Inteligência Artificial

Vector Race

LEI - 2022/2023

Grupo 24

Rui Silva - a97133

Telmo Oliveira - a97018

Hugo Novais - a96267

Universidade do Minho

Índice

List of Figures

1 Introdução

No âmbito da Unidade Curricular de Inteligência Artificial foi-nos proposto a realização dum jogo chamado VectorRace, com o intuito de aplicarmos diversos algoritmos de procura para a resolução do mesmo.

O VectorRace, também conhecido por RaceTrack, trata-se de um jogo de simulação de carros simplificado, que contém um conjunto de movimentos e regras associadas.

Nesta primeira fase tínhamos como objetivo primário o desenvolvimento de um circuito com pelo menos um participante a encontrar o caminho mais curto até à meta, com recurso a um dos algoritmos de pesquisa informada ou não informada.

2 Funcionamento do Jogo

Os movimentos do carro no jogo são bastante simples, uma vez que as ações desenvolvidas são equivalentes a um conjunto de acelerações.

Em concreto, cada jogada tem um custo de 1 unidade e um carro pode assumir três valores de aceleração -1, 0 e 1 unidades em cada direção (linha l e coluna c), ou seja, representamos a aceleração nas duas direções em cada instante por **a = (al, ac)**.

Ao considerarmos **p** como um tuplo que indica a posição de um carro numa determinada jogada **j (pj = (pl, pc))**, e **v** o tuplo que indica a velocidade do carro nessa jogada **(vj = ((vl,vc))**, na próxima jogada o carro irá estar na posição:

$$p_l^{\,j+1} = p_l^{\,j} + v_l^{\,j} + a_l$$

$$p_c^{\,j+1} = p_c^{\,j} + v_c^{\,j} + a_c$$

Figura 1: *Expressões relativas ao tuplo que indica posição de um carro.*

A velocidade dum carro num determinado instante é calculada da seguinte forma:

$$v_l^{\,j+1} = v_l^{\,j} + a_l$$

$$v_c^{\,j+1} = v_c^{\,j} + a_c$$

Figura 2: *Expressões relativas ao tuplo que indica velocidade de um carro.*

Como se trata de uma simulação de carros, temos ainda a possibilidade de o carro sair da pista e, neste caso, o carro terá de voltar para a posição anterior, assumindo um valor de velocidade igual a zero. Ainda neste caso o custo de sair dos limites da pista, em vez de 1 unidade será de 25.

No que diz respeito à representação do circuito em si, decidimos optar pelo mais simples e criar o nosso circuito diretamente num ficheiro .txt sem utilização de quaisquer outros meios aplicacionais.

O circuito que criámos é então o seguinte:

```
XXXXXXXXXXXXXX
XXX-----XX--XX
X-P-XX-----XXX
X--X---XX----X
XX----X----XXX
XX--XX----XXXX
XX---XX---X--F
X--XXX---X---F
X-XXXX---X---F
X--XXX--X--X-X
X---X--X--XX-X
X---XX----X--X
XX--X---XXX--X
X------XXXXXXX
XXXXXXXXXXXXXX
```

Figura 3: *Circuito criado para o projeto.*

Este circuito apresenta 15 linhas e 15 colunas. O circuito é representado por diferentes caracteres sendo que '-' representa a pista, o 'X' representa obstáculos ou a zona fora da pista, o 'P' a posição inicial e 'F' as posições finais (meta).

3 Tarefas e decisões

Como para esta fase do trabalho decidimos que a interface seria em formato de texto, no terminal, a primeira implementação foi da interface dada pelos docentes com algumas modificações para ser possível escolher entre algoritmos, mudar ficheiro da pista, etc.

Em seguida, criamos duas classes: Grafo e Nodo.

Começando pela classe Nodo. Nesta, ao inicializar um novo nodo é fornecido um tipo (pode ser um X, -, P ou F) e as coordenadas do mesmo. As funções incluídas nesta função são construtores portanto foi de certa maneira simples de se implementar.

A classe Grafo é mais complexa. Ao inicializar um Grafo, este contém: uma lista de nodos; um dicionário que guarda a informação de todas as ligações do grafo; uma variável para saber se o grafo é ou não direcionado e o tipo do nodo inicial e final do grafo.

A primeira função chama-se *addEdge()* que dado dois nodos adjacentes, adiciona ambos à lista de nodos do grafo e ao dicionário como sendo uma nova ligação.

Usando dois *loops* e a função *addEdge()*, a função *addEdges()*, dado um circuito, usa o primeiro *loop* para percorrer as linhas e o segundo para percorrer cada nodo da linha e assim adiciona todas as ligações possíveis no circuito ao grafo.

A função *getArcCost()*, dado dois nodos, vai ao dicionário do grafo e compara o segundo nodo com as ligações existentes do primeiro. Quando uma das ligações corresponde ao nodo, devolve o custo da ligação.

Com auxílio da função *getArcCost()*, a função *calculateCost()* soma os valores das ligações todas de uma lista de nodos, devolvendo o valor de custo total dessa lista.

Também criamos uma função *getStart()* para devolver o ponto de partida do circuito ao receber o tipo do mesmo. Para funcionar, só pode haver uma instância desse tipo no circuito.

Dentro da classe Grafo também estão definidas as funções de procura

4 Algoritmos utilizados

Com o objetivo de determinar os diferentes caminhos possíveis entre a partida e a meta do nosso circuito, procedeu-se à implementação dos algoritmos de procura que explicaremos com mais detalhe em seguida.

4.1 BFS - Breadth-first search

Um dos algoritmos que aplicámos para a resolução do jogo foi o Breadth First Search. Este algoritmo de pesquisa consiste essencialmente em visitar todos os nodos a uma distância n do nodo inicial antes de visitar os nodos a uma distância n + 1, ou superior.

Para este projeto separamos os vértices que vamos encontrando em diferentes estruturas, sendo estas, um set() que guarda a informação dos nodos que já foram visitado para evitar que o mesmo seja processamdo mais do que uma vez,uma **queue**,na qual vão sendo armazenados os nodos adjacentes a um nodo que ainda não tenha sido visitado de maneira a serem visitados em iterações futuras do algoritmo. Por último guarda-se também os nodos-pai dos nodos que já foram visitados num dicionário, com o intuito de ao alcançar o objetivo, construir o caminho percorrendo a lista dos pais de cada nodo até chegar ao nodo inicial, ou seja, o nodo que não possui pai, de maneira que ao reverter esta lista tenhamos o melhor caminho construído. Com o caminho construído também calculamos o custo do mesmo, utilizando a nossa função calculateCost().

4.2 DFS - Depth First Search

O segundo algoritmo que implementamos para a resolução do jogo foi o Depth-first search. O algoritmo de pesquisa em profundidade, como se denomina em português, trata-se de um algoritmo que aprofunda a procura tanto quanto possível, isto é, até encontrar um nodo objetivo ou sem filhos e só neste caso é que procede à pesquisa pelos restantes. No nosso projeto decidimos implementar este algoritmo de forma recursiva, uma vez que seria bastante mais simples. Desta forma, na nossa implementação usamos duas estruturas auxiliares, um array de maneira a ir guardando o caminho percorrido desde o nodo inicial até ao nodo objetivo e um set onde se guarda os nodos visitados em cada iteração do algoritmo. O método funciona da seguinte forma, primeiramente ambas as estruturas guardam o nodo inicial e em seguida verifica-se se este é o nodo objetivo e caso o seja, a função retorna um tuplo constituído pelo caminho que obteve até então e o custo associado, respetivamente. Caso o mesmo não se verifique, o método prossegue e para cada nodo adjacente ao nodo inicial, ainda não visitado, o método é chamado recursivamente, mas neste caso, o nodo "inicial" passa a ser o nodo adjacente que se está a considerar.

Por fim, quando se atingir então o nodo objetivo a função retornará um tuplo com o caminho obtido e o seu custo associado como mencionado anteriormente.

4.3 Uniform-cost Search

O algoritmo de procura de custo uniforme (UCS), trata-se de um algoritmo que age de forma semelhante ao algoritmo de procura primeiro em largura com a

pequena diferença que este considera sempre que o próximo nodo a ser expandido será aquele que apresentar menor custo em relação ao nodo inicial.

No que toca à nossa implementação no projeto, primeiramente guarda-se numa lista um tuplo com o nodo inicial e com o custo em relação à origem, respetivamente, que neste caso será 0. De seguida criamos outra lista denominada visited que irá guardar os nodos já visitados e guardamos também nesta o nodo origem. Por último, à semelhança do algoritmo BFS criamos um dicionário que irá guardar os nodos pai e colocamos como pai do nodo origem None. Após estas atribuições iniciais procedemos à procura do nodo objetivo. Esta procura termina assim que encontrarmos o nodo pretendido ou assim que não haja mais tuplos a considerar na primeira lista criada, o que significaria que o nodo objetivo não está no grafo. Durante cada iteração o método procura na lista de tuplos o nodo com o menor custo desde a origem e ao encontrar verifica se este é o nodo objetivo, se não for, são colocados na lista visited todos os nodos adjacentes que ainda não foram visitados, assim como na lista de tuplos, juntamente com o seu custo desde a origem. No dicionário de nodos pai todos os nodos adjacentes ao nodo em consideração ficam com este como nodo pai. Quando o nodo objetivo é finalmente encontrado, a procura termina e vai-se percorrendo a lista de nodos pai desde o nodo objetivo até ser encontrado o None do nodo inicial. Ao mesmo tempo que se percorre o dicionário coloca-se num array denominado path os nodos considerados até chegar à origem, de maneira que ao reverter este array tenhamos o caminho obtido. Por fim calcula-se também o custo deste caminho com recurso à função calculateCost().