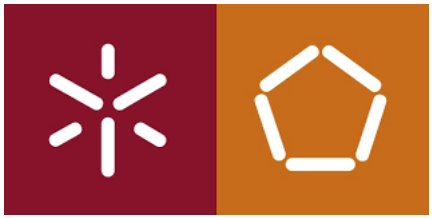

Universidade do Minho

Licenciatura em Engenharia Informática

Ano Letivo 2022/23

Inteligência Artificial

Entrega da Segunda Fase

Grupo 009

Ana Rita Poças (A97284) Inês Ferreira (A97372)
Joana Branco (A96584) João Braga (A97368)

20 de dezembro de 2022

Índice

1 Introdução

Este relatório surge no âmbito da Unidade Curricular de Inteligência Artificial em que nos foi proposto o desenvolvimento de diversos algoritmos de procura para a resolução do jogo *VectorRace*.

O projeto encontra-se dividido em duas fases, e, na primeira fase foi requerido a existência de pelo menos um circuito com um participante do jogo a encontrar o caminho mais curto, com o algoritmo de pesquisa que elegemos.

Já, nesta segunda fase, é suposto desenvolvermos um programa *VectorRace* com maior complexidade, e, assim, foi necessário implementarmos os dois tipos de procura, informada e não informada. Os algoritmos de procura informada tem um custo mais baixo, consomem menos tempo, entre outras vantagens relativamente aos algoritmos não informados.

Nos informados recorremos ao Greedy e A* e nos não informados ao BFS e DFS, sendo que foram todos aprendidos e abordados nas aulas práticas e teóricas da Unidade Curricular. Foi nos proposto, também, a implementação de mais que um jogador e, assim, criar um ambiente competitivo.

2 Descrição do Problema

O *VectorRace* é um jogo de simulação de carros simplificado, que tem um conjunto de movimentos e regras que lhe estão associadas.

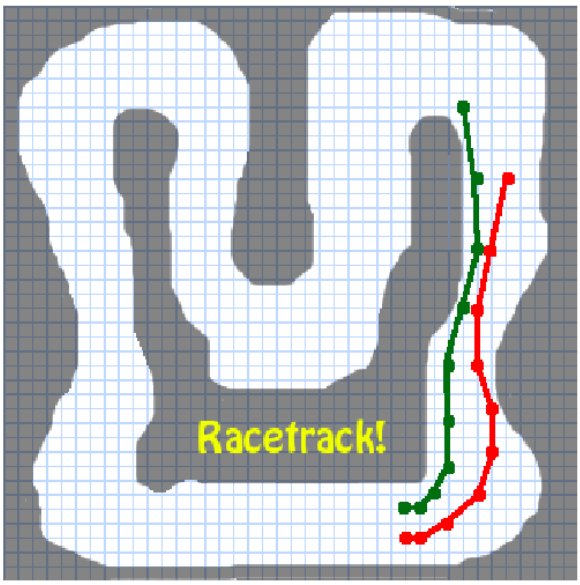

Figura 1: Exemplo de um circuito *VectorRace*

O circuito pode ser tomado como uma matriz que apresenta linhas e colunas, assim sendo num determinado instante o jogador estará numa posição p=(pl,pc). Desta forma, para se movimentar, num determinado instante, o carro pode acelerar -1,0 ou 1 unidades em cada direção logo, temos como acelerações possíveis -1,0,+1, com a=(al,ac) a representar a aceleração de um carro nas duas direções (linha e coluna) num determinado instante.

Sendo p=(pl,pc), o tuplo que representa a posição do jogador num determinado instante, para além da aceleração, o jogador deve ter associado a si um tuplo v=(vl,vc) que indica a velocidade do carro nessa jogada, que deve ser tido em consideração antes de efetuar a próxima jogada.

Sendo este jogo, uma simulação de carros existe a possibilidade de o carro poder sair dos limites da pista, e nesse caso o carro assume velocidade 0, tal como acontece no arranque do programa.

Seguindo as indicações do enunciado, as próximas posições do jogador devem ser calculadas a partir da sua aceleração, velocidade atual e posição atual. Isto pode ser traduzido pela seguinte fórmula: próxima posição = (al + vl + pl, ac + vc + pc) onde al indica a aceleração da linha, vl a velocidade da linha e pl a velocidade da linha, e assim sucessivamente para as

colunas.

Cada movimento do carro numa jogada, de uma posição para outra, está associado a um custo de 1 unidade, à exceção do caso em que sai dos limites da pista (ir contra uma parede), que terá o um custo de 25 unidades.

Um circuito no nosso problema terá a seguinte representação:

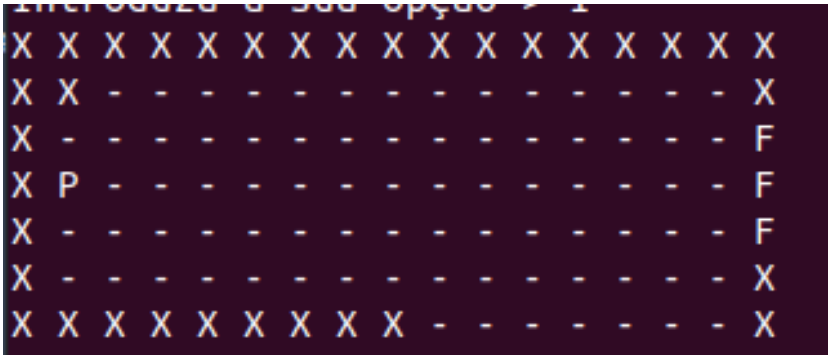

Figura 2: Representação de um circuito no nosso programa

Uma pista terá assim 4 elementos possívies:

- **"P"** que representa a posição inicial;
- **"F"** que representa a posições finais;
- **"X"** que representa um obstáculo/fora da pista;
- **" - "** que representa a pista;

3 Descrição das Tarefas Realizadas

3.1 Reformulação do Problema

Após a entrega da primeira fase e, uma atenta análise ao estado do nosso programa, decidimos alterá-lo da seguinte forma:

- Decidimos que a estratégia anteriormente adotada, nos algoritmos, de representar as posições (x,y) do nosso circuito como xy não era uma opção viável para esta nova fase, visto que faz com que seja mais difícil de perceber as posições, pelo que abandonámos essa estratégia. Para tal, teve de haver uma reformulação na estrutura, isto é, no dicionário que alberga a representação das arestas que fazem a ligação entre os nodos do grafo, ou seja, as posições do circuito.

- Foram usados dois métodos diferentes para a representação do dicionário mencionado anteriormente. No caso dos algoritmos de pesquisa não informada, o dicionário criado começa por ser representado pelas diversas posições do circuito e pela lista de nodos adjacentes destas mesmas posições, considerando apenas o movimento do jogador na horizontal, vertical e diagonal. Já nos algoritmos de pesquisa informada, o grafo inicia a sua formação pela posição inicial do jogador ("P") e todos os seus nodos adjacentes tendo em conta a sua velocidade.

- Ao invés de termos só 1 único circuito disponível no nosso programa, decidimos disponibilizar 4 circuitos para que o utilizador possa escolher o que pretende e assim conseguirmos observar o comportamento dos nossos algoritmos para vários circuitos.

- A nível de algoritmos não-informados, anteriormente, tínhamos implementado o algoritmo de Pesquisa Uniforme, porém nesta fase, optámos por implementar os algoritmos BFS (Breadth First Search) e DFS(Depth First search).

3.2 Algoritmos Implementados

3.2.1 Pesquisa não informada

Os algoritmos de Pesquisa Não Informada implementados nesta fase, foram os seguintes:

- **Pesquisa BFS - Breadth First Search**

- **Pesquisa DFS - Depth First search**

A implementação destes algoritmos foi feita no módulo *naoInformados.py*. Aqui inicializamos o grafo, através da invocação da função *arestinhasParaGrafo*, definida em *arestas.py*. De forma a aplicarmos o **algoritmo DFS** ao nosso grafo, recorremos à implementação da função *procura_DFS* que recebe a posição inicial (sob a forma de um tuplo (x,y)) do jogador ("P"),

uma posição final ("F") (sob a forma de um tuplo (x,y)) e uma lista vazia *path* e um *set* vazio. A partir destes argumentos, aplicamos o algoritmo DFS, isto é, escolhemos o primeiro elemento da lista de estados não expandidos e adicionamos extensões de caminho à frente da lista de estados não expandidos.

De forma a sabermos qual a posição final que a função deveria receber, isto é, qual das metas (posições representadas por "F") é a mais adequada, decidimos recorrer à função *verificaDFSfins* que, quando aplicada à lista que contém todas as posições do circuito representadas por "F", nos permite avaliar qual das posições finais seria menos custosa.

De forma a verificar-se que o jogador está a ir contra uma parede, custo do nodo é 25, este deve regressar para trás, onde apenas adicionamos o nodo à lista de nodos percorridos e regressamos ao nodo que lhe é adjacente, sendo que o nodo parede não é válido para a obtenção do caminho mais curto. No final, imprimimos o caminho devolvido pela pesquisa com o correspondente custo, bem como o caminho percorrido para lá chegar, isto é, os nodos percorridos.

De forma semelhante, para conseguirmos implementar o **algoritmo BFS** no nosso grafo, implementamos a função *procura_BFS* que recebe a posição inicial (sob a forma de um tuplo (x,y)) do jogador ("P") e uma posição final ("F") (sob a forma de um tuplo (x,y)). E, com estes argumentos, aplicamos o algoritmo começando por escolher o primeiro elemento da lista de estados não expandidos, e adicionamos extensões de caminho no final da lista de estados não expandidos. Mais uma vez, de forma a sermos capazes de decidir qual a melhor posição final, recorremos à função *verificaBFSfins* que, determina qual a posição final, de entre as disponíveis, que obteria o caminho menos custoso.

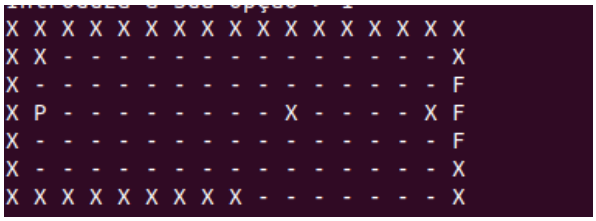

Figura 3: Representação de um circuito1 no nosso programa

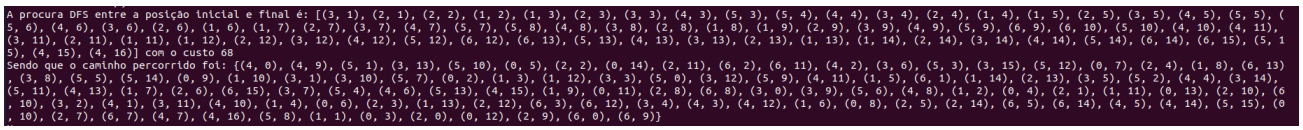

Figura 4: Caminho obtido pelo algoritmo DFS, juntamente com os nodos percorridos para o circuito1

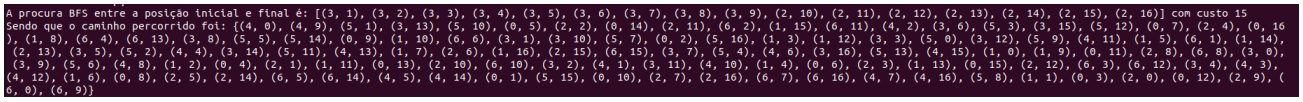

Figura 5: Caminho obtido pelo algoritmo BFS, juntamente com os nodos percorridos para o circuito1

3.2.2 Pesquisa informada

Os algoritmos de Pesquisa Informada implementados nesta fase, foram os seguintes:

- **Pesquisa Greedy**
- **Pesquisa A***

A implementação destes algoritmos foi feita no módulo *informados.py*. Houve também a necessidade de criação de um outro módulo *dictionary.py* onde é construído o grafo como um dicionário. A função *makeGrafo* permite a representação dos nodos do grafo a partir da posição inicial do jogador. Aqui é que são calculadas todas as próximas posições possíveis de um dado jogador tendo em conta a sua velocidade, verificando se as mesmas são pista, parede ou se estão dentro dos limites do circuito.

Atendendo que nos diversos circuitos temos várias posições finais, achamos crucial que o algoritmo seja calculado para essas mesmas posições e apenas é apresentado o que termina com menor custo. Esta observação pode ser tida em conta nas funções *mGreedy* e *mAStar*.

A implementação do **algoritmo Greedy** passou por se recorrer à função *makeHeuristicas* de maneira a ser possível calcular as diferentes heurísticas possíveis, ou seja, a distância em linha reta de qualquer ponto da matriz até à posição final. Para isto, implementamos a *distanciaEuclidiana*. Posto isto, vai se procurar o nodo com menor valor de heurística dos nodos visitados mas com adjacentes que ainda não foram tidos em consideração.

Considerando a velocidade, se o nodo que estivermos a analisar não for adjacente ao seu suposto antecessor, é definida a velocidade do jogador como (0,0). Caso contrário, é atribuído o valor de velocidade ao jogador através da fórmula do enunciado.

De seguida, é analisada a lista de próximas posições do nodo atual e, como a velocidade permite que o jogador faça saltos para posições longínquas, houve a necessidade de validar cada uma dessas transições. Caso o nodo corrente não esteja em nenhumas das listas que sinalizam os seus visitados, é marcado o seu antecessor.

Quando o nodo visto como atual se encontrar na posição final, o caminho percorrido pelo jogador é reconstruído desde a posição naquele preciso momento até à posição do seu estado inicial, assim como também é calculado o custo do mesmo.

Já o **algoritmo A***, inicia-se com uma estrutura semelhante ao anterior, apresentando apenas diferença no cálculo das suas heurísticas. Estas resultam da soma entre o custo acumulado, desde o nodo inicial até ao nodo corrente, e o valor de heurística calculado, onde em *makeHeuristicas* é apresentada a distância em linha reta de qualquer ponto da matriz até à posição final.

De seguida, é atualizado qual o nodo que corresponde ao valor com menor heurística calculada previamente. Para todos os nodos vizinhos do nodo atual, é verificado se esses mesmo

elementos já foram visitados ou não e se é mais rápido visitar o parent em relação ao seu descendente.

Da mesma forma do Greedy, quando o nodo corrente for o nodo da posição final é reconstruído o seu caminho apartir desse ponto e calculado o custo entre todas as ligações realizadas.

3.3 Comparação Teórica dos Algoritmos

Algoritmos Implementados no nosso programa				
Parâmetros	**BFS**	**DFS**	**Greedy**	**A***
É completo?	Sim	Não	Não	Sim
Complexidade Temporal(pior caso)	O(b^d)	O(b^m)	O(b^m)	nº de nodos
Complexidade Temporal(melhor caso)	O(b^d)	O(b^m)	O(bd)	nº de nodos
Complexidade Espaço(pior caso)	O(b^d)	O(bm)	O(b^m)	nº de nodos
Complexidade Espaço(melhor caso)	O(b^d)	O(bm)	O(bd)	nº de nodos
É Ótimo?	Sim (se todos os custos escalonados forem iguais)	Não	Não	Sim (Se a heurística for admissível)

3.4 Comparação dos Resultados Obtidos

Para o seguinte circuito:

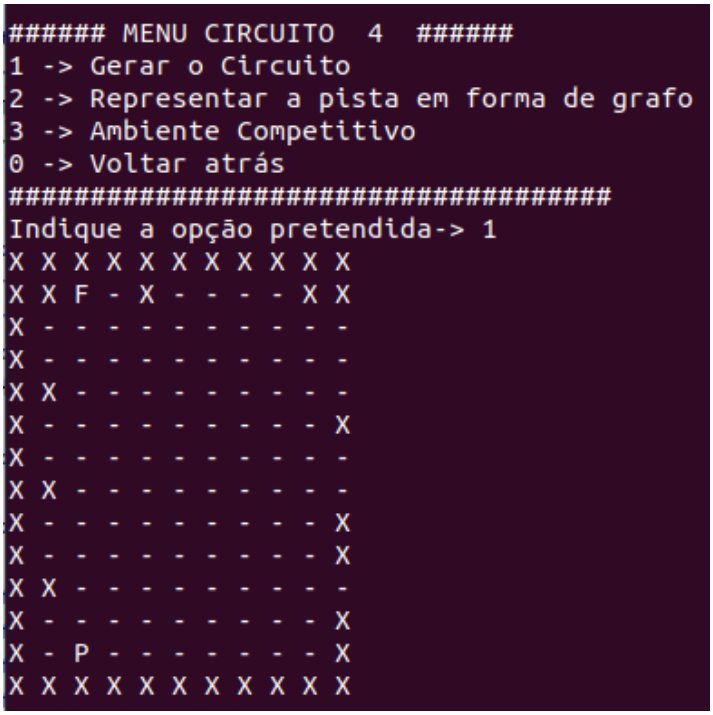

Figura 6: Circuito Exemplo

Quando aplicado o algoritmo DFS, obtemos os seguintes resultados:

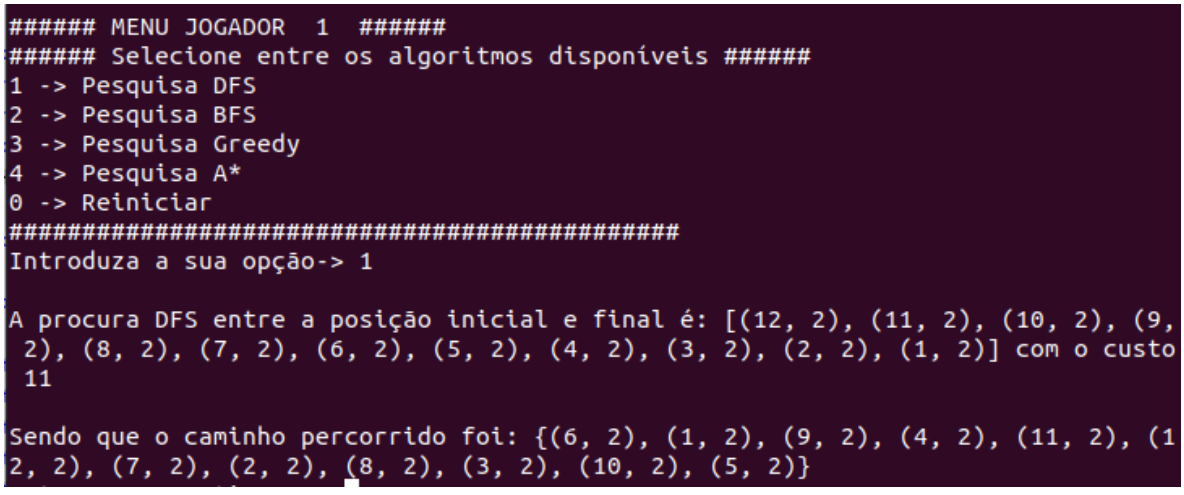

Figura 7: DFS para o circuito

Este algoritmo não é o mais adequado para a procura que pretendemos, isto porque, não

devolve necessariamente o caminho mais curto, mas sim o primeiro que encontra. Ao usar uma stack, DFS pesquisa todos os nodos até não poder mais, mas quando encontramos o nodo objetivo, não temos como garantir que o caminho é o mais curto.

Quando aplicado o algoritmo BFS, obtemos os seguintes resultados:

```
###### MENU JOGADOR  2  ######
###### Selecione entre os algoritmos disponíveis ######
1 -> Pesquisa DFS
2 -> Pesquisa BFS
3 -> Pesquisa Greedy
4 -> Pesquisa A*
0 -> Reiniciar
#############################################
Introduza a sua opção-> 2

A procura BFS entre a posição inicial e final é: [(12, 2), (11, 2), (10, 2), (9, 2), (8, 2), (7, 2),
(6, 2), (5, 2), (4, 2), (3, 2), (2, 2), (1, 2)] com custo 11

Sendo que o caminho percorrido foi: {(12, 4), (4, 0), (4, 9), (5, 1), (8, 0), (5, 10), (8, 9), (10, 6
), (9, 8), (11, 5), (2, 2), (13, 8), (6, 2), (7, 1), (7, 10), (4, 2), (3, 6), (5, 3), (8, 2), (9, 1),
 (9, 10), (11, 7), (2, 4), (13, 1), (13, 10), (1, 8), (6, 4), (7, 3), (3, 8), (5, 5), (8, 4), (9, 3),
 (11, 0), (11, 9), (13, 3), (1, 10), (6, 6), (7, 5), (3, 1), (3, 10), (5, 7), (9, 5), (11, 2), (13, 5
), (1, 3), (7, 7), (12, 6), (3, 3), (5, 0), (5, 9), (9, 7), (11, 4), (10, 8), (13, 7), (1, 5), (6, 1)
, (7, 0), (7, 9), (12, 8), (3, 5), (5, 2), (4, 4), (9, 0), (9, 9), (10, 1), (13, 0), (10, 10), (13, 9
), (1, 7), (2, 6), (7, 2), (12, 1), (12, 10), (3, 7), (5, 4), (4, 6), (9, 2), (8, 6), (10, 3), (13, 2
), (1, 0), (1, 9), (2, 8), (7, 4), (6, 8), (12, 3), (3, 0), (3, 9), (5, 6), (4, 8), (8, 8), (10, 5),
(13, 4), (1, 2), (2, 1), (2, 10), (6, 10), (12, 5), (3, 2), (4, 1), (4, 10), (8, 1), (8, 10), (10, 7)
, (1, 4), (11, 6), (2, 3), (6, 3), (12, 7), (3, 4), (4, 3), (8, 3), (10, 0), (10, 9), (1, 6), (11, 8)
, (2, 5), (6, 5), (12, 0), (12, 9), (4, 5), (8, 5), (10, 2), (9, 4), (11, 1), (11, 10), (2, 7), (6, 7
), (7, 6), (12, 2), (4, 7), (5, 8), (8, 7), (10, 4), (9, 6), (11, 3), (2, 0), (1, 1), (2, 9), (13, 6)
, (6, 0), (6, 9), (7, 8)}
```

Figura 8: BFS para o circuito

O algoritmo BFS é melhor do que o DFS para a procura que pretendemos, visto que o BFS usa uma queue (FIFO) quando percorremos todos os nodos adjacentes camada a camada até ao nodo objetivo, temos como garantir que é o caminho mais curto.

Quando aplicado o algoritmo Greedy, obtemos os seguintes resultados:

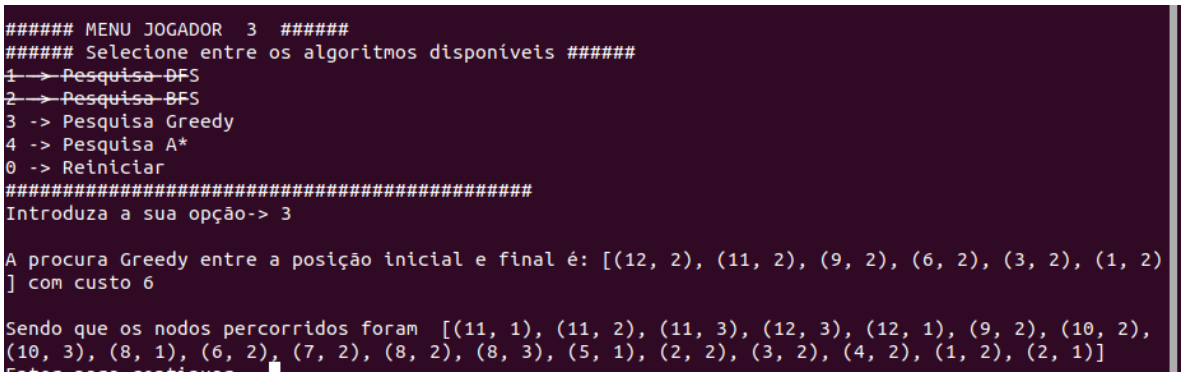

Figura 9: Greedy para o circuito

Visto que o Greedy é um algoritmo informado tem atenção à menor distância ao fim, a cada iteração, o que leva a um caminho mais curto, logo com menor custo a, espera-se que tenha um custo menor que o DFS e BFS, assim como podemos ver pois obtivemos um custo igual a 6, valores inferiores aos anteriores.

Quando aplicado o algoritmo A*, obtemos os seguintes resultados:

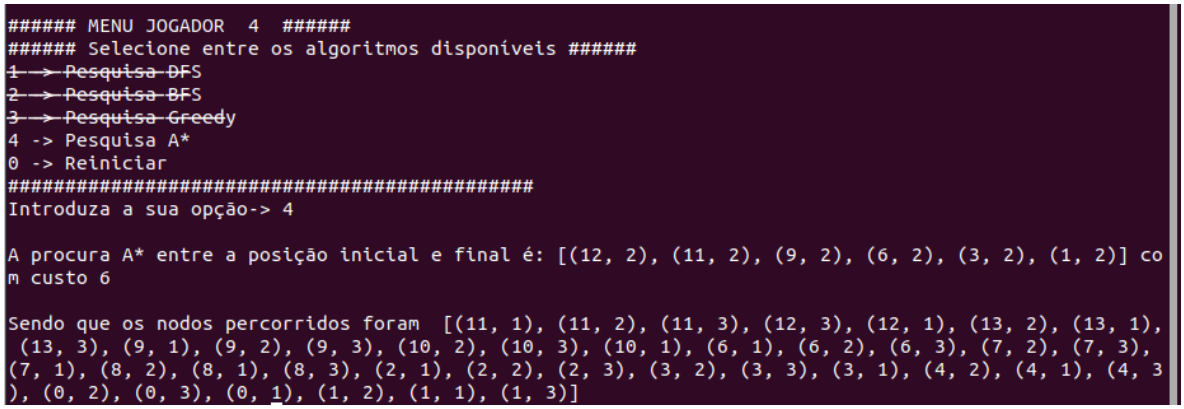

Figura 10: A* para o circuito

O algoritmo A* é melhor do que o outro informado apresentado para a procura que pretendemos, visto que combina o algoritmo Greedy e o de Custo Uniforme minimizando assim a soma do custo do caminho efetuado até ao momento com o menor valor previsto que falta até atingir a posição final. Assim sendo, enquanto que o Greedy apenas estima a menor distância em linha reta até à meta, o A* estima o custo da melhor solução a passar pelo nodo corrente.

Apesar de o A* por norma ser considereado um melhor algoritmo do que o Greedy em alguns casos menos favoráveis para o A*, como visto acima, podem ter o mesmo custo.

3.5 Ambiente Competitivo

Entende-se por ambiente competitivo, a possibilidade de existir mais de um jogador no nosso programa. O grupo teve algumas dificuldades na implementação do conceito no nosso programa. No entanto, implementamos esta funcionalidade da seguinte forma:

```
########## BEM VINDO AO VECTOR RACE  ##########

########## MENU CIRCUITOS ##########
1 -> Circuito 1
2 -> Circuito 2
3 -> Circuito 3
4 -> Circuito 4
0 -> Sair
####################################
Indique a opção pretendida-> 4

###### MENU CIRCUITO  4  ######
1 -> Gerar o Circuito
2 -> Representar a pista em forma de grafo
3 -> Ambiente Competitivo
0 -> Voltar atrás
####################################
Indique a opção pretendida-> 3
Indique o número de jogadores (entre 1 e 4)-> 4

###### MENU JOGADOR  1  ######
###### Selecione entre os algoritmos disponíveis ######
1 -> Pesquisa DFS
2 -> Pesquisa BFS
3 -> Pesquisa Greedy
4 -> Pesquisa A*
0 -> Reiniciar
############################################
Introduza a sua opção-> 1

A procura DFS entre a posição inicial e final é: [(12, 2), (11, 2), (10, 2), (9, 2), (8, 2), (7, 2),
(6, 2), (5, 2), (4, 2), (3, 2), (2, 2), (1, 2)] com o custo 11

Sendo que o caminho percorrido foi: {(6, 2), (1, 2), (9, 2), (4, 2), (11, 2), (12, 2), (7, 2), (2, 2)
, (8, 2), (3, 2), (10, 2), (5, 2)}
```

Figura 11: Excerto 1 do menu para ambiente competitivo

Como é possível observar na figura acima, após selecionar um circuito, o utilizador pode efetuar as seguintes operações:

- Gerar o circuito - imprimir o circuito no terminal.
- Representar a pista em forma de grafo - onde é possível ver todas as arestas do grafo e o respetivo custo.

- Ambiente Competitivo.

Selecionando a opção de **Ambiente Competitivo** (3), é pedido que o utilizador introduza o número de jogadores entre 1 e 4. Foi imposta esta limitação para que cada jogador faça apenas um algoritmo, sem que este mesmo algoritmo possa ser repetido por um outro jogador.

Após selecionar o valor que pretende, por exemplo, 4, o utilizador deve selecionar, de entre os algoritmos disponíveis o que pretende que seja aplicado ao jogador indicado, começando no 1. E após a seleção, é apresentado o caminho resultado do algoritmo e o respetivo custo, bem como o caminho percorrido.

Também é possível observar a movimentação do jogador ("P") pelo circuito em questão, permitindo assim uma melhor compreensão visual do caminho resultado do algoritmo.

Para além disto, o algoritmo anteriormente escolhido deixa de estar disponível para o jogador seguinte, como é possível observar:

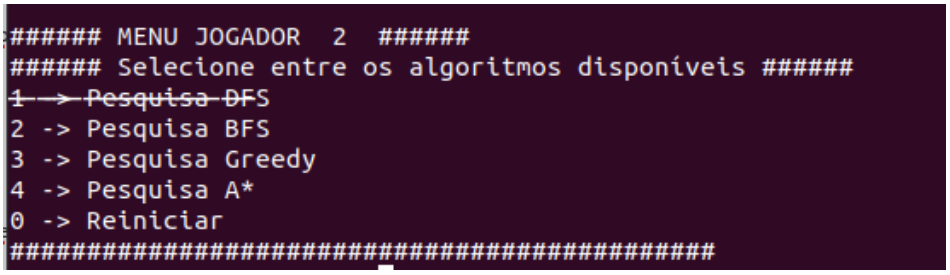

Figura 12: Excerto 2 do menu para ambiente competitivo

Quando todos os jogadores tiverem terminado, o programa sai do ambiente competitivo.

Apesar de considerarmos que esta implementação, está razoável, sabemos que não está totalmente correta, visto que o objetivo era que no caso de 2 jogadores se deslocarem para a mesma posição, fôssemos capazes de escolher qual é que poderia ocupar essa posição. Nesse sentido, o nosso programa apresenta uma falha.

4 Sumário e discussão dos resultados obtidos

Por fim, acreditamos que o nosso trabalho, apesar de não se encontrar perfeito, **está consistente**. Fomos capazes de reformular o programa que havíamos apresentado na primeira fase, e consideramos que **o atual apresenta melhorias visíveis.** Implementamos **4 algoritmos de procura**, dois informados e dois não informados, e desta forma, conseguimos apresentar diversidade na procura existente no nosso programa. Em cada um dos algoritmos, **para além de apresentarmos o caminho mais curto e o respetivo custo, apresentamos também o caminho percorrido ao longo da execução.**

No entanto, apesar de no nosso programa termos disponibilizado o **ambiente competitivo**, consideramos que ele não se encontra plenamente funcional e essa é a falha que consideramos mais relevante no nosso programa.