Universidade do Minho
Escola de Engenharia

Algoritmos de Procura
VectorRace

Trabalho Prático
Inteligência Artificial

Grupo 12

Emanuel Lopes Monteiro da Silva - a95114

Gabriela Santos Ferreira da Cunha - a97393

Miguel de Sousa Braga - a97698

Nuno Guilherme Cruz Varela - a96455


a95114


a97393


a97698


a96455

Conteúdo

1 Descrição do Problema

Para este trabalho prático, foi-nos proposta a implementação de vários algoritmos de procura no contexto do jogo *VectorRace*.

O *VectorRace* consiste num jogo de simulação de carros simplificado em que o carro percorre um circuito predefinido, transitando entre posições definidas por um referencial a duas dimensões.

O carro pode mover-se em qualquer direção, dependendo da aceleração escolhida. Considerando a notação l, que representa a linha e c a coluna para os vetores, num determinado instante, o carro pode acelerar -1, 0 ou 1 unidades em cada direção. Assim, para cada uma das direções o conjunto de acelerações possíveis é {-1, 0, 1} e a é o tuplo que representa a aceleração de um carro nas duas direções nesse instante.

Tendo em conta que p é um tuplo que indica a posição do carro e v um tuplo que indica a velocidade numa determinada jogada j, o movimento do carro rege-se pelas seguintes equações:

$$p_l^{j+1} = p_l^j + v_l^j + a_l$$
$$p_c^{j+1} = p_c^j + v_c^j + a_c$$

$$v_l^{j+1} = v_l^j + a_l$$
$$v_c^{j+1} = v_c^j + a_c$$

Neste problema, tanto a aceleração como a velocidade são grandezas discretas, uma vez que podem apenas tomar valores inteiros.

De forma a simular com o maior realismo possível uma corrida real, é possível que um carro saia da pista, caso em que é introduzido um custo de 25 unidades. Não havendo qualquer situação anormal, o custo de cada movimento, independentemente da deslocação atingida, tem um custo de 1 unidade.

Os circuitos são representados em formato texto, com as posições fora da pista a serem representadas por 'X', os espaços vazios por '-', as posições de partida dos vários jogadores por 'P' e as diversas posições de chegada por 'F'. Na figura 1 é apresentado o exemplo de um circuito criado pelo grupo.

Figura 1: Circuito “Iman“ criado pelo grupo.

Resumindo, o objetivo deste trabalho passa por implementar e comparar diferentes métodos de procura para a resolução do problema apresentado, ou seja, tentar encontrar, no contexto de uma corrida, o melhor caminho para um carro se deslocar desde a casa de partida até à meta.

2 Formulação do Problema

Este é um problema de pesquisa na medida em que queremos encontrar a sequência de ações que nos permite chegar de um estado inicial a um estado objetivo com o menor custo possível. Este problema pode ser formulado, portanto, como um problema de pesquisa num grafo, em que os nodos representam os estados do carro nas diversas fases do seu percurso até ao estado final.

2.1 Estado inicial

Um estado (ou nodo) é caraterizado pela posição (coordenadas x e y) do carro e pela velocidade do mesmo (componentes também em x e em y). Podemos considerar, por isso, que o estado inicial é um estado em que o carro tem uma posição qualquer (representada no mapa) e velocidade 0 nas duas componentes.

2.2 Estado objetivo

Um estado (ou nodo) objetivo é uma dada posição do mapa, sendo caraterizado pelas suas coordenadas x e y. A velocidade neste nodo não é relevante, uma vez que pretendemos apenas que ele chegue a essa posição, independetemente da velocidade com que o faça. Podemos ter vários estados objetivo, marcados por posições 'F' no mapa.

2.3 Operadores

Os operadores de mudança de estado são as possíveis deslocações que o carro pode sofrer, estando dependentes da aceleração que o carro obtém naquele instante. As equações que regem a mudança de estado são as apresentadas na secção 1. Há 9 valores diferentes para a aceleração do carro (3 para a aceleração em x e 3 para a aceleração em y), pelo que, a partir de um estado origem, há, no máximo, 9 estados destino para os quais pode expandir. No entanto, há a possibilidade de escolhas diferentes para a aceleração resultarem num mesmo estado resultante, com posição e velocidade iguais.

Figura 2: Carro na posição (1.5, 1.5)

Por exemplo, na figura acima o carro começa na posição (1.5, 1.5) com velocidade (0,0). O carro pode assim tomar um de 9 deslocamentos (norte, sul, este, oeste, nordeste, noroeste, sudeste, sudoeste e ficar na mesma posição). No entanto, para todos os deslocamentos exceto (+1,0) e (0,0), o carro bate numa parede e volta à posição inicial (1.5,1.5).

2.4 Custo da solução

O custo da solução é dado pela soma dos custos de cada deslocação. Normalmente, o custo de cada deslocação é 1, contudo, esse valor pode ser 25, caso o carro saia da pista na sua deslocação.

3 Descrição das Tarefas Realizadas

3.1 *Parsing* dos circuitos

De modo a fazer a leitura de um circuito, optamos por representar os circuitos em forma matricial, utilizando uma lista de listas para guardar todas as peças do mesmo.

Estas peças são representadas, na matriz, da seguinte forma:

- 'P' (posição inicial) se no mapa se encontra um 'P' nessa posição;
- 'F' (posição final) se no mapa se encontra um 'F' nessa posição;
- 'X' (obstáculo/fora da pista) se no mapa se encontra um 'X' nessa posição;
- '-' (posição livre) se no mapa se encontra um '-' nessa posição.

Implementamos, então, um *parser* que vai ler um circuito de um ficheiro e devolver a matriz do circuito, juntamente com a posição inicial ('P') e as posições finais ('F').

3.2 Criação do grafo

De modo a encontrar um caminho até à posição final, isto é, encontrar uma solução, é necessário gerar o grafo de todos os estados possíveis. Para tal, começamos por definir uma função, explicada mais à frente, que calcula o próximo estado a partir de outro e de um par de aceleração. Com isto, a função de expandir um estado através das 9 possíveis acelerações já pode ser definida e, portanto, o grafo pode ser construído.

O grafo é construído com a ajuda de duas estruturas, uma para guardar os estados e outra para guardar os estados já visitados. Enquanto houver estados na primeira estrutura referida, vamos retirar um estado para ser expandido. Para cada nodo expandido a partir deste estado, criamos uma nova aresta que vai ser adicionada ao grafo. No final, verificamos se o nodo está presente na estrutura dos visitados, de forma a adicionar ou não nessa estrutura. Este último passo é feito para evitar ciclos infinitos na criação do grafo.

3.3 Cálculo do próximo estado dada a aceleração

De modo a gerar o grafo que representa os vários estados possíveis (posição e velocidade) e as transições entre esses estados, começamos por definir uma função auxiliar que, dada uma posição e um deslocamento, calcula a posição final do carro. Esta posição assume que o carro pode deslocar-se na diagonal. Assumimos também que o carro parte do centro de um quadrado e termina no centro de outro quadrado ou do quadrado de onde partiu.

O algoritmo calcula de forma iterativa todos os pontos de fronteira dos quadrados por onde o carro passa.

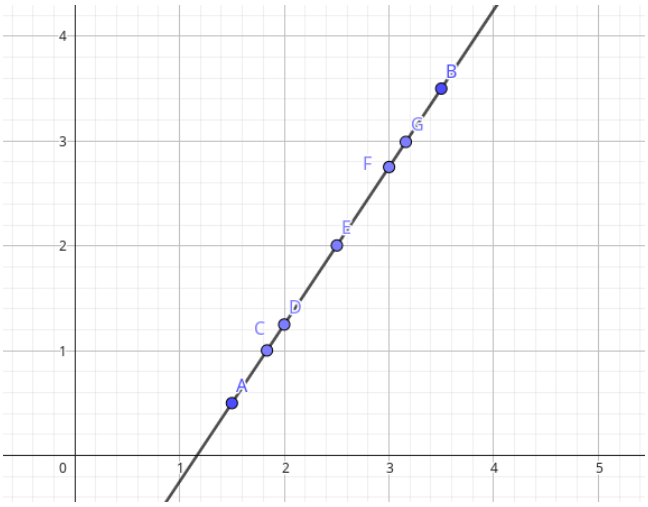


Figura 3: Pontos de fronteira de cada posição

Por exemplo, no deslocamento apresentado na figura, onde o carro parte da posição A(1.5, 0.5) para a posição B(3.5, 3.5), são verificados os pontos C, D, E, F e G.

Para cada ponto, o algoritmo verifica qual é o objeto no próximo quadrado (parede, espaço vazio ou posição final). Ele sabe qual o quadrado a considerar através da posição de fronteira e do deslocamento. Deste modo, caso o objeto encontre uma parede na casa seguinte, mantém-se na casa anterior. Caso encontre uma posição final, pára nessa posição final. Caso a casa seja vazia prossegue o algoritmo. Isto é feito até alguma destas condições de paragem se verificar ou até o deslocamento terminar.

Considerando, por exemplo, o ponto E da reta, o algoritmo deverá verificar qual o objeto existente no quadrado cujo centro é o ponto (2.5, 2.5), testando as condições acima apresentadas.

3.4 Implementação de algoritmos de procura

Como explicitado anteriormente, um dos objetivos do jogo passa por encontrar uma solução para chegar ao final, ou seja, determinar um caminho até às posições finais do circuito. De modo a este caminho ser calculado, temos de implementar algoritmos de procura no grafo.

Deste modo, implementamos algoritmos de procura não-informada (DFS, BFS e Custo Uniforme) e procura informada (Greedy e A*), de modo a construir soluções até às posições finais.

3.4.1 Procura não-informada

Algoritmo *Breadth-First Search*

O algoritmo *Breadth-First Search* é um algoritmo de procura num grafo em largura. Esta estratégia vai começar por procurar em todos os nodos de menor profundidade, isto é, vai percorrer todos os nodos de um certo nível de profundidade e quando todos estes forem percorridos passa a procurar no nível seguinte. Normalmente esta procura demora muito tempo e ocupa muito espaço, pelo que apenas deve ser utilizada em problemas pequenos. Contudo, apresenta sempre soluções com o menor número de nodos e, se o custo de todas as arestas for igual, será também a solução com menor custo.

Algoritmo *Depth-First Search*

O objetivo deste algoritmo passa por expandir o nodo atual sempre num nodo mais profundo da árvore. No entanto, como todos os algoritmos, este tem vantagens e desvantagens. Uma das vantagens é a utilização de pouca memória, o que faz com que seja um algoritmo ideal para problemas com muitas soluções. Por outro lado, este algoritmo não pode ser usado em grafos com profundidade infinita, uma vez que, neste tipo de grafos, entra, por vezes, em caminhos errados.

Algoritmo de Custo Uniforme

Para este algoritmo, é necessário que seja guardado o custo total desde o estado inicial até cada nodo da lista de estados não expandidos. Assim, é sempre expandido o nodo da lista com menor custo. Este algoritmo acaba por se comportar do mesmo modo que o BFS caso os custos dos nodos sejam todos iguais. Este algoritmo apresenta sempre a solução ótima, ou seja solução com menor custo. No entanto, pode não ser a solução com menor número de nodos.

3.4.2 Procura informada

Heurísticas utilizadas

Os algoritmos de procura informada baseiam-se numa heurística para encontrar o próximo melhor nodo, de entre os nodos disponíveis. Assim, neste jogo, consideramos duas heurísticas: a menor distância em linha reta a uma das posições de chegada e o simétrico do módulo da velocidade do carro (de modo a podermos minimizar o valor da heurística). Estes valores são atribuídos a cada um dos estados gerados na criação do grafo.

Algoritmo *Greedy*

O estado escolhido pelo algoritmo *Greedy* é o estado com menor heurística dos estados atualmente expandidos. Este algoritmo recorre, assim, a uma *open list* (conjunto de nodos que podem vir a ser visitados) e a uma *closed list* (conjunto de nodos que já não podem ser visitados). Este algoritmo começa por inicializar a *open list* com o estado inicial, escolhendo em seguida, a cada iteração, o nodo dessa lista com menor heurística, adicionando à mesma, no final, os vizinhos não visitados. O processo é repetido até se encontrar um nodo objetivo.

Algoritmo A*

O algoritmo A* opta por uma estratégia semelhante, no entanto armazena o custo até cada um dos estados alcançados. O método de seleção do próximo estado da *open list* é a minimização da soma do custo atingido nodo com a heurística desse nodo. Assim, para cada nó (estado) expandido é necessário calcular o custo até esse nó (custo até ao nó anterior mais custo da aresta). Este algoritmo encontra solução ótima caso a heurística seja aceitável.

3.5 Contexto multijogador

De modo a cumprir com o requisito de poder simular vários jogadores ao mesmo tempo, elaboramos uma função que, para cada jogador na pista, determina o caminho percorrido ao longo da pista, mediante um dado algoritmo.

A primeira estratégia que pensamos passou por gerar um grafo em que cada nodo continha as posições e velocidades de todos os jogadores do mapa. Cada nodo era, no fundo, um possível estado dos N jogadores. As transições de estado representariam os N movimentos de todos os carros em jogo.

Contudo, para valores de N maiores, esta solução tornar-se-ia impensável uma vez que o fator de ramificação do grafo seria, no pior caso, 9^N, onde N é o número de jogadores. Isto traduzir-se-ia num tempo de geração de grafo exponencial, o que não seria viável.

Outra estratégia considerada foi permitir que cada jogador jogasse à vez, tendo em consideração as posições dos restantes jogadores. As posições dos outros jogadores afetavam as jogadas que um dado jogador poderia realizar, pois poderia haver embates. Portanto, precisávamos de gerar um novo grafo a cada jogada, algo que rapidamente vimos que não seria a melhor solução.

Assim, a estratégia utilizada passou por aplicar para cada jogador, por ordem, o seu algoritmo de procura no grafo inicialmente gerado a partir de todos os estados da lista de partida dos vários jogadores. Com isto, os jogadores aplicam o algoritmo à vez e, no final, é apresentada uma simulação onde as jogadas de todos os jogadores são sincronizadas por iteração e apresentadas ao mesmo tempo.

Para este contexto competitivo entre vários jogadores, as posições dos mesmos devem ser introduzidas a partir do circuito de *input*. Desta forma, os jogadores terão obrigatoriamente de partir de casas diferentes. Temos noção de que no jogo oficial os jogadores partem da mesma casa mas optamos por esta decisão uma vez que se aproxima da realidade, visto que, num contexto real, nenhum carro pode estar numa mesma posição que outro carro.

3.5.1 Tratamento de colisões

Seguindo a estratégia indicada acima, os algoritmos de procura passam então a considerar os caminhos percorridos pelos jogadores que já tenham aplicado o algoritmo respetivo anteriormente. Na iteração i de um algoritmo de procura genérico, deverão ser avaliadas as posições de cada carro nessa mesma iteração. Por exemplo, o algoritmo DFS, na sua primeira iteração deverá considerar todas as posições dos carros que já efetuaram a sua procura na primeira iteração. Assumindo que a próxima opção do algoritmo DFS é a posição (x_j, y_j), isto é, a posição onde o carro j se encontrava nessa iteração, o algoritmo deverá passar ao próximo nodo a avaliar, seguindo o mesmo raciocínio. Da mesma forma, se ao realizar o deslocamento para uma dada posição, encontrar um outro jogador a meio desse deslocamento, deverá descartar esse estado adjacente. Isto é realizado, recorrendo aos métodos do módulo position_calculator, que lida com deslocamentos posição a posição. Deste modo, cada jogador aplica o seu algoritmo de procura tendo em conta os caminhos já efetuados pelos jogadores que efetuaram a procura no grafo, evitando colisões entre estes.

Por exemplo, no ambiente representado na figura 4, é o carro mais à esquerda o primeiro a jogar. Assumindo que o carro já tem velocidade (1,0), a escolha mais óbvia é adquirir uma aceleração de (1,0), permitindo-lhe permanecer no caminho mais curto até ao final. No entanto, um outro jogador já se encontra nessa posição. Com este algoritmo, obrigamos o primeiro carro a adotar um plano de contingência, contornando o obstáculo.

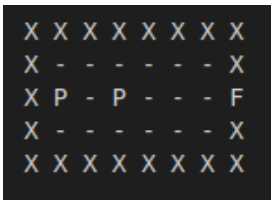


Figura 4: Tratamento de colisões no ambiente multijogador.

No entanto, há casos em que, com esta estratégia, é impossível avançar. Consideremos o exemplo seguinte:

Figura 5: Situação especial - Partida.

Neste circuito, assumindo que os jogadores jogam da esquerda para a direita e que fazem a escolha óbvia para a próxima jogada, o jogador roxo movimenta-se uma casa para a direita, o jogador amarelo uma casa para baixo, ficando o jogador azul bloqueado, pois na casa à sua esquerda já se encontra o jogador roxo.

Figura 6: Situação especial - 1ª Jogada.

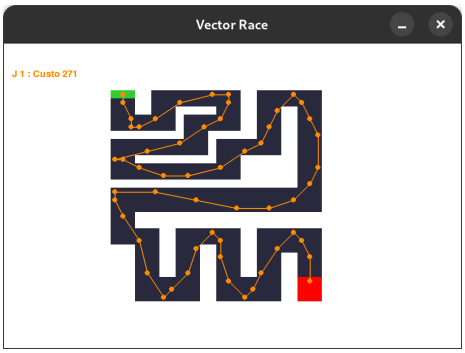
Nesta situação, o jogador azul deve ficar parado, permanecendo no mesmo local até que a única posição válida seja desocupada.

Figura 7: Situação especial - 2ª Jogada.

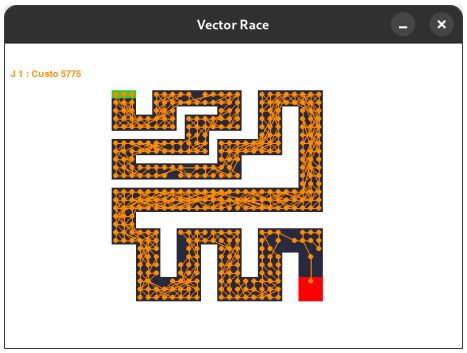
Finalmente, quando o jogador roxo avançar, o jogador azul poderá ocupar a casa que desejava e o algoritmo prossegue normalmente.

4 Discussão dos Resultados Obtidos

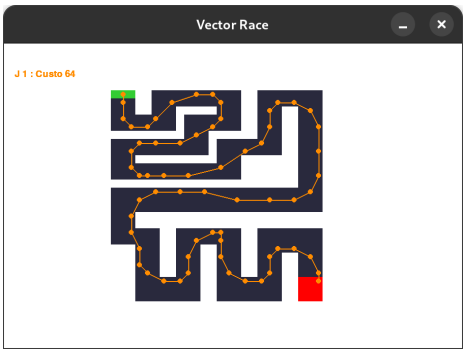
De modo a testar e a comparar o comportamento dos diferentes algoritmos, usamos o circuito "Vector" como ambiente de teste.



(a) Algoritmo BFS.



(b) Algoritmo DFS.



(c) Algoritmo do Custo Uniforme.

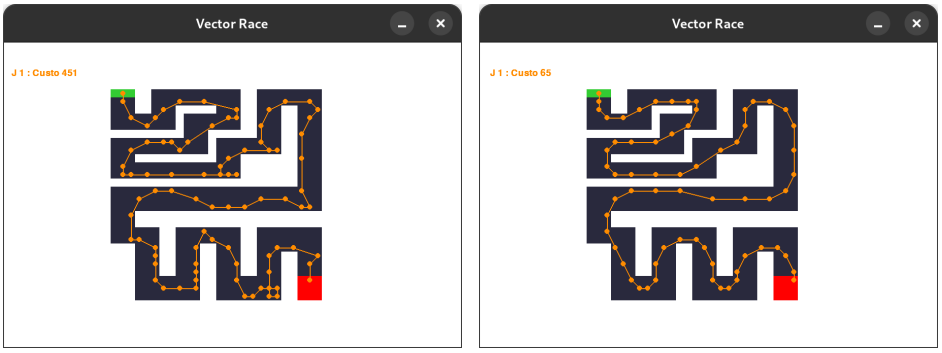
Figura 8: Pesquisas não-informadas no circuito "Vector".

O algoritmo BFS realiza uma procura sistemática, por níveis, pelo que é esperado que encontre uma solução com o menor número de jogadas possível. No entanto, isso não impede que o mesmo bata em paredes, aumentando o custo da sua solução. Caso as arestas da solução sejam de custo 1, então será a solução ótima.

Realizando uma procura em profundidade, o algoritmo DFS é, de todos, o que encontra pior solução. A escolha do primeiro estado não visitado da lista de estados da expansão não traz qualquer vantagem ao algoritmo, pois faz com que este se "perca" em caminhos inconsequentes, tal como podemos ver na imagem. Para além disso, não há qualquer tipo de mecanismo que previna o algoritmo de se deslocar por arestas com custo 25 (embates com obstáculos).

O algoritmo de custo uniforme é, de todos os algoritmos referentes a pesquisas não-informadas, aquele que apresenta melhor resultado. Pelo facto de poder

calcular, para cada nó da expansão, qual o custo desde a origem até esse nodo, este algoritmo pode escolher sempre o nodo que minimiza o custo total da solução, apresentando deste modo a solução ótima.



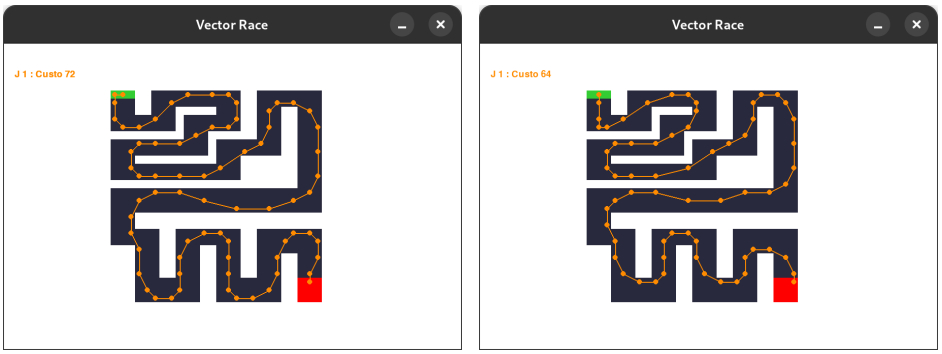
(a) Algoritmo *Greedy*.

(b) Algoritmo A*.

Figura 9: Pesquisas informadas com heurística "distância".

Analisando a figura, podemos verificar que a heurística "distância euclidiana à meta" não é a mais apropriada para o algoritmo *Greedy*, uma vez que o leva a ficar temporariamente bloqueado em cantos. Este tipo de heurísticas seria capaz de encontrar melhores soluções em circuitos com poucas curvas.

Já o algoritmo A* é capaz de minimizar as ineficiências trazidas pela escolha desta heurística. Mantendo o custo acumulado do caminho até um dado estado, este algoritmo procura minimizar a soma entre o custo acumulado e a distância até ao final. A complexidade espacial deste algoritmo é, no entanto, maior, uma vez que terá de guardar todos os custos até aos nós atualmente visitados, para poder "recuar" e analisar melhores alternativas.



(a) Algoritmo *Greedy*. (b) Algoritmo A*.

Figura 10: Pesquisas informadas com heurística "velocidade".

Uma particularidade interessante da heurística "velocidade" é o facto de evitar ao máximo que o carro bata em obstáculos (paredes), mantendo ao mesmo tempo uma boa velocidade.

Por exemplo, no circuito acima, seria de prever, primeiramente, que o algoritmo *Greedy* optasse por estados onde a velocidade aumentasse. No entanto, não é isso que acontece e podemos comprovar que o carro não bate mesmo nos obstáculos, verificando que o custo da solução é igual ao número de deslocações. O facto de o algoritmo poder "recuar" permite que ele encontre sempre uma solução em que a velocidade é a máxima possível para aquele trajeto. Por exemplo, caso o carro esteja com uma velocidade bastante alta antes de uma curva, de tal modo que é impossível não embater numa parede, o algoritmo é levado a considerar uma posição melhor, expandindo outro estado anterior da lista (*open list*).

Assim, comparando as duas heurísticas utilizadas (distância em linha reta à meta e velocidade do carro), podemos verificar que a heurística "velocidade" apresenta muito melhores resultados. A heurística define um critério de preferência entre dois estados adjacentes a um primeiro. A preferência pela distância leva o carro a procurar uma próxima posição o mais perto do destino, mesmo implicando um embate numa parede. A preferência por um estado com maior velocidade leva o carro a evitar situações de embate, mantendo sempre uma velocidade estável, o que o permite travar atempadamente em muitas circunstâncias.

Em relação à comparação dos dois algoritmos de pesquisa informada, pelas figuras 9 e 10 podemos verificar que o algoritmo A* apresenta normalmente melhores resultados.

5 Interface Gráfica

Nesta segunda fase do projeto, para atingir uma melhor apresentação dos caminhos escolhidos pelos diferentes algoritmos, elaboramos uma interface gráfica. Para isso, recorremos à biblioteca "pygame", tornando assim mais elegante e funcional a apresentação dos algoritmos de procura quer no modo singleplayer quer no modo multiplayer e facilitando o debug dos mesmos.

É necessária a instalação da biblioteca através do comando:

```
pip install pygame
```

Para executar o jogo, é necessário o seguinte comando:

```
python3 main.py
```

Os controlos básicos, estendidos para todos os menus, são os seguintes:

- "Esc" - Voltar para o menu anterior
- Setas - Navegação dentro dos menus
- "Enter" - Avançar para o menu seguinte

O menu inicial permite ao utilizador escolher o circuito sobre o qual pretende fazer a simulação. Os circuitos apresentados por predefinição são os circuitos criados pelo grupo. Contudo, é também possível utilizar um outro circuito, inserindo apenas o nome do circuito (sem a diretoria), sendo que este deverá estar na pasta "circuits".

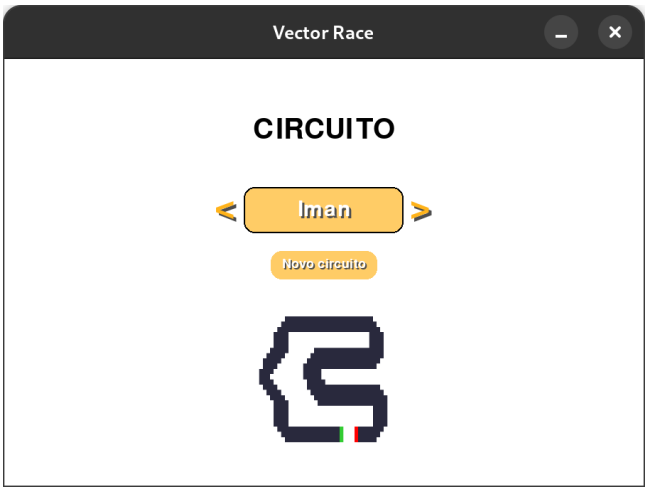
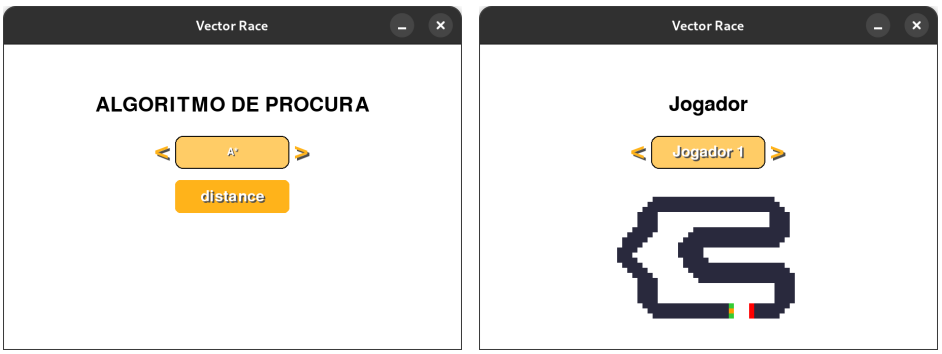


Figura 11: Escolha do circuito.

Em seguida, o utilizador pode escolher o modo de jogo entre Singleplayer (apenas um carro) ou Multiplayer (modo competitivo com vários carros).

Dependendo da escolha do modo jogo, ser-lhe-à apresentado um menu onde deve escolher o(s) algoritmo(s) a utilizar pelo(s) carro(s) na corrida. As 5 possibilidades neste momento disponibilizadas são os algoritmos enumerados acima, com a possibilidade de escolha de heurística nos algoritmos de procura informada.

Figura 12: Escolha dos algoritmos no modo Multiplayer.



(a) Escolha do algoritmo.

(b) Escolha do jogador.

Figura 13: Escolha do algoritmo e do jogador no modo Singleplayer.

Por fim, são apresentadas as simulações, onde o utilizador pode avançar ou recuar utilizando as setas do teclado, podendo manter estas teclas premidas para uma simulação mais acelerada.

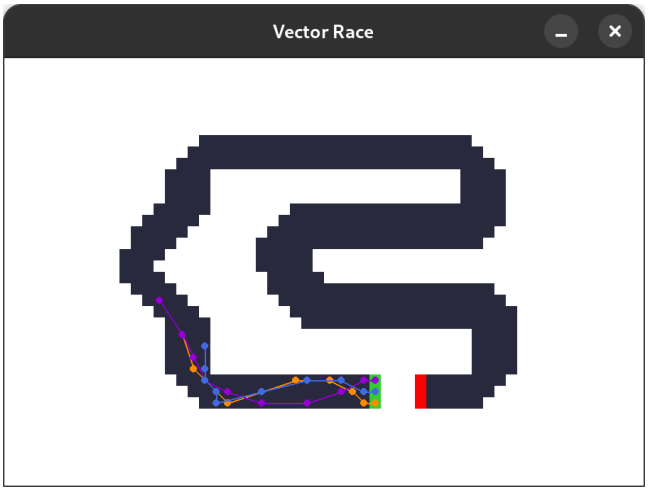


Figura 14: Simulação Multiplayer.

No modo singleplayer, na janela de simulação, o utilizador pode aceder à simulação em modo *debug* ao premir a tecla "D", onde é apresentado todo o caminho percorrido ao longo da execução do algoritmo. O modo para avançar a simulação é semelhante, utilizando as setas.

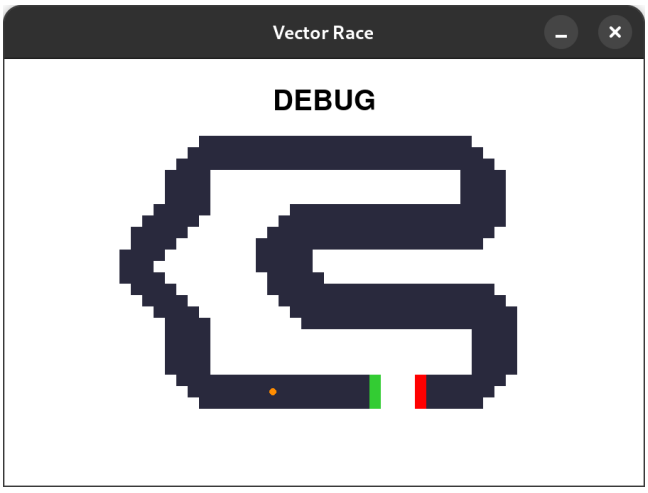


Figura 15: Modo Debug.

6 Conclusão

Em jeito de conclusão, os objetivos impostos pelo grupo passaram por ler circuitos de um ficheiro de texto, gerar um grafo de estados, implementar algoritmos de procura informada e não-informada, desenvolver um contexto competitivo que apresenta vários jogadores e construir uma interface gráfica.

A implementação de vários algoritmos de pesquisa em grafos foi objeto de preocupação do grupo e, portanto, acabamos por implementar mais 3 algoritmos em relação à primeira fase, de modo a ter um número considerável de algoritmos apresentados.

O modo multijogador foi o alvo de maior reflexão neste trabalho, uma vez que ambicionávamos construir algo sólido e coerente. Deste modo, após várias estratégias pensadas para resolver o problema, optamos pelo contexto competitivo já referido.

A representação do caminho mais curto e do caminho percorrido ao longo da execução dos algoritmos foi também um dos entraves do trabalho. Primeiramente, começamos por recorrer à biblioteca "Matplotlib" para representar os caminhos. Contudo, achamos por bem que o utilizador deveria conseguir visualizar a construção dos caminhos finais etapa a etapa, pelo que prosseguimos à realização de uma funcionalidade adicional, a interface gráfica, permitindo uma melhor experiência ao utilizador.

Reconhecemos que existem alguns aspetos que podiam vir a ser melhorados no projeto, tal como a apresentação de mais algoritmos. Depois de implementarmos o algoritmo de procura iterativa, chegamos à conclusão de que a nossa implementação era extremamente ineficiente a nível de tempo ao encontrar a solução com menor número de nodos, principalmente em circuitos de grande dimensão. Assim, decidimos não apresentá-la.

Com isto, consideramos que cumprimos todos os objetivos propostos de uma maneira bastante satisfatória, o que permitiu ao grupo consolidar o conhecimento adquirido ao longo das aulas, nomeadamente na geração de grafos e na implementação de algoritmos de procura, tanto não-informada como informada.