement"*, concluído com a inserção da peça na variável *materialCut*. São ainda contados os cortes e o material usado.

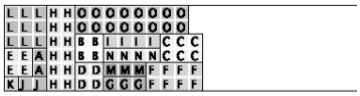

Figura 4 - Ilustração da matriz *materialCut*

4. Função de Avaliação Utilizada

Com o objetivo de saber se um individuo é melhor ou pior que outro, pelos motivos já falados, energia gasta em cortes, comprimento de material gasto é necessário a aplicação saber avaliar quando um individuo é melhor que outro.

Dentro do método *computeFitness*. Já com a *materialCut* definida, e com a contagem de cortes e tamanho do material usado como referenciado no capítulo 3. É somado os cortes mais o material usado devolvendo assim o *fitness*.

Considero o valor de fitness bem estruturado, sem necessidade de dar pesos diferentes entre tipos de cortes ou o tamanho de material usado, pois o custo de energia de uma máquina de corte (por exemplo CNC) tem uma relação direta com o tamanho da peça.

5. Algoritmo genético

O algoritmo genético é um método de otimização que visa simular o desenvolvimento e processo de vida de um ser (individuo). Fazendo este passar por mutações (equiparado ao desenvolvimento) e recombinações (juntando-se com outros seres criando novos).

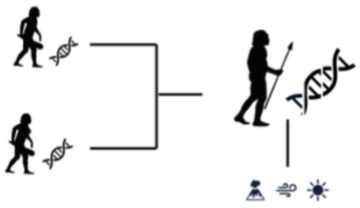

Figura 5 - Representação do algoritmo genético

6. Operadores Genéticos Desenvolvidos

Os operadores genéticos são usados para simular o desenvolvimento (mutação) e junção de indivíduos (recombinação). Existem infinitas opções de operadores. Assim nós disponibilizamos os seguintes:

- Operadores de mutação:
  - Displacement (DM)
  - Exchange (EM)
  - Insert (Dado pelos docentes)
  - InsertionSimple (ISM)
  - InversionDisplacement (IDM)
  - SimpleInversion (SIM)
- Operadores de recombinação:
  - ModifiedCycle
  - Order

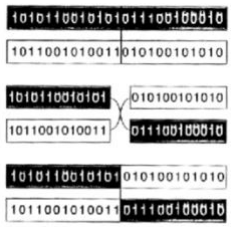

Figura 6 - Operadores genéticos

  - PartialMapped (Dado pelos docentes)

6.1 Operadores de mutação

6.1.1 Displacement (DM)

O operador *Displacement* consiste em selecionar um subconjunto de genes aleatório e colocá-lo numa nova posição aleatória.

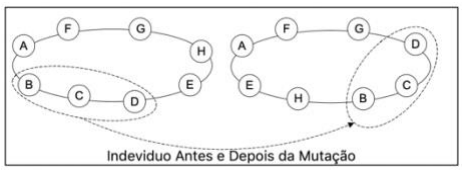

Figura 7 - Aplicação do operador - Displacement Mutation

Para tal, criámos um subconjunto aleatório, fazendo dois cortes aleatórios (usando o método estático *ramdom* da classe abstrata "*Algorithm*", estes têm de estar contidos entre 0 e o número de genes ou peças menos 1) no *genome*, e guardando o subconjunto numa variável inserindo os valores desde o primeiro ao segundo corte, gerámos a nova posição também aleatória. Após sabermos tanto o subconjunto como a nova posição, é guardado num vetor todos os valores/genes que não pertencem ao subconjunto.

Finalizando com a inserção do subconjunto num último vetor já na nova posição, para depois preencher os valores restantes anteriormente guardados nas posições vazias até ao momento.

6.1.2 Exchange (EM)

Este operador consiste em selecionar dois genes aleatoriamente no cromossoma/*genome* de um individuo e trocá-los.

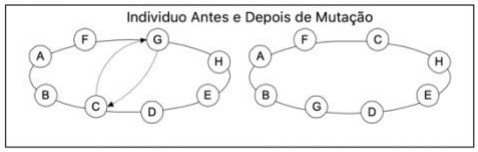

Figura 8 - Aplicação do operador - Exchange Mutation

São gerados dois valores aleatórios (usando o método estático *ramdom* da classe abstrata "*Algorithm*", estes têm de estar contidos entre 0 e o número de genes ou peças menos 1) e trocamos os genes das posições relativas aos valores gerados.

6.1.3 Insert

Este operador consiste em selecionar um subconjunto, usando o método de cortes com valores aleatórios, e inverter os valores dentro de si, descendentemente.

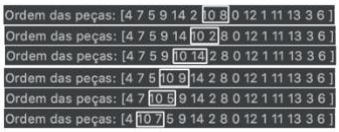

Figura 9 - Aplicação do operador - Insert Mutation

6.1.4 InsertionSimple (ISM)

A aplicação deste operador consiste em selecionar um gene aleatoriamente do nosso cromossoma e escolher uma nova posição para o mesmo.

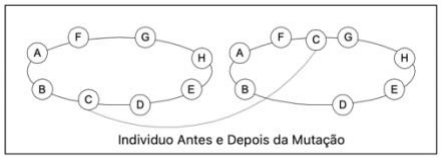

Figura 10 - Aplicação do operador - InsertionSimple

Para tal geramos duas posições aleatórias baseadas no *genome* ao qual usamos uma das posições para retirar o valor e a segunda para colocá-lo num vetor vazio com o mesmo tamanho que o número de genes. Assim, todas as posições vazias desse vetor serão preenchidas com os restantes valores.

6.1.5 InversionDisplacement (IDM)

Este operador é similar ao já referido *Displacement*, ao qual seleciona um subconjunto e coloca numa nova posição. Porém agora esse subconjunto é inserido de ordem inversa ao original.

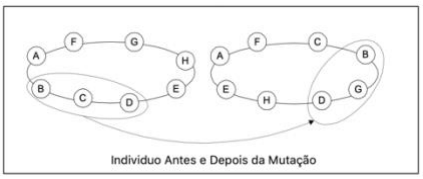

Figura 12 - Aplicação do operador - InversionDisplacement Mutation

À semelhança da mutação *displacement*, é criado um subconjunto baseado em dois cortes sobre o *genome*, a ordem deste é invertida. É gerado ainda um valor respetivo à nova posição, ao qual é usado para fazermos a inserção num novo vetor do subconjunto. Assim adicionamos os restantes valores que não estão presentes no subconjunto.

6.1.6 SimpleInversion (SIM)

A aplicação este operador de mutação consiste em selecionar um subconjunto aleatório, baseado em dois cortes sobre o *genome* e trocar os valores da primeira e última posição, da segunda e da penúltima posição, e assim sucessivamente.

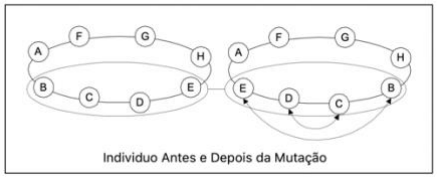

Figura 11 - Aplicação do operador - SimpleInversion Mutation

6.2 Operadores de recombinação

Um operador de recombinação usa dois "pais" e devolve dois "filhos". Estas expressões de pais e filhos, progenitores e descendentes, vão ser usadas ao longo da descrição dos operadores como auxílio à explicação.

6.2.1 ModifiedCycle (CX2)

Operador de crossover de ciclo modificado (CX2) foi proposto por Hussain et al. É uma forma modificada de CX, por isso a abreviatura CX2. Da mesma forma que o CX, ele gera os dois filhos dos pais usando o(s) ciclo(s) até o último bit.

Esta técnica é usada para criar descendência de tal forma que o primeiro bit do segundo pai seja o primeiro bit do primeiro filho e, em seguida, pesquise esse bit no primeiro pai e escolha o mesmo bit na localização do segundo pai e pesquise novamente no primeiro pai e novamente escolhe o mesmo bit na localização do segundo pai e esse bit é o primeiro bit da segunda descendência.

Por exemplo, considere os estes dois pais:

CX2: Caso 1

$$P1 = (3\ 4\ 8\ 2\ 7\ 1\ 6\ 5),$$

$$P2 = (4\ 2\ 5\ 1\ 6\ 8\ 3\ 7).$$

O primeiro bit do segundo pai é o primeiro bit do primeiro filho:

$$O1 = (4 \times \times \times \times \times \times \times).$$

O bit selecionado é 4, então procuramos o bit 4 que se encontra na segunda posição no primeiro pai e o bit nesta posição no segundo pai é 2.

Repete-se o processo pesquisando o bit 2 no primeiro pai e está na quarta posição e 1 está na mesma posição no segundo pai, então 1 é selecionado para o segundo filho:

$$O2 = (1 \times \times \times \times \times \times \times).$$

O bit anterior era 1 e está localizado na 6ª posição no primeiro pai e nessa posição o bit é 8 no segundo pai, então:

$$O1 = (4\ 8 \times \times \times \times \times \times).$$

E com dois movimentos para preencher o segundo filho, como abaixo de 8 é 5 e abaixo de 5 é 7, então:

$$O2 = (1\ 7 \times \times \times \times \times \times).$$

Portanto, da mesma forma,

$$O1 = (4\ 8\ 6\ 2\ 5\ 3\ 1\ 7),$$

$$O2 = (1\ 7\ 4\ 8\ 6\ 2\ 5\ 3).$$

Vemos que o último bit do segundo filho é 3, que foi o primeiro bit do primeiro pai. Portanto, o esquema proposto termina num ciclo.

CX2: Caso 2

Às vezes, não termina num ciclo, considere outro exemplo:

$$P1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8),$$

$$P2 = (2\ 7\ 5\ 8\ 4\ 1\ 6\ 3).$$

Utilizando o exemplo anterior completamos os filhos:

$$O1 = (2\ 1\ 6\ 7 \times \times \times \times),$$

$$O2 = (6\ 7\ 2\ 1 \times \times \times \times).$$

Paramos porque preenchemos o bit 1 do segundo filho que estava na 1ª posição do primeiro pai. Um ciclo acabou porque o próximo bit a ser preenchido seria o 2 que estava na primeira posição do segundo pai e já se encontra no filho. Então antes de iniciar outro ciclo, combinamos os bits do primeiro filho com o segundo pai ou vice-versa e deixamos de fora os bits existentes com sua posição em ambos os pais como:

$$P1 = (\bullet\ \bullet\ 3\ 4\ 5\ \bullet\ \bullet\ 8),$$

$$P2 = (\bullet\ \bullet\ 5\ 8\ 4\ \bullet\ \bullet\ 3).$$

No nosso projeto, preferimos criar dois auxiliares, offspring1 e offspring2, para não modificar os pais.

Agora, as posições preenchidas de pais e '×' de filhos são consideradas 1ª, 2ª e 3ª posições, etc., para que possamos concluí-lo como de costume:

$$O1 = (2\ 1\ 6\ 7 \mid 5\ 3\ 8\ 4),$$

$$O2 = (6\ 7\ 2\ 1 \mid 8\ 4\ 5\ 3).$$

E assim o esquema acaba.

6.2.2 Oder (OX)

Neste operador um descendente é construído com um subconjunto (baseado em dois cortes) de um pai, e os restantes valores de um outro pai preservando a sua ordem após o índice do segundo corte usado para construir o subconjunto. Ao encontrar valores repetidos estes são ignorados passando para o próximo. O mesmo acontece para o segundo descendente, mas com os focos de pais invertidos.

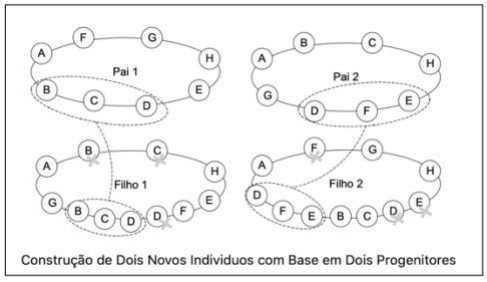

Figura 13 - Aplicação do operador - Order Crossover

Na prática aquilo que é feito é o seguinte, são gerados dois valores para os cortes, estes valores não podem ser iguais e o primeiro corte tem de ser inferior ao segundo corte. Com base nesses cortes são gerados os subconjuntos para ambos os descendentes como ilustra a Figura 13 (recorrendo ao método *create_Segments*). São criadas as sequências a acrescentar aos subconjuntos para cada descendente, do filho1 a sequência é baseada nos genes do pai2 e a sequência do filho2 baseada nos genes do pai1 (usando o método *create_Sequences*). Para finalizar só temos de juntar a sequência de cada descendente com o seu subconjunto garantindo que não existem valores repetidos (usando o método *cossOver*).

6.2.3 PartialMapped (PMX)

O operador *partial-mapped* cria os descendentes escolhendo dois subconjuntos (baseados em dois cortes) de ambos os pais e então realiza a troca dos mesmos. No segundo passo, os elementos pertencentes ao subconjunto enviado são mapeados sobre os genes do cromossomo, que coincidem com os do segmento recebido. O mapeamento evita a geração de indivíduos inválidos.

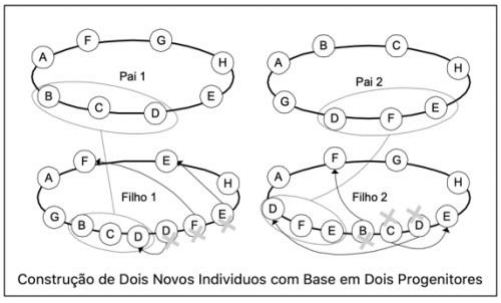

Figura 14 - Aplicação do operador - Partial-Mapped Crossover

7. Implementação do Algoritmo Aleatório

Nem sempre a utilização do algoritmo genético é melhor. Para ultrapassar problemas como a rapidez e ou simplicidade que o algoritmo genético não tem, neste capítulo vamos então falar sobre o algoritmo aleatório e a nossa implementação ao mesmo.

Existem dois grandes grupos de algoritmos aleatórios bem conhecidos são, "Monte Carlo" focados na fidelidade da resposta e "Las Vegas" focados no tempo de resposta. A escolha de um destes grupos depende somente do caso real ou em estudo.

Passando à implementação e acreditando que esta se integra no grupo "Las Vegas" devido a alguns testes realizados e resultantes de uma grande diferença de tempo relativamente ao algoritmo genético, e os resultados fitness não serem melhores.

8. Resultados Obtidos

Neste capítulo iremos relatar os resultados que obtivemos em cada dataset testado. Com o objetivo de mostrar e provar qual a melhor configuração para cada um com a ajuda de tabelas e gráficos.

A configuração dos computadores onde os testes realizados:

- Problema 1
  - Processador: I7 8º Geração
  - Disco: SSD
  - Ram: 8Gb DDR4
- Problema2,3,4,5
  - Processador: I3 6º Geração
  - Disco: SSD
  - Ram: 8Gb DDR4 2400Hz

Para guia inicial da realização de testes, foram feitos testes preliminares ao problema1 estes seguidos para os restantes datasets. Estes testes preliminares servem para reduzirmos o número de combinações nos testes gerais.

Tabela 1 - Testes preliminares à mutação sem variância de probabilidade de mutação

Population si	Max generat	Selection:	Recombinati	Recombinati	Mutation:	Mutation prob.:	Average:	StdDev:
50	100	Tournament(	PMX	0,7	EM	0,1	285,72	1,3272528
50	100	Tournament(	PMX	0,7	Insert	0,1	285,78	1,31590273
50	100	Tournament(	PMX	0,7	IDM	0,1	286,36	1,59699718
50	100	Tournament(	PMX	0,7	SIM	0,1	286,52	1,6277592
50	100	Tournament(	PMX	0,7	DM	0,1	286,8	1,58745079
50	100	Tournament(	PMX	0,7	ISM	0,1	287,32	2,06339526

Começamos por fazer testes à mutação, no intuito de arranjar três mutações "preferidas".

Este teste foi realizado com os valores por defeito, fazendo apenas variar a mutação. Concluímos então que nesta fase a mutação Exchange (EM) destacou-se. Não achando os testes conclusivos variámos a probabilidade de mutação.

Tabela 2 - Testes preliminares à mutação com variância de probabilidade de mutação

Population si	Max generat	Selection:	Recombinati	Recombinati	Mutation:	Mutation prob.:	Average:	StdDev:
50	100	Tournament(	PMX	0,7	Insert	0,3	284,7	1,02469508
50	100	Tournament(	PMX	0,7	EM	0,3	284,72	0,89532117
50	100	Tournament(	PMX	0,7	EM	0,2	285,12	1,33626345
50	100	Tournament(	PMX	0,7	Insert	0,2	285,14	1,29630243
50	100	Tournament(	PMX	0,7	SIM	0,3	285,6	1,2

Neste ponto conseguimos ter uma melhor certeza de quais mutações devemos escolher daqui para a frente. Destacaram-se as mutações, *Insert*, *Exchange* (*EM*), *SimpleInvertion* (*SIM*).

Com isto passamos para os testes preliminares relativos à recombinação, usámos como base o melhor dos testes anteriores variando assim os algoritmos de recombinação e a sua probabilidade.

Population si	Max generat	Selection:	Recombinati	Recombinati	Mutation:	Mutation prob.:	Average:	StdDev:
50	100	Tournamenti		0,6	Insert	0,3	285,02	1,06752049
50	100	Tournamenti	PMX	0,7	Insert	0,3	284,7	1,02469508
50	100	Tournamenti		0,8	Insert	0,3	284,42	0,60299254
50	100	Tournamenti		0,6	Insert	0,3	286,08	1,50784615
50	100	Tournamenti	OX	0,7	Insert	0,3	286,4	1,32664992
50	100	Tournamenti		0,8	Insert	0,3	286,48	1,51314243
50	100	Tournamenti		0,6	Insert	0,3	286,26	1,68297356
50	100	Tournamenti	CX2	0,7	Insert	0,3	286,96	1,41364776
50	100	Tournamenti		0,8	Insert	0,3	286,98	1,47634684

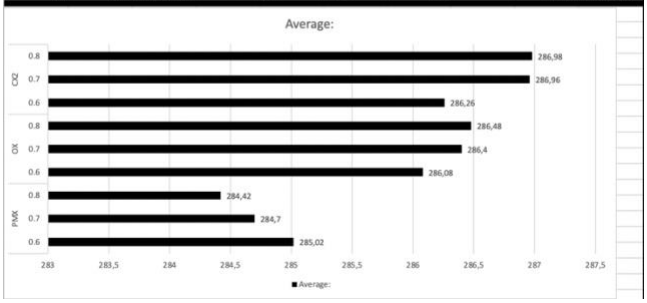

Gráfico 1 – Testes preliminares à recombinação com variância de probabilidade de recombinação

Testes preliminares à recombinação para termos perceção/sensibilidade da probabilidade de recombinação

Com variação da probabilidade entre 0.6, 0.7, 0.8: (*Average*)

Melhores valores de cada operador de recombinação.

- PMX - 0.8 - 284.42
- OX - 0.6 - 286.08
- CX2 - 0.6 - 286.26

Não achando conclusivos os testes relativos à recombinação, iremos então prosseguir com estes valores para mais alguns testes preliminares

Iremos então realizar mais alguns testes adicionando mais algumas variantes nos seguintes campos, *max_generation*, *recombination*, *recombination_probability*, *mutation_probability*.

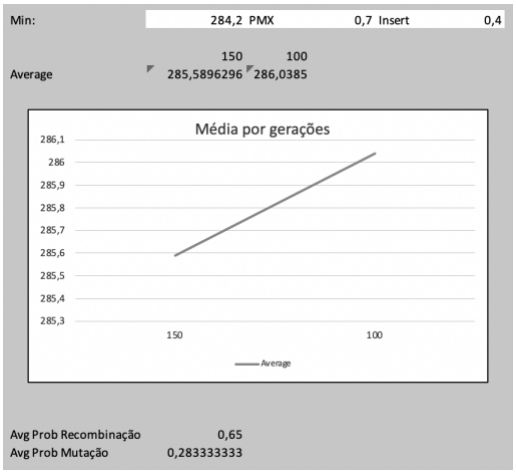

Gráfico 2 - Testes perliminares variados

Pudemos confirmar que a probabilidade de recombinação pendeu para baixo, a maior parte dos melhores valores são (0.6 e 0.7)

Obtivemos como melhor resultado:

Fitness	Recombinação	Prob. Recombinação	Mutação	Prob. Mutação
284.2	PMX	0.7	Insert	0.4

A diferença entre gerações não foi suficientemente grande, então assumimos que o valor mais baixo (100) já é estável.

O algoritmo de recombinação que tem conseguido valores mais baixos é o PMX.

A probabilidade de mutação que alcançou um valor de fitness mais baixo foi 0.7, no entanto o valor que mais aparece é 0.6

A probabilidade de mutação mantém-se estável entre 0.2 e 0.3.

- Média de Probabilidade de Recombinação: 0.65
- Média de Probabilidade de Mutação: 0.28333...

Com base nos parâmetros experienciados nos testes preliminares, procedemos às experiências específicos aos datasets.

8.1 Testes ao Problema1

8.1.1 Testes Gerais

Achando relevante mostrar o ficheiro de configuração dos testes gerais.

```
Runs: 50
Population_size: 50, 100, 150
Max_generations: 100
//--------------------------------
Selection: tournament
Tournament_size: 2, 4, 6
//--------------------------------
Recombination: pmx, ox, cx2
Recombination_probability: 0.6, 0.7, 0.8
//--------------------------------
Mutation: insert, exchange, displacement
Mutation_probability: 0.2, 0.3, 0.4
//--------------------------------
Problem_file: Problema1.txt
//--------------------------------
Statistic: BestIndividual
Statistic: BestAverage
```

Figura 15 – Problema1 - Ficheiro de configuração usado nos testes gerais

Population	Max genera	Selection:	Recombina	Recombina	Mutation:	Mutation prob.:	Average:	StdDev:
150	100	Tournamer	PMX	0,6	EM	0,3	284,04	0,195959
150	100	Tournamer	PMX	0,6	Insert	0,4	284,06	0,237487
150	100	Tournamer	PMX	0,8	Insert	0,4	284,12	0,324962
150	100	Tournamer	PMX	0,6	Insert	0,3	284,14	0,4005
150	100	Tournamer	PMX	0,7	EM	0,2	284,14	0,4005
100	100	Tournamer	PMX	0,6	Insert	0,4	284,16	0,366606
150	100	Tournamer	PMX	0,7	Insert	0,2	284,18	0,433128
150	100	Tournamer	PMX	0,7	EM	0,4	284,18	0,384187
150	100	Tournamer	PMX	0,6	EM	0,4	284,2	0,447214
100	100	Tournamer	PMX	0,6	Insert	0,3	284,22	0,46

Tabela 3 - Problema1 - Testes gerais relativos

Como podemos observar o melhor resultado obtido foi 284,04 com a Mutation EM (0,3). Mas a Insert também aparece muita vez com bons resultados, por isso não a vamos deixar de lado. A pior foi a DM que não aparece nos melhores 10.

8.1.2 Testes ao tamanho da população e número de gerações

Variámos a população e as gerações em:

Population_size: 50, 100, 150, 200

Max_generations: 50, 100, 150, 200

Population	Max genera	Selection:	Recombina	Recombina	Mutation:	Mutation prob.:	Average:	StdDev:
200	150	Tournamen	PMX	0,6	Insert	0,3	284	0
200	150	Tournamen	PMX	0,6	EM	0,3	284	0
200	200	Tournamen	PMX	0,6	Insert	0,3	284	0
200	200	Tournamen	PMX	0,6	EM	0,3	284	0
150	150	Tournamen	PMX	0,6	Insert	0,3	284,02	0,14
150	150	Tournamen	PMX	0,6	EM	0,3	284,02	0,14
150	200	Tournamen	PMX	0,6	Insert	0,3	284,02	0,14
150	200	Tournamen	PMX	0,6	EM	0,3	284,02	0,14
150	100	Tournamen	PMX	0,6	EM	0,3	284,04	0,195959179
200	100	Tournamen	PMX	0,6	Insert	0,3	284,06	0,237486842

Tabela 4 – Problema1 -Testes ao tamanho da população e número de gerações

Gráfico 3 - Problema1 - População vs Gerações

Population	Max genera	Average:	Average/(Population*MaxGeneration)
pop 150	ger 100	284,04	0,018936
pop 200	ger 100	284,06	0,014203
pop 150	ger 150	284,02	0,012623111
pop 150	ger 150	284,02	0,012623111
pop 150	ger 200	284,02	0,009467333
pop 150	ger 200	284,02	0,009467333
pop 200	ger 150	284	0,009466667
pop 200	ger 150	284	0,009466667
pop 200	ger 200	284	0,0071
pop 200	ger 200	284	0,0071

Gráfico 4 - Problema 1 - Average/(Population*MaxGeneration)

Em relação à média dos resultados, o melhor seria com População 150 e Gerações 100, que são 50 a menos em ambos os campos do melhor resultado acima. Será que 0,4 a menos de *Average* vale esses 50 valores a mais de população + 50 valores a mais de *MaxGeneration*?

Tendo isto em conta, vamos então usar a seguinte combinação:

Tabela 5 - Problema 1 - Melhor resultado relativo aos testes de população e gerações

Population size:	Max generations:	Recombination:	Recombination prob.:	Mutation prob.:
150	100	PMX	0,6	0,3

8.1.3 Testes ao tamanho do torneio

Variámos o valor do torneio em:

Tournament_size: 2, 4, 6, 8, 10

Population	Max genera	Selection:	Recombina	Recombina	Mutation:	Mutation prob,:	Average:	StdDev:
150	100	Tournamer	PMX	0,6	EM	0,3	284,04	0,195959
150	100	Tournamer	PMX	0,6	EM	0,3	284,5	0,8544
150	100	Tournamer	PMX	0,6	EM	0,3	284,6	1,03923
150	100	Tournamer	PMX	0,6	EM	0,3	284,86	0,98
150	100	Tournamer	PMX	0,6	EM	0,3	284,88	1,259206

Selection:	Average:
Tournamer	284,04
Tournamer	284,5
Tournamer	284,6
Tournamer	284,86
Tournamer	284,88

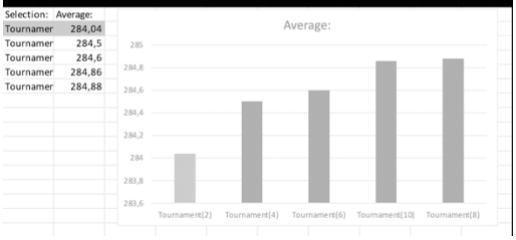

Gráfico 5 - Problema 1 - Testes ao tamanho do torneio

Os melhores resultados vieram do torneio (2)

Prosseguindo com o melhor resultado para o próximo teste

8.1.4 Testes à recombinação

Relativamente à recombinação tentámos alargar a nossa visão relativa às probabilidades, já que os testes preliminares não tinham sido conclusivos.

- *Recombination*: pmx, ox, cx2
- *Recombination_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

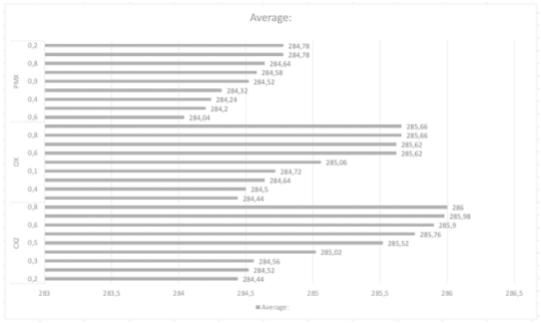

Gráfico 6 - Problema 1 - Testes à recombinação

Prosseguindo então com o melhor resultado dos testes anteriores para os próximos testes

Tabela 6 - Problema 1 - Melhor resultado relativo aos testes de recombinação

Recombination	Recombination Prob.	Average Fitness
PMX	0,6	284,04

8.1.5 Testes à mutação

No desenvolvimento do projeto, como extra, criámos algumas mutações extra, então nestes testes fizémos variar as mesmas com probabilidades demutação diferentes:

- *Mutation*: *insert*, *exchange*, *displacement*, *simpleinversion*, *invertionsimple*, *inversiondisplacement*
- *Mutation_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

Pudemos então afirmar que a mutação que mais se destacou por ter um valor de fitness mais baixo foi a 'EM' com uma probabilidade de mutação de 0.3.

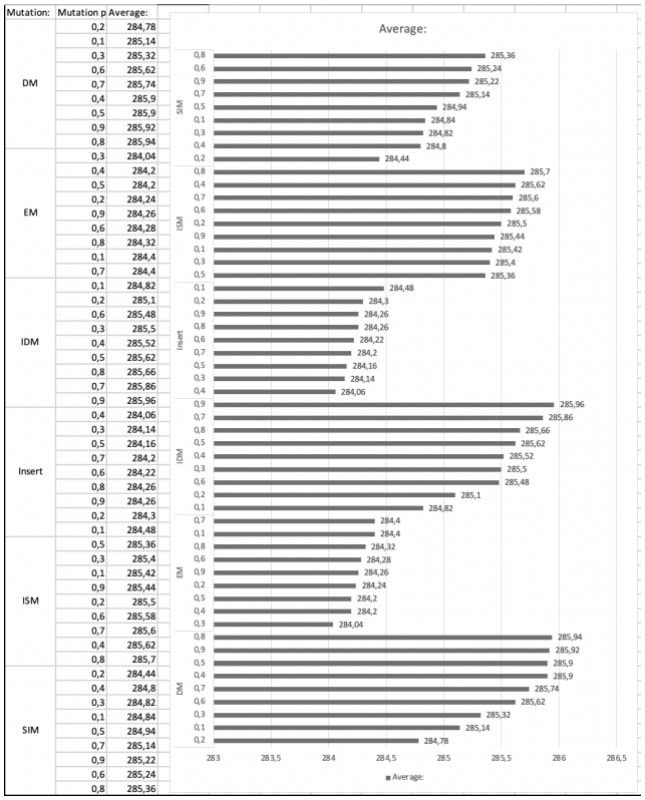

Gráfico 7 - Problema 1 - Testes à mutação

Assim, depois destes testes assumimos que os melhores parâmetros são:

Assim, depois destes testes assumimos que os melhores parâmetros são:						
Population size:	Max generations:	Selection:	Recombination:	Recombination prob.:	Mutation:	Mutation prob.:	Average:
150	100	Tournament(2)	PMX	0.6	EM	0,3	284,04

Figura 16 - Problema 1 - Melhores resultados finais

8.2 Testes ao Problema2

Para o dataset Problema2 começámos baseados nos testes preliminares feitos ao dataset Problema1.

Assim executando testes gerais mais precisos.

8.2.1 Testes Gerais

```
Runs: 50

Population_size: 50, 100, 150

Max_generations: 100, 150

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

Selection: tournament

Tournament_size: 2, 4, 6

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

Recombination: pmx, ox, cx2

Recombination_probability: 0.6, 0.7, 0.8

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

Mutation: insert, exchange, displacement

Mutation_probability: 0.2, 0.3, 0.4

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

Problem_file: Problema2.txt

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

Statistic: BestIndividual
Statistic: BestAverage
```

Figura 17 - Problema 2 - Ficheiro de configuração usado nos testes gerais

Com isto pudemos concluir que por ser um dataset mais pequeno não havia muita discrepância de fitness. No entanto começamos a ver certas combinações a destacarem-se por aparecerem mais frequentemente com valores mais baixos.

Tabela 7 - Problema 2 - Testes Gerais

Population si	Max generat	Selection:	Recombinati	Recombinati	Mutation:	Mutation prob,:	Average:
150	150	Tournament(4)	PMX	0,8	EM	0,4	305,22
150	150	Tournament(6)	PMX	0,8	EM	0,4	305,38
100	150	Tournament(2)	PMX	0,6	EM	0,4	305,8
150	150	Tournament(4)	PMX	0,7	EM	0,3	305,92
150	150	Tournament(4)	PMX	0,7	EM	0,4	305,94
150	100	Tournament(4)	PMX	0,8	EM	0,4	306,12
150	150	Tournament(4)	PMX	0,8	EM	0,3	306,32
100	150	Tournament(4)	PMX	0,6	EM	0,4	306,4
150	150	Tournament(6)	PMX	0,7	EM	0,4	306,42
150	150	Tournament(2)	PMX	0,6	EM	0,3	306,52

Seguimos então com o melhor valor dos resultados acima para os testes do tamanho da população

8.2.2 Testes ao tamanho da população e número de gerações

Variámos a população e as gerações em:

- *Population_size*: 50, 100, 200, 400
- *Max_generations*: 25, 50, 100, 200

Por lápso variámos também a mutação

- Mutation: insert, exchange, displacement

Efetuamos os calculos de Media sobre a População multiplicado pelas Gerações

Para sabermos a relação direta entre a média do fitness e (Population*Generations)

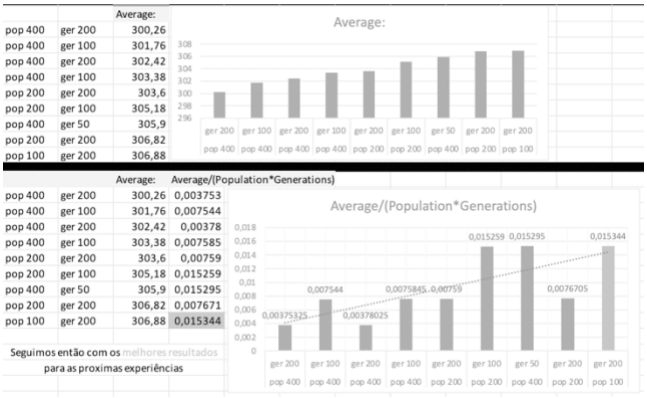

Gráfico 8 - Problema 2 - Testes ao tamanho da população

8.2.3 Testes ao tamanho do torneio

Variámos o tamanho do torneio em:

- *Selection*: tournament
- *Tournament_size*: 2, 4, 6, 8, 10

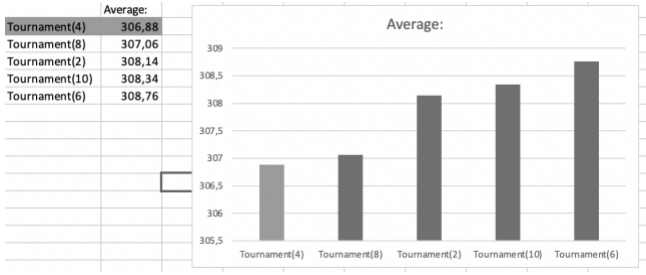

Gráfico 9 - Problema 2 - Testes ao tamanho do torneio

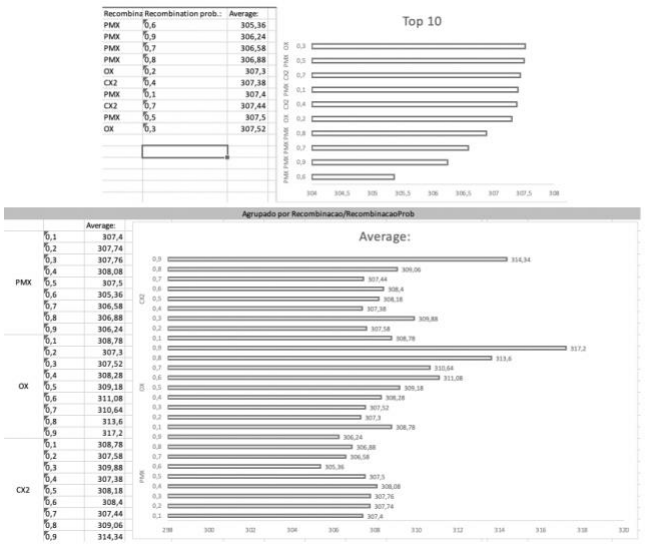

Gráfico 10 - Problema 2 - Testes à recombinação

Os melhores resultados vieram do torneio (4)

Prosseguindo com o melhor resultado para o próximo teste

8.2.4 Testes à recombinação

Relativamente à recombinação tentámos alargar a nossa visão relativa às probabilidades, já que os testes preliminares não tinham sido conclusivos.

- *Recombination*: pmx, ox, cx2
- *Recombination_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

Ao concluir estes testes pudemos agora dizer com certezas que a recombinação que mais se destacou pelos melhores resultados foi a *'pmx'*, relativamente à probabilidade de recombinação conseguimos afirmar que valores [0.2, 0.7] destacaram-se na generalidade das recombinações, no entanto, na recombinação *pmx*, a que obteve melhores resultados, os melhores valores mantiveram-se entre [0.6, 0.9].

Prosseguindo então com o melhor resultado dos testes anteriores para os próximos testes

Recomb.	Recomb.Prob	Average
PMX	0,6	305,36

8.2.5 Testes à mutação

Como já mencionado, criámos algumas mutações extra, então nestes testes fizemos também variar as mesmas com probabilidades de mutação diferentes:

- *Mutation*: *insert*, *exchange*, *displacement*, *simpleinversion*, *invertionsimple*, *inversiondisplacement*
- *Mutation_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

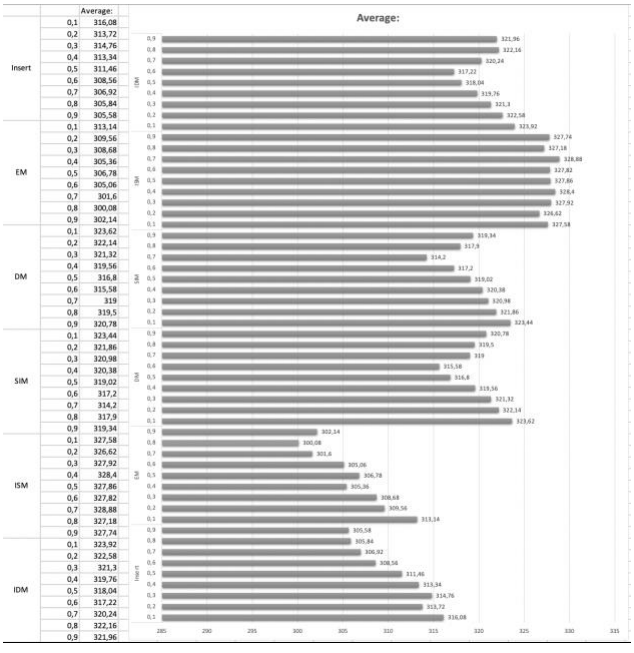

Gráfico 11 - Problema 2 - Testes à mutação

Pudemos então afirmar que a mutação que mais se destacou por ter um valor de fitness mais baixo foi a 'EM' com uma probabilidade de mutação de 0.8.

No geral a mutação que obteve valores mais baixos de fitness foi a 'EM', em qualquer probabilidade superou todas outras.

Assim, depois destes testes assumimos que os melhores parâmetros são:

Population size:	Max generations:	Selection:	Recombination:	Recombination prob.:	Mutation:	Mutation prob.:	Average:
100	200	Tournament(4)	PMX	0.6	EM	0.8	300,08

Figura 18 - Problema 2 - Melhor Resultado Final

8.3 Testes ao Problema3

Para o dataset Problema3 começámos baseados nos testes preliminares feitos ao dataset Problema1.

Assim executando testes gerais mais precisos.

8.3.1 Testes Gerais

```
Runs: 50

Population_size: 50, 100, 150

Max_generations: 125, 175

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

Selection: tournament

Tournament_size: 2, 4, 6

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

Recombination: pmx, ox, cx2

Recombination_probability: 0.6, 0.7, 0.8

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

Mutation: insert, exchange, displacement

Mutation_probability: 0.2, 0.3, 0.4

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

Problem_file: Problema3.txt

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

Statistic: BestIndividual
Statistic: BestAverage
```

Figura 19 - Problema 3 - Configuração para os testes gerais

Population	Max genera	Selection:	Recombina	Recombina	Mutation:	Mutation prob.:	Average:	StdDev:
150	175	Tournament(2)	PMX	0,6	EM	0,4	339,48	##########
150	175	Tournament(2)	PMX	0,6	EM	0,4	339,48	##########
150	175	Tournament(2)	PMX	0,6	EM	0,4	339,48	##########
100	175	Tournament(2)	PMX	0,6	EM	0,4	340,2	##########
100	175	Tournament(2)	PMX	0,6	EM	0,4	340,2	##########
100	175	Tournament(2)	PMX	0,6	EM	0,4	340,2	##########
150	175	Tournament(2)	PMX	0,6	EM	0,3	341,32	##########
150	175	Tournament(2)	PMX	0,6	EM	0,3	341,32	##########
150	175	Tournament(2)	PMX	0,6	EM	0,3	341,32	##########
150	175	Tournament(2)	PMX	0,6	Insert	0,3	341,52	##########

Tabela 8 - Problema 3 - Testes Gerais

A melhor combinação acabou com uma média de 339,48 e seguimos com ela para os testes do tamanho da população.

8.3.2 Testes ao tamanho da população e número de gerações

Variámos a população e as gerações em:

Population_size: 50, 100, 150, 200, 400

Max_generations: 100, 125, 150, 175

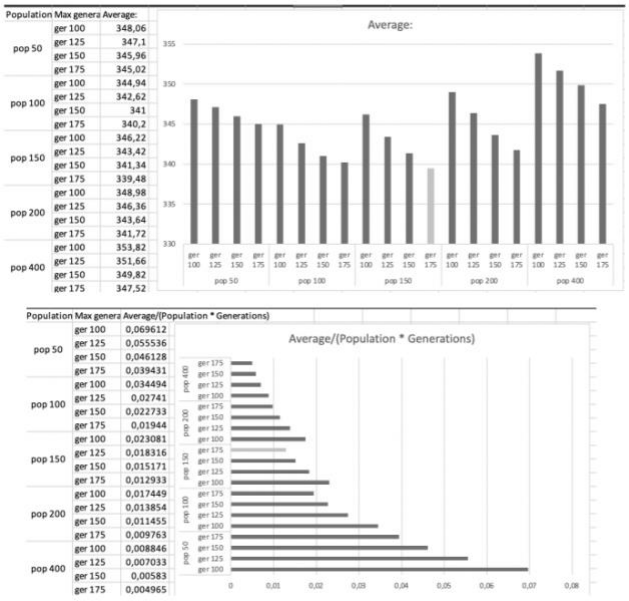

Gráfico 12 - Problema 3 - Testes ao tamanho da população e ao número de gerações

Efetuamos os cálculos de Media sobre a População multiplicado pelas Gerações para sabermos a relação direta entre a média do fitness e (Population*Generations)

A melhor combinação em relação ao cálculo de Average/(Popupation*Generations) é de pop 50 e ger 100. Mas a média aumenta de 339,48 para 348,06 por isso vamos optar pela combinação de pop 150 e ger 175.

8.3.3 *Testes ao tamanho do torneio*

Varámos o torneio em:

Tournament_size: 2, 4, 6, 8, 10

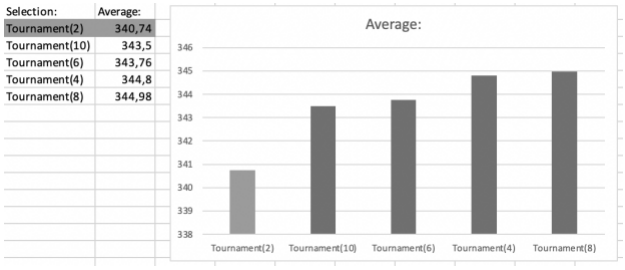

Gráfico 13 - Problema 3 - Testes ao tamanho do torneio

Os melhores resultados vieram do torneio (2)

Prosseguindo com o melhor resultado para o próximo teste

8.3.4 *Testes à recombinação*

Relativamente à recombinação tentámos alargar a nossa visão relativa às probabilidades, já que os testes perliminares não tinham sido conclusivos.

- *Recombination*: pmx, ox, cx2
- *Recombination_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

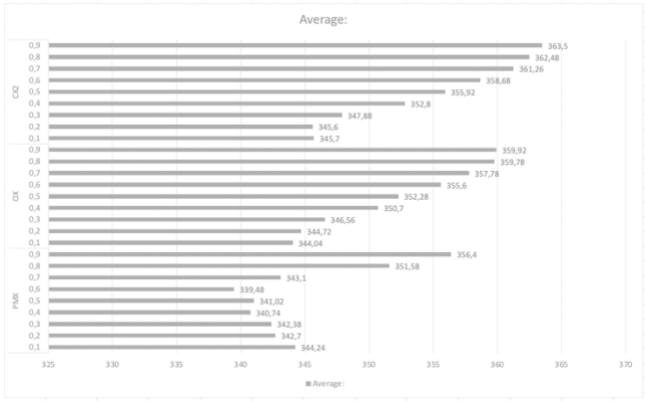

Gráfico 14 - Problema 3 - Testes à recombinação

Prosseguindo então com o melhor resultado dos testes anteriores para os próximos testes

Recombinação	Rec.Prob	Average
PMX	0,6	339,48

8.3.5 *Testes à mutação*

No desenvolvimento do projeto, como extra, criámos algumas mutações extra, então nestes testes fizemos variar as mesmas com probabilidades de mutação diferentes:

- *Mutation*: *insert*, *exchange*, *displacement*, *simpleinversion*, *invertionsimple*, *inversiondisplacement*
- *Mutation_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

Pudemos então afirmar que a mutação que mais se destacou por ter um valor de fitness mais baixo foi a 'EM' com uma probabilidade de mutação de 0,4.

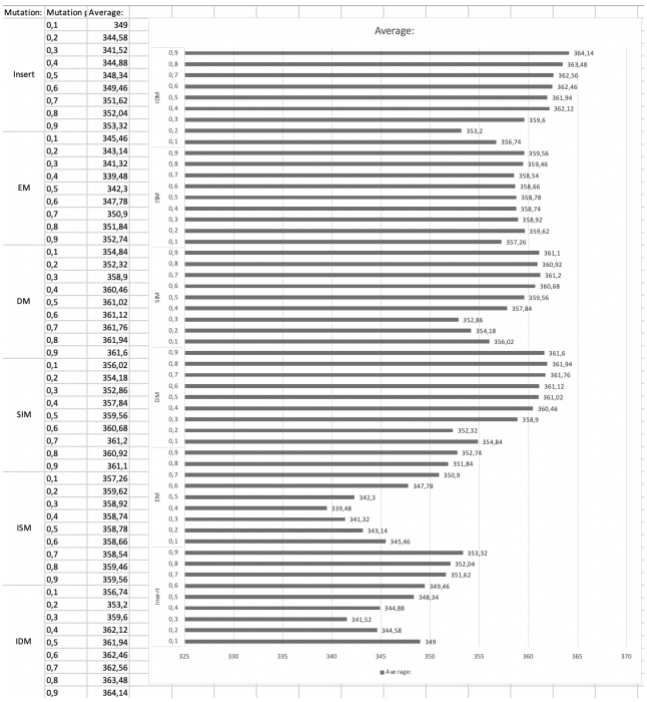

Gráfico 15 - Problema 3 - Testes à mutação

Assim, depois destes testes assumimos que os melhores parâmetros são:

Population size:	Max generations:	Selection:	Recombination:	Recombination prob.:	Mutation:	Mutation prob.:	Average:
150	175	Tournament(2)	PMX	0,6	EM	0,4	339,48

Figura 20 - Problema 3 - Melhores resultados finais

8.4 Testes ao Problema4

Para o dataset Problema3 começámos baseados nos testes preliminares feitos ao dataset Problema1.

Assim executando testes gerais mais precisos.

8.4.1 Testes Gerais

```
Runs: 50

Population_size: 50, 100, 150

Max_generations: 150, 200

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

Selection: tournament

Tournament_size: 2, 4, 6

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

Recombination: pmx, ox, cx2

Recombination_probability: 0.6, 0.7, 0.8

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

Mutation: insert, exchange, simpleinversion

Mutation_probability: 0.7, 0.8, 0.9

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

Problem_file: Problema4.txt

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

Statistic: BestIndividual
Statistic: BestAverage
```

Figura 21 - Problema 4 - Configuração para os testes gerais

Tabela 9 - Problema 4 - Testes gerais

Population si	Max generat	Selection:	Recombinati	Recombinati	Mutation:	Mutation prob,:	Average:	StdDev:
150	200	Tournament(4)	PMX	0,8	EM	0,9	243,02	4,53206355
100	200	Tournament(4)	PMX	0,8	EM	0,9	243,56	2,94047615
150	200	Tournament(4)	PMX	0,6	EM	0,9	243,56	3,25674684
150	150	Tournament(4)	PMX	0,8	EM	0,9	243,76	5,01421978
150	200	Tournament(6)	PMX	0,8	EM	0,9	244,12	4,72288048
150	150	Tournament(4)	PMX	0,6	EM	0,9	244,26	3,3033922
100	150	Tournament(4)	PMX	0,8	EM	0,9	244,3	3,28785644
150	200	Tournament(6)	PMX	0,7	EM	0,9	244,3	4,75920161
150	150	Tournament(6)	PMX	0,8	EM	0,9	244,5	4,93659802
150	200	Tournament(4)	PMX	0,7	EM	0,9	244,5	4,22018957

A melhor combinação acabou com uma média de 243,02 e seguimos com ela para os testes do tamanho da população.

8.4.2 Testes ao tamanho da população e número de gerações

Variámos a população e as gerações em:

- *Population_size*: 50, 100, 150, 200, 400
- *Max_generations*: 100, 125, 150, 175, 200

Efetuamos os cálculos de Media sobre a População multiplicado pelas Gerações para sabermos a relação direta entre a média do fitness e (Population*Generations)

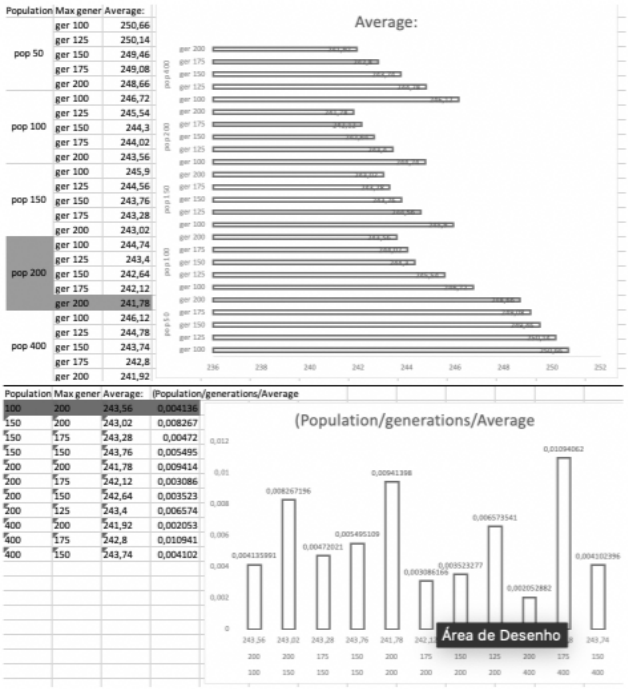

Gráfico 16 - Problema 4 - Testes ao tamanho da população e ao número de gerações

A melhor combinação em relação ao cálculo de Average/(Popupation*Generations) é de pop 100 e ger 200. E a média aumenta de 241,78 para 243,56 o que não é muito por isso vamos optar pela combinação de pop 100 e ger 200.

8.4.3 Testes ao tamanho do torneio

Tournament_size: 2, 4, 6, 8, 10

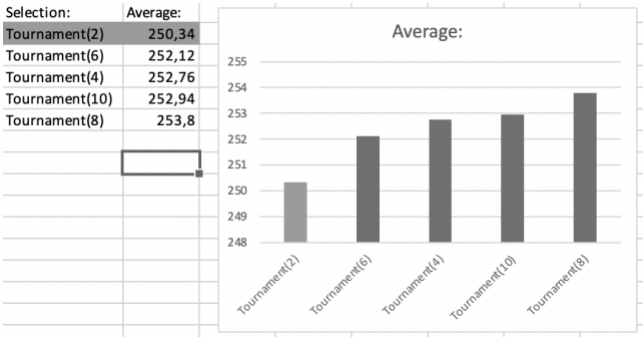

Gráfico 17 - Problema 4 - Testes ao tamanho do torneio

Os melhores resultados vieram do torneio (2)

Prosseguindo com o melhor resultado para o próximo teste

8.4.4 Testes à recombinação

Relativamente à recombinação tentámos alargar a nossa visão relativa às probabilidades, já que os testes preliminares não tinham sido conclusivos.

- *Recombination*: *pmx, ox, cx2*
- *Recombination_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

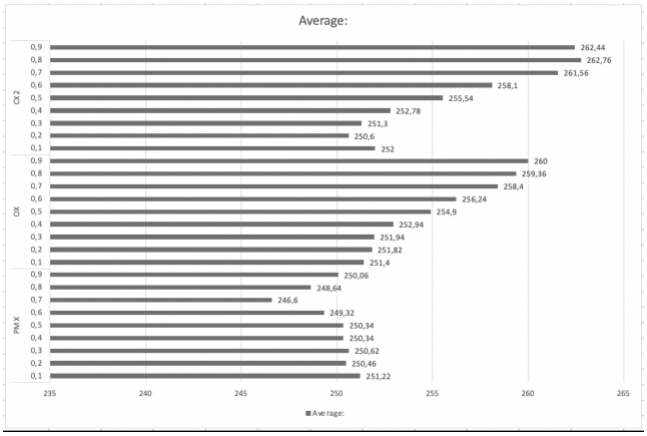

Gráfico 19 - Problema 4 - Teste à recombinação

Prosseguindo então com o melhor resultado dos testes anteriores para os próximos testes

Rec.	Rec.Prob	Average
PMX	0,7	246,6

8.4.5 Testes à mutação

No desenvolvimento do projeto, como extra, criámos algumas mutações extra, então nestes testes fizemos variar as mesmas com probabilidades de mutação diferentes:

- *Mutation*: *insert, exchange, displacement, simpleinversion, invertionsimple, inversiondisplacement*
- *Mutation_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

Pudemos então afirmar que a mutação que mais se destacou por ter um valor de fitness mais baixo foi a 'EM' com uma probabilidade de mutação de 0,4.

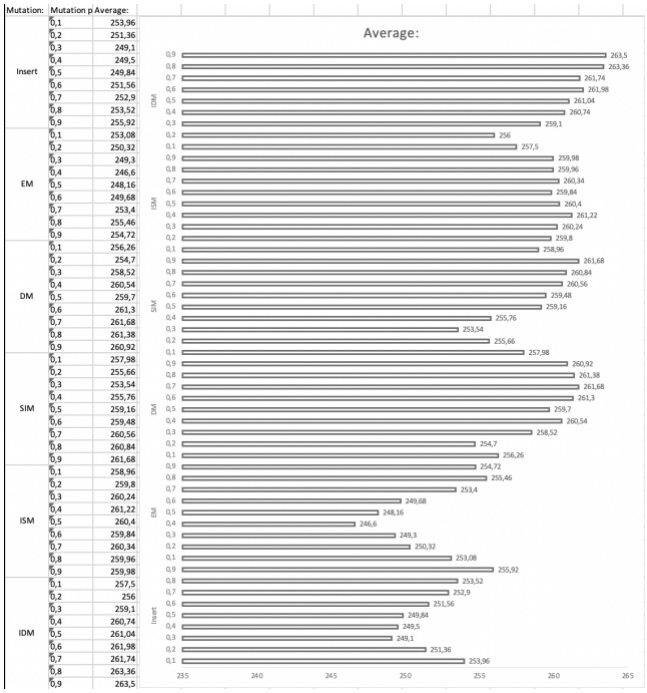

Gráfico 18 - Problema 4 - Testes à mutação

Assim, depois destes testes assumimos que os melhores parâmetros são:

Population size:	Max generations:	Selection:	Recombination:	Recombination prob.:	Mutation:	Mutation prob.:	Average:
100	200	Tournament(2)	PMX	0,7	EM	0,4	246,6

Figura 22 - Problema 4 - Melhores resultados finais

8.5 Testes ao Problema5

8.5.1 Testes Gerais

```
Runs: 50

Population_size: 50, 100, 150

Max_generations: 50, 75

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

Selection: tournament

Tournament_size: 2, 4, 6

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

Recombination: pmx, ox, cx2

Recombination_probability: 0.6, 0.7, 0.8

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

Mutation: insert, exchange, displacement

Mutation_probability: 0.2, 0.3, 0.4

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

Problem_file: Problema5.txt

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

Statistic: BestIndividual
Statistic: BestAverage
```

Figura 23 - Problema 5 - Configuração para os testes gerais

Tabela 10 - Problema 5 - Testes gerais

Population	Max genera	Selection:	Recombina	Recombina	Mutation:	Mutation prob,:	Average:	StdDev:
150	75	Tournament(6)	PMX	0,8	EM	0,4	1056,44	7,743798
150	75	Tournament(6)	PMX	0,6	EM	0,4	1059,5	7,958015
150	75	Tournament(4)	PMX	0,8	Insert	0,4	1059,64	7,659661
150	75	Tournament(4)	PMX	0,8	EM	0,4	1059,66	8,853496
150	75	Tournament(4)	PMX	0,7	EM	0,4	1060,42	7,472858
150	75	Tournament(6)	PMX	0,7	EM	0,3	1060,56	7,459651
150	50	Tournament(6)	PMX	0,8	EM	0,4	1060,94	6,842251
150	75	Tournament(4)	PMX	0,6	EM	0,4	1061,24	7,522127
150	75	Tournament(4)	PMX	0,7	EM	0,3	1061,94	6,900464
150	75	Tournament(4)	PMX	0,7	Insert	0,4	1062,26	7,604762

A melhor combinação acabou com uma média de 1056,44 e seguimos com ela para os testes do tamanho da população.

8.5.2 Testes ao tamanho da população e número de gerações

Variámos a população e as gerações em:

- *Population_size*: 50, 100, 150, 200, 400
- *Max_generations*: 75, 125, 150, 175, 200

Efetuamos os cálculos de Média sobre a População multiplicado pelas Gerações para sabermos a relação direta entre a média do fitness e (Population*Generations)

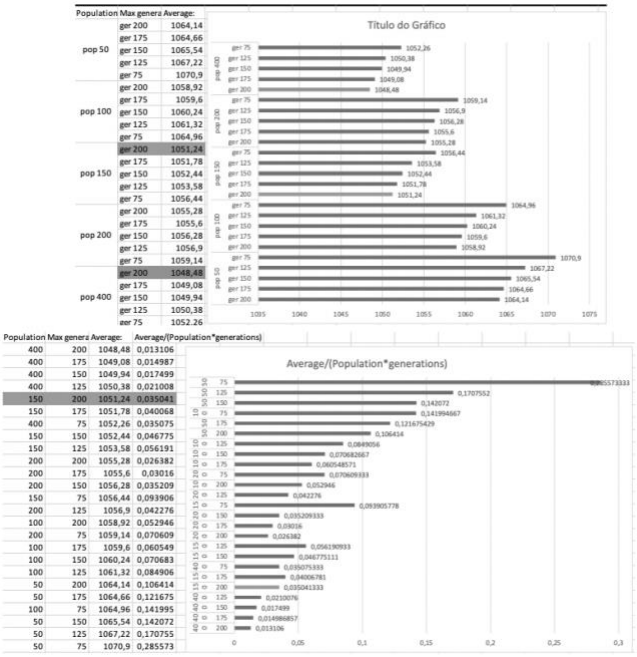

Gráfico 20 - Problema 5 - Testes ao tamanho da população e número de gerações

A melhor combinação em relação ao cálculo de Average/(Popupation*Generations) é de pop 150 e ger 200. E a média aumenta de 1048,48 para 1051,24 o que não é muito por isso vamos optar pela combinação de pop 150 e ger 200.

8.5.3 Testes ao tamanho do torneio

Tournament_size: 2, 4, 6, 8, 10

Selection:	Average:
Tournament(6)	1051,24
Tournament(4)	1055
Tournament(8)	1055
Tournament(10)	1055,52
Tournament(2)	1060,7

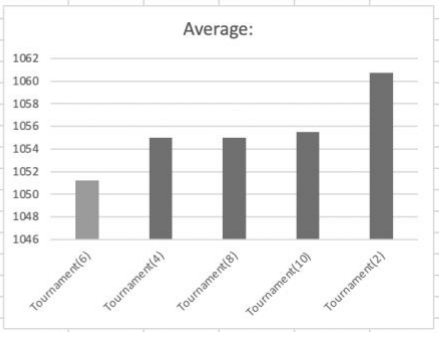

Gráfico 21 - Problema 5 - Testes ao tamanho do torneio

Os melhores resultados vieram do torneio (6)

Prosseguindo com o melhor resultado para o próximo teste

8.5.4 Testes à recombinação

Relativamente à recombinação tentámos alargar a nossa visão relativa às probabilidades, já que os testes perliminares não tinham sido conclusivos.

- *Recombination*: *pmx*, *ox*, *cx2*
- *Recombination_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

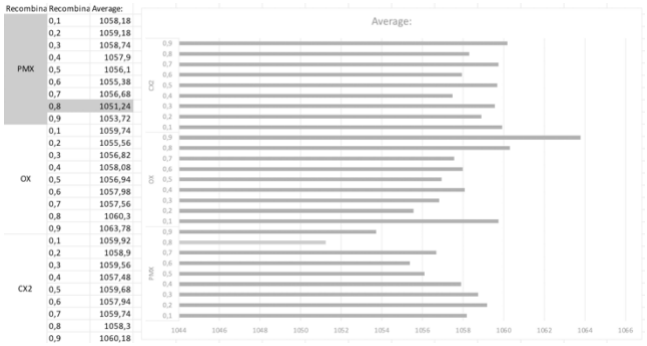

Gráfico 22 - Problema 5 - Testes à recombinação

Rec.	Rec.Prob	Average
PMX	0,8	1051,24

8.5.5 Testes à mutação

No desenvolvimento do projeto, como extra, criámos algumas mutações extra, então nestes testes fizemos variar as mesmas com probabilidades de mutação diferentes:

- *Mutation*: *insert*, *exchange*, *displacement*, *simpleinversion*, *invertionsimple*, *inversiondisplacement*
- *Mutation_probability*: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

Pudemos então afirmar que a mutação que mais se destacou por ter um valor de fitness mais baixo foi a 'EM' com uma probabilidade de mutação de 0,8.

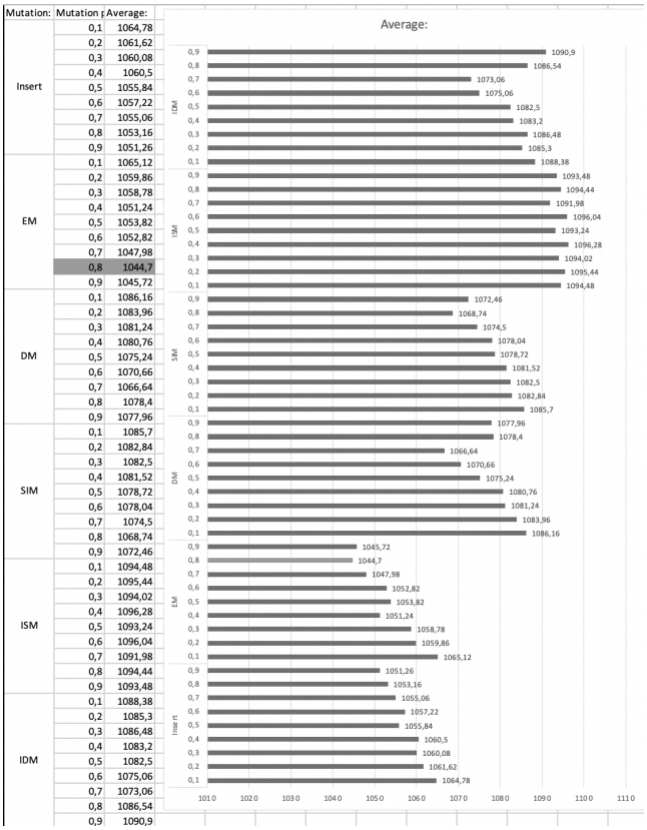

Gráfico 23 - Problema 5 - Testes à mutação

Assim, depois destes testes assumimos que os melhores parâmetros são:

Population size:	Max generations:	Selection:	Recombination:	Recombination prob.:	Mutation:	Mutation prob.:	Average:
150	200	Tournament(6)	PMX	0.8	EM	0.8	1044,7

Gráfico 24 - Problema 5 - Melhores resultados finais

9. Outros Aspetos Relevantes

9.1 Extras

Como extras sugeridos, pelos docentes:

- Implementar uma versão do problema em que as peças possam ser rodadas;
- Implementar um mecanismo de elitismo no algoritmo genético;
- Implementar uma estrutura gráfica do material usado;

9.1.1 Implementar uma versão do problema em que as peças possam ser rodadas

Como já referenciado no capítulo 2, a inserção das peças no material não sofria qualquer rotação. E o facto de pudermos rodarmos uma peça pode facilitar muito o encaixe da mesma. Assim iremos explicar como implementámos esta funcionalidade.

Para guardarmos a rotação de uma peça, essa mesma rotação teria de estar diretamente ligada à própria peça, assim o método escolhido para que a rotação acompanhasse sempre a sua peça foi acrescentarmos essa informação no *genome*.

$$NRot\_Peça\_n = [0;3] \in N \text{ com,}$$

$$0º => 0$$

$$90º => 1$$

$$180º => 2$$

$$270º => 3$$

$$P1 = (NPeça1, NRot\_Peça1, NPeça2, NRot\_Peça2, NPeça\_n, NRot\_Peça\_n)$$

Assim, todos operadores tiveram de ser alterados de maneira que trabalhassem de duas em duas posições e movessem sempre o gene de rotação atrás do mesmo do gene com o número da peça.

Para além do já falado a última coisa que precisou de ajustes a construção da matriz (*materialCut*) a qual precisou de rodar as peças (recorrendo ao método *rotateMatrixBy90Degrees*) antes de ser verificado o posicionamento (com o método chackValidPlacement) e a inserção da mesma na *materialCut.*

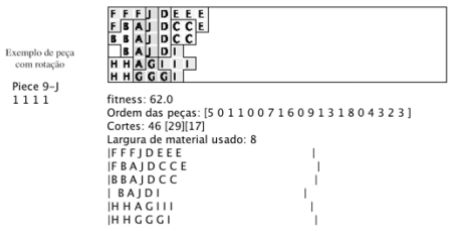

Figura 24 - Representação de rotação de uma peça

9.1.2 Implementar um mecanismo de elitismo no algoritmo genético

Um mecanismo de elitismo visa guardar o melhor individuo de uma geração para a outra.

Para implementar este mecanismo na classe "*GeneticAlgorithm*" dentro do método "*run*" substituímos um individuo aleatório da nova geração pelo melhor da geração anterior.

9.1.3 Implementar uma estrutura gráfica do material usado

Com o objetivo de melhorar a perceção do material e da disposição das peças no mesmo, criámos uma secção na parte gráfica para despormos o nosso material. Diferença entre peças, cortes, e a relação com o tamanho máximo do material.

Essa expressa todo o processo de desenvolvimento da matriz de peças (*matrizCut*).

Figura 25 - Representação de estrutura gráfica do material

9.1.4 Implementar um temporizador de execução

Com o objetivo de contabilizar e comparar o tempo de execução entre execuções implementámos um temporizador. Esta implementação passa por criar uma variável que marca o tempo em que a execução tem início e outra no fim da execução. Subtraem-se resultando o tempo de execução.

$$Temp\_Exec = Temp\_Final - Temp\_Inicial$$

10. Referências

[1] https://www.hindawi.com/journals/cin/2017/7430125/ - Operadores Recombinação

[2] https://core.ac.uk/download/pdf/30364387.pdf - Operadores

[3] https://www.redalyc.org/pdf/2652/265219635002.pdf - Operação

[4] Conteúdo teórico e prático lecionado e exposto pela UC de Inteligência Artificial.

[5] Breves conteúdos da UC de Álgebra. – Relacionados a matriz.

[6] Google Imagens