Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de
Inteligência Artificial

Ano Letivo de 2022/2023

Relatório do Projeto
Implementação de Algoritmos de Pesquisa para Resolução autónoma do jogo Vector Race

Grupo 21

Luís Alberto Barreiro Araújo	A96351
Pedro Miguel da Cruz Pacheco	A61042
Pedro Calheno Pinto	A87983
Rafael Arêas	A86817

13 de Dezembro de 2022

Index

1 Introdução

Em 1958 Alan Turing questionou-se "Conseguem as máquinas pensar?". A partir dessa pergunta ele formulou um teste, atualmente chamado de "Teste de *Turing*", onde um interrogador humano tenta distinguir entre respostas dadas por uma pessoa e por um computador. Embora este teste tenha sofrido muito escrutínio desde a sua publicação, é inegável a sua contribuição para o campo da Inteligência Artificial.

Um dos manuais de Inteligência Artificial (AI) preponderantes é "*Artificial Intelligence: A Modern Approach*" de Stuart Russell e Peter Norvig. Nele, os autores debruçam-se sobre quatro potenciais objetivos ou definições de AI que diferenciam sistemas de computação da seguinte forma:

- **Human Approach**:

  Sistemas que pensam como humanos

  Sistemas que agem como humanos

- **Ideal Approach**:

  Sistemas que pensam racionalmente

  Sistemas que agem racionalmente

O conceito de *Turing* estaria inserido na categoria de "Sistemas que agem como humanos", mas um grande número de definições de Inteligência Artificial têm surgido ao longo das décadas, e no entanto, na sua forma mais simplista, Inteligência Artificial é o campo que combina Ciência da Computação com grandes *datasets* de modo a criar soluções para resolução de problemas.

Poderíamos continuar a aprofundar de uma forma extensa o campo da AI mas isso iria inevitavelmente sair do âmbito daquilo que é este projeto e, embora sendo um tema fascinante, ocuparia demasiado tempo ao leitor desviando-se simultaneamente da mensagem que queremos passar aqui. Consequentemente, manteremos o foco e falemos sobre uma das formas de guardar informação em memória que é usada por algoritmos habitualmente empregues em Inteligência Artificial - os Grafos.

Um grafo é um tipo de dados abstrato que pode ser usado para representar um conjunto complexo de ligações não-lineares entre objetos. Os elementos nos quais é armazenada a informação que compõe um grafo são denominados por nodos e as ligações entre os diferentes nodos são chamados de arestas. Este tipo de dados é usado nos mais diferentes contextos, como por exemplo numa rede de estradas em que os nodos de um grafo representam diversas cidades e as suas arestas as diferentes ligações rodoviárias entre essas cidades; ou até mesmo a própria Internet, onde os nodos simbolizam os diferentes *routers* e as arestas as ligações entre eles. Qualquer que seja a área usada, os grafos são estruturas que, quando devidamente implementados, podem modular de uma forma simples um problema que era inicialmente complexo.

Para que um grafo seja usado eficazmente torna-se necessário que as formas de fazer a travessia entre os seus nodos sejam também elas o mais eficaz possível, de modo a que um problema de pesquisa num grafo seja facilmente solucionado. Desta forma, ao longo dos anos foram-se desenvolvendo algoritmos com o intuito de encontrar estas soluções.

Para se solucionar um problema de pesquisa, é essencial não só encontrar uma sequência de ações que atinjam o objetivo esperado, mas que essa sequência seja a sequência ótima dentro de certos critérios. Formalmente, um problema de pesquisa define-se através da especificação dos seus componentes:

- O estado inicial em que se começa. Esse estado é normalmente um modelo matemático da realidade do problema;
- As ações que podem ser tomadas num qualquer estado;
- Um teste de verificação que averigua se um estado é o estado objetivo pretendido;
- Uma função 'custo' que atribui um custo numérico a um caminho entre dois estados;

Quaisquer que sejam os componentes, o objetivo será sempre encontrar a sequência de ações que corresponde ao caminho de menor custo entre o estado inicial e o estado final.

No campo da Inteligência Artificial, o uso dos algoritmos de travessia de grafos foi catalogado de duas formas - Algoritmos de Pesquisa Informada e Algoritmos de Pesquisa Não Informada.

A Pesquisa Não Informada, também às vezes chamada de Pesquisa Cega, não consegue diferenciar entre estados objetivo e estados não-objetivo, nem consegue inspecionar a estrutura interna de um estado para estimar quão próximo está do objetivo. Por outras palavras, este tipo de algoritmo não consegue saber se um estado está mais próximo do estado final do que o estado que o precedeu. Isto traduz-se em tempos de procura mais largos ou menos eficazes porque, dados dois estados s_a e s_b, o algoritmo pode escolher percorrer todos os sub-estados de s_a antes dos sub-estados de s_b, mesmo que s_b esteja mais próximo do estado objetivo. São exemplos de algoritmos de Pesquisa Informada, a Pesquisa em Largura ou a Pesquisa em Profundidade.

Em contraste, a Pesquisa Informada, usa estratégias com informação adicional àquela que é dada na definição do problema. Esta informação extra é calculada através de uma função chamada de 'Heurística' que recebe como input um estado e estima a sua proximidade ao estado objetivo. Com o uso de uma heurística, uma estratégia de procura consegue diferenciar entre estados objetivo e estados não-objetivo, e focar-se apenas naqueles que lhe parecem ser mais promissores. É por esta razão que as técnicas de Pesquisa Informada são mais rápidas a encontrar um qualquer estado final do que um algoritmo de Pesquisa Não Informada, contanto que a função de heurística seja bem definida. O algoritmo A* (A-Estrela) e o algoritmo de Pesquisa Gulosa são duas das mais famosas técnicas de Pesquisa Informada.

2 Caso de Estudo

2.1 Descrição do problema

Foi nos proposto em contexto académico a realização do jogo *VectorRace*, um jogo baseado num conjunto de movimentos e acelerações numa determinada direção escolhida por um jogador. A versão doravante apresentada neste relatório, diverge da versão clássica na medida em que há pouca intervenção humana, ou seja, a escolha de que aceleração impor e que direção escolher é toda calculada através do computador, mas as regras que estipulam o funcionamento do *VectorRace* continuam a ser aquelas que restringem esta versão.

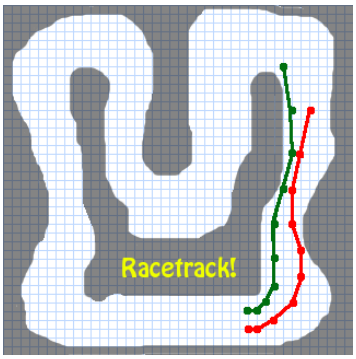

Figure 1: Exemplo de um circuito

O movimento do carro é implementado através de um conjunto de acelerações. 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 (linha e coluna). Consequentemente, para cada uma das direções o conjunto de acelerações possíveis é Acel = -1, 0, +1, com a $= (a_l, a_c)$ a representar a aceleração de um carro nas duas direções num determinado instante.

Tendo em conta o tuplo *p* que indica a posição de um carro numa determinada jogada *j* ($p_j = (p_l, p_c)$), e *v* o tuplo que indica a velocidade do carro nessa jogada ($v_j = ((v_l, v_c))$), na seguinte jogada o carro irá estar na posição:

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

A velocidade do carro num determinado instante é calculada:

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

Sendo uma simulação de corrida de carros, existe a possibilidade de o carro sair da pista, e quando essa situação ocorrer o carro volta para a posição anterior, assumindo um valor de velocidade zero. Cada movimento de um carro numa determinada jogada, de uma determinada posição para outra, terá um custo de 1 unidade, sendo que quando o mesmo sair dos limites da pista, o custo é de 25 unidades.

2.2 Representação do Circuito

A representação dos circuitos é feita através de ficheiros de texto com diferentes caracteres para cada tipo de peça. O exemplo seguinte apresenta uma pista de 10 colunas e 7 linhas com a posição inicial identificada por 'P' e três posições finais representadas por 'F'.

```
X X X X X X X X X X
X X - - - X X - - X
X - - - - - - - - F
X P - - X X X - - F
X - - - - - - - - F
X X X X - - - - X X
X X X X X X X X X X
```

Figure 2: Representação de um circuito em formato texto

2.3 Formulação do problema como um Problema de Pesquisa

Para a formulação do problema consideramos a notação *l*, que representa a linha e *c* a coluna para os vetores. Num determinado instante o carro pode avançar para qualquer peça adjacente, incluindo as peças diagonais e não pode avançar para fora da pista. Para a representação das peças no circuito declaramos da seguinte forma:

- As peças válidas do circuito são representadas por '-'
- As peças que representam as barreiras do circuito são os 'X'
- A peça que representa o ponto de partida de um carro no circuito é o 'P'
- A peça que representa a chegada, o ponto final do circuito é o 'F'

Para resolver este problema, tornou-se vital transformar circuitos inicialmente em formato texto em representações em memória guardadas em grafos e implementar algoritmos de procura que encontram a melhor sequência de ações que levam ao estado final desejado, neste caso em particular, descobrir o melhor caminho para percorrer o circuito desde a posição de partida até à posição da meta. Esta solução consistirá sempre na sequência de ações com o menor custo.

Utilizando a notação em cima, definimos então as componentes do Problema de Pesquisa da seguinte forma:

- Estados Inicial e Final: respetivamente os nodos do grafo com a posição 'P' e a posição 'F'
- Possíveis ações: dado um nodo *n*, deslocar-se para qualquer um dos nodos adjacentes a *n*
- Teste de verificação: dado um nodo *n*, verificar se ele contém a peça 'F'
- Função Custo: dado um nodo *n*, a soma do custo acumulado desde o nodo inicial até *n*

3 Descrição da Solução Proposta para o Caso de Estudo

3.1 Circuitos Criados e Usados

Em seguida apresentamos os circuitos que criamos na elaboração deste trabalho e nos quais foram testados os algoritmos de procura implementados.

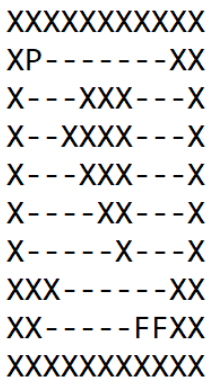

Figure 3: Circuito 1

Posição inicial : (1, 1, 'P') Posição final : (8, 8, 'F')

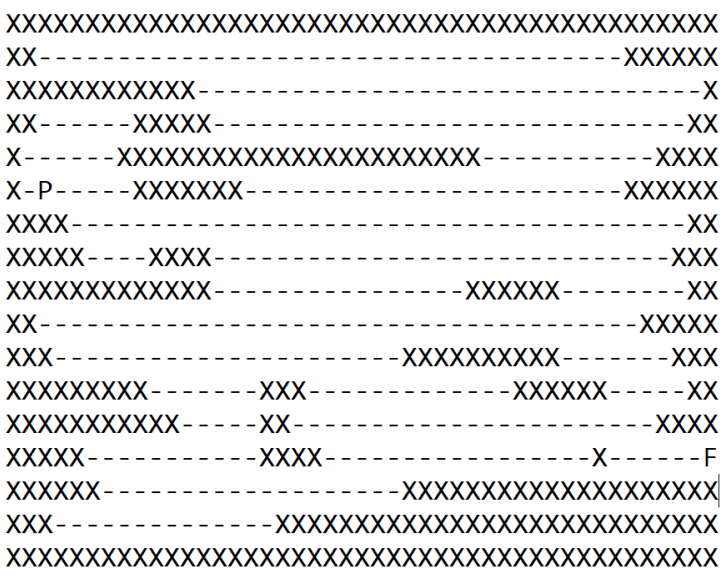

Figure 4: Circuito 2

Posição inicial : (5, 2, 'P') Posição final : (13, 44, 'F')

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXP------------------------------------XXXXXX
XX--------------------------------------XXXXX
XX------XXXXXXXXXXXXXXXXXXXXXXX-----------XXX
X-------XXXXXXXXXXXXXXXXXXXXXX-----------XXXX
XXX------XXXXXXXX------------------------XXXX
XXXX------XXXXX-----------------XXXXXXXXXXXXX
XXXXX------XXXX--------XXXXXXXXXXXXXXXXXXXXXX
XXXXX-------XXXX--------------XXXXXXXXXXXXXXX
XXXX-------XXXXXXX-------------------XXXXXXXX
XXX--------XXXXXXXXXXXX------------------XXXX
XXXXX-------XXXXXXXXXXXXXXXXXX-------------XX
XXXXXX-------XXXXXXXXXXXXXXXXXXX------------F
XXXXXXX--------XXXXXXXXX-----------------XXXX
XXXXXXXX------------------------------XXXXXXX
XXXXXXXXXXX--------------------XXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure 5: Circuito 3

Posição inicial : (1, 2, 'P') Posição final : (12, 44, 'F')

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXX----------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX-------F-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXX-------XXXX--------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXX------XXXXXXXX---------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXX------XXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXX------XXXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXX------XXXXXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX------XXXXXXXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX------XXXXXXXXXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX------XXXXXXXXXXXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X------XXXXXXXXXXXXXXXXXXXXXXX--------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X-----XXXXXXXXXXXXXXXXXXXXXXXXXXX--------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X-----XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X-----XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX---------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X------------XXXXXXXXXXXXXXXXXXXXXXXXXXXXX-------------XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX----------------------XXXXXXXXXXXXXXXXXXXXXXXX--------------XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXX----------------------XXXXXXXXXXXXXXXXXXXXXXX--------------XXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXX-------------------XXXXXXXXXXXXXXXXXXXXXX-------------XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXX-----------------------XXXXXXXXXXXXXXXXXXXXX-------------XXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX------P--------------XXXXXXXXXXXXXXXXXXXXX--------------XXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX--------------------XXXXXXXXXXXXXXXXXXX------------XXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX--------------------XXXXXXXXXXXXXXXX----------XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX---------------------XXXXXXXX------------XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX---------------------------XXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-------------------XXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX------------XXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Figure 6: Circuito 4

Posição inicial : (21, 34, 'P') Posição final : (3, 15, 'F')

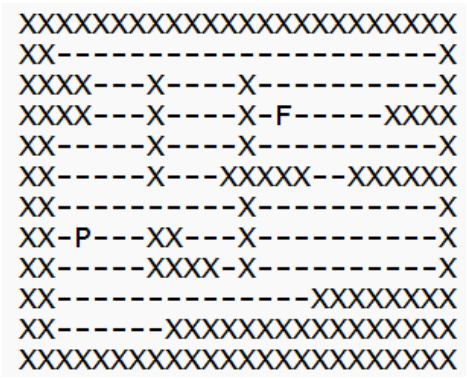

Figure 7: Circuito 5

Posição inicial : (7, 3, 'P') Posição final : (3, 14, 'F')

3.2 Construtores e Criação de Estruturas

A resolução do problema proposto começou por se fazer a partir da leitura dos ficheiros de texto que contêm os vários circuitos para uma matriz (para cada circuito) de tuplos de quatro posições - (x, y, peça, distância). Cada tuplo representa uma posição em que o veículo pode estar, sendo x e y as suas coordenadas na matriz, peça o tipo de coordenada que é (uma casa livre, uma parede, uma posição final ou a posição inicial) e distância a distância euclidiana desde aquela coordenada até à coordenada da posição final.

Esta matriz de tuplos é posteriormente usada para criar um grafo com a sua informação. Com esse intuito, foi desenvolvida uma classe 'Nodo' com a seguinte implementação:

```
def __init__(self, no):
        self.m_name = "(" + str(no[0]) + ", " + str(no[1]) + ") : " + str(no[2]) + str
    (no[3])
        self.x = no[0]
        self.y = no[1]
        self.peca = no[2]
        self.dist = no[3]
```

Listing 1: Construtor da classe Nodo

O construtor do Nodo usa o tuplo da matriz para criar os seus campos. Estes nodos serão a parte central de um grafo (definido na estrutura *Graph*) que vai representar todas as possíveis deslocações no circuito.

A estrutura *Graph* é então criada à custa de dois métodos: *createGraph()* e *addedge()*. Em *createGraph()* a matriz de tuplos é lida e para cada dois tuplos em posições adjacentes é invocado o método *addedge()*. Aqui são criados os nodos com base nos tuplos parametrizados através do construtor da classe 'Nodo', e adicionado ao dicionário *m_graph* a existência de uma aresta que liga os dois nodos com o seu respetivo custo.

De notar que para as classes '*Graph*' e 'Nodo', um ponto de partida para a nossa implementação foi parte do código utilizado nas aulas práticas da UC de Inteligência Artificial.

```
def __init__(self, directed=False):
    self.m_nodes = []
    self.m_directed = directed
    self.m_graph = {}
    self.m_h = {}
```

Listing 2: Construtor da classe Graph

```
with open("circuito1.txt") as file1:
    matrix4 = ([list(line.strip().replace(" ", "")) for line in file1.readlines()
])

def calculaDist(node1, node2):
    a = pow((node2[0] - node1[0]),2) + pow((node2[1] - node1[1]),2)
    b = math.sqrt(a)
    n = list(node1)
    n[3] = round(1.5*b,2)
    return n

def createMatrix(matrix):
    mat = []
    matAux = []
    for x, line in enumerate(matrix):
        for y, piece in enumerate(line):
            matAux.append((x, y, piece, 0))
            if (piece == "F"):
                nodeFinal = (x, y, piece, 0)

    for n in matAux:
        a = calculaDist(n, nodeFinal)
        n = tuple(a)
        mat.append(n)

    return mat

def createGraph(input):
    g=Graph()
    if(input==1):
        mat=createMatrix(matrix1)
    elif(input==2):
        mat = createMatrix(matrix2)
    elif(input==3):
        mat=createMatrix(matrix3)
    elif(input==4):
        mat=createMatrix(matrix4)
    else:
        mat = createMatrix(matrix5)

    for index, node in enumerate(mat):
```

```
            for node2 in mat[index:]:
                if (node2[0] == (node[0] + 1) and node2[1] == node[1]) or (
                        node2[0] == (node[0] - 1) and node2[1] == node[1]) or (
                        node2[0] == node[0] and node2[1] == (node[1] + 1)) or (
                        node2[0] == node[0] and node2[1] == (node[1] - 1)) or (
                        node2[0] == (node[0] - 1) and node2[1] == (node[1] - 1)) or (
                        node2[0] == (node[0] + 1) and node2[1] == (node[1] - 1)) or (
                        node2[0] == (node[0] - 1) and node2[1] == (node[1] + 1)) or (
                        node2[0] == (node[0] + 1) and node2[1] == (node[1] + 1)):
                    if (node[2] != "X" and node2[2] != "X"):
                        g.addedge(node, node2, 1, 1)
                    if (node[2] != "X" and node2[2] == "X"):
                        g.addedge(node, node2, 25, 1)
                    if (node[2] == "X" and node2[2] != "X"):
                        g.addedge(node, node2, 1, 25)

        return g
```

Listing 3: Exemplo da transformação de um circuito.txt num grafo através dos vários métodos usados.

3.3 Algoritmos Implementados

3.3.1 Pesquisa Não Informada

Para este tipo de algoritmos de procura, onde não é levada em conta uma heurística auxiliar, foram implementados dois tipos de algoritmos: o de procura DFS(Depth-first Search) e o de Procura BFS (Breadth-first search).

- DFS

Primeiramente apresentamos o algoritmo de procura DFS, que se baseia na estratégia de uma procura em profundidade que consiste na expansão recursiva de um dos nodos de uma árvore de nodos, até que seja encontrado o mais profundo. Embora com a possibilidade de ser muito mais rápido que o algoritmo BFS (ver mais abaixo), é demarcado pelo facto de o resultado desta procura ser o primeiro caminho encontrado entre o nodo origem e o nodo destino, não sendo esse caminho necessariamente o mais curto.

```
def procura_DFS(self, start, end, path=[], visited=set()):
    path.append(start)
    visited.add(start)

    if start == end:
        custoT = self.calcula_custo(path)
        return path, custoT
    for (adjacente, peso) in self.m_graph[start]:
        if adjacente not in visited and peso!=25:
            resultado = self.procura_DFS(adjacente, end, path, visited)
            if resultado is not None:
                return resultado
    path.pop()
    return None
```

Listing 4: Algoritmo de procura DFS.

Começa por se declarar duas variáveis - *path* e *visited* - a primeira diz respeito ao caminho percorrido e a segunda aos nodos já visitados. O *nodo_inicial* é adicionado às duas e é feita a verificação se é o *nodo_final*. Caso seja, é calculado o custo do *path* e retornado o seu valor junto com o *path*, caso não seja, para cada nodo adjacente *m* ao *nodo_inicial*, se ele não estiver na lista de visitados é chamado recursivamente o método com o *nodo_inicial* igual a *m*, e o *path* e *visited* até àquele momento.

- BFS

Em seguida apresentamos o algoritmo de procura BFS, que consiste num algoritmo de procura em largura e tem como estratégia expandir todos os nodos de menor profundidade. Uma das mais valias deste algoritmo é a capacidade que ele tem de calcular o melhor caminho entre dois nodos (leia-se, o de menor custo), mas este benefício é contraposto com a sua necessidade de um maior tempo de execução.

```
def procura_BFS(self, start, end):
    visited = set()
    fila = Queue()

    fila.put(start)
    visited.add(start)

    parent = dict()
    parent[start] = None

    path_found = False
    while not fila.empty() and path_found == False:
        nodo_atual = fila.get()
        if nodo_atual == end:
            path_found = True
        else:
            for (adjacente, peso) in self.m_graph[nodo_atual]:
                if adjacente not in visited and peso!=25:
                    fila.put(adjacente)
                    parent[adjacente] = nodo_atual
                    visited.add(adjacente)

    path = []
    if path_found:
        path.append(end)
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse()
        custo = self.calcula_custo(path)
    return path, custo
```

Listing 5: Algoritmo de procura BFS.

Inicialmente são declaradas três variáveis *visited*, *fila* e *parents*{}. A primeira diz respeito à lista de nodos já visitados, a segunda à lista de nodos para visitar e a terceira é um dicionário que guarda para cada nodo já visitado qual nodo é o seu pai. *fila* e *visited* são inicializados com o *nodo_inicial* e *parents[nodo_inicial]* inicializado a *None*. Enquanto houver elementos em *fila* e nenhum desses elementos for o *nodo_final*, procura-se a lista de nodos adjacentes a cada um e caso não estejam em *visited* adicionam-se a *fila* e a *visited* e atualizam-se os seus nodos pai. Assim que a fila estiver vazia ou se tenha encontrado o *nodo_final*, reconstrói-se o caminho desde *nodo_inicial* até *nodo_final* e calcula-se o seu custo devolvendo esses dois valores.

3.3.2 Pesquisa Informada

De maneira a encontrar os caminhos mais curtos de forma mais rápida e inteligente, foram implementados dois algoritmos de Pesquisa Informada - o A* e o Pesquisa Gulosa (*Greedy*) - usando dois tipos de heurísticas - uma contando apenas com a distância euclidiana desde um nodo até ao nodo final, e outro fazendo o uso dessa distância assim como a aplicação de um sistema de movimentação através de acelerações e desacelerações falado na secção 2.

Usando Distância Euclidiana como Heurística : *procura_aStar()* e *Greedy()*

Nota: estes algoritmos foram parcialmente importados da implementação realizada nas aulas da UC e foram modificados com o intuito de os moldar à estrutura deste projeto.

- *Algoritmo A**

```
def procura_aStar(self, start, end):
    g = {}

    g[start] = 0

    parents = {}
    parents[start] = start
    n = None
    while len(open_list) > 0:
        function
        calc_heurist = {}
        flag = 0
        for v in open_list:
            if n == None:
                n = v
            else:
                flag = 1
                calc_heurist[v] = g[v] + self.getH(v)
        if flag == 1:
            min_estima = self.calcula_est(calc_heurist)
            n = min_estima
        if n == None:
            print('Path does not exist!')
            return None

        start_node
        if n == end:
            reconst_path = []

            while parents[n] != n:
                reconst_path.append(n)
                n = parents[n]

            reconst_path.append(start)

            reconst_path.reverse()
```

```
            return (reconst_path, self.calcula_custo(reconst_path))


        for (m, weight) in self.getNeighbours(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight

            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None
```

Listing 6: Algoritmo de Procura A*.

Para o algoritmo A* que usa apenas a distância euclidiana como heurística criaram-se dois métodos auxiliares - *getH()* e *calcula_est()* - que ajudam na obtenção da distância euclidiana de cada nodo ao nodo final e na determinação de qual nodo contém o valor de heurística mais baixo numa lista de nodos. Posto isto, eis o funcionamento da pesquisa A*:

São criadas duas variáveis chamadas *open_list* e *closed_list* e dois dicionários - $g\{\}$ e $parents\{\}$. As primeiras duas são respetivamente uma lista de nodos que já foram visitados mas cujos nodos adjacentes ainda não foram todos inspecionados; e uma lista de nodos que já foram visitados com todos os nodos adjacentes inspecionados. *open_list* é inicializada com o *nodo_inicial*. $g\{\}$ e $parents\{\}$ guardam relativamente a cada nodo o seu custo acumulado assim como a informação do nodo pai. O primeiro é inicializado a zero para o *nodo_inicial*, e o segundo como sendo ele mesmo o seu pai. Começa então um ciclo que decorrerá enquanto houver elementos na *open_list*. Para cada elemento *n* na *open_list* é calculada a sua heurística e adicionado à lista de heurísticas $calc\_heurist\{\}$. Usando o método *calcula_est()* é calculado o nodo com menor valor dessa lista e verificado se ele é o *nodo_final*. Caso seja, é reconstruído o caminho desde o *nodo_inicial* até ele e calculado o seu custo total, devolvendo esses dois valores; caso não seja usa-se o método *getNeighbours()* para calcular todos os nodos *m* adjacentes a *n*. Averigua-se depois se este nodo *m* já foi previamente visitado e caso contrário adiciona-se a *open_list* e atualiza-se o seu pai e o seu custo acumulado. Na hipótese de *m* já ter sido visitado antes, compara-se o custo total por *n* com aquele já guardado em *g[m]*: se mais barato substitui-se o custo acumulado e o nodo pai, fazendo as adições e remoções de *open_list* e *closed_list* que sejam necessárias.

- *Algoritmo Pesquisa Gulosa (Greedy)*

```
def greedy(self, start, end):
    open_list = set([start])
    closed_list = set([])

    parents = {}
    parents[start] = start

    while len(open_list) > 0:
        n = None

        for v in open_list:
            if n == None or self.m_h[v] < self.m_h[n]:
                n = v

        if n == None:
            print('Path does not exist!')
            return None

        if n == end:
            reconst_path = []

            while parents[n] != n:
                reconst_path.append(n)
                n = parents[n]

            reconst_path.append(start)

            reconst_path.reverse()

            return (reconst_path, self.calcula_custo(reconst_path))

        for (m, weight) in self.getNeighbours(n):
            list
            if m not in open_list and m not in closed_list and weight!=25:
                open_list.add(m)
                parents[m] = n

        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None
```

Listing 7: Algoritmo de Pesquisa Gulosa.

O algoritmo Greedy começa por inicializar as variáveis *open_list*, *closed_list* e *parents*{} de forma análoga ao A*. A sua execução gira em torno da presença de nodos na *open_list*. Caso existam, é selecionado o nodo *n* com menor distância ao *nodo_final*, e os nodos adjacentes a este que não estejam presentes na lista são adicionados a ela e ao dicionário de pais. Assim que todos os adjacentes a *n* estejam em *open_list*, este nodo é removido dela e adicionado a *closed_list*. A execução termina assim que encontrar o *nodo_final* na *open_list* sendo devolvido o caminho até ele e o seu custo.

Usando Distância Euclidiana e Aceleração como Heurística : *procura_aStar_velocidade()*

- *Procura A* com Velocidade*

```
def procura_aStar_velocidade(self, start, end):
    open_list = {start}
    closed_list = set([])

    g = {}
    vels = {}
    g[start] = 0
    vels[start] = (0,0)

    parents = {}
    parents[start] = start
    n = None
    while len(open_list) > 0:
        calc_heurist = {}
        flag = 0
        for v in open_list:
            if n == None:
                n = v
            else:
                flag = 1
                calc_heurist[v] = g[v] + self.getH(v)
        if flag == 1:
            min_estima = self.calcula_est(calc_heurist)
            n = min_estima
        if n == None:
            print('Path does not exist!')
            return None

        if n == end:
            reconst_path = []

            while parents[n] != n:
                reconst_path.append(n)
                n = parents[n]

            reconst_path.append(start)
            reconst_path.reverse()

            return (reconst_path, g[end])

        aux = self.getPossiveisPosicoes(n, vels[n])
        paredes = self.bateParede(n, aux)

        for (m) in aux:

            if m in paredes:
                nv, acc = self.actualiza_velocidade(n,m,vels[n])
                new = self.checkPosicaoSeguinte(n, acc)
                if new not in open_list and new not in closed_list:
                    open_list.add(new)
                    parents[new] = n
                    g[new] = g[n] + 25
                    vels[new] = (0,0)
                    self.escreveficheiro([n,new,(0,0),(0,0), vels[n]])
                else:
                    if g[new] > g[n] + 25:
```

```
                    g[new] = g[n] + 25
                    vels[new] = (0,0)
                    self.escreveficheiro([n,new,(0,0),(0,0), vels[n]])
                    parents[new] = n

                    if new in closed_list:
                        closed_list.remove(new)
                        open_list.add(new)
            else:
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + 1
                    nv, acc = self.actualiza_velocidade(n,m,vels[n])
                    vels[m] = nv
                    self.escreveficheiro([n,m,nv,acc, vels[n]])

                else:
                    if g[m] > g[n] + 1:
                        g[m] = g[n] + 1
                        nv, acc= self.actualiza_velocidade(n,m,vels[n])
                        vels[m] = nv
                        self.escreveficheiro([n,m,nv,acc,vels[n]])
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None
```

Listing 8: Algoritmo de Procura A* com distância e velocidade como heurística.

Nesta versão do algoritmo A*, para além dos dois métodos descritos anteriormente (*calcula_est()* e *getH()*), foram criados outros que auxiliam no funcionamento do algoritmo. Eis uma pequena síntese desses métodos e o que cada um faz:

- **getPossiveisPosicoes()**: Este método recebe como argumento um grafo, um nodo nesse grafo e uma velocidade, e calcula quais os nodos possíveis de alcançar no grafo partir do nodo inicial com a velocidade estipulada e diferentes possíveis acelerações.
- **bateParede()**: À semelhança do método anterior, este pega num nodo inicial num grafo e uma lista de nodos alcançáveis a partir dele (calculada através de *getPossiveisPosicoes()*) e verifica agora quais desses nodos o carro teria que bater numa parede para o poder alcançar. Este método faz uso de outros dois: *bresenham()*[1]: que verifica em que coordenadas uma linha entre dois nodos passa; e *checkWall()* que verifica se um nodo no grafo é um elemento parede.
- **actualiza_velocidade()**: Método que calcula a velocidade final que o carro terá depois de se mover de um $nodo_a$ para um $nodo_b$ com $velocidade(a)_i$. Retorna essa nova velocidade e qual a aceleração que o carro sofreu.

[1] mais informação sobre o Algoritmo de Bresenham pode ser encontrada através do link na bibliografia

- **checkPosicaoSeguinte()**: Finalmente, aqui é calculada uma posição final para um carro que bateu na parede. Este cálculo é feito à base do nodo de onde o veículo partiu e qual o vetor aceleração a que foi sujeito. É verificado se o nodo adjacente ao nodo inicial na direção do vetor aceleração é uma posíção do grafo viável. Se for, devolve essa posição, se não for, verifica qual dos outros nodos adjacentes o é.

Explicados cada um dos métodos auxiliares, descrevemos agora o funcionamento do algoritmo A* usando heurística como velocidade.

É começado por se criar duas variáveis chamadas *open_list* e *closed_list* que consistem respetivamente numa lista de nodos que já foram visitados mas cujos nodos adjacentes ainda não foram todos inspecionados; e uma lista de nodos que já foram visitados e todos os nodos adjacentes já foram inspecionados. A primeira é inicializada com o *nodo_inicial*.

São depois criados três dicionários, *g*{}, *vels*{} e *parents*{} que guardam relativamente a cada nodo o seu custo e velocidade acumuladas assim como a informação do nodo pai. Os primeiros dois são inicializados a zero para o *nodo_inicial*, e o terceiro como sendo ele mesmo o seu pai. É então inicializada a execução do algoritmo que consiste em ir sucessivamente verificando os nodos que estão contidos em *open_list*, removendo e adicionando nodos quando necessário. De notar que, da maneira que implementamos este algoritmo, o custo acumulado em *g*{*n*} para um qualquer *n* depende da velocidade acumulada até então, pois se o carro tiver um excesso de velocidade irá sem dúvida embater em parede e o seu custo acumular 25 unidades.

De uma forma simples, o funcionamento deste algoritmo é o seguinte: para cada nodo *v* em *open_list* é calculada a sua heurística (o custo acumulado até *v* + a distância euclidiana até ao *nodo_final*); essa heurística é adicionada a um dicionário e depois o método *calcula_est()* irá escolher o nodo *n* com valor mais baixo desse dicionário; é verificado se *n* é o *nodo_final* - se for, é reconstruído e devolvido o caminho desde o *nodo_inicial* até ele, assim como o custo acumulado; se não for, são calculados quais os nodos possíveis de alcançar a partir de *n* (guardados em *aux*) e quais aqueles que bateu na parede para alcançar (guardados em *paredes*); seguidamente, para cada elemento *m* em *aux* verifica se ele também está em *paredes* - se está, calcula qual a aceleração que o carro sofre para chegar de *n* a *m*, procura por um nodo adjacente (*new*) a *n* que seja viável, i.e. sem parede, e confere se *new* não está em *open_list* nem em *closed_list*. Caso não esteja, adiciona-o a *open_list*, atualiza o seu pai como sendo *n* e o seu custo e velocidade acumuladas como sendo *g[n]* + 25 e (0,0) (pois bateu em parede); caso não esteja verifica se o custo para chegar a *new* batendo em parede é mais barato do que aquele previamente guardado em *g[new]* e assim sendo faz as necessárias modificações a *g[new]*, *vels[new]* e *parents[new]* removendo-o da *closed_list* e adicionando-o a *open_list* se necessário.

Se *m* não estiver em *paredes* nem em *open_list* nem em *closed_list* então isso significa que *m* é um nodo viável e que é a primeira vez que está a ser visitado. É adicionado a *open_list*, calculados e atualizados os seus novos custo e velocidade acumulados assim como o seu nodo pai. Se *m* não estiver em *paredes* mas estiver em *closed_list* ou *open_list* então é porque *m* é um nodo viável mas já foi previamente visitado. É portanto necessário investigar se o seu novo custo acumulado por *n* é mais baixo que aquele guardado e fazer as consequentes alterações em *g*{}, *vels*{} e *parents*{}, assim como remoção e adição de *m* de *closed_list* e *open_list* se necessário.

4 Menu Interativo

O Menu Interativo é a face do projeto. Ele permite a escolha e visualização das diferentes funcionalidades implementadas pela equipa no desenvolvimento do trabalho. Inicialmente, o programa pede ao utilizador que insira qual o circuito no qual pretende trabalhar e é seguidamente apresentado com as várias opções que pode escolher.

```
Escolha o circuito: 1, 2, 3, 4 ou 5
1
1-Imprimir Grafo
2-Desenhar Grafo
3-Imprimir nodos de Grafo
4-Imprimir arestas de Grafo
5-DFS
6-BFS
7-A*
8-Greedy
9-A* Velocidade
10-Escolher Circuito
11-Imprimir Circuito
12-Simular corrida
0-Saír
introduza a sua opcao->
```

Figure 8: Menu de Utilizador

Como é possível observar na figura 8, o utilizador tem acesso a várias formas de representação dos grafos do circuito escolhido, sejam elas por via gráfica ou textual, assim como a possibilidade de executar os vários algoritmos implementados.

Eis algumas funcionalidades implementadas:

- Impressão dos nodos de um Grafo

```
introduza a sua opcao-> 3
dict_keys([(1, 1, 'P', 14.85), (1, 2, '-', 13.83), (1, 3, '-', 12.9), (1, 4, '-', 12.09), (1, 5, '-', 11.42), (1, 6, '-', 10.92), (1, 7, '-',
10.61), (1, 8, '-', 10.5), (2, 1, '-', 13.83), (2, 0, 'X', 15.0), (2, 2, '-', 12.73), (2, 3, '-', 11.72), (2, 4, 'X', 10.82), (2, 5, 'X',
10.06), (2, 6, 'X', 9.49), (2, 7, '-', 9.12), (2, 8, '-', 9.0), (1, 9, 'X', 10.61), (2, 9, '-', 9.12), (3, 1, '-', 12.9), (3, 0, 'X', 14.15), (
3, 2, '-', 11.72), (3, 3, 'X', 10.61), (3, 4, 'X', 9.6), (3, 7, '-', 7.65), (3, 6, 'X', 8.08), (3, 8, '-', 7.5), (3, 9, '-', 7.65), (2, 10,
'X', 9.49), (3, 10, 'X', 8.08), (4, 1, '-', 12.09), (4, 0, 'X', 13.42), (4, 2, '-', 10.82), (4, 3, '-', 9.6), (4, 7, '-', 6.18), (4, 6, 'X',
6.71), (4, 8, '-', 6.0), (4, 9, '-', 6.18), (4, 10, 'X', 6.71), (5, 1, '-', 11.42), (5, 0, 'X', 12.82), (5, 2, '-', 10.06), (5, 3, '-', 8.75), (
4, 4, 'X', 8.49), (5, 4, '-', 7.5), (5, 7, '-', 4.74), (5, 6, 'X', 5.41), (5, 8, '-', 4.5), (5, 9, '-', 4.74), (5, 10, 'X', 5.41), (6, 1, '-',
10.92), (6, 0, 'X', 12.37), (6, 2, '-', 9.49), (6, 3, '-', 8.08), (6, 4, '-', 6.71), (5, 5, 'X', 6.36), (6, 5, '-', 5.41), (6, 7, '-', 3.35), (
6, 6, 'X', 4.24), (6, 8, '-', 3.0), (6, 9, '-', 3.35), (6, 10, 'X', 4.24), (7, 0, 'X', 12.09), (7, 1, 'X', 10.61), (7, 2, 'X', 9.12), (7, 3,
'-', 7.65), (7, 4, '-', 6.18), (7, 5, '-', 4.74), (7, 6, '-', 3.35), (7, 7, '-', 2.12), (7, 8, '-', 1.5), (7, 9, 'X', 2.12), (7, 10, 'X',
3.35), (8, 2, '-', 9.0), (8, 3, '-', 7.5), (8, 4, '-', 6.0), (8, 5, '-', 4.5), (8, 6, '-', 3.0), (8, 7, 'F', 1.5), (8, 8, 'F', 0.0), (8, 9,
'X', 1.5), (9, 1, 'X', 10.61), (9, 2, 'X', 9.12), (9, 3, 'X', 7.65), (9, 4, 'X', 6.18), (9, 5, 'X', 4.74), (9, 6, 'X', 3.35), (9, 7, 'X',
2.12), (9, 8, 'X', 1.5), (9, 9, 'X', 2.12)])
prima enter para continuar
```

Figure 9: Nodos do Grafo referente ao circuito 3

- Impressão do Grafo em formato texto

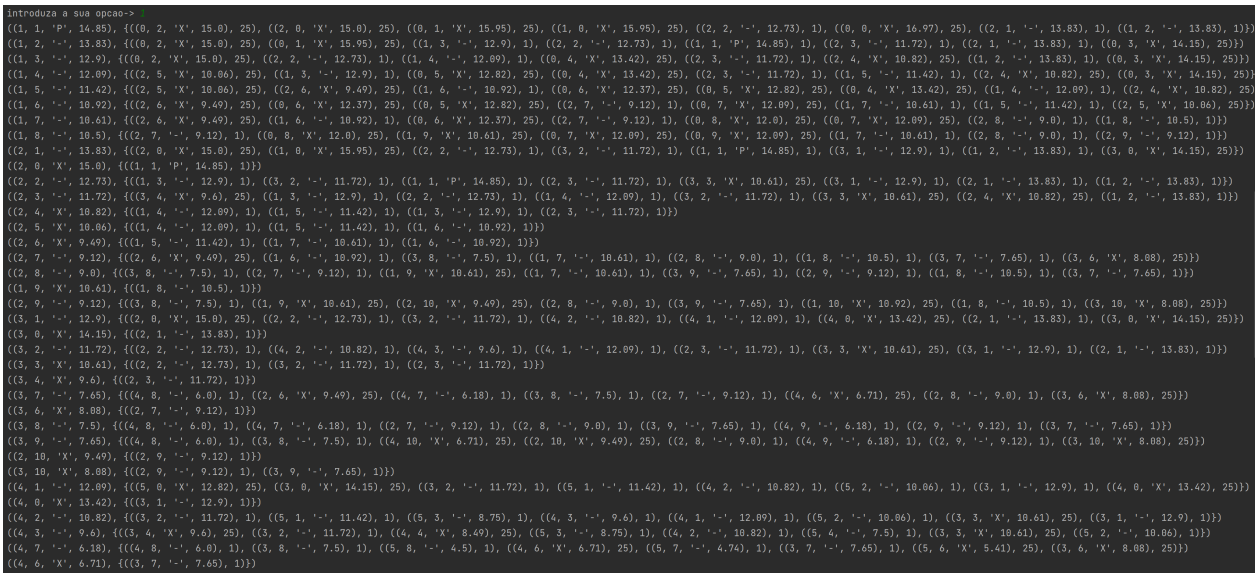

Figure 10: Excerto da representação textual do Grafo referente ao Circuito 1

- Desenho do Grafo

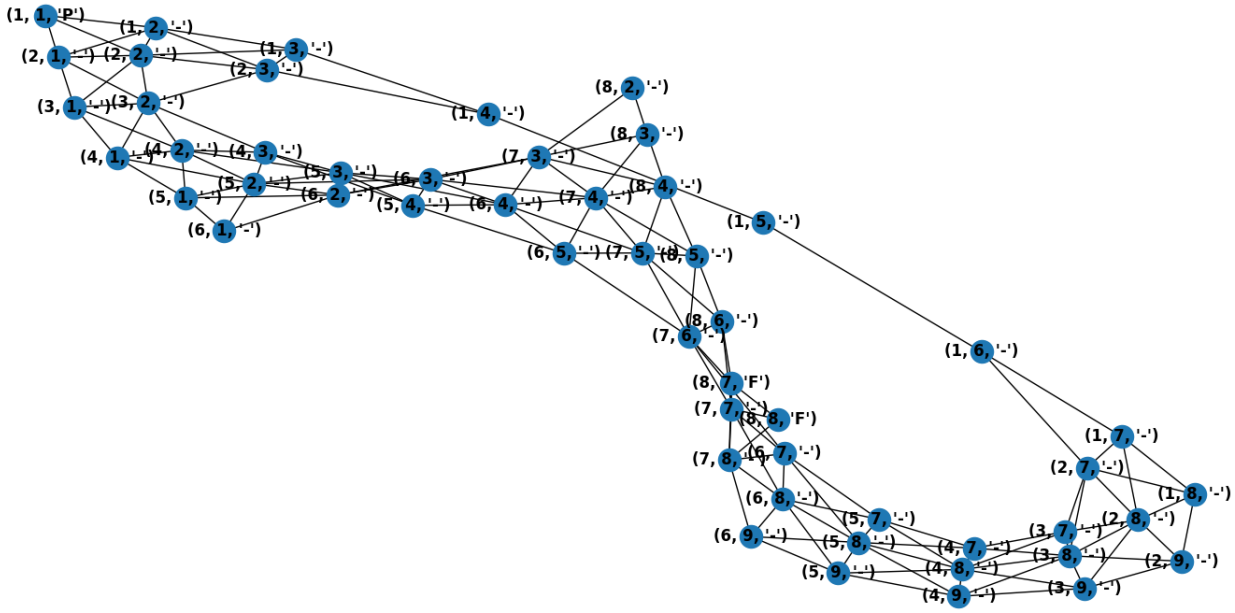

Figure 11: Representação Gráfica do Grafo do Circuito 1

- Execução do Algoritmo BFS

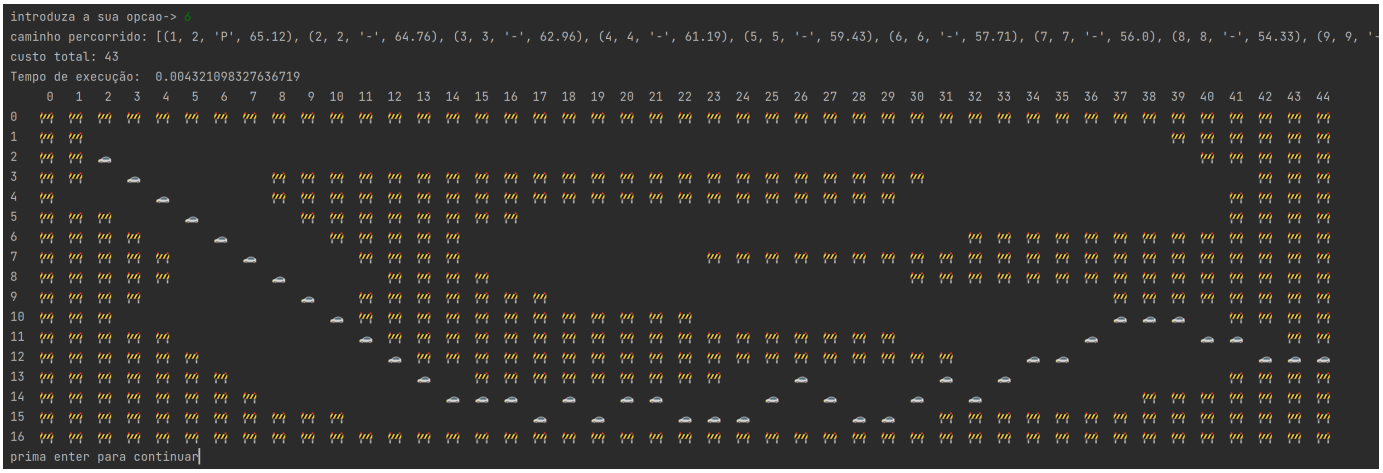

Figure 12: Execução do Algoritmo BFS para o circuito 3

- Simulação de uma corrida com dois jogadores

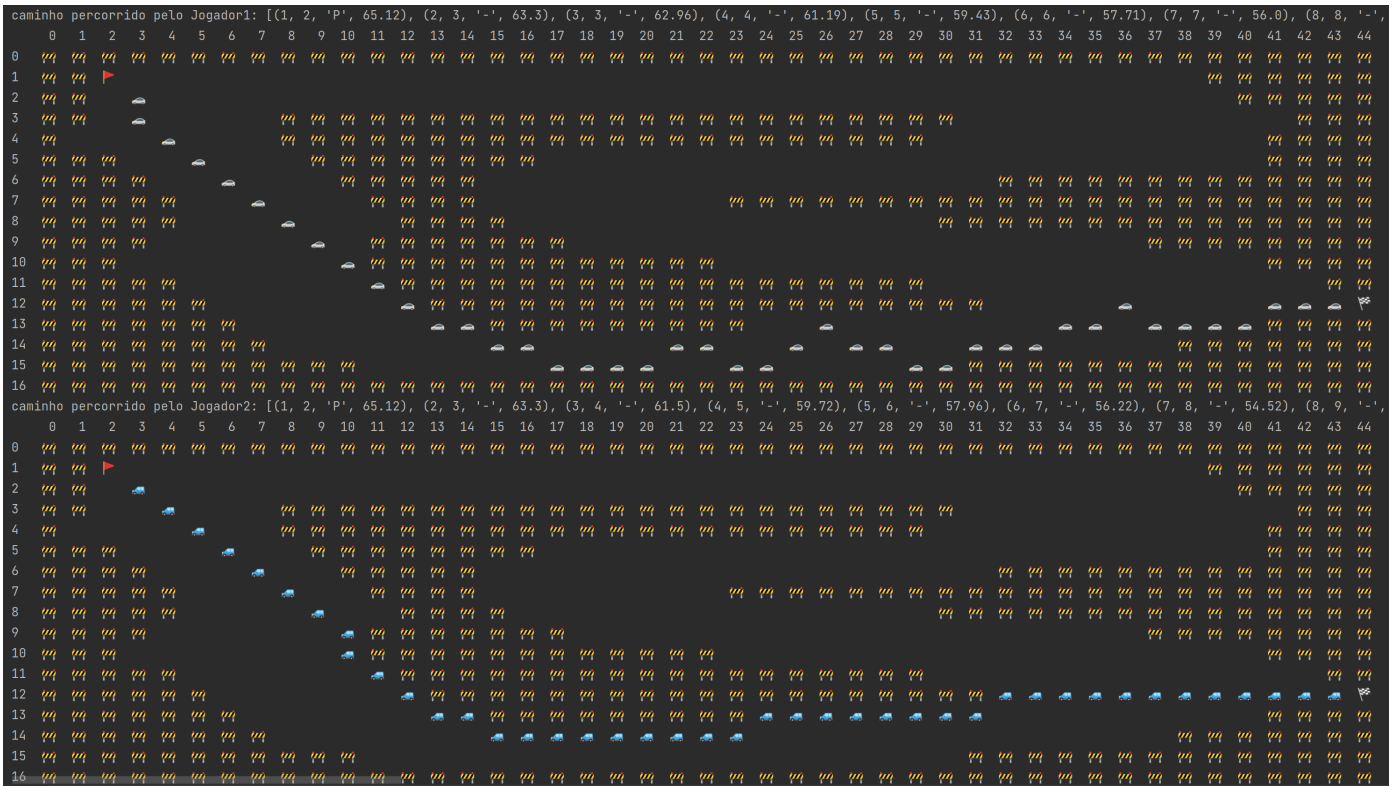

Figure 13: Exemplo de dois algoritmos a correrem um contra o outro. Em cima: BFS, Em baixo: Greedy

5 Discussão das decisões tomadas e Comparação dos resultados obtidos

Aquando do desenvolvimento dos algoritmos de Pesquisa Não Informada a equipa debateu qual seria a melhor forma de lidar com a fórmula de cálculo dos custos do trajeto. Decidiu-se então não contabilizar as arestas que vão para os nodos que modulam uma coordenada do circuito em que existe uma parede, efetivamente impedindo que o carro vá contra a parede e procure o melhor caminho à volta da mesma. Esta escolha traduz-se na impossibilidade de o carro sair fora da pista e 25 unidades serem somadas ao seu custo acumulado.

Já durante a implementação das técnicas de Pesquisa Informada, os nodos parede foram tomados em consideração. A maneira que a equipa decidiu resolver essa possibilidade foi a de, sempre que o carro bater numa parede, somar 25 unidades ao custo acumulado e reduzir a velocidade acumulada para 0 (tal como era proposto nas regras do problema) e o carro voltar a uma posição adjacente àquela de onde tinha partido. Foram implementados métodos para, se possível, a posição adjacente ser aquela na direção do vetor aceleração da jogada que bateu na parede.

Seguidamente apresentam-se comparações dos resultados obtidos usando os diferentes algoritmos num mesmo circuito. De modo a não ocupar imenso espaço, os resultados apresentados dirão respeito ao Circuito 1 pois este é o mais curto.

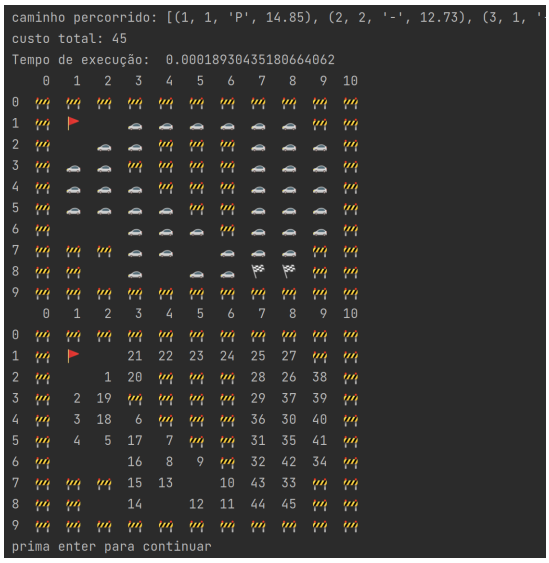

Figure 14: Solução do Algoritmo DFS - Representação Gráfica e Representação passo a passo do caminho percorrido

Custo Total = 45, Tempo de execução = 0.000189 segundos

Como se pode observar na figura 14, embora o tempo de execução seja bastante rápido, a procura em profundidade é um algoritmo bastante ineficiente pois tenta "às cegas" encontrar o primeiro caminho entre o nodo inicial e o nodo final, algo que tem grande probabilidade de devolver alto custo.

Figure 15: Solução do Algoritmo BFS

Custo Total = 8, Tempo de execução = 0.00105 segundos

Embora ligeiramente mais lento, podemos claramente ver que a procura em largura é imensamente mais eficaz que a procura em profundidade, o que faz com que o custo seja muito menor.

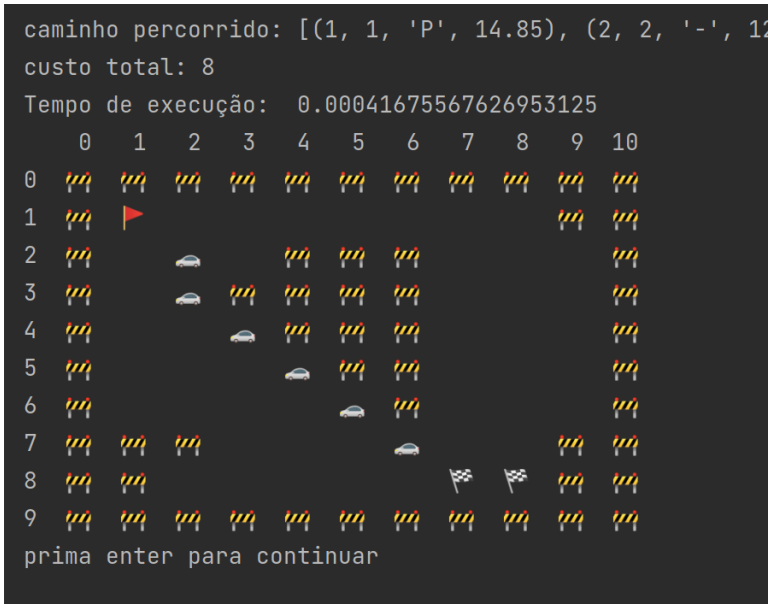

Figure 16: Solução do Algoritmo da Pesquisa Gulosa

Custo Total = 8, Tempo de Execução = 0.000417 segundos

O primeiro os algoritmos de pesquisa informada. Aqui foi usada apenas a distância euclidiana como heurística, e muito embora o custo seja o mesmo que na pesquisa em largura, verifica-se uma acentuada diminuição no tempo necessário para o caminho até ao nodo final ser encontrado.

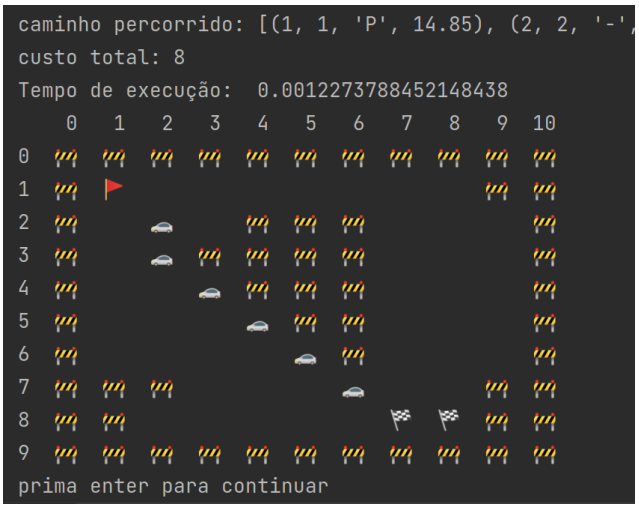

Figure 17: Solução do Algoritmo de Pesquisa A*

Custo Total = 8, Tempo de Execução = 0.00123 segundos

Comparativamente aos resultados obtidos a partir da pesquisa gulosa, seria de pensar que o algoritmo A* é imensamente pior. Na verdade é importante notar que a distância euclidiana não é a melhor das heurísticas e que neste caso em particular (o circuito ser pequeno e o facto de o A* fazer mais cálculos para saber o caminho), torna-se difícil comparar os resultados. No entanto, o A* tende a encontrar o melhor caminho possível, ao passo que o Greedy tem essa possibilidade, mas não a certeza.

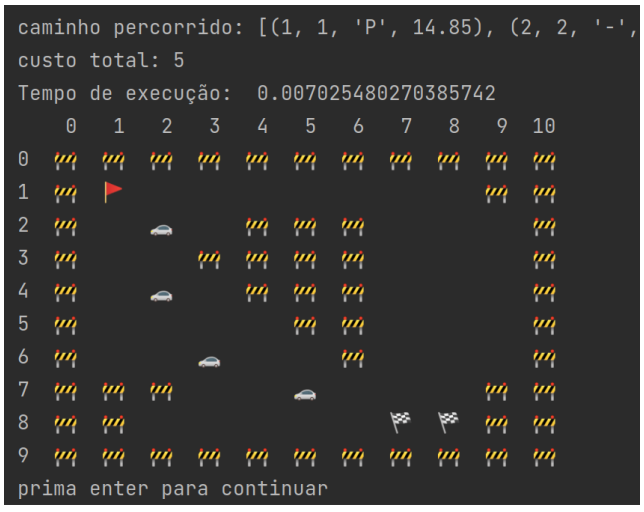

Figure 18: Solução do Algoritmo A* com Velocidade como Heurística

Custo Total = 5, Tempo de Execução = 0.00703

Aqui podemos constatar que o uso de uma melhor heurística traduz-se no cálculo de um caminho com menor custo entre os nodos inicial e final.

6 Conclusão

Durante as aulas da Unidade Curricular de Inteligência Artificial fomos providos das bases teóricas sobre os diferentes algoritmos que distinguem as Pesquisas Informada e Não-Informada, mas o desenvolvimento deste trabalho guarneceu-nos do inestimável valor da experiência que só a resolução prática de um problema pode dar. O facto de podermos efetivamente verificar cada um deles e comparar o resultado obtido de um com o de outro, foi uma mais valia para a nossa compreensão de como estas técnicas realmente funcionam. Conseguimos também constatar que, na resolução de um problema desta génese, a escolha da heurística é um elemento fulcral para obter resultados que mais se aproximam do valor ótimo.

A necessidade de termos que escrever o código fonte deste trabalho na linguagem *Python* foi um desafio acrescido pois nenhum de nós era proficiente neste paradigma de programação o que levou inevitavelmente a muitas horas dedicadas à aprendizagem da linguagem, assim como a algumas dificuldades encontradas na resolução dos problemas que nos tínhamos proposto fazer.

Ainda que o grupo tenha tentado o seu melhor para a solucionar, o cálculo do custo da jogada quando um carro bate na parede e consequente regresso ao ponto de partida ainda apresenta algumas falhas que não conseguiram ser colmatadas. Achamos que isto se deve à aplicação do algoritmo de *Bresenham* para a determinação das coordenadas em que se encontram paredes. A informação que recolhemos é que o algoritmo em si não é perfeito no seu cálculo e, por dependência imediata dele, o nosso cálculo de coordenadas também não o é. Existem portanto duas lacunas no nosso projeto: em situações raras o carro passar através de uma parede e; o retorno do carro à posição de onde partiu quando bate em parede foi nos difícil de implementar portanto arranjamos a alternativa que falamos na secção 5.

Em modo de finalização e tendo plena noção que nem todos os desafios conseguiram ser superados, estamos satisfeitos com aquilo que conseguimos aprender durante este trabalho pois sentimos que estamos na posse de muito mais informação agora do que a que tínhamos no começo deste projeto.

7 Bibliografia

Código base retirado de : **Licenciatura em Engenharia Informática - Mestrado Integrado em Engenharia Informática - Inteligência Artificial - Ficha Prática n.º 3 - Tema: Formulação de Problemas de Pesquisa e Resolução de Problemas utilizando Pesquisa Não Informada**

Norvig, P., Russell, S., Pearson; Artificial Intelligence: A Modern Approach, 1995, 3rd edition (December 1, 2009)

Boschloo, H. (no date) Vector racer, harmmade.com. Available at: https://harmmade.com/vectorracer/ (Accessed: November 30, 2022).

Adjacency matrix (no date) Programiz. Available at: https://www.programiz.com/dsa/graph-adjacency-matrix (Accessed: November 30, 2022).

Python - convert matrix to coordinate dictionary (2020) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/python-convert-matrix-to-coordinate-dictionary/ (Accessed: November 30, 2022).

Julie RaswickJulie Raswick 20.4k33 gold badges1515 silver badges33 bronze badges et al. (1957) How to read a file line-by-line into a list?, Stack Overflow. Available at: https://stackoverflow.com/questions/3277503/how-to-read-a-file-line-by-line-into-a-list (Accessed: November 30, 2022).

What is Artificial Intelligence (AI)? available at: https://www.ibm.com/cloud/learn/what-is-artificial-intelligence

Bresenham's Algorithm available at: https://en.wikipedia.org/wiki/Bresenham_line_algorithm