Inteligência Artificial

Algoritmos de procura aplicado ao VectorRace

João Almeida - A95191
Daniel Du - A97763
Afonso Amorim - A97569


João Almeida


Daniel Du


Afonso Amorim

Grupo 50
30 de novembro de 2022

Conteúdo

1 2ª Fase

1.1 Introdução

Este relatório surge na resolução da segunda fase do exercício em grupo da Unidade Curricular de Inteligência Artificial.

O exercício proposto basea-se na elaboração de algoritmos de procura para a resolução de um jogo, nomeadamente o *VectorRace*, também conhecido como *RaceTrack*.

Nesta fase do projeto, iremos desenvolver um circuito com N participante e encontrar vários caminhos, através dos algoritmos de procura em largura, em profundidade, greedy e o A*.

Para além do que já foi enunciado, também melhoramos implementações que tinham ficado pendentes na primeira fase.

Para a resolução deste exercício, usaremos a linguagem *Python*.

1.2 Descrição do problema

Nesta fase do projeto foi-nos pedido para criar mais circuitos VectorRace e mais complexos, a representação de uma pista em forma de grafo e desenvolver estratégias de procura informada ou não informada.

A pista é um conjunto de linhas e colunas, onde temos diversas posições (linha,coluna), o que facilita bastante o uso de vetores.

1.2.1 Legenda de um circuito

- X representam as paredes
- P representam a posição inicial
- F representam a posição de chegada
- \- representam caminho possível

1.2.2 Regras

- O carro pode acelerar -1, 0 ou 1 unidades em cada direção (linha,coluna).
- Se o carro sair da pista, o carro terá de voltar à posição anterior, assumindo um valor de velocidade igual a 0, aumentando em 25 o custo.

1.3 Formulação do problema

1.3.1 Carro

Criamos uma classe carro, em que cada carro tem como paramentros 2 tuplos, um é a velocidade e o outro é a posição do carro. O carro é usado como um nodo no grafo de listas de adjacencia.

1.3.2 Estado inicial

O estado inicial é representado por um carro na posição inicial do mapa, isto é, onde está o carater *P*, e com uma velocidade nula.

1.3.3 Estado final

Para representar um estado final utilizamos uma lista de coordenadas onde se encontra o carater 'F' no mapa. Conseguimos saber se um carro se encontra no estado final quando as suas coordenadas tiverem na lista de coordenadas finais.

1.3.4 Calcular a Próxima posição

O programa calcula a próxima posição do carro, dada uma acelaração e tendo em conta as fórmulas apresentadas no enunciado (velocidade e posição).

1.3.5 Próximo estado

Operação que avalia todas as hipóteses possíveis para a próxima posição. A função ***nextestados*** devolve uma lista carros nas próximas posições possíveis. Primeiro calculamos todos vetores acelarações existentes e após calcularmos as próximas posições e velocidades possíveis filtramo-las pelas condições:

- Carros que coordenadas estão numa parede
- Carros que passam por cima de paredes

1.3.6 Algoritmos usado para verificação de colisão de paredes

Para verificar a colisão do carro com paredes a meio do caminho, calculamos a interseção de dois caminhos com paredes, cuja decomposição vetotiral da diferença das posições fazem 2 triangulos como se ve na figura:

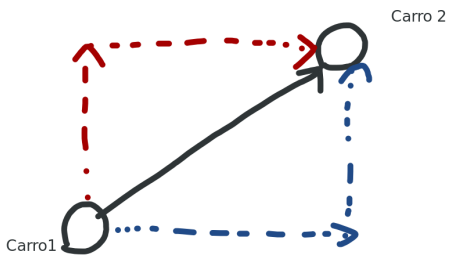


Figura 1: Pista 1

Sendo que o *carro 1* representa a posição inicial de um carro e o *carro 2* é a posição final desse mesmo carro depois de mover. Calculamos a interseção da fronteira do triangulo azul com paredes e o triangulo bordô com paredes. Se ambas intersetarem paredes admitimos que o carro não pode deslocar-se. Basta haver um dos triangulos que não intersete paredes para considerar a deslocação válida.

Quando geramos o grafo, temos que, chamar este algoritmo muitas vezes e por isso foi alvo de otimizações.

Para otimizar este algoritmo, já na segunda parte, começamos por verificar se a posição do destino era uma parede e só depois verificar se o caminho até lá intersetava paredes. Só esta pequena mudança notou-se subtancialmente, passando a gerar, por

exemplo o grafo da pista 3, em 1min e 20s em vez de 2min e 40s como tínhamos na primeira fase.

Como ainda não nos encontrávamos satisfeitos, tentamos otimizar ainda mais usando a função *extends* uma vez que esta função adiciona multiplos elementos muma lista, em vez de iterarmos as posições e adiciona-las numa lista uma de cada vez como estávamos a fazer. Desta forma, passou a gerar o grafo aproximadamente em 47s.

1.4 Representação da pista em forma de grafo

Para representar um grafo utilizamos listas de adjacencia e construímos um grafo que associa um carro a uma lista de outros carros.

Para fazermos este dicionário de listas de adjacência partimos do estado inicial e calculamos os estados seguintes e as suas respetivas heurísticas.

No grafo, o nó inicial é a chave para os valores dessa mesma lista e continuamos o processo para esses estados possíveis, para além disso, mantivemos guardado os nós que já tinhamos visitado de forma a não os repetir.

Até não haver mais nodos a visitar repetimos o processo.

Por motivos de rapidez e comodidade, em vez de estar sempre a recalcular e a gerar um grafo novo sempre que corremos o programa, optamos por guardar o grafo gerado num ficheiro *pickle* para posterioremente obtermos o grafo instantaneamente.

1.5 Custo da solução

O custo é calculado na função que gera os próximos estados possíveis. Cada movimento de um carro tem 1 unidade ao custo e, caso ele saia da pista, são acrescentadas 25 unidades.

1.6 Heurística

Para cada nodo estamos a usar uma heurística que é a distância em linha reta dessa corrdenada até ao fim.

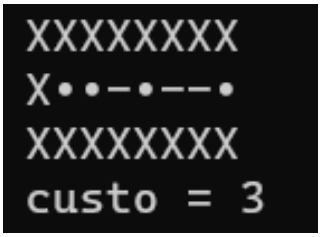
1.7 Pistas e análise de resultados

Nas imagens em baixo estão as pistas que usamos para testar os nossos algoritmos e as pistas após aplicar a função BFS, imagemdo lado esquerdo, e a função e a DFS, imagem do lado direito, onde o carater '•' representa as posições por onde o carro

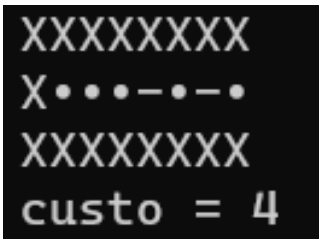
A pista 1, representada na figura 2, como podemos ver é uma pista muito básica, é só uma linha reta.

Para este circuito, o tempo de geração do grafo e a execução do algoritmo é quase instantâneo.

Figura 2: Pista 1



BFS aplicada à pista 1



DFS aplicada à pista 1

De seguida, a pista 2, representada na figura 3, embora ainda sem paredes ou obstáculos no meio do circuito, já implica o carro andar em diferente direção e sentido.

Para este circuito tempo de geração do grafo e correr o algoritmo é aproximadamente 1 segundo.

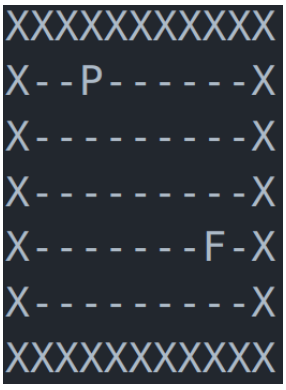
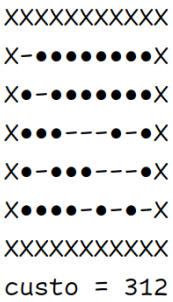


Figura 3: Pista 2

BFS aplicada à pista 2



DFS aplicada à pista 2

A pista 3, a mais complexa utilizada até ao momento, já implica maior dificuldade na geração do grafo.

Para este circuito tempo de geração do grafo e correr o algoritmo é aproximadamente 47 segundos.

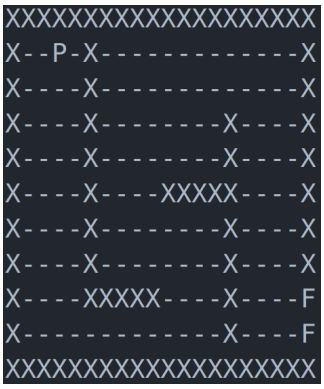
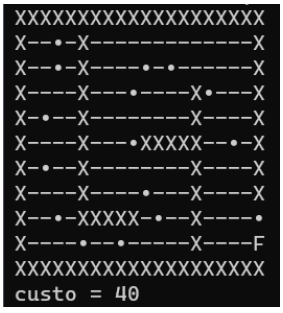
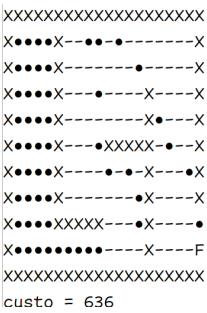


Figura 4: Pista 3



BFS aplicada à pista 3



DFS aplicada à pista 3

Também criamos uma pista ainda mais difícil, a pista 5, demorou 14min e 27 segundos a gerar o grafo. Não conseguimos executar a função BFS porque como é grafo é muito grande excedemos o limite de chamadas recursivas que o python nos deixa fazer.

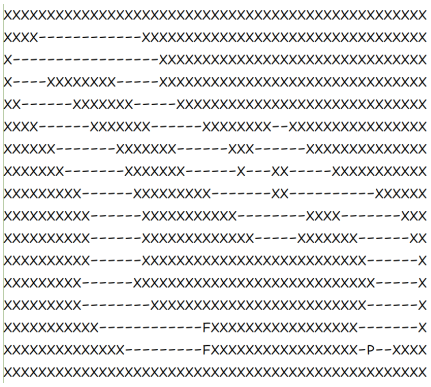
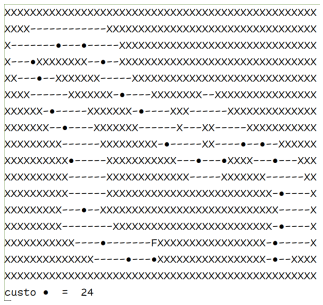
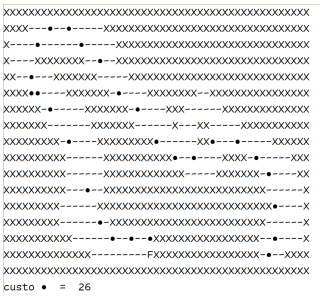


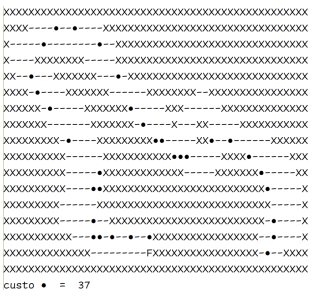
Figura 5: Pista 5



BFS aplicada à pista 5



A* aplicada à pista 5



Greedy aplicada à pista 5

1.8 Multiplos carros no sistema

Num ambiente competitivo estamos a lidar com as colisões da seguinte forma:

- Sempre que um carro deseja ir para a posição onde está outro carro, este ficará no mesmo sitio com velocidade nula.

1.8.1 Como funiona o algoritmo

Para gerar vários caminhos para vários carros temos os seguintes passos:

1. Calcular o caminho para todos os carros até ao fim
2. Percorrer iterativamente todos os carros e ver se a primeira posição para que queriam ir já está a ser ocupada pelos anteriores
   2.1. Caso a posição final já esteja ocupada o carro fica parado.
   2.2. Depois disso guarda-se a primeira posição de cada solução de cada carro como sendo defenitiva e repete-se o processo até chegar ao fim

1.9 Interface

Para representar diferentes carros usamos os seguintes carateres redondos:

Figura 6: carateres

Neste momento só temos 12 carateres por isso teoricamente só temos forma de desenhar 12 carros, mas o nosso programa consegue ter N carros possíveis quantos a pista aguentar.

Criamos uma função de *pretty print* que mostra iteração a iteração o percursos dos vários carro, o que nos permite realizar uma análise das corrida.

1.9.1 Análise de uma corrida iteração a iteração

Exemplo de correr uma função BFS no grafo 3 com 3 jogadores ao mesmo tempo:

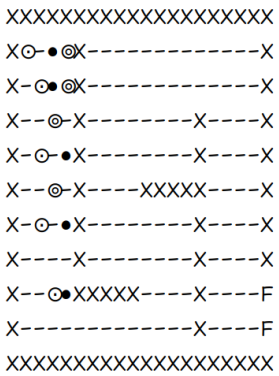


Figura 7: 1ºiteração

Para a iteração seguinte ambos os carros querem ir para a mesma casa.

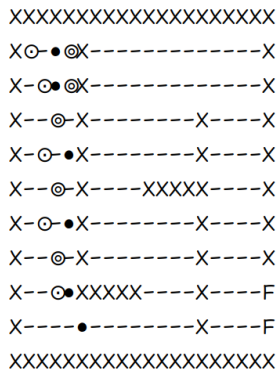


Figura 8: 2ºiteração

Com o conflito de destinos, um dos carros ficou no mesmo síteo que na iteração anterior, ao mesmo tempo que o outro carro andou para a frente.

Podemos ver que na iteração seguinte o carro que estava parado vai 1 casa para a direita, entrou novamente em disputa mas agora com o carro que estava mais atrás. Desta vez o o carro mais atrás vai parar para deixar o outro passar.

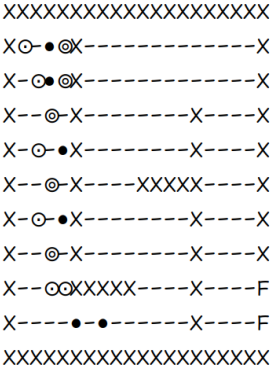


Figura 9: 3ºiteração

A partir deste momento os carros foram um atrás dos outros e percorreram todos o mesmo caminho.

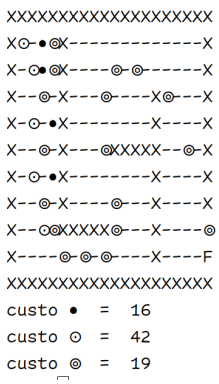


Figura 10: Fim

Para introduzir vários carros na pista basta editar o ficheiro da pista e colocar o carater *'P'* onde queisermos identificar um carro. Se tivermos a correr a main não é preciso reniciar-la.

1.10 Discussão dos resultados obtidos

No nosso trabalho a parte que mais custa computacionalmente é gerar o grafo do nosso problema, fazemos bruteforce de todas os estados possíveis, gasta principalmente muita memória e tempo. Apesar de guardarmos o grafo num pickle é uma coisa que se tem que melhorar

Para o nosso trabalho decidimos implementar algoritmos de procura não informada*BFS (Breath First Search)* e *DFS (Depth First Search)* e procura informada *Greedy Search* e *A* Search* .

1.10.1 BFS

O *BFS* ocupa muita memória e é muito demorado para problemas de grandes dimenções. Neste contexto o algoritmo percorre em todas as direções ao mesmo tempo cada nível de cada vez ate chegar ao destino e apresenta a solução com menos etapas para chegar ao destino

1.10.2 DFS

O *DFS* ocupa pouca memória para problemas com muitas soluções, a desvantagem é que a sua solução não é ótima, apresenta a primeira solução que encontrar seja esta boa ou má. Neste problema o algoritmo escolhe sempre um caminho "aleatorio"a percorrer até chegar ao destino.

1.10.3 Greedy

O *Greddy* se tiver uma boa heurística o tempo de execução pode diminuir consideravelmente. No contexto do nosso trabalho, em cada iteração o carro escolhe qual a melhor jogada possível com base na heurística. O problema deste algoritmo é que no final da sua execução, pode não ter chegado à solução ideal, contudo, de forma geral, tendo em conta a nossa implementação consigos obter um bom tempo e solução.

1.10.4 A*

O $A*$ Não tem garantia de encontrar o caminho mais rápido. Faz um escolha de caminho com o equilíbrio da heurítica e do custo de determinao caminho, a heurítica tem o papel central deste algoritmo. É mais eficiente que algoritmos brute force como é caso BFS e também nao gasta tanta memória a expandir para todos os nosos.

1.11 Conclusão

Em suma, mesmo tendo em conta as adversidades que fomos encontrando ao longo do desenvolvimento do projeto, consideramos que fomos capazes de contornar os obstáculos que foram aparecendo.

Achamos que fizemos um bom trabalho mas que tínhamos espaço por onde melhorar, por exemplo, em termos de organização de código, o facto da nossa função BFS com a existência de mais de 1 carro, em simultaneo, não funcionar, e a documentação.

Este trabalho, após a simulação da corrida e a procura do caminho mais curto, ajudou-nos a entender na prática o funcionamento de diversos algoritmos e a tirar conclusões de eficiência e exatidão a partir de comparações entre estes algoritmos.