Relatório IA

Projeto realizado por:

Miguel Melro, 2212879
Ricardo Crespo, 2211047

1. Introdução

Este relatório descreve o projeto de desenvolvimento de uma aplicação que visa otimizar a recolha de produtos num armazém, conhecido como processo de "picking". O objetivo do sistema é distribuir os produtos entre os agentes responsáveis pela recolha (forklift) e determinar a ordem em que cada agente deve recolher os produtos atribuídos, minimizando o tempo de entrega do último produto no ponto de entrega, com uso de algoritmos genéticos. Além disso, o objetivo é minimizar a distância total percorrida pelos agentes.

2. Descrições

2.1 Descrição do estado

A classe WarehouseState representa o estado do problema relacionado com a recolha de produtos no armazém. O estado é caracterizado pela posição atual do agente (forklift) e por uma referencia à matriz inicial, que descreve a disposição dos elementos no armazém.

Dentro desta classe, são implementados os métodos que verificam quais ações o agente pode tomar. Essas ações estão relacionadas à movimentação do agente nas quatro direções disponíveis: para cima, para baixo, para a esquerda ou para a direita.

Por exemplo, o método 'can_move_up()' verifica se o agente pode mover-se para cima. Isso é determinado verificando se a posição acima do forklift não contém uma prateleira ou um produto. Caso seja possível mover-se para cima, o método retorna 'True'; caso contrário, retorna `False`. Os métodos 'can_move_right()', 'can_move_down()' e 'can_move_left()' seguem uma lógica similar.

Esses métodos de verificação de movimento são fundamentais para controlar as ações permitidas para o agente, garantindo que ele não ultrapasse as restrições do problema.

Dessa forma, a classe 'WarehouseState' encapsula a lógica relacionada ao estado do problema, fornecendo os meios para verificar as ações permitidas e controlar a movimentação do agente dentro do armazém.

2.2 Descrição da heurística

A classe 'HeuristicWarehouse' representa a heurística desenvolvida para ser utilizada em conjunto com o algoritmo A* para resolver o problema de recolha de produtos no armazém.

A heurística implementada é baseada na distância de Manhattan entre a posição atual do forklift (representada por 'state.line_forklift' e 'state.column_forklift') e a posição objetivo (representada por 'self._problem.goal_position.line' e 'self._problem.goal_position.column'). A distância de Manhattan é calculada pela soma das diferenças absolutas entre as coordenadas vertical e horizontal das duas posições.

O método 'compute(self, state: WarehouseState) -> float' é responsável por calcular o valor da heurística para um determinado estado. Ele retorna a distância de Manhattan entre a posição atual do agente e a posição objetivo.

2.3 Descrição da representação dos indivíduos utilizada no algoritmo genético

A classe `WarehouseIndividual` representa os indivíduos utilizados no algoritmo genético para resolver o problema de recolha de produtos no armazém.

Esses indivíduos são representados por um vetor de números inteiros e herdam da classe `IntVectorIndividual`, que, por sua vez, herda da classe abstrata `Individual`. A classe `IntVectorIndividual` fornece uma estrutura básica para indivíduos que possuem um genoma representado por um vetor de números inteiros.

A classe 'WarehouseIndividual' possui os seguintes atributos:

- 'num_steps': Representa o número máximo de passos dados por um dos forklifts no indivíduo.

- 'forklifts_path': Uma lista de listas que armazena os caminhos percorridos por cada empilhadeira (forklift). Cada lista interna representa o caminho percorrido por uma empilhadeira específica.

Os métodos principais da classe são:

- 'compute_fitness()': Calcula o *fitness* do indivíduo. Esse método é responsável por percorrer o genoma do indivíduo e calcular o valor do fitness com base nas células visitadas e produtos coletados durante o percurso.

- 'obtain_all_path()': Retorna uma lista de listas contendo os caminhos percorridos por cada agente (forklift) no indivíduo, juntamente com o número máximo de passos dados por qualquer empilhadeira.

- 'better_than(other: "WarehouseIndividual") -> bool': Compara a aptidão do indivíduo com outro indivíduo fornecido como argumento e retorna `True` se a aptidão do próprio indivíduo for melhor do que a do outro, caso contrário, retorna `False`.

2.4 Descrição da função de avaliação (função de fitness) utilizada

A função ‘compute_fitness’ é responsável por calcular a aptidão de um indivíduo no contexto do problema de recolha de produtos no armazém. Esta função percorre o genoma do indivíduo, que representa a ordem de recolha dos produtos, e realiza várias etapas para determinar a aptidão do indivíduo.

Primeiro, a função inicializa o número de passos dados pelos forklifts e variáveis auxiliares, como j (numero do forklift). Em seguida, o genoma é dividido em subgenomas para cada cada agente, separando os produtos que cada uma deles deve recolher.

De seguida, a função percorre cada subgenoma e analisa os genes (produtos) na ordem especificada pelo genoma. Para cada gene, a função determina a célula de origem (a posição atual da empilhadeira) e a célula de destino (o produto que deve ser recolhido).

Com base nessas informações, a função verifica se existe um par correspondente na lista de pares de movimento definida no problema. Se encontrar um par correspondente, a função adiciona as células visitadas por esse par ao caminho da empilhadeira correspondente. Além disso, o fitness do indivíduo é incrementado com o valor do par e o número de passos dados pelo forklift nesse movimento.

É importante notar que os pares são construídos de forma especifica: os produtos menores são sempre definidos como a origem no par de movimento entre dois produtos. Caso o genoma contenha um par que não siga essa regra, os produtos são invertidos para garantir que o par correto seja encontrado. Ao inverter um par, também é necessário inverter o caminho percorrido

A função continua a percorrer os genes, atualizando a célula de origem para a próxima iteração e registando o produto atual como o produto anterior. Quando todos os genes de um subgenoma são percorridos, a função adiciona o último par de movimento da empilhadeira em direção à saída do armazém.

Em seguida, a função atualiza o número máximo de passos dados por um agente, com o valor máximo entre o número de passos do forklift atual e o valor anteriormente registado. Por fim, a função retorna o fitness calculado para o indivíduo.

2.5 Descrição do método de criação da população inicial

O método de criação da população inicial é responsável por gerar um conjunto de indivíduos que constituem a primeira geração do algoritmo genético. Essa população é criada de forma aleatória, seguindo algumas restrições definidas pelo problema.

O método começa por criar um vetor vazio para armazenar os indivíduos gerados. Em seguida, um loop é executado um número de vezes igual ao tamanho da população desejada.

Dentro desse loop, um novo indivíduo é criado utilizando a classe `WarehouseIndividual`, que é uma subclasse da classe `IntVectorIndividual`. Essa subclasse define a representação genética dos indivíduos como um vetor de números inteiros, onde cada gene representa a ordem de recolha dos produtos no armazém.

O genoma do novo indivíduo é inicializado com uma sequência de números inteiros de 1 a N, onde N é o número total de produtos no armazém. Essa sequência é baralhada aleatoriamente para garantir que cada indivíduo tenha uma ordem de recolha única.

Após a criação do novo indivíduo, ele é adicionado ao vetor que armazena a população inicial.

Esse processo é repetido até que o número desejado de indivíduos seja gerado, resultando na formação da população inicial completa. Isso estabelece a base inicial para o processo de evolução do algoritmo genético.

2.6 Descrição dos operadores genéticos desenvolvidos

A classe Mutation2 herda da classe Mutation e representa uma mutação do tipo "troca de genes" num indivíduo do tipo IntVectorIndividual. No método `mutate`, primeiro verifica-se se a mutação deve ocorrer com base na probabilidade fornecida. Se a condição for satisfeita, duas posições aleatórias diferentes são selecionadas no genoma do indivíduo. Em seguida, os genes nas posições selecionadas são trocados, ou seja, o valor do gene na posição pos1 é trocado com o valor do gene na posição pos2. Essa troca de genes contribui para a introdução de diversidade genética na população.

A classe Mutation3 também herda da classe Mutation e realiza uma mutação do tipo "inversão de genes". No método `mutate`, novamente é verificado se a mutação deve ocorrer com base na probabilidade fornecida. Se sim, duas posições aleatórias diferentes são selecionadas no genoma do indivíduo. Em seguida, a ordem dos genes entre pos1 e pos2 (inclusive) é invertida, ou seja, os genes são reorganizados em ordem inversa nesse intervalo. Essa inversão de genes pode levar a uma exploração de soluções diferentes no espaço de busca, ajudando a evitar a convergência prematura.

A classe ‘Recombination2’ é uma subclasse da classe `Recombination` e representa um operador de recombinação chamado "crossover" para indivíduos do tipo `Individual`. No método ‘recombine’, são selecionados dois pontos de crossover aleatórios. O segmento entre os pontos de crossover de `ind1` é copiado para criar `offspring1`. Os genes restantes de `ind2` que não estão presentes em `offspring1` são adicionados a `offspring1`. Da mesma forma, o segmento entre os pontos de crossover de `ind2` é copiado para criar `offspring2`, e os genes restantes de `ind1` são adicionados a `offspring2`. Por fim, os genomas de `ind1` e `ind2` são atualizados com as novas crias.

A classe `Recombination3` também é uma subclasse da classe `Recombination` e representa um operador de recombinação chamado "ciclo" para indivíduos do tipo `Individual`. No método `recombine`, é usada uma técnica de crossover baseada em ciclos para criar os filhos. Primeiro, o primeiro ciclo é criado começando

com o primeiro gene de `ind1` e encontrando a posição correspondente do gene em `ind2`. O ciclo continua até que todas as posições tenham sido preenchidas. Em seguida, as posições restantes nos filhos são preenchidas usando os genes correspondentes do outro progenitor. Por fim, os genomas de `ind1` e `ind2` são atualizados com as novas crias.

Estes operadores genéticos introduzem diversidade genética na população combinando informações genéticas de dois indivíduos progenitores. Esses operadores foram escolhidos para o problema porque são operadores genéticos baseados em permutações e não introduzem novos valores.

3. A apresentação e discussão dos resultados obtidos nos testes realizados

NOTA Devido à duração dos testes e má gestão de tempo por parte do grupo, tivemos de optar por cortar em alguns elementos nos testes, pelo que as recombinações e as mutações nos datasets 4 e 5 não foram terminadas a tempo da entrega do projeto.

NOTA nos gráficos do experimento "Recombination", o operador recombination2 está com o nome desconfigurado. Consideremos que os valores são Recombination2(0.N), onde N vai incrementando (no sentido de baixo para cima) de 0.1 a 0.9.

Testes realizados

1. **GeneralTests**

   Neste experimento vamos testar o algoritmo na sua generalidade, com vários valores em todos os parâmetros do ga, parâmetros esses que são o o tamanho da população, o numero máximo de gerações, método de seleção e o tamanho do torneio (o único método de seleção utilizado é o método "tournament", pelo que apenas se altera o seu tamanho). Temos também os métodos de recombinação e mutação, bem como a probabilidade de ocorrerem.

   Nos pontos seguintes vamos abordar mais detalhadamente a influencia destes parâmetros no algoritmo genético.

2. **Population**

   O objetivo deste experimento foi investigar o impacto da combinação de dois parâmetros: o tamanho da população e o número de gerações, para o Dataset N. O tamanho da população refere-se à quantidade de indivíduos na população de cada geração, e o número de gerações representa a quantidade de iterações que o algoritmo percorre para encontrar uma solução.

3. **Tournament**

   O objetivo deste experimento foi investigar como o tamanho do torneio influencia o desempenho geral do algoritmo. O tamanho do torneio refere-se ao número de indivíduos selecionados aleatoriamente da população para "competir" entre eles em cada geração. O indivíduo com a melhor fitness dentro do torneio é selecionado para reprodução e geração da próxima população. Como referido anteriormente o único método de seleção utilizado foi o "Tournament" pelo que o único parâmetro alterado foi o seu tamanho.

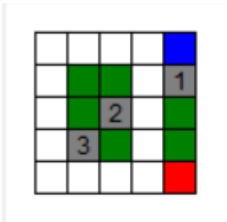
4. **Recombination**

   O objetivo deste experimento foi avaliar como diferentes métodos de recombinação e suas probabilidades afetam o desempenho geral do algoritmo. A recombinação é um operador genético que combina informações de dois ou mais indivíduos para criar descendentes na próxima geração. Além do método de recombinação, a probabilidade de aplicação desse operador também é um parâmetro importante. Esta probabilidade determina a chance de realizar a recombinação em um determinado par de indivíduos selecionados para reprodução.

5. **Mutation**

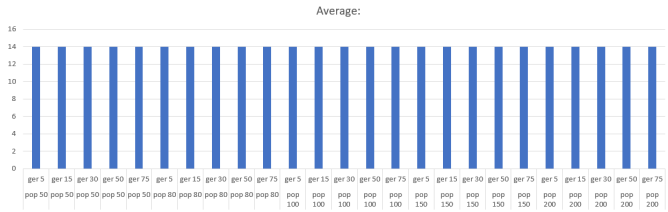
O objetivo deste experimento foi investigar como diferentes métodos de mutação e as suas probabilidades afetam o desempenho geral do algoritmo. A mutação é um operador genético que introduz variação aleatória nas informações genéticas dos indivíduos da população. Essa variação é fundamental para garantir a diversidade genética e evitar a estagnação do algoritmo num max/min local. Além do método de mutação, a probabilidade de aplicação deste operador é um parâmetro essencial. Esta probabilidade determina a chance de realizar a mutação em cada gene de um indivíduo selecionado para reprodução.

3.1. Dataset1



Numero de runs: 30

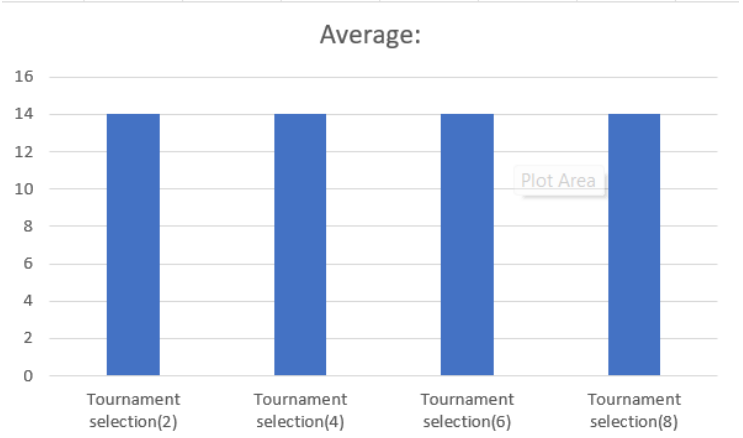
3.1.1. Population - Dataset1



Melhor resultado obtido (30 runs):

- Tamanho da populacao : 50
- Numero max. de gerações : 200
- Tamanho do torneio : 4
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 14

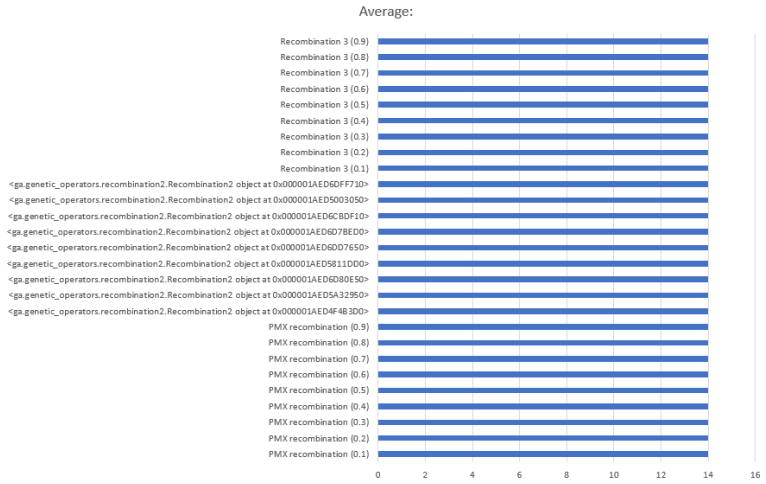
3.1.2. Tournament – Dataset1



Melhor resultado obtido (30 runs):

- Tamanho da populacao : 50
- Numero max. de gerações : 5
- Tamanho do torneio : 4
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 14

3.1.3. Recombination – Dataset1

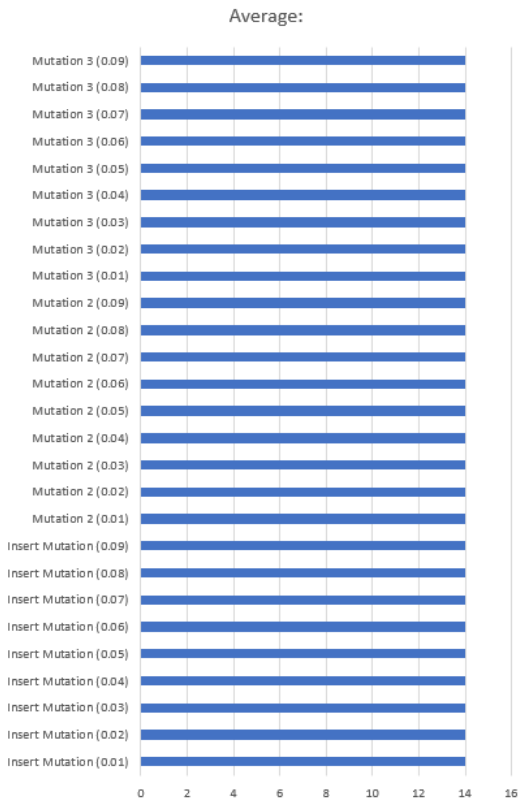


Melhor resultado obtido (30 runs):

- Tamanho da populacao : 50
- Numero max. de gerações : 5
- Tamanho do torneio : 2

- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 14

3.1.4. Mutation – Dataset1



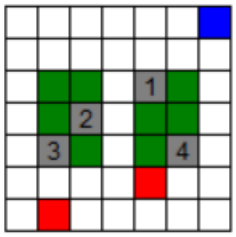
Melhor resultado obtido (30 runs):

- Tamanho da populacao : 50
- Numero max. de gerações : 5
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 14

3.1.5 Conclusão dataset1

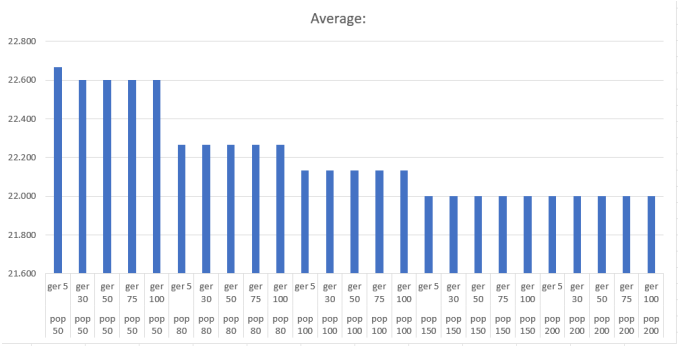
Devido a simplicidade deste dataset, todas as combinações de parâmetros do GA obteram sempre o mesmo *average* (14), que corresponde à melhor solução do problema.

3.2. Dataset2



Numero de runs: 30

3.2.1. Population – Dataset2

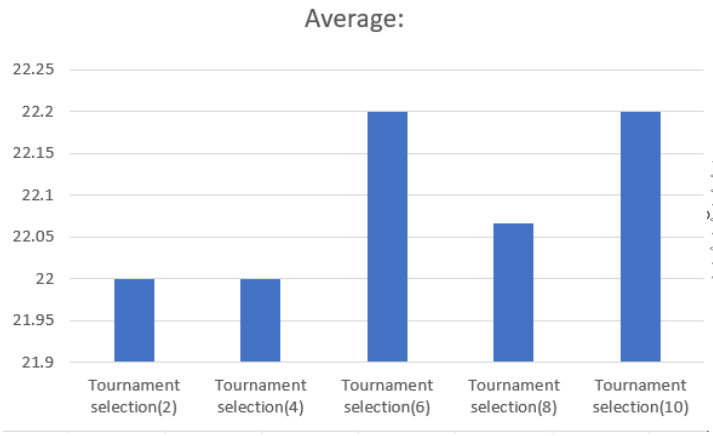


Com base na observação do gráfico do experimento "Population" do dataset2, podemos perceber que quando a população por geração é igual ou superior a 150, vai ser sempre encontrado o melhor average (y), independentemente do número de gerações imposto. Devido a isto, optou-se por utilizar o tamanho da população de 150 e um número máximo de gerações a 5, com 30 runs para os próximos testes.

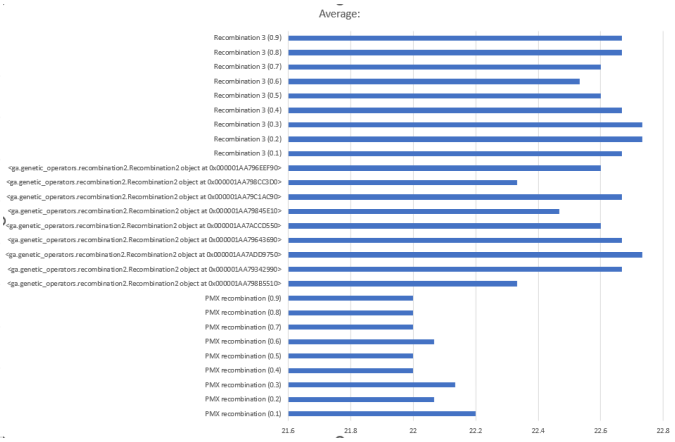
Melhor resultado obtido (30 runs):

- Tamanho da populacao : 50
- Numero max. de gerações : 5
- Tamanho do torneio : 4
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 22

3.2.2. Tournament – Dataset2



3.2.3. Recombination – Dataset2



Como é possível verificar no gráfico "Tournament" do dataset2, o melhor average é atingido nos menores tamanhos deste método de seleção (tamanho 2 e 4) e a partir destes valores o melhor average já não é alcançado.

O tamanho 6 e 10 foram os que apresentaram uma maior divergência do melhor average. O tamanho 8, mesmo tendo uma menor divergência, não conseguiu alcançar o melhor average (22) por 0.05.

Com isto, foi utilizado o tournament_size = 2 para o próximo experimento.

Melhor resultado obtido (30 runs):

- Tamanho da populacao : 150
- Numero max. de gerações : 5
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 22

Observando o gráfico relativo ao experimento "recombination" no dataset2, o melhor *average* é alcançado pelo método de recombinação PMX, especificamente com probabilidade 0.4, 0.5, 0.7, 0.8, 0.9.

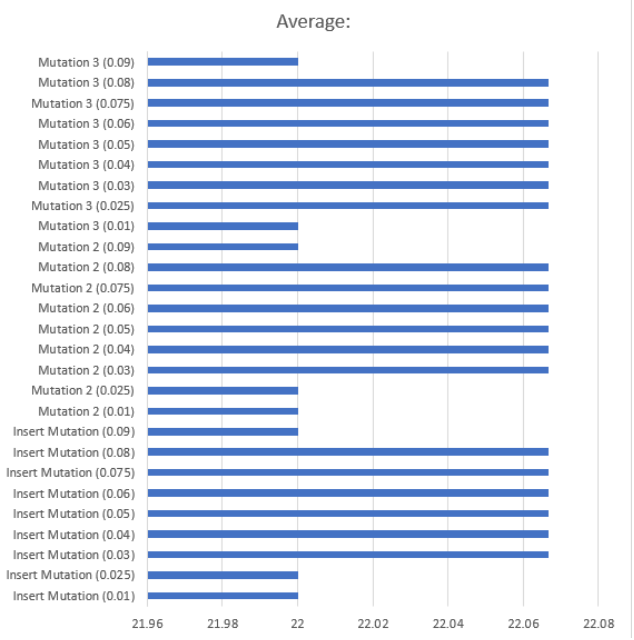
Os valores com um maior desvio da melhor solução encontram-se distribuídos pelos restantes métodos de recombinação usados, especialmente em recombination2 com probabilidade de 0.3 e recombination3 com probabilidades de 0.2 e 0.3.

O gráfico apresenta pouca regularidade ao longo dos 3 métodos de recombinação, contudo o método PMX destaca-se em relação aos outros, atingindo sempre melhores soluções que as restantes recombinações, independentemente da probabilidade. Com isto, usufruímos da recombinação pmx, com probabilidade de 0.7

Melhor resultado obtido (30 runs):

- Tamanho da populacao : 150
- Numero max. de gerações : 5
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 22

3.2.4. Mutation – Dataset2

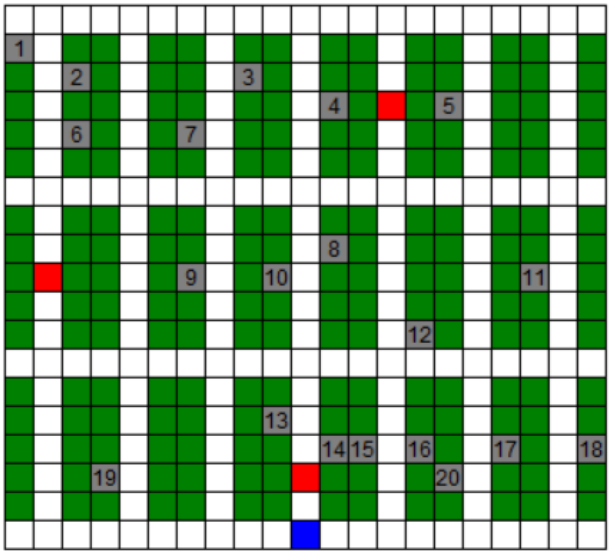


Com base no gráfico acima representado, existem apenas 2 valores de *average* resultantes deste experimento. O melhor average observa-se em todos os tipos de mutação, sempre quando a probabilidade de mutação é 0.9 e 0.01. é também encontrada a melhor solução e mutation2, com probabilidade de 0.025. O resto das mutações nas probabilidades restantes encontraram todos a mesma solução, sendo esta a pior solução das duas.

Melhor resultado obtido (30 runs):

- Tamanho da populacao : 150
- Numero max. de gerações : 5
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 22

3.3. Dataset3



Numero de runs: 30 em generalTests e Population, 15 no resto dos experimentos

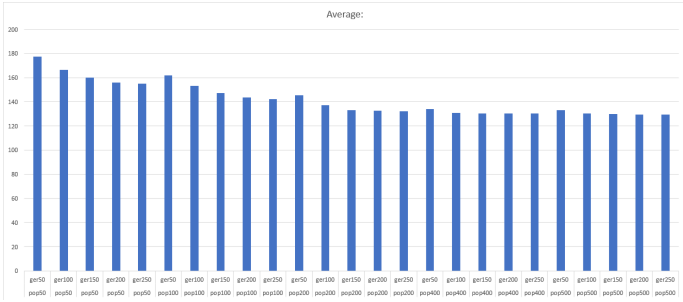
3.3.1. GeneralTests

NOTA porque a velocidade dos testes foi muito mais prolongada do que o esperado, usámos o resultado do experimento "GeneralTests" do dataset3 para a realização dos testes no dataset4 e dataset5.

Melhor resultado obtido (30 runs):

- Tamanho da populacao : 200
- Numero max. de gerações : 100
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Mutation2
- Probabilidade de mutação : 0.01
- Fitness : 120

3.3.2. Population - Dataset3

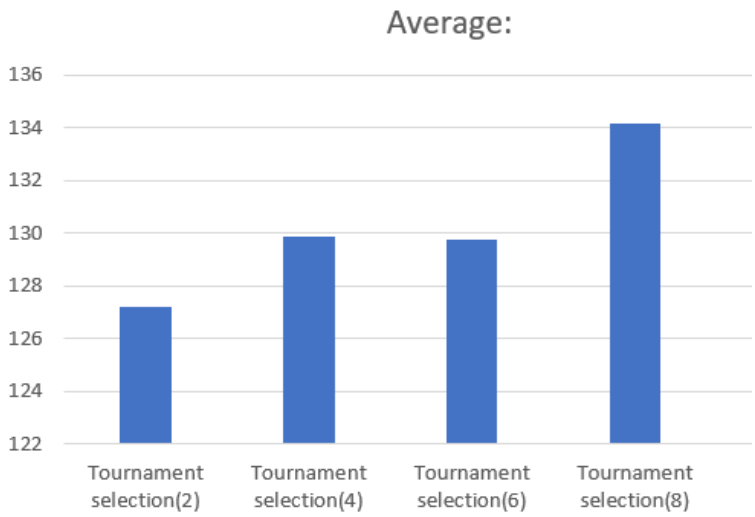


Com referência ao gráfico do experimento "Population" do dataset3, podemos perceber que quanto a população por geração é, maior a tendência para se encontrar a melhor solução. O melhor average é encontrado por duas combinações diferentes, com população de 500 e 200 gerações e com população de 500 e 250 gerações (~129.3). Devido a isto, optou-se por utilizar o tamanho da população de 500 e um número máximo de gerações a 200, com 15 runs para os próximos testes.

Melhor resultado obtido (30 runs):

- Tamanho da populacao : 500
- Numero max. de gerações : 50
- Tamanho do torneio : 4
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 118

3.3.3. Tournament - Dataset3



Levando em consideração o gráfico. "Tournament" do dataset3, o melhor average é atingido no menor tamanho deste método de seleção (tamanho 2) e a partir deste valor o melhor average já não é alcançado.

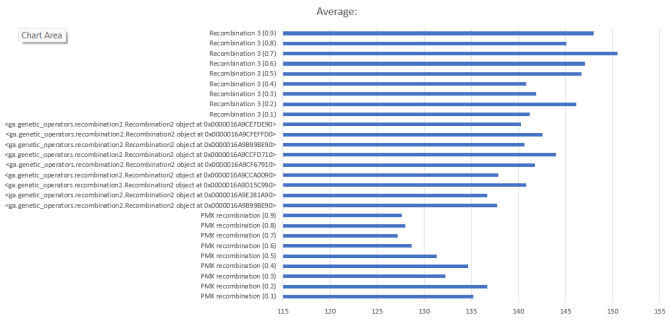
O tamanho 8 foi os que apresentarou uma maior divergência do melhor average, enquanto que os valores intermédios (4,6) atingiram valores semelhantes, porém não constituem a melhor solução.

Com isto, foi utilizado o tournament_size = 2 para o próximo experimento.

Melhor resultado obtido (15 runs):

- Tamanho da populacao : 500
- Numero max. de gerações : 200
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 116

3.3.4. Recombination - Dataset3



Observando o gráfico relativo ao experimento "recombination" no dataset2, o melhor *average* é alcançado pelo método de recombinação PMX, especificamente com probabilidade 0.7.

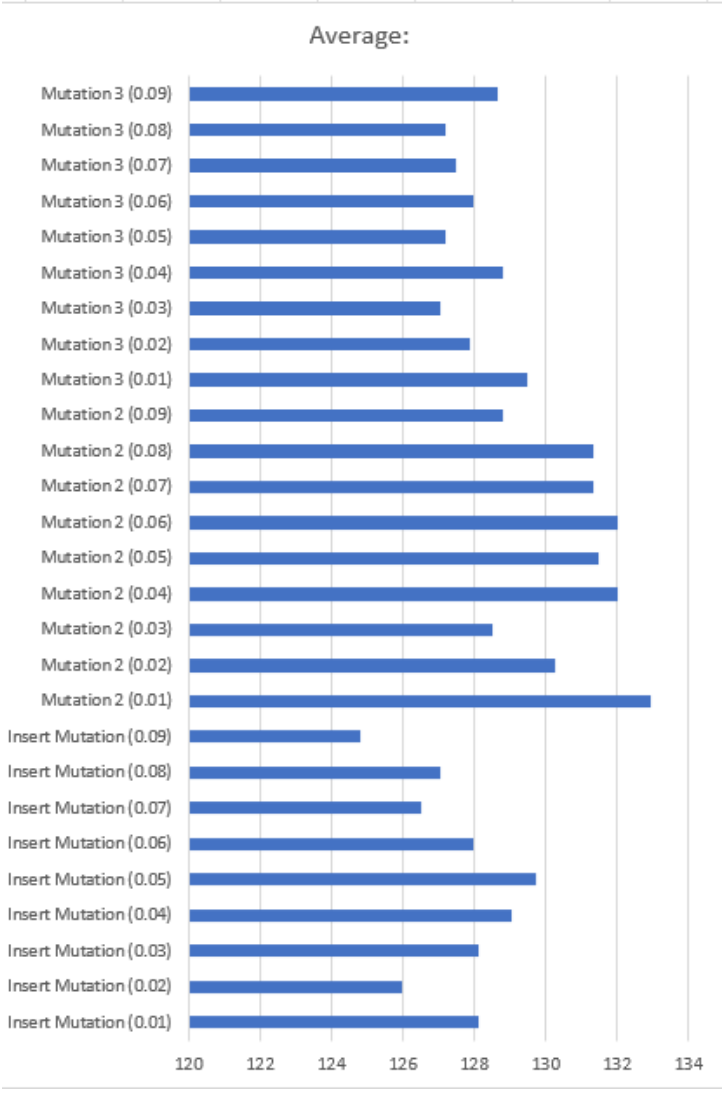
Os valores com um maior desvio da melhor solução encontram-se distribuídos pelos restantes métodos de recombinação usados, especialmente em recombination3 com probabilidade de 0.7, foi o metodo de recombinação que teve menor sucesso a encontrar o melhor average.

O gráfico apresenta pouca regularidade ao longo dos 3 métodos de recombinação, contudo o método PMX destaca-se em relação aos outros, atingindo sempre melhores soluções que as restantes recombinações, independentemente da probabilidade. Com isto, usufruímos da recombinação pmx, com probabilidade de 0.7

Melhor resultado obtido (15 runs):

- Tamanho da populacao : 500
- Numero max. de gerações : 200
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.3
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 116

3.3.5. Mutation – Dataset3



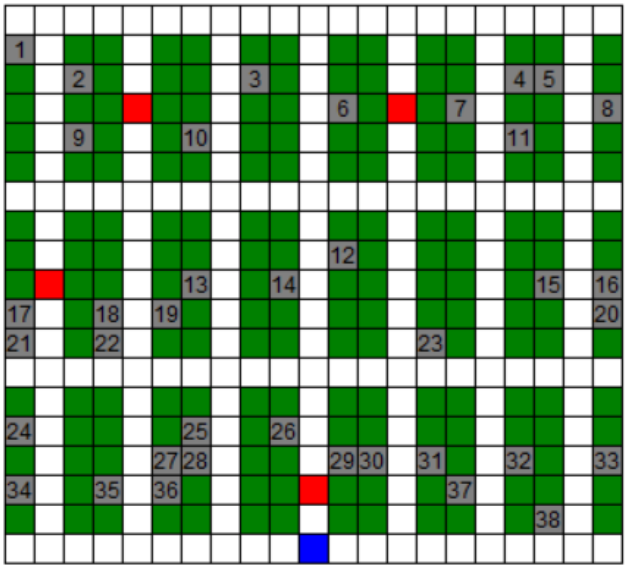
A partir da análise do gráfico acima representado. O melhor *average* observa-se em Insert mutation, sempre quando a probabilidade de mutação é 0.09. O resto das mutações nas probabilidades restantes encontraram soluções piores, predominantemente na mutação 2.

Melhor resultado obtido (15 runs):

- Tamanho da populacao : 500
- Numero max. de gerações : 200
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert

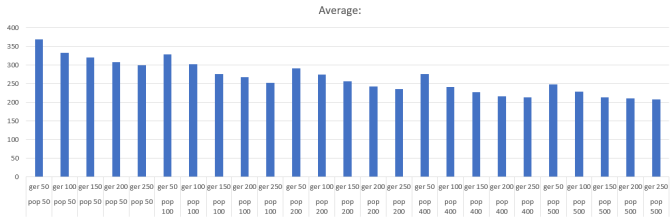
- Probabilidade de mutação : 0.01
- Fitness : 116

3.4. Dataset4



Numero de runs : 10

3.4.1. Population – Dataset4

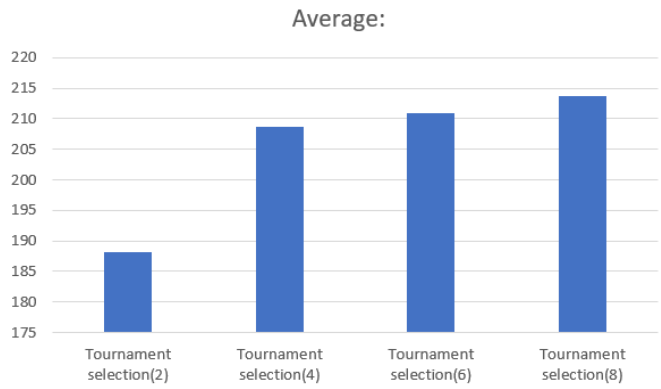


Com base na observação do gráfico do experimento "Population" do dataset4, verifica-se que á medida que o numero de gerações aumenta, o avarege diminui, e que o mesmo acontece para o tamanho da população com o mesmo numero de gerações. Verifica-se tambem que o melhor avarege foi atingido com o tamanho da população de 500 e um número máximo de gerações de 250. Devido a isto iremos utilizar estes valores para os proximos testes do dataset4.

Melhor resultado obtido (10 runs):

- Tamanho da populacao : 500
- Numero max. de gerações : 250
- Tamanho do torneio : 4
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.01
- Fitness : 116

3.4.2. Tournament – Dataset4



Como é possível verificar no gráfico "Tournament" do dataset4, a medida que o tamanho do torneio aumenta o avarege tambem aumenta, havendo grande disparidade do tamanho 2 para os outros tamanhos, porem dentro dos tamanhos 4,6,8 a divergencia no avarege é muito menor. Com isto, foi utilizado o tournament_size = 2 para o continuação dos experimentos no dataset4.

Melhor resultado obtido (10 runs):

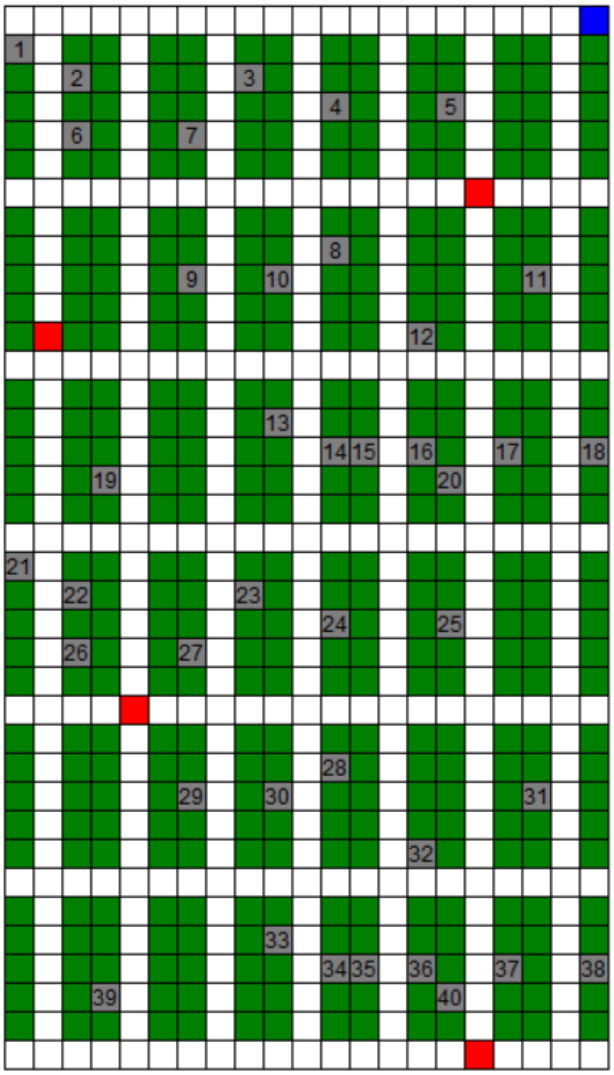
- Tamanho da populacao : 500
- Numero max. de gerações : 250
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 167

3.4.3. Recombination – Dataset4

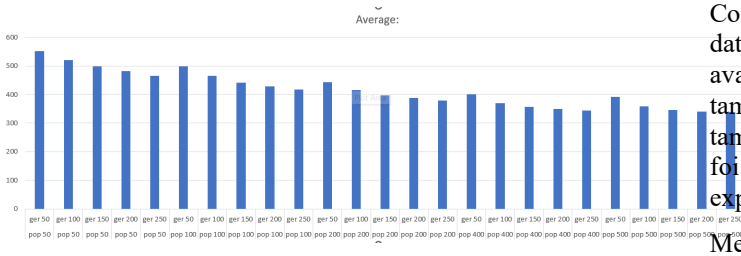
3.4.4. Mutation – Dataset4

- Probabilidade de mutação : 0.025
- Fitness : 169

3.5. Dataset5



3.5.1. Population – Dataset5

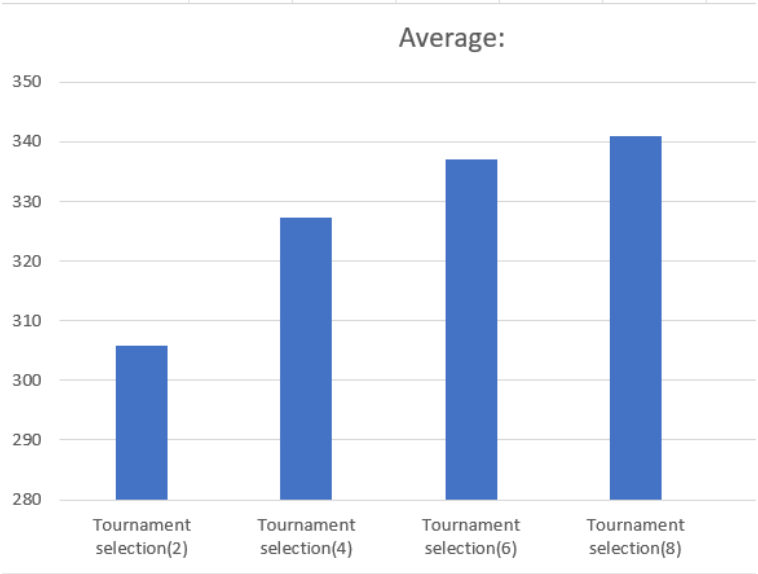


Com base na observação do gráfico do experimento "Population" do dataset5, verifica-se que á medida que o numero de gerações aumenta, o avarege diminui, e que o mesmo acontece para o tamanho da população com o mesmo numero de gerações. Verifica-se tambem que o melhor avarege foi atingido com o tamanho da população de 500 e um número máximo de gerações de 250. Devido a isto iremos utilizar estes valores para os proximos testes do dataset5.

Melhor resultado obtido (10 runs):

- Tamanho da populacao : 400
- Numero max. de gerações : 200
- Tamanho do torneio : 4
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7
- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 292

3.5.2. Tournament – Dataset5



Como é possível verificar no gráfico "Tournament" do dataset5, á medida que o tamanho do torneio aumenta o avarege tambem aumenta, havendo grande disparidade do tamanho 2 para os outros tamanhos, porem para os tamanhos 6 e 8 a diferença de valores foi menor. Com isto, foi utilizado o tournament_size = 2 para o continuação dos experimentos no dataset5.

Melhor resultado obtido (10 runs):

- Tamanho da populacao : 500
- Numero max. de gerações : 250
- Tamanho do torneio : 2
- Metodo de recombinação : PMX
- Probabilidade de recombinação : 0.7

- Metodo de mutação : Insert
- Probabilidade de mutação : 0.025
- Fitness : 274

3.5.3. Recombination – Dataset5

3.5.4. Mutation – Dataset5

4. Conclusão

Neste projeto, desenvolvemos uma aplicação para otimizar o processo de recolha de produtos em um armazém, conhecido como "picking". Utilizamos algoritmos genéticos e o algoritmo de procura A* para resolver o problema, visando minimizar o tempo de entrega do último produto e a distância percorrida pelos agentes, enquanto evitamos colisões entre eles.

Através da aplicação desenvolvida, foi possível representar a estrutura do armazém, a localização dos produtos, o ponto de recolha e a posição inicial dos agentes numa matriz.

Os pares são representados por um dicionário, onde a key é a hash do par e o valor é o próprio par. Esses pares representam a combinação de células que precisam ser percorridas pelos agentes do armazém para coletar produtos.

Implementamos a classe 'WarehouseState', que permitiu definir os estados do problema e as ações que os agentes podem tomar. Também desenvolvemos a classe WarehouseProblemSearch, que utilizou o algoritmo A* para calcular os caminhos ótimos entre os estados. Além disso, foi implementada a classe HeuristicWarehouse, que forneceu uma heurística para guiar o algoritmo de procura.

Através da aplicação, foi possível visualizar a simulação da melhor solução encontrada, que representa a distribuição de produtos entre os agentes e a ordem de recolha. Os resultados obtidos demonstraram a eficácia dos algoritmos utilizados, permitindo uma otimização significativa do processo de "picking" no armazém.

Em conclusão, o projeto proporcionou uma oportunidade de aplicar algoritmos genéticos e o algoritmo A* para resolver um problema real de otimização logística. A aplicação desenvolvida pode ser utilizada como uma ferramenta útil para melhorar a eficiência do processo de recolha em armazéns, contribuindo para reduzir custos e tempo de entrega.