PROJETO IA 2022/23

Francisco Rosa
2201756
2201756@my.ipleiria.pt

Mariana Pereira
2200679
2200679@my.ipleiria.pt

1. INTRODUÇÃO

O objetivo deste projeto é o desenvolvimento de uma aplicação que permita otimizar a recolha de produtos das prateleiras de um armazém, um processo conhecido por *picking*. O objetivo do sistema consiste em distribuir os *picks* (produtos) pelos agentes responsáveis pela recolha, bem como definir a ordem pela qual cada agente deve recolher os produtos que lhe foram atribuídos de modo a minimizar o tempo de entrega do último produto a ser entregue no ponto de entrega. Pretende-se também minimizar a distância total percorrida pelos agentes e o número de colisões entre os agentes.

2. EXPLICAÇÃO DE CÓDIGO

2.1 Search

Começámos por implementar no *warehouse_state* os *move*'s e os *can_move*'s para o agente conseguir saber em que posições se conseguia mover, ou seja: se houver estiver no final da matriz, ou se existir um obstáculo como uma prateleira ou um produto, os *can_move*'s irão retornar falso.

```
def can_move_up(self) -> bool:
    return self.line_forklift > 0 and \
        self.matrix[self.line_forklift - 1][self.column_forklift] != constants.SHELF and \
        self.matrix[self.line_forklift - 1][self.column_forklift] != constants.PRODUCT
```

Foram realizadas algumas funções auxiliares que poderiam ser úteis no futuro, tais como se uma célula tem produto ou se está vazia.

```
def cell_has_product(self, x, y):
    return self.matrix[x][y] == constants.PRODUCT

def cell_is_empty(self, x, y):
    return self.matrix[x][y] == constants.EMPTY
```

Depois tivemos que implementar o *is_goal* para o agente saber se a célula em que está é a célula objetivo que lhe indica que o seu percurso terminou.

```
def is_goal(self, state: WarehouseState) -> bool:
    return state.line_forklift == self.goal_position.line and state.column_forklift == self.goal_position.column
```

Após isso, tivemos que pensar numa heurística válida e eficaz para o algoritmos de procura A*, para que o agente conseguisse achar da melhor forma o caminho mais curto da sua posição a um produto ou a uma saída, optámos por implementar a heurística da táxi-distância (*Manhattan Distance*), que envolve calcular a soma das diferêncças absolutas entre a posição do agente e da saída.

```
def compute(self, state: WarehouseState) -> float:
    goal_position = self.problem.goal_position
    manhattan_distance = abs(state.line_forklift - goal_position.line) + abs(state.column_forklift - goal_position.column)
    return manhattan_distance
```

Finalmente chegou o nosso primeiro desafio, calcular as distâncias dos pares. O método *run* itera sobre todos os pares do agente, calculando a distância entre cada par. Inicialmente, o estado é definido de acordo com o primeiro elemento do par. Se o primeiro elemento é um produto, o agente é posicionado ao lado dele, de acordo com a disponibilidade na matriz do ambiente inicial.

Em seguida, o objetivo é definido com base no segundo elemento do par. Novamente, se o objetivo é um produto, a sua posição é ajustada para o lado adjacente disponível na matriz do ambiente inicial.

Então, um problema do tipo *WarehouseProblemSearch* é definido usando o estado atualizado e a posição do objetivo, e o agente resolve o problema. A solução retorna o custo, que é a distância entre os dois elementos do par. Esse custo é então associado ao par correspondente.

Além disso, a solução é utilizada para atualizar o dicionário de distâncias e o dicionário de caminhos do agente, utilizando as coordenadas das células como chave. O custo calculado é utilizado para definir o valor associado a cada par, que corresponde à distância entre os dois elementos do par.

No final, o caminho encontrado para alcançar a solução é armazenado para cada par. Todo este processo é repetido para cada par. Por fim, os resultados são exibidos na interface gráfica.

2.2 Genoma e Indivíduo

Quando avançámos para a parte do algoritmo genético, começámos por ter em mente que este problema que nos foi proposto é um problema de permutação, diferente do que foi resolvido nas aulas, por isso tivemos que ter alguns detalhes em mente, nomeadamente saber como representar os agentes no genoma e quais os produtos que lhes são atribuídos.

Chegámos à conclusão que a melhor estratégia era incluir N-1 agentes no genoma, definidos como inteiros sequenciais maiores que o maior produto, em que o primeiro agente apanhava por ordem os produtos no genoma, até existir um gene agente, atribuindo assim os próximos produtos a esse agente até que outro gene agente aparecesse ou que o genoma terminasse.

Cria-se o genoma da seguinte maneira:

```
# Vamos criar uma lista vazia, que vai ser o genoma
self.genome = []

# Vamos adicionar os inteiros de 1 a num_genes ao genoma
for gene_index in range(1, num_genes + 1):
    self.genome.append(gene_index)

# Vamos misturar os inteiros do genoma, usando o random da classe GeneticAlgorithm
GeneticAlgorithm.rand.shuffle(self.genome)
```

E os indivíduos desta maneira, dependendo do número de agentes no nível:

```
def generate_individual(self) -> "WarehouseIndividual":
    if self.forklifts == 1:
        new_individual = WarehouseIndividual(self, len(self.products))
    else:
        new_individual = WarehouseIndividual(self, len(self.products) + len(self.forklifts) - 1)

    return new_individual
```

2.3 Compute Fitness

A função *compute_fitness* é responsável por calcular o "fitness", ou aptidão, de um indivíduo. Essa aptidão é representada pela soma dos custos de todas as ações realizadas pelo agente de acordo com a sequência definida pelo genoma do indivíduo.

Na solução do algoritmo genético, buscamos nos pares (*cell1*, *cell2*) aonde *cell1* representa o agente e *cell2* é o inteiro do genoma. Para um indivíduo com genoma 1-2-3, por exemplo, o cálculo seria feito da seguinte forma: calcula-se a distância do agente ao produto 1, do produto 1 ao 2, do produto 2 ao 3 e, finalmente, do produto 3 à saída. A soma dessas distâncias constitui o fitness do indivíduo.

Nesta função, também foram definidas duas variáveis, *pesoDistTotal* e *pesoDistMax*, que dão diferentes pesos à distância total percorrida por todos os agentes e à distância máxima percorrida por um único agente, respetivamente. No final, o "fitness" é calculado somando o produto da distância total pelo seu peso com o produto da distância máxima pelo seu peso.

É importante notar que, ao longo do processo, pode ser que o agente mude a sua posição para ir buscar outro produto, ou que não exista um produto a ser buscado. Nesses casos, a distância a ser considerada é a distância do agente até a saída.

Adicionalmente, a função verifica se a distância obtida é infinita. Caso seja, o código tenta obter a distância inversa, ou seja, de cell2 para cell1. Isso ocorre pois a distância de cell1 para cell2 pode não estar definida no dicionário de distâncias, mas a de cell2 para cell1 sim. No final, a função retorna o "fitness" do indivíduo, que é um valor representativo da sua aptidão. Quanto menor o "fitness", melhor o indivíduo, pois significa que ele cumpre o seu objetivo com um custo menor.

2.4 Obtain All Path

A função *obtain_all_path* tem o objetivo de construir o caminho que cada agente vai percorrer, dada a sequência definida no genoma do indivíduo. Esses caminhos são necessários para a parte gráfica do sistema, que vai ilustrar o percurso de cada agente.

Os caminhos de cada agente são armazenados na variável forklift_paths, que é uma lista onde cada elemento corresponde ao caminho percorrido por um agente. Além disso, a função calcula max_steps, que é o número máximo de passos que um agente pode dar.

Para cada valor no genoma, a função verifica se o valor corresponde a um agente ou a um produto. Se for um agente, o caminho a ser considerado será do último produto visitado à saída. Se não houver um último produto visitado, o caminho será do agente à saída. Esses caminhos são obtidos a partir do dicionário paths que é uma propriedade de agent_search, e são adicionados à variável path, que contém o caminho do agente atual.

Se o valor do genoma corresponder a um produto, o caminho será do agente (ou do último produto visitado) até o produto atual. Caso o caminho não esteja definido no dicionário *paths* (ou seja, se a distância for infinita), o código tenta obter o caminho inverso, de cell2 para cell1, e reverte o caminho obtido. Isso acontece pois a distância de cell1 para cell2 pode não estar definida no dicionário de distâncias, mas a de cell2 para cell1 sim. O caminho obtido também é adicionado à variável path.

Depois de passar por todos os valores do genoma, a função verifica se ainda há um último produto visitado. Se houver, ela obtém o caminho do último produto até a saída e adiciona à variável path. Caso contrário, ela obtém o caminho do agente até a saída. Finalmente, o caminho do agente é adicionado à lista de caminhos forklift_paths e a função retorna esta lista, junto com o número máximo de passos.

Ambos os dicionários foram instanciados no ficheiro *warehouse_agent_search.py*.

```
self.distances = {} # dicionário de distâncias entre pares de células
self.paths = {} # dicionário de caminhos
```

2.5 Mutações e Recombinações

Para além das mutações e recombinações já implementadas no código base, criámos mais duas de cada, após uma pesquisa breve acerca de operadores genéticos para problemas de permutação:

- **Mutação *Swap***
- **Mutação *Insert*** (inicialmente ***Inversion***, até saber que esse afinal era o que estava já implementado)
- **Recombinação *Cycle Crossover***
- **Recombinação *Order Crossover***

Vamos à explicação breve de cada uma das implementadas.

Mutação por Inserção (*Insert Mutation*): Esta operação de mutação é caracterizada pela remoção de um gene (elemento) do genoma (lista de genes de um indivíduo) em uma posição aleatória e inserindo-o novamente em uma posição diferente. O gene a ser movido e a nova posição de inserção são escolhidos aleatoriamente. Isso pode resultar em uma variação significativa no indivíduo, dependendo da posição do gene que é removido e onde é reinserido.

Mutação por Troca (*Swap Mutation*): Este tipo de operação de mutação envolve a seleção de dois genes em posições aleatórias no genoma de um indivíduo e trocando suas posições. Esta é uma forma de mutação menos disruptiva do que a mutação por inserção, uma vez que não muda a ordem relativa dos outros genes no genoma. A troca de duas posições pode resultar em uma mudança significativa na qualidade da solução se as posições trocadas forem críticas para a solução.

Ambas as classes de mutação herdam da classe abstrata *Mutation*, o que garante que ambas implementem o método *mutate*, responsável por executar a operação de mutação em um indivíduo. A probabilidade de mutação é passada para a classe pai através do construtor.

Recombinação por Ordem (*Order Crossover* - OX): Este operador de recombinação começa por selecionar um segmento de genes de um dos pais. Em seguida, preenche o resto do genoma com os genes do outro progenitor que ainda não estão presentes, mantendo a sua ordem original. O mesmo processo é então realizado para o segundo descendente, mas desta vez começando com o segundo progenitor. Esta recombinação é eficaz para manter sequências de genes que trabalham bem juntas.

Recombinação por Ciclo (*Cycle Crossover* - CX): Este operador de recombinação inicia identificando "ciclos" de genes que são trocados entre os dois pais. Em seguida, os ciclos são alternadamente copiados de cada pai para criar a descendência. Isto garante que cada gene aparece exatamente uma vez em cada descendente, tornando-o útil para problemas onde os genes não podem ser repetidos, como o problema do caixeiro-viajante.

Ambas as classes de recombinação herdam da classe abstrata *Recombination*, o que garante que ambas implementem o método recombine, responsável por executar a operação de recombinação em dois indivíduos. A probabilidade de recombinação é passada para a classe pai através do construtor.

3. EXPERIÊNCIAS

Para as experiências, optámos sempre por fazer 30 *Runs*. Estas são as gerações usadas para cada dataset:

Ficheiro	Gerações
problem1.txt	5
problem2.txt	10
problem3.txt	50
problem4.txt	50
problem5.txt	100

E estas são as características da máquina usada para realizar as experiências:

- **Processador:** Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
- **Memória RAM:** 16 GB
- **SO:** Windows 10 Home 22H2

Por falta de tempo, infelizmente não foi possível realizar os testes gerais para o dataset5, por essa razão optámos por usar os melhores parâmetros dos testes gerais do dataset4 para realizar os testes parciais do dataset5.

3.1 Dataset 1

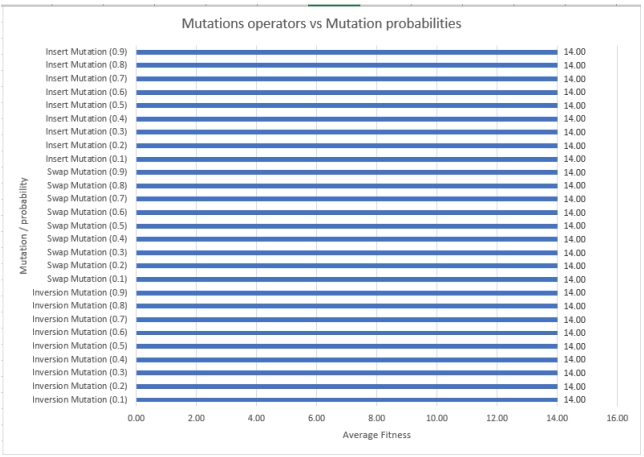
Ficheiro config.txt dos testes gerais:

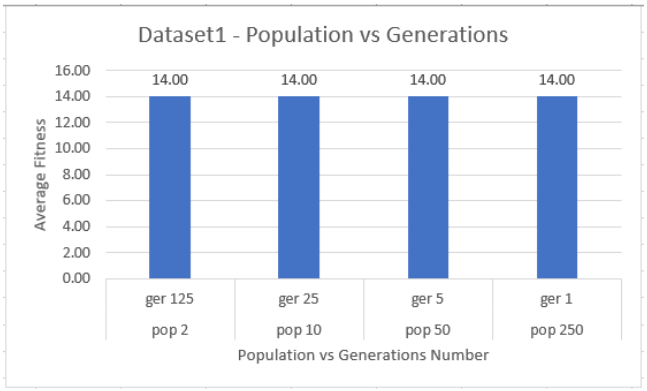
```
Runs: 30

Population_size: 50, 100, 200

Max_generations: 5

# --------------------------------

Selection: tournament

Tournament_size: 2, 4, 6

# --------------------------------

Recombination: pmx, cx, ox

Recombination_probability: 0.7, 0.8, 0.85

# --------------------------------

Mutation: inversion, swap, insert

Mutation_probability: 0.1, 0.2, 0.3

# --------------------------------

Problem_file: ./problem1.txt

# --------------------------------

Statistic: BestIndividual
Statistic: BestAverage
```

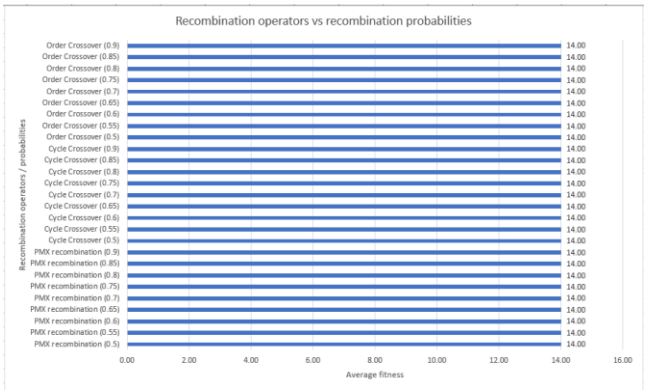
Como este problema é extremamente simples, não deu para obter os melhores parâmetros para o dataset, então decidimos por optar aleatóriamente por uns parâmetros para realizar os testes parciais:

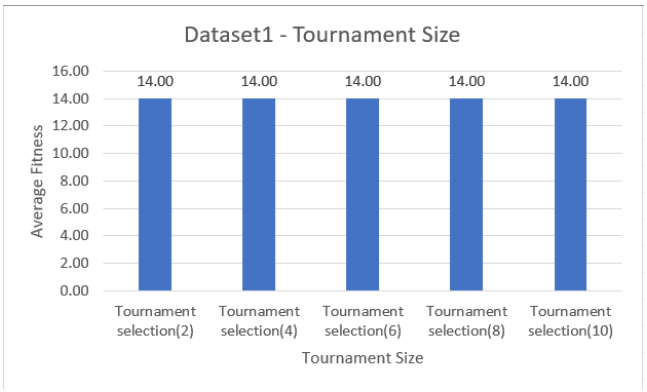
Population size:	Max generations:	Selection:	Recombination:	Mutation:	Average:	StdDev:
50	5	Tournament selection(2)	PMX recombination (0.7)	Inversion Mutation (0.1)	14	0

Tanto o average como o melhor fitness foi sempre o mesmo (14), aqui estão os gráficos respeitantes aos testes parciais:









Com este dataset não conseguimos retirar grandes conclusões por ser demasiado simples. Vamos então avançar para o próximo.

3.2 Dataset 2

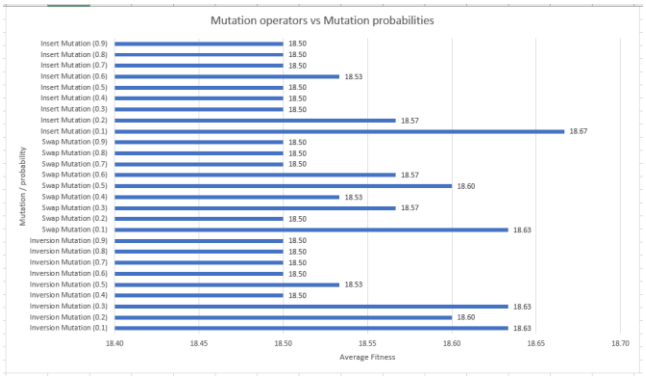
Config.txt para os testes gerais 2:

```
Runs: 30

Population_size: 50, 100, 200

Max_generations: 10

# ---------------------------------

Selection: tournament

Tournament_size: 2, 4, 6

# ---------------------------------

Recombination: pmx, cx, ox

Recombination_probability: 0.7, 0.8, 0.85

# ---------------------------------

Mutation: inversion, swap, insert

Mutation_probability: 0.1, 0.2, 0.3

# ---------------------------------

Problem_file: ./problem2.txt

# ---------------------------------

Statistic: BestIndividual
Statistic: BestAverage
```

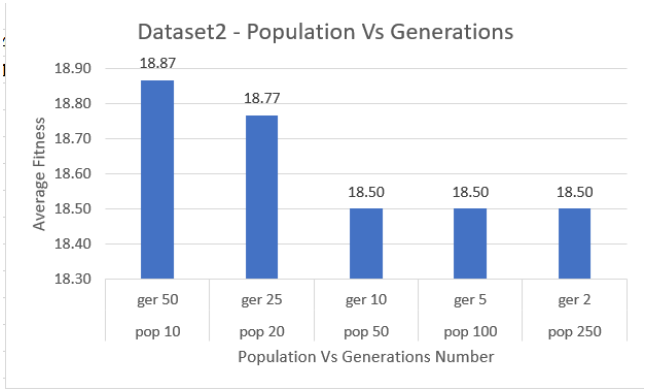
Os testes gerais do Dataset 2 ainda deram alguns parâmetros todos com o mesmo melhor average fitness, optámos por escolher estes:

Population size:	Max generations	Selection:	Recombination:	Mutation:	Average:	StdDev:
50	10	Tournament selection(2)	Order Crossover (0.8)	Swap Mutation (0.2)	18.50	0

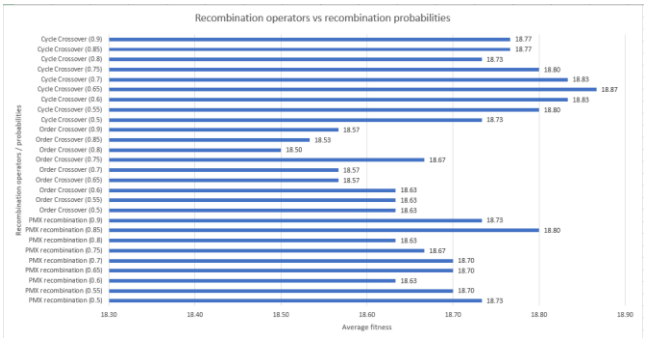
Vamos então analisar os gráficos:



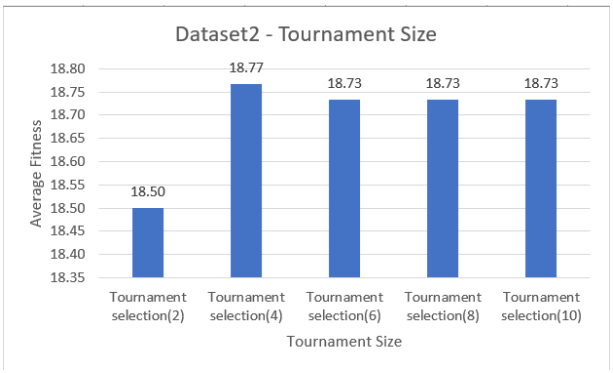
Através deste gráfico analisamos que perante todas as possibilidades de operadores de mutação, o melhor parâmetro são várias possibilidades.



Concluimos neste gráficos que para o Dataset2, várias populações dão o melhor average fitness.



Melhor Recombinação do dataset 2: Order Crossover (0.8)



Melhor Torneio: Tamanho 2

3.3 Dataset 3

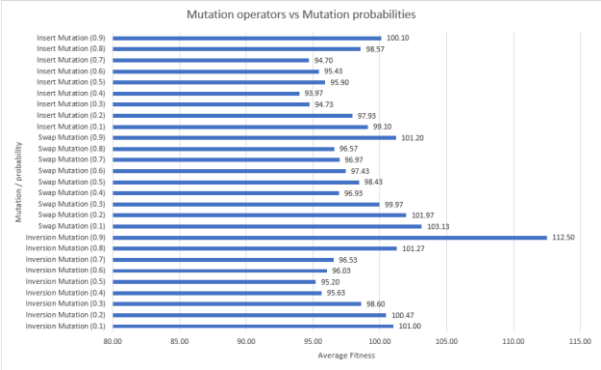
Config.txt do Dataset 3:

```
Runs: 30

Population_size: 50, 100, 200

Max_generations: 50

# ---------------------------------

Selection: tournament

Tournament_size: 2, 4, 6

# ---------------------------------

Recombination: pmx, cx, ox

Recombination_probability: 0.7, 0.8, 0.85

# ---------------------------------

Mutation: inversion, swap, insert

Mutation_probability: 0.1, 0.2, 0.3

# ---------------------------------

Problem_file: ./problem3.txt

# ---------------------------------

Statistic: BestIndividual
Statistic: BestAverage
```

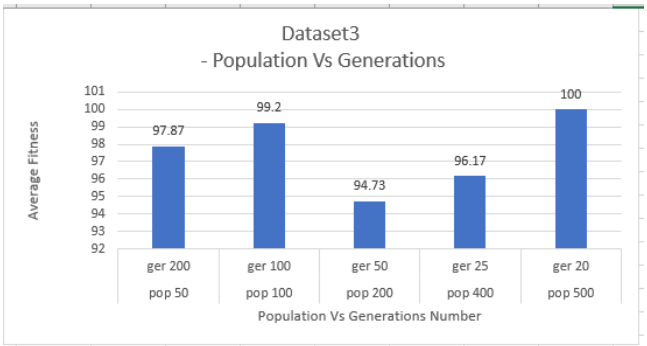
Os melhores parâmetros dos testes gerais para o Dataset 3 são os seguintes:

Population size:	Max generations:	Selection:	Recombination:	Mutation:	Average:
200	50	Tournament selection(6)	PMX recombination (0.85)	Insert Mutation (0.3)	94.73333333

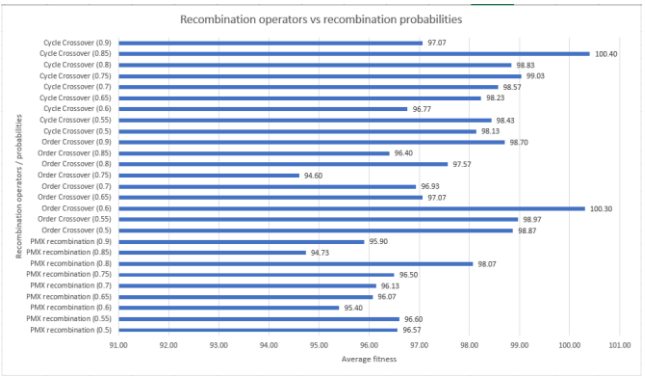
Vamos então analisar os gráficos dos testes parciais com esses parâmetros.



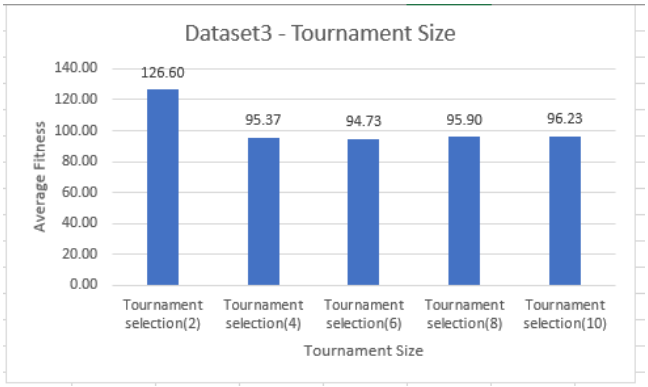
Melhor operador de mutação do dataset 3: Insert Mutation (0.4)



Melhor população do dataset 3: população de 200 com 50 gerações



Melhor recombinação do dataset 3: Order Crossover (0.75)



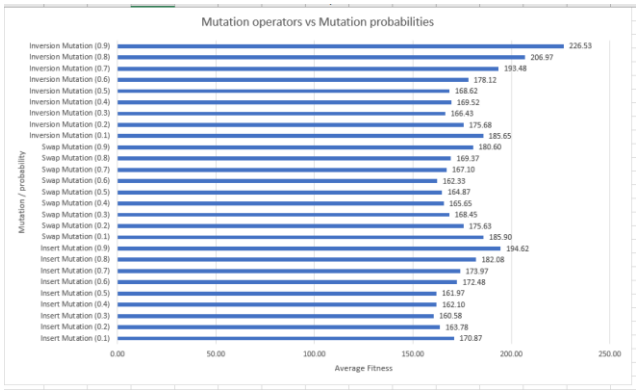
Melhor Torneio do dataset 3: Tamanho 6

3.4 Dataset 4

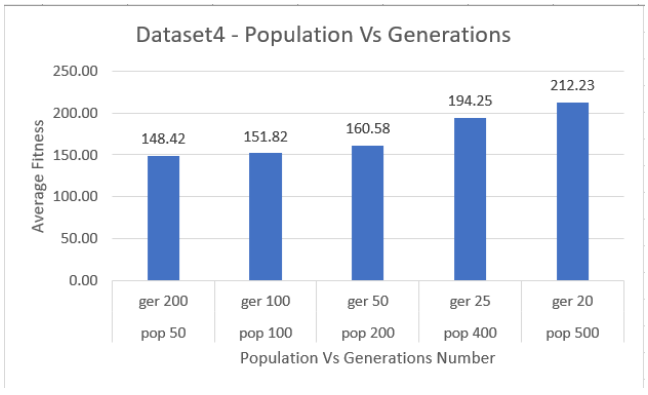
O ficheiro config.txt para os testes gerais é igual ao do dataset3.

Population size:	Max generations:	Selection:	Recombination:	Mutation:	Average:
200	50	Tournament selection(6)	PMX recombination (0.8)	Insert Mutation (0.3)	160.58

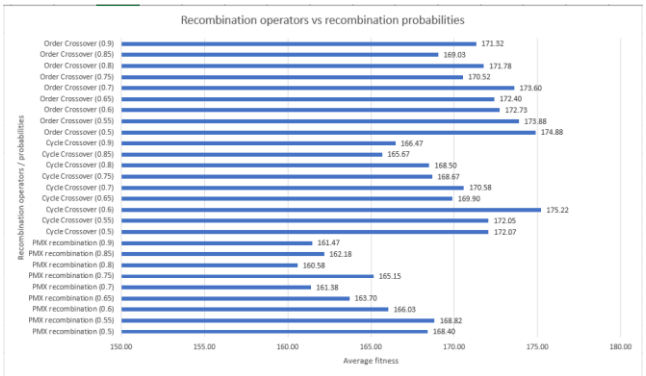
Vamos analisar os gráficos:



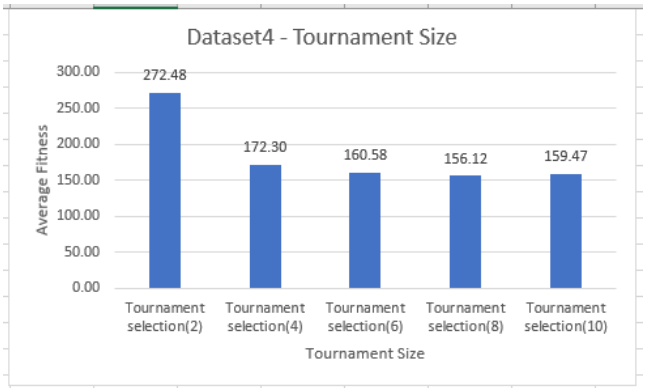
Melhor operador de mutação do dataset 4: Insert Mutation (0.3)



Melhor população do dataset 4: 200 gerações, população de 50



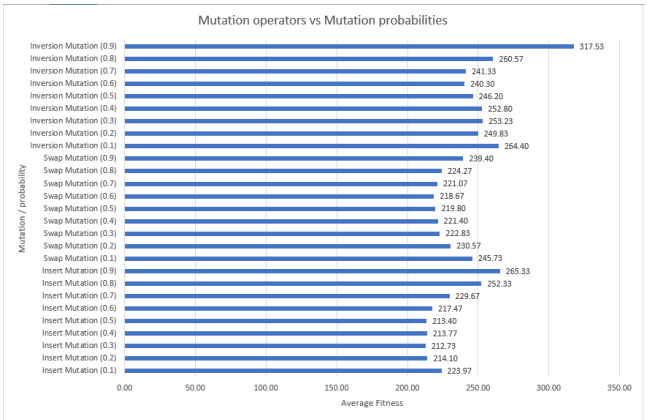
Melhor recombinação do dataset 4: PMX (0.8)



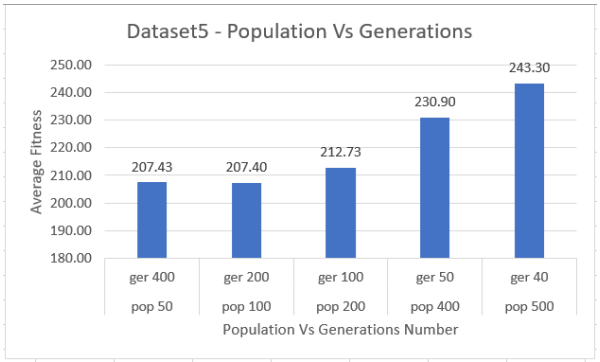
Melhor torneio do dataset 4: Tamanho 8

3.5 Dataset 5

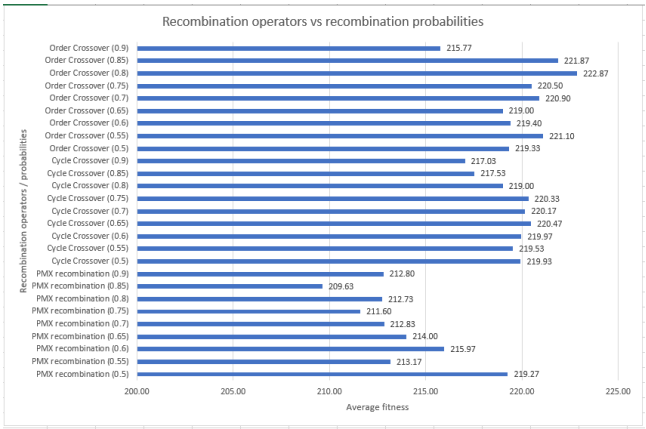
No Dataset 5 foi usado os mesmos parâmetros dos testes gerais do dataset 4, devido à falta de tempo para realizar as experiências gerais para o dataset 5.



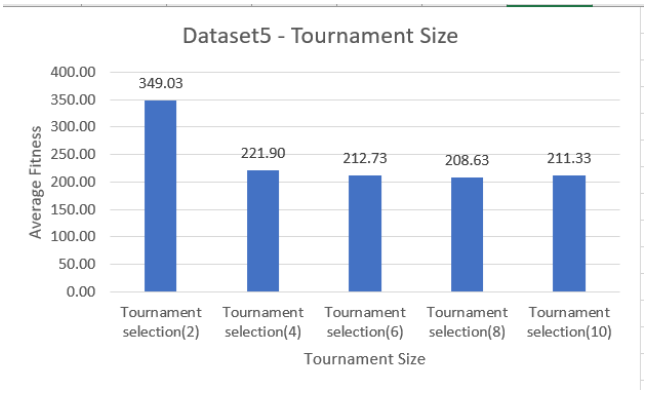
Melhor operador de mutação do dataset 5: Insert Mutation (0.5)



Melhor população do dataset 5: População de 100, com 200 gerações.



Melhor recombinação do dataset 5: PMX (0.85)



Melhor torneio do dataset 5: tamanho 8

4. ASPETOS RELEVANTES

A distribuição de trabalho por ambos os elementos foi justa, tanto a Mariana e o Francisco trabalharam e contribuíram para a implementação e o sucesso do projeto, trabalhando em equipa para ajudar nos problemas que cada um enfrentou ao longo do desenvolvimento. O Francisco, no entanto, ficou responsável pelas tarefas mais difíceis e complexas pois estava mais

confortável com a programação, mas não existiu falta de empenho e iniciativa por parte da Mariana.

Para além das funcionalidades exigidas no enunciado, foram ainda feitos os seguintes extras:

- Mutação Insert (considerámos isto um extra tendo em conta que a mutação no código base era afinal a Inversion)
- Uso de dicionários no programa em vez de tabelas de Hash para melhorar significativamente o desempenho não só do programa como das experiências
- Tentativa de deteção de colisões, infelizmente não foi implementada porque existia um bug que não conseguimos resolver após bastante esforço, decidimos à mesma deixar a função comentada no ficheiro *warehouse_individual.py*.

5. REFERÊNCIAS

[1] Slides e material disponibilizado no moodle

[2] Google, Stackoverflow, ChatGPT, Github Copilot

[3] Genetic Algorithm for permutation problems (PermGA) https://xloptimizer.com/features/genetic-algorithms-ga/genetic-algorithm-for-permutation-problems-permga