VectorRace

Trabalho realizado por:

Beatriz Ribeiro Monteiro
Bianca Araújo do Vale
João Carlos Fernandes Novais
João Pedro Machado Ribeiro

A95437
Beatriz Monteiro

A95835
Bianca Vale

A96626
João Novais

A95719
João Ribeiro

Grupo 29
Inteligência Artificial 2022/2023
Universidade do Minho

Índice

Índice de Imagens

1 Introdução

No presente relatório iremos elaborar e explicar o trabalho prático desenvolvido na Unidade Curricular de Inteligência Artificial. Este relatório engloba todo o trabalho desenvolvido, isto é, as tarefas desenvolvidas tanto na primeira fase do projeto como na segunda.

Foi nos proposto pela equipa docente desta UC desenvolver vários algoritmos de procura para a resolução do jogo *VectorRace*, também conhecido como *RaceTracker*. Este é um jogo de simulação de carros, que contém um conjunto de movimentos e regras associadas.

Na primeira fase tivemos que criar um circuito, com um participante, e implementar pelo menos uma estratégia de procura , enquanto na segunda fase foi proposto tornar o trabalho mais complexo, implementendo mais estratégias de procura, informada ou não informada, e ter dois ou mais participantes.

Para a construção deste projeto foi utilizada a linguagem de programação *python*.

2 Descrição e Formulação do Problema

2.1 Descrição do Problema

O *VectorRace* é um jogo que simula uma corrida de carros num mapa que é representado por uma grelha, em que os jogadores movem-se de um ponto do mapa para outro, e têm que tentar chegar ao final com o menor número de mudanças de direção posível. Neste jogo existem posições jogáveis e não jogáveis. Estas últimas são caracterizadas pelo jogador sair do mapa ou se encontrar numa parede.

Na elaboração do trabalho adotamos a notação y para representar as linhas e x para as colunas.

O carro pode acelerar -1, 0, ou 1 unidades em cada direção. Por consequência, para cada uma das direções, o conjunto de acelerações possíveis é Acel=-1, 0, +1, sendo que a=(ay,ax) representa a aceleração de um carro nas duas direções num determinado instante.

Como tuplo, p indica a posição de um carro numa determinada jogada $j(p^j = (p_y, p_x))$, e v o tuplo que indica a velocidade do carro nessa jogada $(v^j = ((v_y, v_x))$, na seguinte jogada o carro encontrar-se-á na posição:

$p_y^{j+1} = p_y^j + v_y^j + a_y$

$p_x^{j+1} = p_x^j + v_x^j + a_x$

A velocidade do carro num determinado instante é calculada por:

$v_y^{j+1} = v_y^j + a_y$

$v_x^{j+1} = v_x^j + a_x$

Sempre que o carro sair da pista o carro terá que voltar para a posição anterior, assumindo um valor de velocidade 0.

Cada movimento de um carro numa certa jogada, de uma determinada posição para outra, terá um custo de 1 unidade, sendo que se este sair da pista, o custo é de 25 unidades.

2.2 Formulação do Problema

Antes de começar a formulação do problema como um problema de pesquisa, é importante referir algumas coisas a ter em conta. Primeiramente, os diferentes estados deste problema são definidos pela posição e velocidade que o jogador tem nesse momento, sendo que a posição é relativa a um mapa. Este mapa possui paredes (representadas por "#"

na representação em forma de texto do mapa) e espaços que são possíveis de conduzir (representados por " " ou "-" na representação textual do mapa).

Ao gerar o grafo para a pesquisa de soluções para o problema, para além do estado é considerada a posição anterior do jogador.

2.2.1 Estado Inicial

No estado inicial temos o jogador em uma posição inicial, representado no formato textual através de um "I", e no formato visual através da cor verde, com velocidade igual a zero em ambas as componentes.

2.2.2 Estado Objetivo

No estado final, o jogador está numa posição final, representado no formato textual através de um "F" e no formato visual com cor vermelha, sendo que a velocidade acaba por não importar, neste caso.

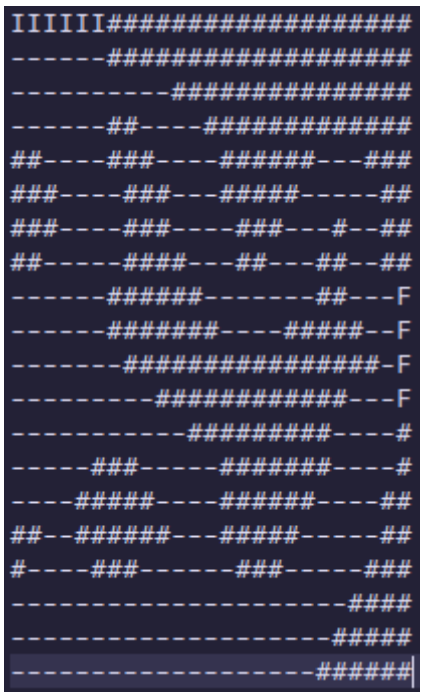

Figure 1: Representação textual de um mapa

2.2.3 Operadores

Para a expansão do problema existem 2 casos possíveis, sendo estes:

- O jogador encontra-se numa posição onde não se pode movimentar, isto é, está fora do mapa ou na posição de uma parede, e é usado para a expasão desse estado o operador "jogadorFora"
- O jogador encontra-se numa posição jogável e movimenta-se para outra posição, sendo usado para isso os restantes operadores.

Tendo isso em consideração, apresentam-se agora os diferentes operadores:
Operador 1
Nome: jogadorFora.

Pré-condição: O jogador está numa posição não jogável, isto é, encontra-se numa parede ou fora do mapa.

Efeitos: O jogador volta para a posição onde estava antes de sair da pista, com velocidade 0 em ambas as componentes.

Operador 2

Nome: acelerarXabrandarY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade na componente x do jogador aumenta 1, a velocidade na componente y diminui em 1 e a posição varia consoante a velocidade.

Operador 3

Nome: acelerarXmanterY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade na componente x do jogador aumenta 1, a velocidade na componente y mantém-se constante e a posição varia consoante a velocidade.

Operador 4

Nome: acelerarXacelerarY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade do jogador aumenta 1 em ambas as componentes e a posição varia consoante a velocidade.

Operador 5

Nome: abrandarXabrandarY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade do jogador diminui 1 em ambas as componentes e a posição varia consoante a velocidade.

Operador 6

Nome: abrandarXmanterY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade na componente x do jogador diminui 1, a velocidade na componente y mantém-se constante e a posição varia consoante a velocidade.

Operador 7

Nome: abrandarXacelerarY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade na componente x do jogador diminui 1, a velocidade na componente y aumenta 1 e a posição varia consoante a velocidade.

Operador 8

Nome: manterXabrandarY

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade na componente x do jogador mantém-se constante, a velocidade na componente y diminui 1 e a posição varia consoante a velocidade.

Operador 9

Nome: manterXmanterY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade do jogador mantém-se constante em ambas as componentes e a posição varia consoante a velocidade.

Operador 10

Nome: manterXacelerarY.

Pré-condição: O jogador está numa posição jogável.

Efeitos: A velocidade na componente x do jogador mantém-se constante, a velocidade na componente y aumenta 1 e a posição varia consoante a velocidade.

2.3 Custo de solução

O custo total de cada solução será igual ao custo somado de cada movimento. Nos casos em que o carro se move e mantém-se na pista o custo de movimento é igual a 1. Nos casos em que o carro bate contra uma parede ou vai para fora do mapa, o custo de movimento é igual a 1 por se mover mais 25 por bater contra a parede.

3 Geração de circuitos VectorRace

Apesar de termos um algoritmo que gera caminhos aleatoriamente, no ficheiro *GeracaoAutomaticaPerlinNoise.py*, muitas vezes os mapas nele gerado não são adequados para a execução do trabalho, pelo que necessitam de retoques, pelo que nesta fase criamos 4 mapas, com as mesmas características dos que foram apresentados em cima, 1 gerado à mão (o mais simples) e sendo os outros gerados no ficheiro referido acima, apenas levando uns retoques. Para a execução do nosso trabalho, após o utilizador escolher o mapa que pretende utilizar no menu, este será carregado e é feita a criação do grafo, explicada abaixo.

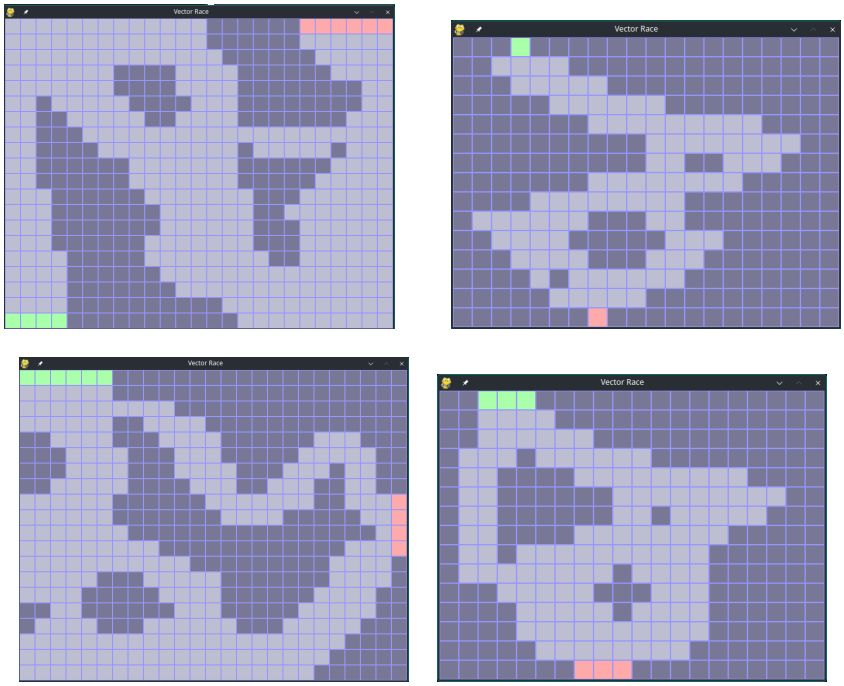

Figure 2: Mapas

4 Grafo

4.1 Guardar o Objeto do Grafo em Ficheiro

Antes de passar para a explicação da construção do grafo em si, primeiro temos que explicar uma coisa que fizemos no trabalho. Para tornar a experiência mais agradável decidimos ao gerar um grafo guardar o objeto criado que representa o mesmo num ficheiro, de forma a que se se decidir usar novamente esse mapa noutra execução do programa não seja necessário gerar o mesmo novamente.

4.2 Construção do Grafo

Na construção do grafo, para além da informação de cada estado em si, consideramos ainda a posição anterior à atual, de forma a facilitar a geração do grafo, e para além disso tiramos partido de um mapa de inteiros mapeado com a distância a que cada posição está de uma posição final, utilizado para atribuir heurísticas aos nodos em cada ponto.

Começamos por criar uma lista para guardar os nodos a expandir, e uma lista com os nodos já expandidos. Os nodos são expandidos enquanto a lista de nodos a expandir não estiver vazia.

Através de uma função auxiliar chamada *expandState* expandimos o estado atual. Nessa função verificamos se no estado atual o jogador se encontra fora de pista ou numa parede e, se sim, volta para a posição anterior com velocidade zero. Caso contrário, a expansão do estado dá origem a 9 estados filhos (tendo em conta que se a velocidade do estado pai for 0 em ambas as componentes, não é possível manter a velocidade).

Ao gerar os 9 estados filhos, usamos ainda outra função auxiliar, que calcula se durante o seu movimento o jogador embate contra uma parede ou vai para fora do mapa. Essa função chama-se *validVector* e o funcionamento da mesma é o seguinte: primeiramente começamos por aplicar a aceleração ao estado, aumentado a velocidade, e calcular quais serão as coordenadas finais caso não embata. De seguida fazemos algumas verificações, sendo estas:

- Caso a posição final do jogador seja fora do mapa ou numa parede, o jogador vai para essa posição.
- Caso a velocidade na componente x do jogador seja zero, é verificado se nessa coluna existe alguma parede, e caso haja, o jogador fica parado na mesma, caso contrário segue até à posição final.
- Caso a velocidade na componente y do jogador seja zero, é verificado se nessa linha existe alguma parede, e caso haja, o jogador fica parado na mesma, caso contrário segue até à posição final.
- Caso a diferença absoluta entre alguma das componentes da posição final e inicial for igual a 1, verificamos se no retângulo gerado pelas posições há alguma parede da seguinte forma (exemplificado na figura seguinte)

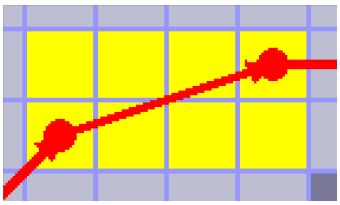

Figure 3: Posições verificadas caso a diferença absoluta de alguma comonente da posição seja 1

- Caso não se adapte a nenhum dos casos acima, verificamos através de equações lineares, usando como variável independente a posição no eixo x numa e a posição no eixo y na outra. Durante essa verificação não só é verificada a posição que se encontre a intercetar com a equação como a posição à frente, caso o x aumente, ou atrás caso contrário, para o caso da equação que usa x como variável independente, e a posição acima caso o movimento diminua no eixo do y, e abaixo caso aumente, para a equação que usa y como variável independente, possível de ver na figura seguinte:

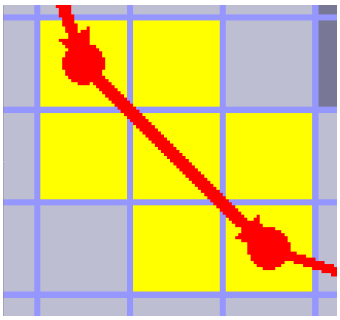

Figure 4: Posições verificadas caso não se encaixe com os outros casos

Se em alguma dessas posições estiver uma parede, o jogador fica nela, senão fica na posição final.

Ao expandir um nodo adicionamos ainda uma heurística ao mesmo, sendo essa heurística constituída pela soma entre o quociente entre a distância de um estado até ao fim (visto no mapa de distâncias ao fim) com a norma do vetor velocidade somado com 1 (para evitar divisão por zero) com a diferença entre as normas do vetor associado à velocidade do nodo pai e a norma do vetor associado à velocidade do nodo filho multiplicado por 0,5 (de forma a não dar números muito grandes).

Após analisar todos os estados a expandir o nodo fica criado e é guardado em ficheiro para uso futuro, e é possível assim passar para a pesquisa de caminhos.

5 Estratégias de procura implementadas

Neste projeto foi-nos proposto o desenvolvimento de estratégias de procura, **informada** ou **não informada**. Nós optamos por implementar 4 estratégias de procura, sendo 2 delas **não informadas** e as outras **informadas**.

As estratégias de procura **não informada** utilizam unicamente as informações disponíveis na definição do sistema enquanto que nas heurísticas, ou estratégias de procura **informadas**, são dadas "dicas" ao algoritmo sobre a adequação de diferentes estados.

As estratégias de procura não-informadas que implementamos foram a BFS (*Breadth-First Search*) e DFS (*Depth-First Search*).

A procura BFS costuma ser melhor para problemas mais pequenos, mas é de se esperar um tempo de execução maior por analisar todos os nós de menor profundidade, já na procura DFS é possível obter um tempo menor por não necessitar de muita memória para ser exectuada, mas também se espera uma solução menos ótima.

Os algoritmos informados que implementamos foram a pesquisa gulosa e o algoritmo A^*, e para as heurísticas usamos a soma do quociente entre a distância de um estado até ao fim (visto no mapa de distâncias ao fim) com a norma do vetor velocidade somado com 1 (para evitar divisão por zero) e a diferença entre as normas do vetor associado à velocidade do nodo pai e a norma do vetor associado à velocidade do nodo filho multiplicado por 0,5.

A procura A^* apresenta resultados bons se as heurísticas forem bem definidas, e foi por esse motivo que usamos o método acima para o cálculo das heurísticas, e funciona tendo em conta a distância percorrida até um nodo e a heurística do mesmo para escolher qual caminho seguir, enquanto a procura gulosa tem apenas em conta a heurística dos nodos. O bom destes algoritmos é que normalmente geram caminhos de forma rápida.

A procura Gulosa apresenta resultados bons em maioria dos casos porque definimos de forma satisfatória as heurísticas, uma vez que o que faz com que este algoritmo seja mau em alguns casos é o facto de este algoritmo não ter em conta o custo das ligações,

mas como neste caso a maior parte dos caminhos entre nodos tem custo 1, acaba por gerar um caminho satisfatório.

Com a implementação de quatro estratégias de procura diferentes é possível ver os diferentes algoritmos a trabalharem em simultâneo e a diferença entre eles, isto é, o caminho escolhido por cada um, bem como o tempo que demoram a encontrar um caminho.

6 Formato Competitivo

Na segunda fase do trabalho, foi nos proposto que fosse adicionado um ambiente com, no mínimo, dois jogadores. De forma a cumprir esse requisito, conseguimos adaptar o código realizado na fase anterior.

6.1 Modificações em relação à Primeira Fase

Em relação à Primeira Fase, devido à maneira como tínhamos o projeto estruturado, não foi necessário mudar muita coisa, foi apenas necessário adicionar alguma maneira de calcular caminhos que não passassem numa certa posição (visto que dois carros não podem ocupar a mesma posição ao mesmo tempo).

Desta forma, de modo a conseguir implementar essa mudança adicionamos 2 parâmetros nos métodos de procura, um com uma lista de posições em que não pode passar na primeira jogada (porque é esta a posição que leva a um novo cálculo do caminho), e uma *flag* que dita se o algoritmo pode ignorar a verificação da validade da posição inicial.

6.2 Funcionamento do Modo Competitivo

O método que implementa esta procura recebe apenas um argumento, que é um dicionário em que cada chave é a identificação de um jogador e o valor associado a cada chave é o algoritmo que este deve utilizar.

Para o cálculo dos caminhos em si, é feito um ciclo que é executado enquanto houver algum jogador que ainda não tenha atingido o fim. Uma iteração desse ciclo corresponde a uma posição que cada jogador ativo anda, e em cada iteração a lista de jogadores é ordenada de forma a que o jogador que se encontre mais longe da meta seja o primeiro a jogar. Cada jogador tem um caminho que pretende seguir, e caso não seja possível é calculado um novo.

6.3 Como aceder ao formato competitivo

O formato competitivo pode ser acedido seguindo os seguintes passos:

1. No Menu Principal, escolher a opção "Selecionar Mapa", como se vê na figura 5

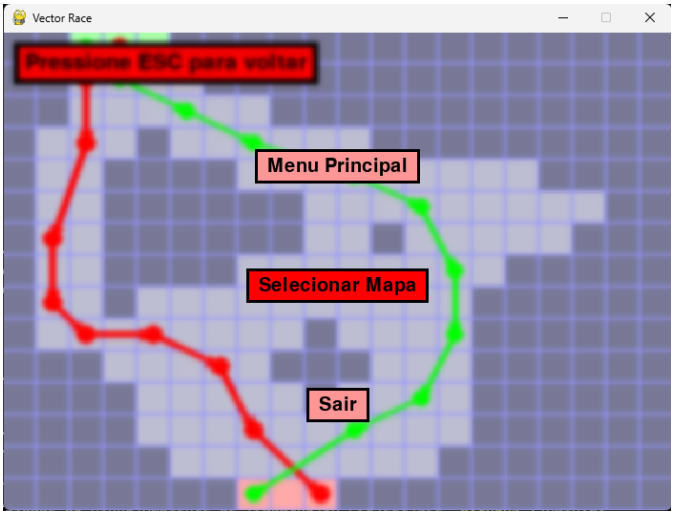

Figure 5: Menu Principal

2. De seguida seleciona-se um mapa (figura 6), utilizando as setas para navegar o menu, e utilizado a tecla Enter entra-se na opção "Competicao" do menu de Simulação (figura 7).

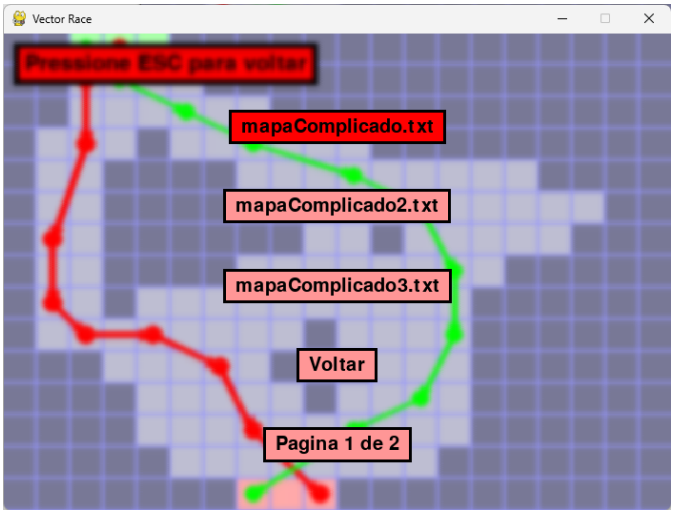

Figure 6: Menu de Selecionar Mapa

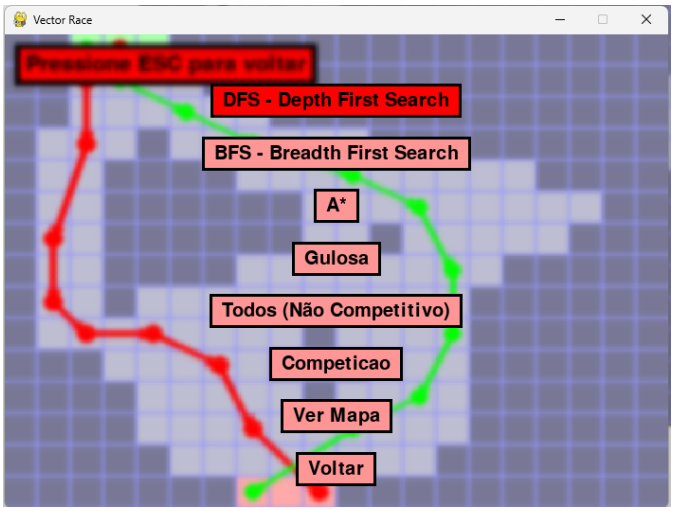

Figure 7: Menu de Simulação

3. Com as setas para a esquerda e direita escolhe-se o número de Jogadores e com a tecla Enter seleciona-se o número (figura 8), e finalmente escolhem-se os algoritmos de procura que cada Jogador deverá utilizar (figura 9).

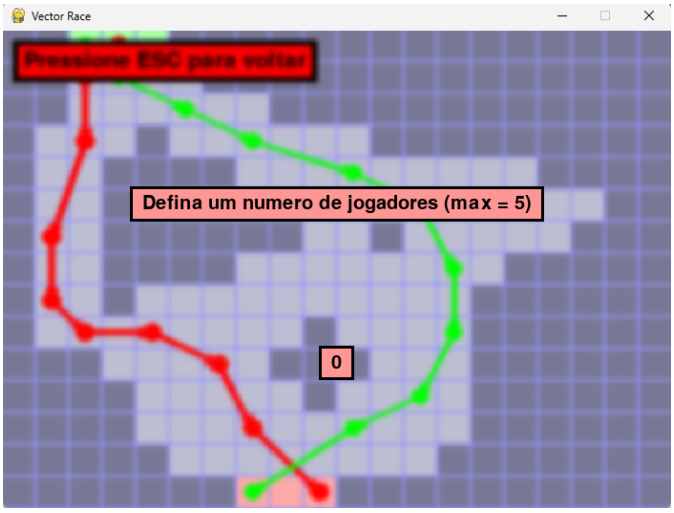

Figure 8: Escolher Número de Jogadores

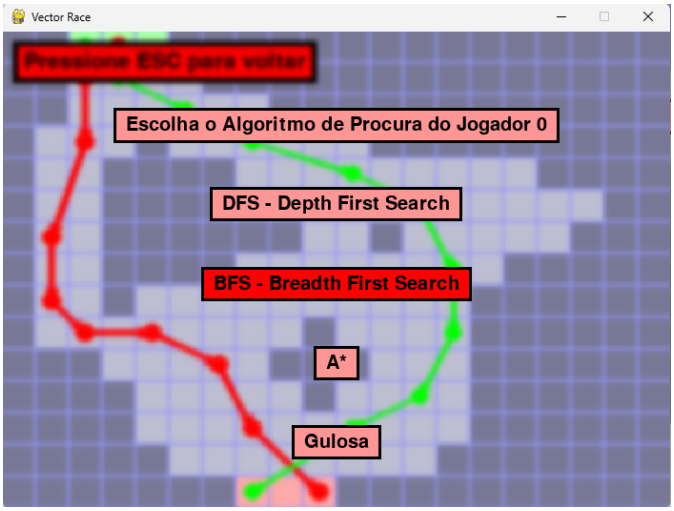

Figure 9: Escolher algoritmo de procura que jogador deve utilizar

4. Após esperar que os resultados sejam calculados, é apresentado no ecrã uma pequena simulação da corrida (figura 10).

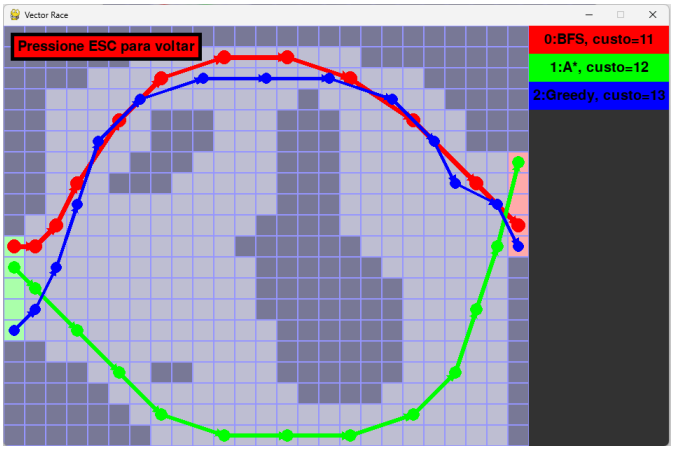

Figure 10: Resultado do Modo Competitivo

7 Discussão dos Resultados Obtidos

A procura BFS é a que tem melhores resultados em termos de custo do caminho, sendo que ele econtra sempre um caminho ótimo. Em contrapartida, comparado aos outros algoritmos de pesquisa, este demora mais tempo a gerar um caminho, devido ao que foi explicado acima.

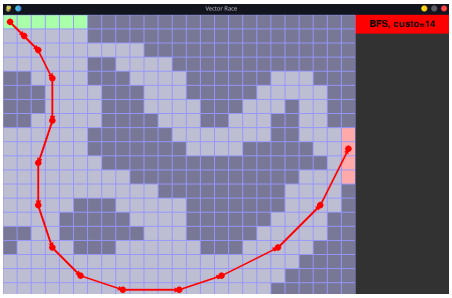

Figure 11: Procura BFS

A procura DFS teve resultados bastante maus, é possível observar que o jogador encontra mais dificuldades em sair da zona inicial, executando praticamente todos os caminhos possíveis, chegando ao fim ainda por caminhos não ideais, indo contra paredes e para fora do mapa.

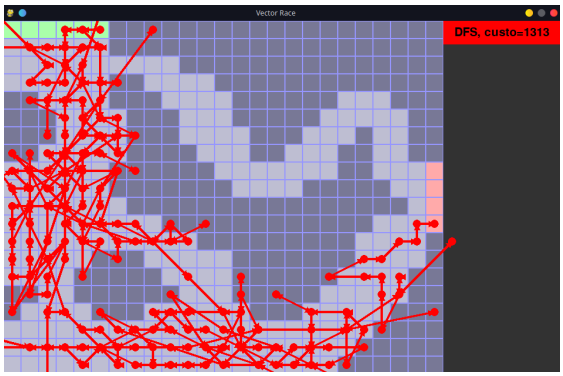

Figure 12: Procura DFS

A procura gulosa teve resultados razoáveis, não obtendo o caminho ótimo mas obtendo um caminho muito próximo disso.

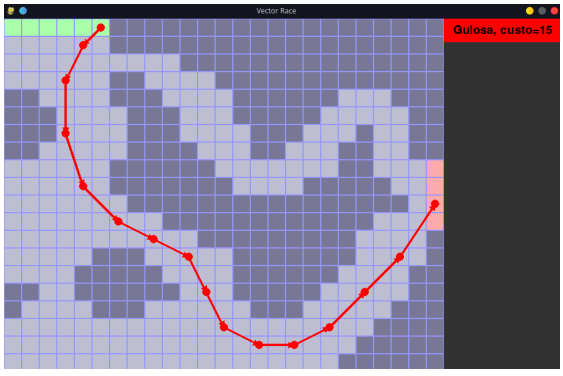

Figure 13: Procura Gulosa

A procura A*, apesar de nem sempre encontrar um caminho ótimo, devido ao facto

da nossa heurística não ser perfeita, é a que consideramos melhor, uma vez que tem tendência a gerar caminhos com custo próximos do ótimo (muitas vezes com custo ótimo até), sendo que o tempo que demora a encontrar o mesmo é muito menor que o único algoritmo que a ele se compara em termos de custo das soluções, o BFS.

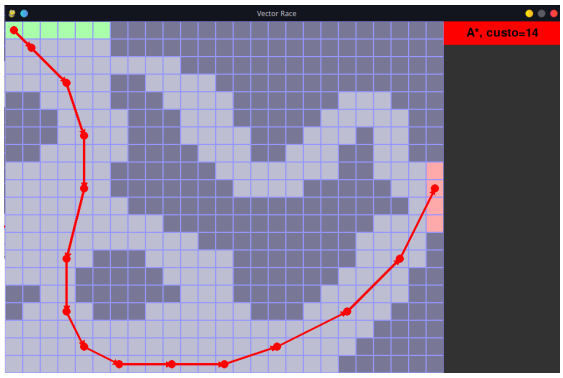

Figure 14: Procura A*

Podemos ver pela imagem a seguir as procuras DFS(vermelho), BFS (verde), A* (azul) e Gulosa (amarelo) a serem executadas ao mesmo tempo, e pode-se chegar a conclusão que a procura BFS e A* são parecidas até um certo ponto, e depois de divergirem a procura BFS vai por um caminho com menos curvas, acabando por ter um custo total ligeiramente melhor que a procura A*. A procura DFS eventualmente chega ao fim, mas quase nem server como termo de comparação com as outras três. Quanto à procura gulosa, nota-se que vai sempre para o ponto com a menor heurística, não sendo tão bom como as outras (excluindo a DFS).

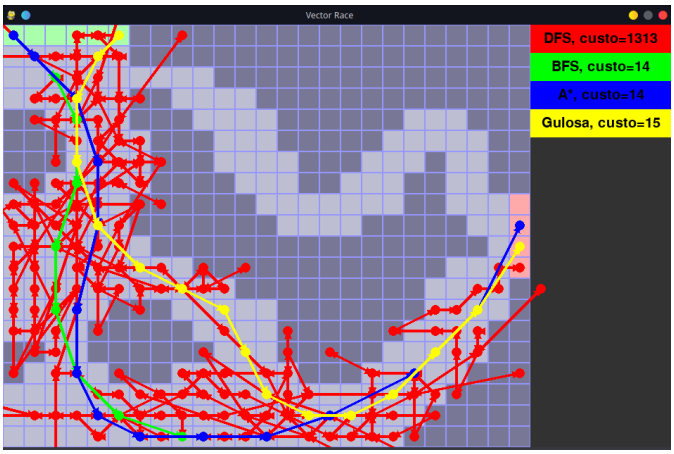

Figure 15: Mapa com Todos

Na figura apresentada em baixo, está representado o menu, onde se podem escolher os difernetes algoritmos de procura, neste caso, o algoritmo selecionado é o DFS.

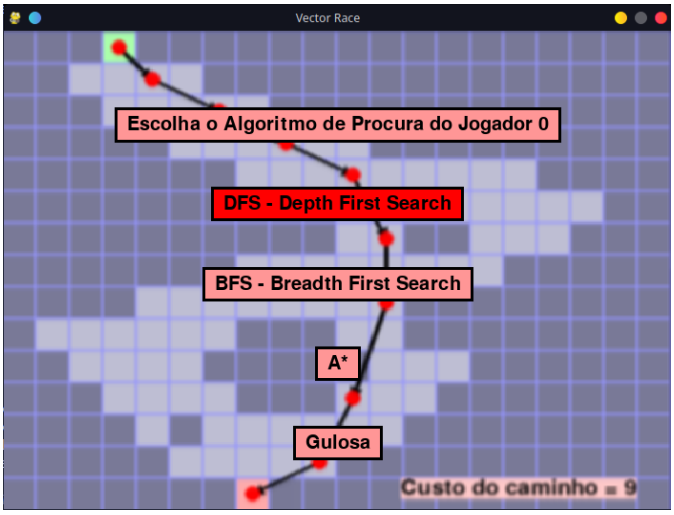

Figure 16: Menu com os diferentes algortimos de procura

7.1 Resultados do Ambiente Competitivo

Na seguinte figura podemos analisar um exemplo do ambiente competitivo entre 2 jogadores ao mesmo tempo. Um deles usando o algoritmo A* e o outro o algoritmo Guloso (Greedy). É possível ver que ambos percorreram caminhos parecidos, sendo que nunca chegam a ocupar a mesma posição ao mesmo tempo. Neste mapa era possível ter um máximo de 4 jogadores, que é o número de posições iniciais que o mesmo tem.

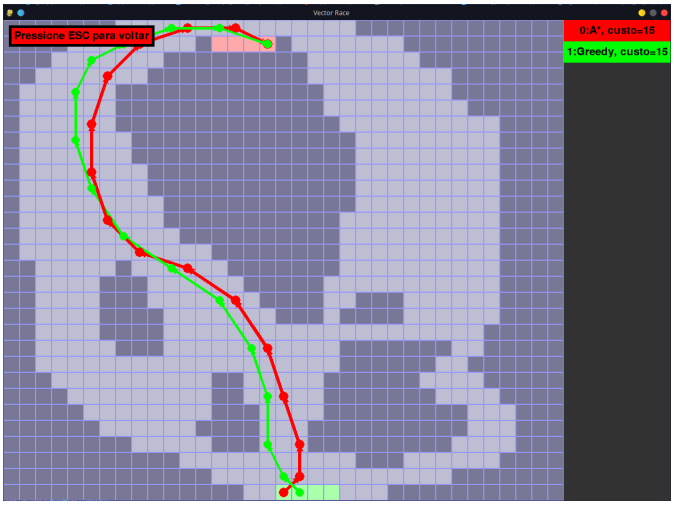

Figure 17: Mapa com Procura A* e Greedy

Numa nota mais geral, é possível ver os resultados são próximos do que se esperava, os concorrentes nunca ocupam a mesma posição ao mesmo tempo, e devido ao cálculo da sua heurística, podemos verificar que os algoritmos de procura informada têm uma tendência a seguir o caminho que lhes permita acelerar mais/desacelerar menos, que neste ambiente permitem com que os seus resultados sejam tanto ou mais satisfatórios do que os de procura não-informada (neste caso apenas o algoritmo de procura em largura, visível no ambiente competitivo na figura 10), sendo o único problema que o jogador

que consegue ser o primeiro a jogar acaba por tendencialmente ser o que tem o melhor resultado.

8 Conclusão

Em jeito de conclusão, a realização do trabalho prático no geral, não só na primeira, como também na segunda, permitiu que consolidassemos diversos conhecimentos obtidos durante as aulas da Unidade Curricular de Inteligência Artificial, como a resolução de problemas através da conceção e implementação de algoritmos de procura, e também relativamente à linguagem de programação ***python***.

À semelhança da primeira fase, atribuímos um balanço bastante positivo ao trabalho desenvolvido pelo grupo num todo, sendo que, numa nota mais introspetiva, há coisas que poderiam ser melhoradas na conceção do trabalho, visto que a geração de grafos é uma tarefa muito demorada devido à maneira como a implementá-mos, às vezes os caminhos dos jogadores no modo competitivo cruzam-se, sendo que poderia-mos ter arranjado alguma maneira de verificar isso, e podiamos ter implementado mais algoritmos de procura.

9 Referências

Na entrega do último relatório faltou referir uma coisa acerca do nosso projeto. Ao desenhar no ecrã os caminhos, é possível ver setas a irem de um nodo para outro. Essa seta é desenhada na função com a seguinte assinatura:

```
1 def draw_arrow(
2         surface: pygame.Surface,
3         start: pygame.Vector2,
4         end: pygame.Vector2,
5         color: pygame.Color,
6         body_width: int = 2,
7         head_width: int = 4,
8         head_height: int = 2,):
```

Listing 1: Assinatura da função para desenhar uma seta no ecrã

Este código foi retirado da seguinte discussão no site *Reddit*:

https://www.reddit.com/r/pygame/comments/v3ofs9/draw$_{a}rrow_{f}unction$/.