Picking, Recolha de Produtos em Armazéns

Bernardo Paulo Melo
2211000

Renato Luz de Oliveira
2200232

RESUMO

A aplicação tem como objetivo otimizar a recolha de produtos de um armazém, de forma a minimizar o tempo de entrega do último produto e a distância total percorrida pelos agentes.

1. INTRODUÇÃO

O presente trabalho enquadra-se na temática da Inteligência Artificial, no âmbito da UC de Inteligência Artificial que integra o Curso de Engenharia Informática. Este terá como objetivo aplicar os conhecimentos básicos adquiridos durante as aulas pratico-laboratoriais/teóricas e também desenvolver e aprofundar o nosso conhecimento sobre esta temática, muito presente na nossa atualidade, através do uso de algoritmos de pesquisa, algoritmos genéticos, operadores de mutação e operadores de recombinação, com o objetivo de resolver um dado problema.

O projeto consiste numa aplicação, com o objetivo de recolher produtos das prateleiras de um armazém (processo conhecido por *picking*), desta forma, os produtos são distribuídos pelos agentes responsáveis pela sua recolha (os forklifts/empilhadoras). Para resolvermos esta problemática, iremos recorrer a um algoritmo de procura, conhecido por A*, ,este irá calcular os caminhos mais curtos e que em conjunto com recurso a um algoritmo genético e a operadores de mutação e recombinação, irão otimizar os vários caminhos, isto é, irão permitir definir a ordem pela qual cada agente deve recolher os produtos que lhe foram atribuídos, conduzindo ao grande objetivo do projeto, que é o de conseguir minimizar o tempo de entrega do último produto a ser entregue, diminuindo assim a distancia total percorrida pelos *forklifts* e o número de colisões entre estes. Permitindo desta forma, encontrar o caminho mais eficiente, para fazer a recolha dos produtos.

2. DESCRIÇÃO DA IMPLEMENTAÇÃO

descrição da implentação(estado, heuristica, individuos,...)

No projeto foi usado o algoritmo A* com o objetivo de calcular os caminhos mais curtos, com vista a efetuar-se posteriormente a recolha dos produtos de forma mais eficaz. Deste modo, o projeto irá apresentar uma heurística, que consiste num método que tem como objetivo, resolver o nosso problema, isto é, encontrar uma solução de forma rápida, que seja ao mesmo tempo temporalmente eficiente e que produza resultados confiáveis.

2.1 Estado

O estado do problema vai ser representado na classe WarehouseState.py, onde foi definida a forma como se irá alterar a posição do foklift, uma vez que nesta encontram-se métodos específicos com o objetivo de definir o movimento do *forklift* no armazém, este vai ser composto por espaços livres onde os *forklifts* podem circular, por prateleiras vazias ou com produtos que mais tarde ou mais cedo irão ser recolhidos pelo *forklift*, além disso também irão apresentar um ponto de entrega. Desta forma, estes terão de se movimentar de forma, a circularem pelos espaços, para poderem ir recolher o produto, e assim sucessivamente até recolherem todos os produtos que eram supostos serem recolhidos por si. Por outro lado, o *forklift* para recolher um terminado produto deverá colocar-se na célula adjacente onde consta o produto a recolher e não deve ir para a célula onde se encontra o produto. Depois de recolher o último produto, deverá ir para a saída. O agente, poderá movimentar-se para cada uma das células adjacentes, isto é, poderá avançar para Norte, Sul, Este, Oeste, desde que a célula não seja uma prateleira, de forma que os caminhos evitem as prateleiras, isto é que sejam caminhos ótimos. Além disso, foi implementado no warehouse_probemforSearch.py um método, denominado *is_goal*, que terá como objetivo ver se alcançamos o goal State, isto é, ver se o forklift chega ao local final, que corresponde à saída. Desta forma, enquanto o *forklift* não chegar á *goal position*, vamos ter de ter atenção às células adjacentes ao nosso *forklift* e se este se encontra posicionado adequadamente. Quanto este se encontrar na goal position, o método irá retonar *True*, caso contrário, irá retornar *False*.

Os métodos “can_move_up”, “can_move_right”, “can_move_down” e “can_move_left”, possuem a função de verificar se o *forklift* se pode movimentar numa dada direção, ambos irão retornar True se o *forklift* não se encontra nas extremidades do armazém e se célula adjacente na direção desejada não for uma prateleira.

O método “move_up”, “move_right”, “move_down” e “move_left” movem o forklift numa direção válida, tenho em consideração as limitações do armazém.

2.2 Heurística

A heurística implementada no projeto encontra-se localizada, no ficheiro “HeuristicWarehouse.py” na diretoria warehouse, neste caso irá ter como objetivo calcular todas as distâncias entre pontos importantes, denominado por *pairs*, com recurso ao algoritmo A*. Neste caso irá ser usada a fórmula da distância euclidiana entre 2

pontos, que nos irá devolver o valor absoluto entre a distância entre os 2 pontos de um *pair* (ex: 1-4, 2-2...), (esta distância irá ser usada mais tarde para calcular a fitness na função compute fitness, pertencente ao algoritmo genético). Para cada par vai ser então calculado o valor da distância entre 2 pontos, que posteriormente irão ser otimizadas, com vista a minimizar as distâncias.

2.3 Mutações e Recombinações

Para além do método de mutação presente na classe *MutationInsert*, foram adicionados outros 2 operadores de mutação. De uma forma geral, os operadores de mutação são aplicados no processo de criação de uma nova geração de indivíduos. Neste caso é o equivalente á criação de uma geração com vários produtos, dispostos aleatoriamente e associados a um *forklift* que os irá transportar. As mutações vão ser baseadas em probabilidades, e estas deverão ser usadas com uma baixa probabilidade, com o objetivo de não prejudicar a qualidade de cada indivíduo criado, o que poderá causar a produção de resultados muitos díspares dos pretendidos. As mutações irão levar a que, por exemplo, um *forklift* passe a transportar um dado produto, que anteriormente era transportado por outro. Um dos operadores de mutação será do tipo" swap *mutation*", que é caracterizada pela seleção aleatória de 2 genes do genoma, e logo a seguir procede-se á troca dos seus valores, o que poderá conduzir á troca de produtos transportados por dois *forklifts*, à mudança dos genes transportados pelos *forklifts* ou mesmo á troca de todos os produtos entre 2 *forklifts* (correspondente ao caso que um *forklift* é substituído por outro). Foi escolhido este tipo de mutação, uma vez que pode ser aplicado a vetores com uma certa ordem de elementos, fazendo com que os novos vetores originados, continuem a possuir os mesmos genes que os originais, isto é, não irá ocorrer a perda de produtos, nem de *forklifts*, o qual não faria sentido no contexto do problema.

O segundo operador de mutação usado, será do tipo "*inversion mutation*", que consistirá na seleção aleatória de genes, sendo depois a ordem dos genes invertida. Foi escolhido este operador, pelo mesmo motivo, mencionado anteriormente em relação ao" *swap mutation*".

Os operadores de recombinação permitem a recombinação de genes entre 2 indivíduos distintos, por outro lado, a mutação ocorre no mesmo indivíduo. Desta forma, podemos definir uma recombinação como uma troca de segmentos de genes entre 2 indivíduos. No projeto foram usados, 3 operadores de recombinação diferentes, desde os quais o pwx, "partially mapped crossover" que irá selecionar 2 indivíduos e irá proceder á seleção aleatória de segmentos de genes e depois á troca entre eles, reparando ao mesmo tempo algum problema causado pela troca de genes. Desta forma, irá se obter outros 2 indivíduos, cada um deles não irá apresentar genes repetidos e respeitará a ordem relativa dos genes no vetor do genoma, o que é de salientar, pois no problema em questão, não pode existir a repetição do mesmo forklift, nem do produto no genoma. Os outros 2 operadores de recombinação que irão ser usados serão: o Order Crossover (OX) e o Two-Point Crossover (TPX), uma vez que também como o anterior respeitam a ordem relativa dos genes no vetor do genoma. Este operador de recombinação foi criado por "De Jong", e consiste na seleção aleatória de 2 pontos no genoma na mesma posição para cada indivíduo, desta forma, os genes dos indivíduos irão ser separadas pelos pontos de crossover, e depois ocorrerá troca dos genes entre os indivíduos, criando 2 "offsprings" diferentes. O Order Crossover (OX), irá copiar um segmento aleatório do indivíduo 1, seguidamente, irá copiá-lo para uma variável auxiliar. Depois o restante desta irá ser preenchido com genes do individuo 2. Depois de copiar-se um segmento do individuo 2, para uma segunda variável auxiliar, esta irá ser preenchida com genes oriundos do individuo 1.

No algoritmo genético, irá ser usado o método de seleção "Tournament", que irá escolher aleatoriamente os indivíduos e irá e selecionar os melhores deles.

2.4 Definição do genoma e cálculo fitness

No "vectIntIndividual.py", encontra-se definido o vetor que irá representar o genoma do nosso problema, este será um vetor de inteiros e irá ser preenchido, inicialmente com zeros. No "Warehouse_problemforGa.py", irá ser definido o tamanho do vetor, este será definido pela soma da quantidade de produtos, mais a quantidade de forklifts -1, depois de criado o vetor, os seus genes irão ser baralhados de forma aleatória e além disso, este terá início no índice 1.

O compute fitness incluído na classe "WarehouseIndividual" irá calcular o quão "apta" a distribuição dos produtos pelos forklifts é, isto é, o quão bom esta resolverá o problema. Este método, irá percorrer os caminhos, feitos por cada *forklift* no armazém, baseando-se na sua posição inicial. Neste caso não irão ser contabilizadas as colisões, que ocorrem durante o caminho, no entanto é de salientar que estas terão um impacto no fitness. As colisões teriam de ser contabilizadas, sempre que dois *forklifts*, se encontram na mesma posição em algum momento do percurso.

O valor de aptidão calculado é armazenado na variável *self.fitness* e será do tipo *float*.

No compute fitness presente no ficheiro "WarehouseIndividual", como o genoma, vai ser composto pelas encomendas e pelos *forklifts*, cada elemento vai ser representado por números, sem repetição, isto é não pode haver duas encomendas iguais, nem 2 *forklifts* iguais. De forma a representar estes elementos corretamente, o número de *forklifts* é sempre o número de encomendas+1, cada forklift irá funcionar como um separador, isto é, irá delimitar no genoma as encomendas que cada forklift irá recolher. Nesta função foram implementas algumas proteções, desde as quais que o separador não pode ser o primeiro elemento do genoma, uma vez que assim não estaria a delimitar corretamente as encomendas a recolher e além disso, que também não pode haver um genoma sem separador, isto é, tem de haver sempre um *forktlift* para transportar as encomendas. Também não poderá existir o caso em que o *forklift* não tem encomendas para transportar. Primeiro vamos ver quais os elementos do genoma são separadores e vamos guardá-los numa lista. Sempre que o

separador está na primeira posição, a iteração vai ser ignorada. Assim sempre que o forklift, encontrar-se onde está o produto, vai ser calculada a distância da célula até a próxima célula, que vai ser somada á variável *self.total_distance*,

Quando um *forklift* vai ter um caminho vazio, isto é que não possua produtos para recolha, vai calcular a distância a partir da célula atual do *forklift* até à saída.

Para o primeiro produto no caminho do *forklift*, vai se calcular a distância a partir da célula atual, até a célula do produto.

Para o último produto no caminho do *forklift*, vai se calcular a distância a partir célula do produto até a célula de saída. Para os restantes, vai ser calculada a distância entre a célula atual do produto e a próxima célula do produto.

Assim o fitness é a distância total, que os *forklifts* percorreram.

O *obtain_all_path* deverá rastrear a distância completa e o número total de colisões, desta forma iria-se calcular os caminhos completos percorridos pelos forklifts e devolver uma lista de células, isto é, as células percorridas por cada forklift e além disso também devia devolver o número de passos necessários para percorrer todos os caminhos e o caminho mais longo percorrido pelo forklift.

3. RESULTADOS

(apresentação e descrição)

Primeiro foram realizados testes a todos os datasets, com uma configuração geral, com os seguintes parâmetros:

Runs: 30; Population_size: 50, 100; Max_generations: 50; Population_size: 50, 100; Max_generations:50; Recombination: pmx; Recombination_probability: 0.6, 0.7; Mutation: insert; Mutation_probability: 0.01, 0.03, 0.1

No entanto, não foram concluídos os testes para os problemas 3,4 e 5, devido á elevada quantidade de tempo que levaria para serem efetuados, não é possível afirmar os seus valores com grandes certezas. No mínimo para os resultados terem eficiência estatística, será preciso fazer 30 runs. Para o problema 1 a fitness média foi de 10, para todas as combinações de variáveis. Para o problema 2 foi de 25. Para o problema 3 a média do fitness é de aproximadamente 178, para o 4 é de 393 e para o 5 é de 586.

Para se testar a population, foram usados os seguintes parâmetros:

Runs: 20; Population_size: 50, 100, 150; Max_generations: 12; Runs: 30; Population_size: 50, 100; Max_generations:25; Recombination: pmx; Recombination_probability: 0.6; Mutation: insert; Mutation_probability: 0.1; Selection: Tournament selection: 2. Neste teste apenas foi feito variar a população, os restantes parâmetros mantiveram-se constantes. Após analisar o gráfico da figura1, tendo em conta que foram feitas 20 execuções, os resultados podem não ser os mais fiáveis nem os que apresentam a melhor eficiência estatística (no entanto já nos dão uma ideia relativamente "fiável" dos resultados), mas podemos concluir que o melhor resultado foi obtido com a combinação dos parâmetros anteriormente referidos com uma população de 100 elementos. Sempre que for mencionado, qual é o melhor resultado, estaremos a referir-nos ao que apresentou o valor do desvio padrão mais baixo, isto é, o que indica uma baixa dispersão dos dados em torno de média.

Para se testar o trournament, foram usados os seguintes Parametros:

Runs: 20; Population_size: 100; Max_generations: 12; Runs: 30; Population_size: 50, 100; Max_generations:25; Recombination: pmx; Recombination_probability: 0.6; Mutation: insert; Mutation_probability: 0.1; Selection: Tournament selection: 2, 4, 6,8 e 10. Neste teste apenas foi feito variar o tournament, os restantes parâmetros mantiveram-se constantes. Após analisar o gráfico da figura2, e tendo em consideração que foram feitas 20 execuções, os resultados podem não ser os mais fiáveis, mas podemos concluir que o melhor resultado foi obtido com a combinação dos parâmetros anteriormente referidos com o valor do tournament a 4.

Para se testar as mutações, foram usados os seguintes Parametros:

Runs: 20; Population_size: 100; Max_generations: 12; Runs: 30; Population_size: 50, 100; Max_generations:25; Recombination: pmx; Recombination_probability: 0.6; Mutation: insert, mutation2 e muttaion3; Mutation_probability: 0.01, 0,03; 0,1; e 0,2; Selection: Tournament selection: 2. Neste teste apenas foi feito variar os operadores e a sua probabilidade, os restantes parâmetros mantiveram-se constantes. Após analisar o gráfico da figura3, tendo em conta que foram feitas 20 execuções, os melhores resultados foram:a Insert Mutation com 0.2%, a mutation2 com 0,1% e no caso da mutation3, o teste com 0.1% de probabilidade. No entanto, podemos concluir que de forma geral os resultados obtidos com a insertion, foram melhores.

Para se testar as recombinações, foram usados os seguintes Parametros:

Runs: 20; Population_size: 100; Max_generations: 12; Runs: 30; Population_size: 50, 100; Max_generations:25; Recombination: pmx; Recombination_probability: 0,2; 0,3; 0,4; 0.6 e 0,8; Mutation:insert; Mutation_probability: 0,1; Selection: Tournament selection: 2. Neste teste apenas foi feito variar os operadores e a sua probabilidade, os restantes parâmetros mantiveram-se constantes. Após analisar o gráfico da figura4, tendo em conta que foram feitas 20 execuções e apenas foram feitos testes para o operador de recombinação PMX, os melhores resultados foram: o PMX a 0.8%.

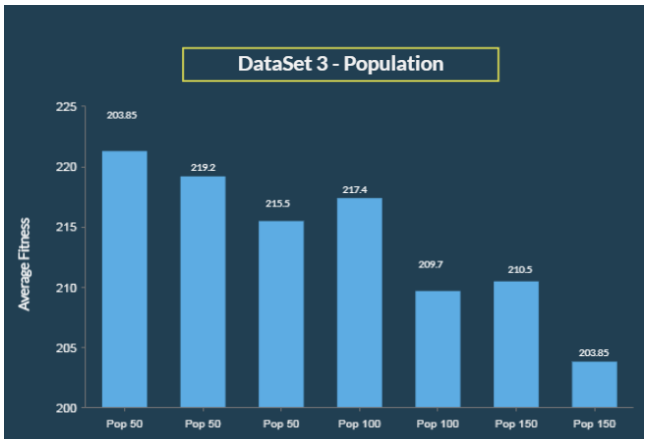

Figura 1- Gráfico Population

Figura 2- Gráfico Tournament Size

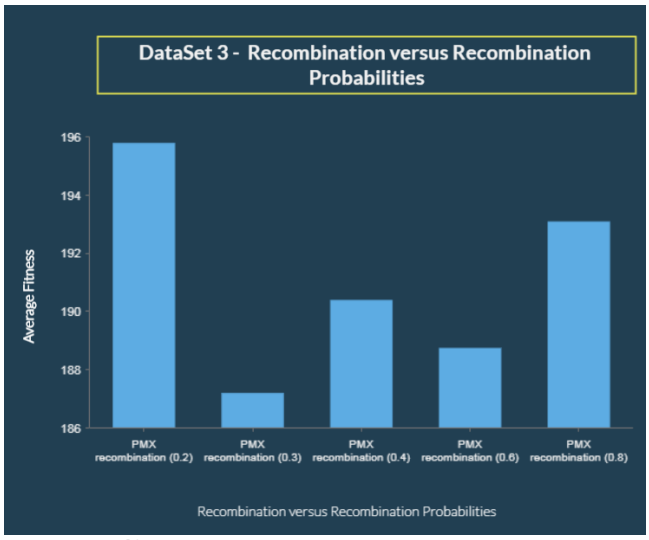

Figura 3- Gráfico Recombination

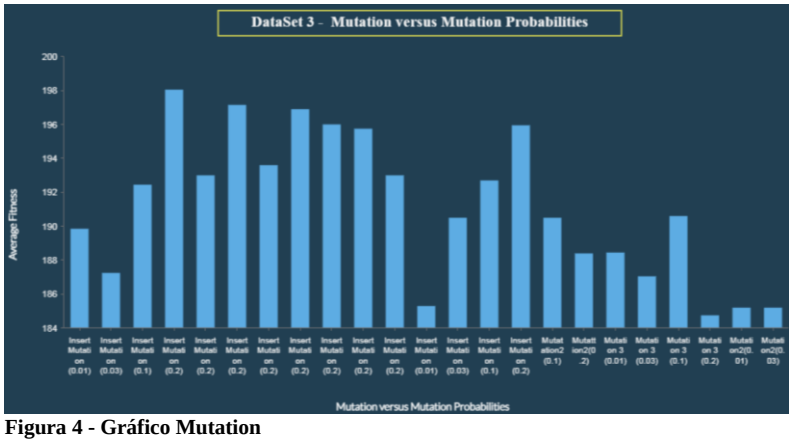

Figura 4 - Gráfico Mutation

4. Conclusões

De uma forma geral, foi feito o desenvolvimento das classes: warehouseStatee; warehouseProblemSearch, a heuristicWarehouse, warehouseProblemGa, ou seja, a aplicação consegue calcular a distância entre os *pairs*, aplicando o A*. Na classe warehouseIndividual, não ficou a funcionar a função obtain_all_path, não estando a sua implementação correta, por outro lado, foi feita a função compute_fitness, que calcula o valor do fitness, com base nas distâncias entre os pares calculadas anteriormente, os valores destas distâncias são mostrados na Gui da aplicação. Nesta também se encontra o valor do fitness. Por outro lado, não foi implementada a simulação na Gui da melhor solução encontrada. A fitness também não se encontra, o mais otimizada possível, pois não são contabilizadas as colisões nem o tempo que cada *forklift* demora.

Além disso, também foram definidos, 2 operadores de mutação e de recombinação para além dos que, já eram previamente fornecidos, no entanto é de salientar que estes operadores, são bem conhecidos, não tendo sido criados por nós, apenas foram implementados no projeto.

Por fim, não foram feitos os testes, a todos os *datasets* e os que foram feitos podem conter alguma incorreção.

5. BIBLIOGRAFIA

[1] Apontamentos disponibilizados pelos docentes da disciplina em causa.

[2] https://www.mecs-press.org/ijisa/ijisa-v7-n11/IJISA-V7-N11-3.pdf

[3] https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6