Inteligência Artificial

Vector Race

Licenciatura em Engenharia Informática

Relatório Final

2022/2023

GIT: https://github.com/MiguelJacinto99

Grupo 16:

- Miguel Jacinto Dias Carvalho (A84518)
- Adélio José Ferreira Fernandes (A78778)
- Gabriel Alexandre Monteiro da Silva (A97363)
- Gonçalo Peres Costa (A93309)

Índice

Descrição do Problema

Os problemas de procura do caminho mais curto através de ambientes definidos matematicamente têm ganho cada vez mais importância como objetos de estudo. Métodos como o algoritmo de Dijkstra apareceram para solucionar estes problemas.

A base da solução deste problema é a pesquisa, pesquisa essa que pode ser de dois tipos: informada e não informada. A maior diferença entre elas é a sua base de informação, onde a pesquisa informada fornece orientações/indicações sobre como chegar à solução. Por outro lado, a pesquisa não informada parte do absoluto zero, onde não tem indicações de como chegar à solução final.

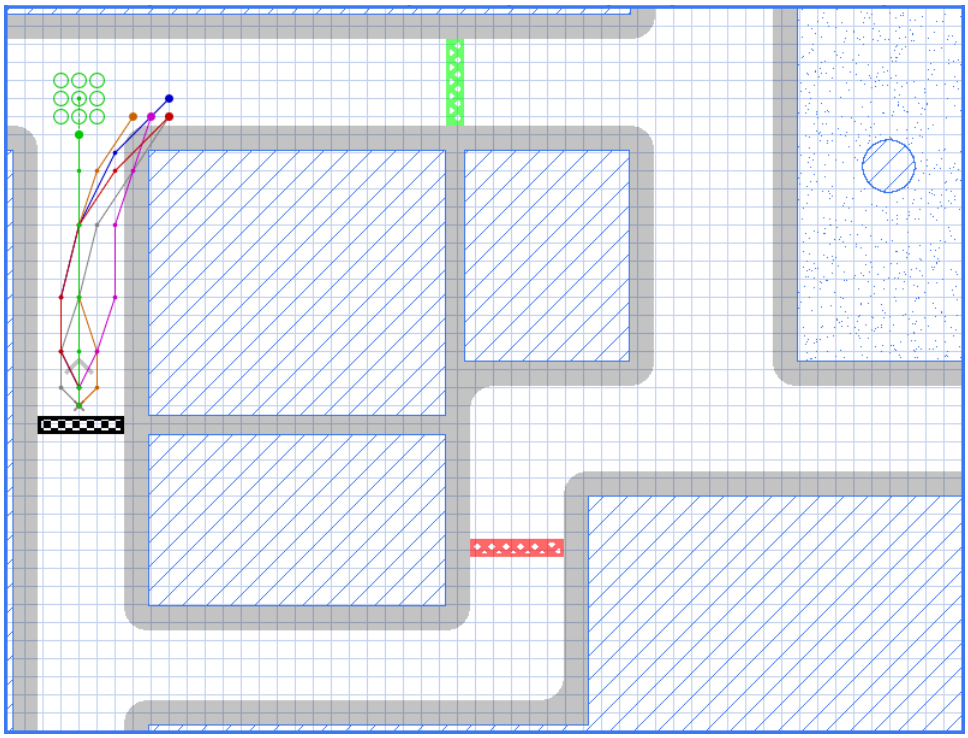


Figura 1 – Exemplo de um Estado VectorRace.

Pesquisa Informada

Um algoritmo de pesquisa informada utiliza informações disponíveis e especificadas em cada problema para fornecer pistas sobre a resolução do mesmo. Dito isto, podemos afirmar que a pesquisa informada pode ser mais vantajosa em termos de custo da solução, onde a sua otimização derivará destas pistas obtidas.

A pesquisa da solução ideal numa estratégia de pesquisa informada centra-se na inserção dos nodos mais promissores de um grafo numa função heurística. Esta irá retornar um número real não negativo, que representa um custo aproximado do caminho encontrado entre o ponto de partida e o nodo de destino.

A função heurística é a parte mais importante desta estratégia de pesquisa, visto que ajuda na transmissão de conhecimento adicional do problema ao algoritmo.

Pesquisa não Informada

A pesquisa não informada difere da pesquisa informada pelo simples facto de não existirem heurísticas na definição do problema. Deste modo, a estratégia de pesquisa não informada ficou conhecida como pesquisa cega. Dois algoritmos de pesquisa não informada são por exemplo o Depth-First Search (DFS) e o Breadth-First Search (BFS).

Formulação do Problema

Representação do Problema

Estado Inicial: Carater P, que representa o nodo de partida.

Estado Objetivo: Conjunto de carateres F, que representam a linha da meta.

Estados: Carater -, que representa o circuito navegável.

Operações: Fazer a travessia do circuito até ao estado objetivo.

Operadores: Algoritmo Depth-First Search (DFS), Algoritmo Breadth-First Search (BFS), Algoritmo Seguro e Algoritmo A*.

Estratégia Utilizada

Neste tópico iremos explicar detalhadamente a estratégia utilizada.

Construção de Circuitos

Criamos dez circuitos para testar os algoritmos de procura:

1: Circuito Aberto

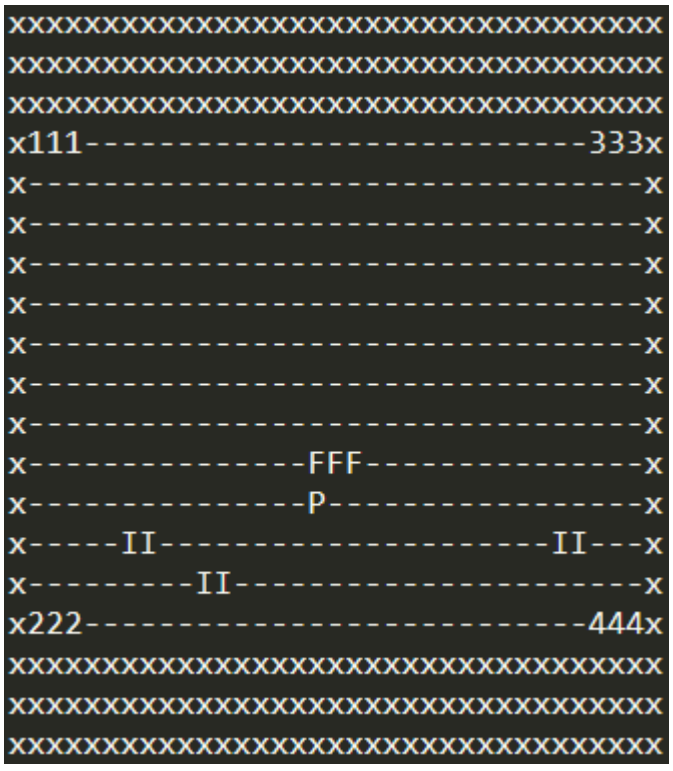


Figura 2 – Circuito Aberto

2: Circuito Circular

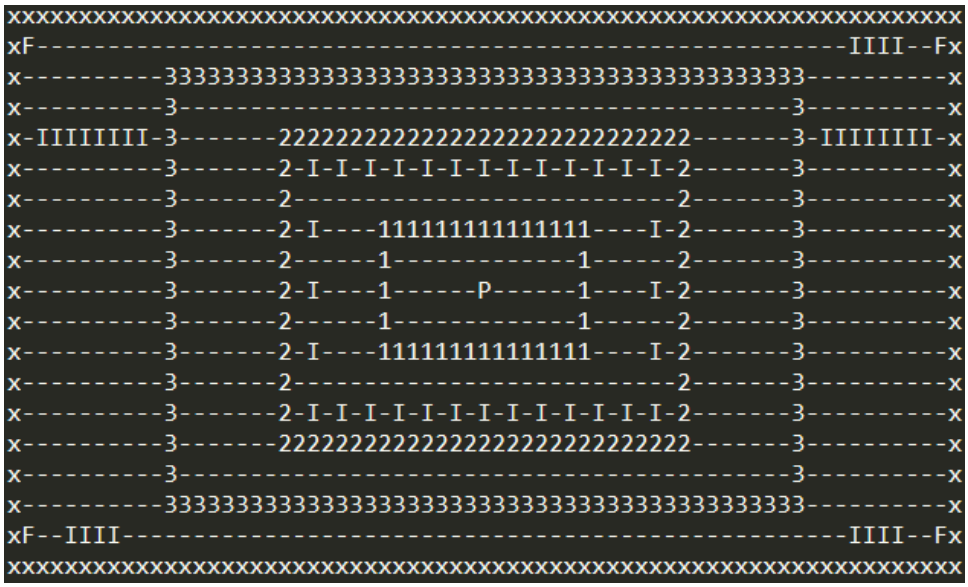


Figura 3 – Circuito Circular

3: Circuito Labirinto

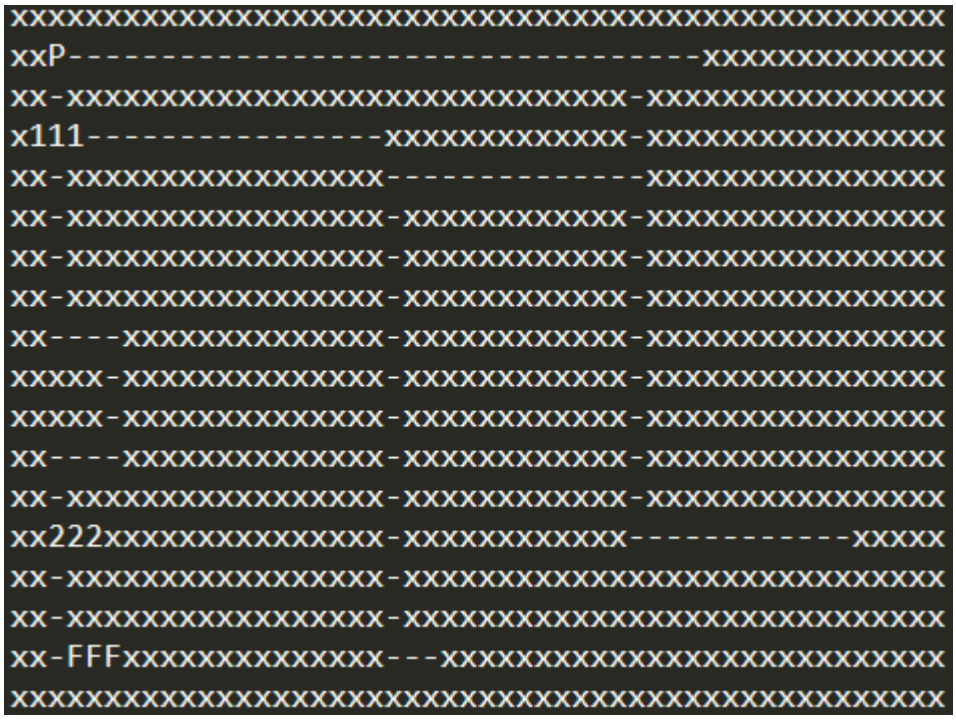


Figura 4 – Circuito Labirinto

4: Circuito Oval

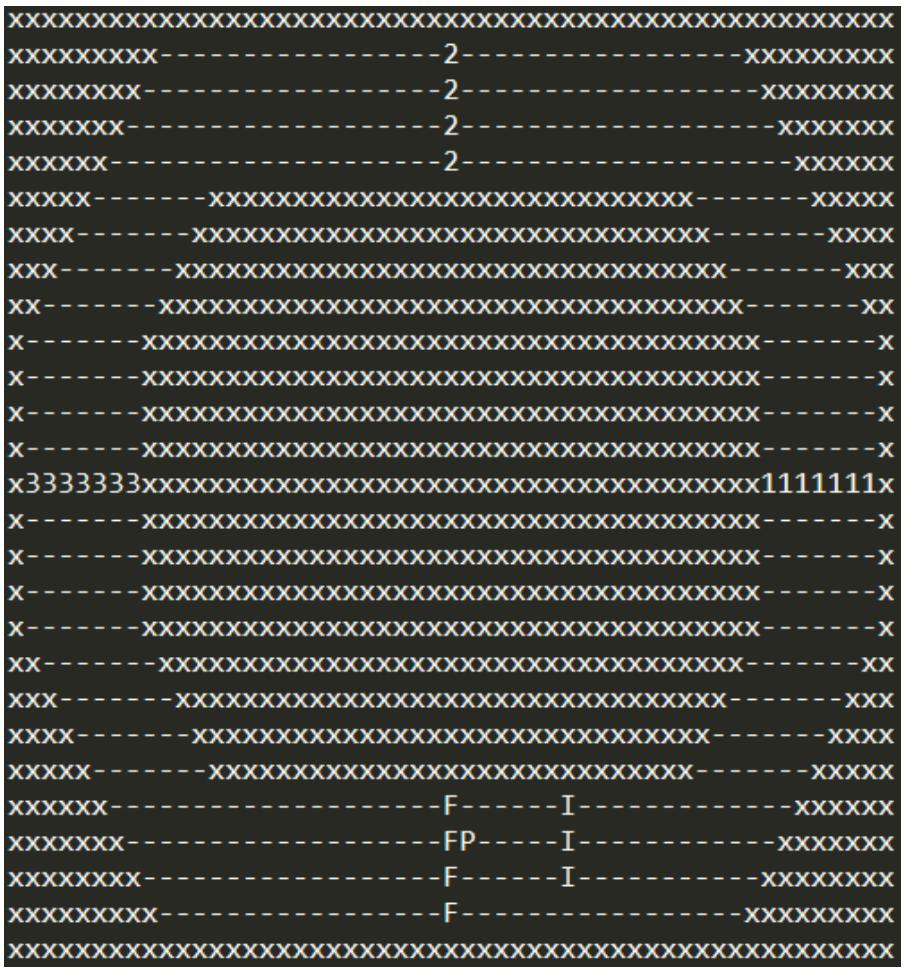


Figura 5 – Circuito Oval

5: Circuito Reta

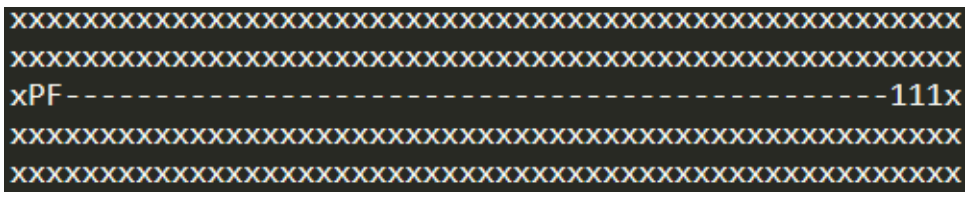


Figura 6 – Circuito Reta

6: Circuito Buraco

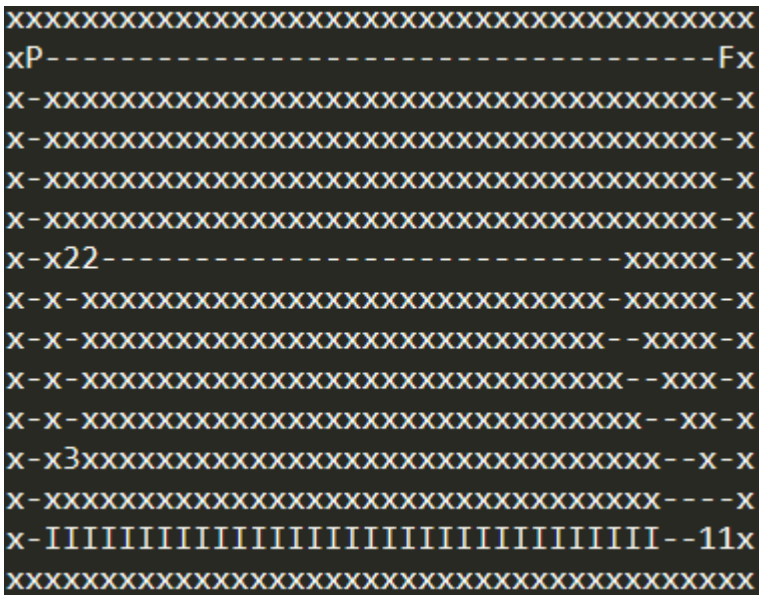


Figura 7 – Circuito Buraco

7: Circuito Estrada

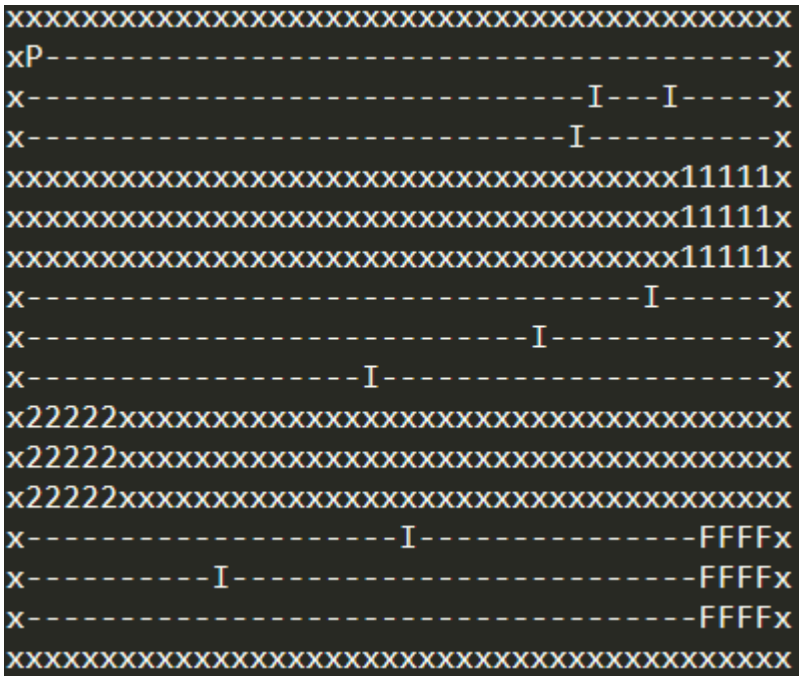


Figura 8 – Circuito Estrada

8: Circuito Caminho Apertado

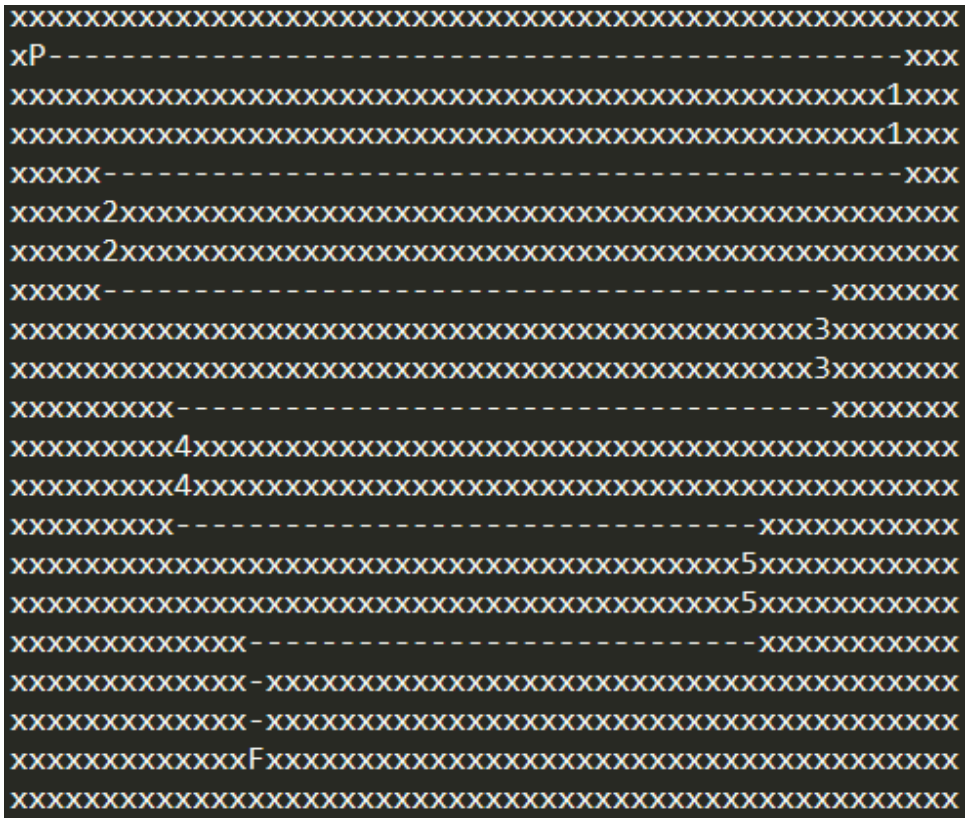


Figura 9 – Circuito Caminho Apertado

9: Circuito Espiral

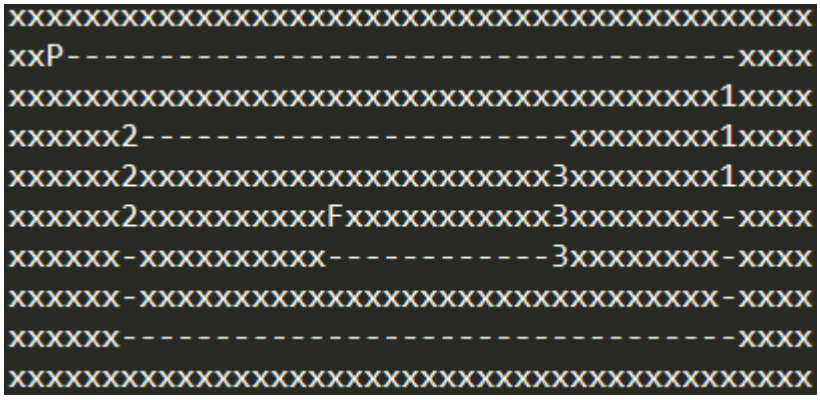


Figura 10 – Circuito Espiral

10: Circuito Hexa

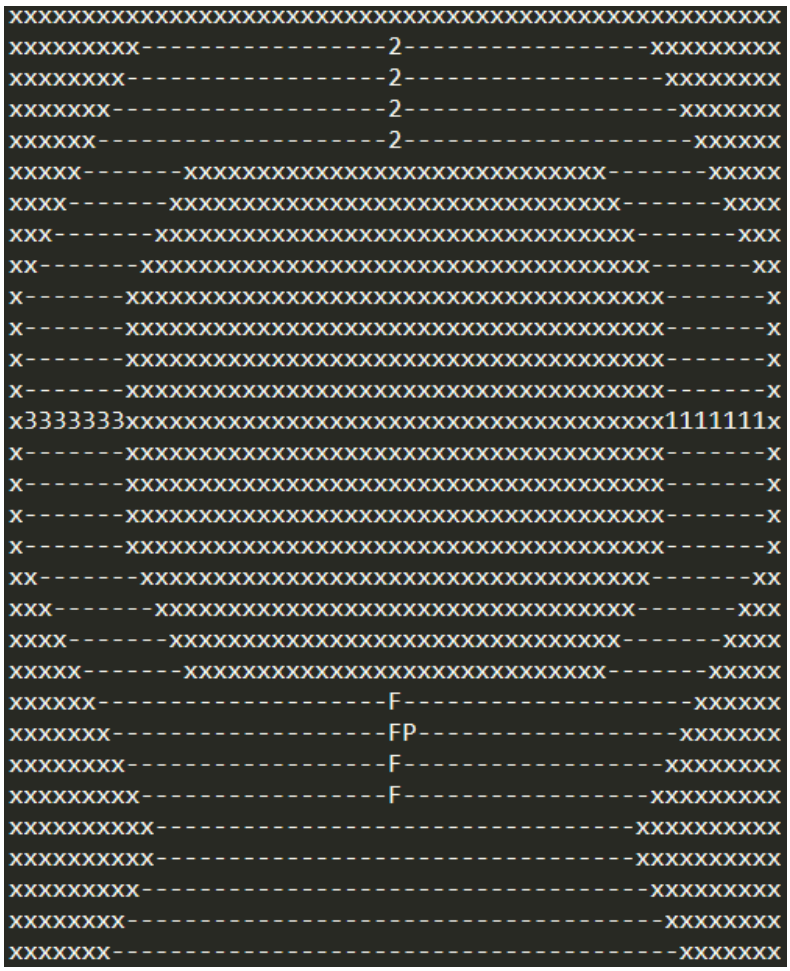


Figura 11 – Circuito Hexa Parte 1

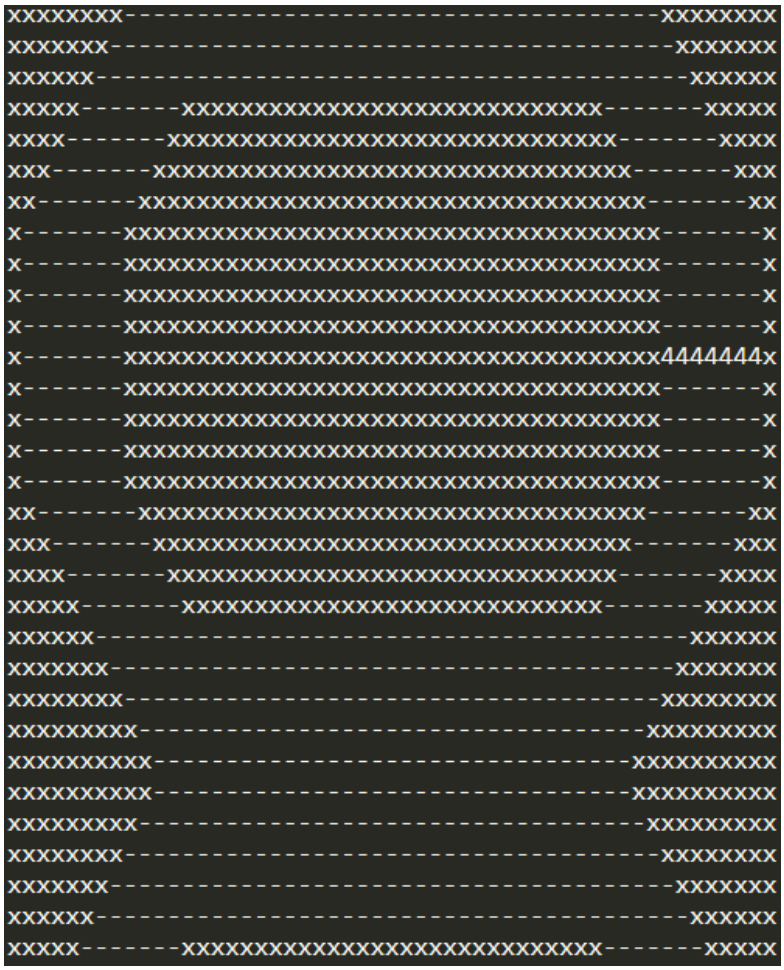


Figura 12 – Circuito Hexa Parte 2

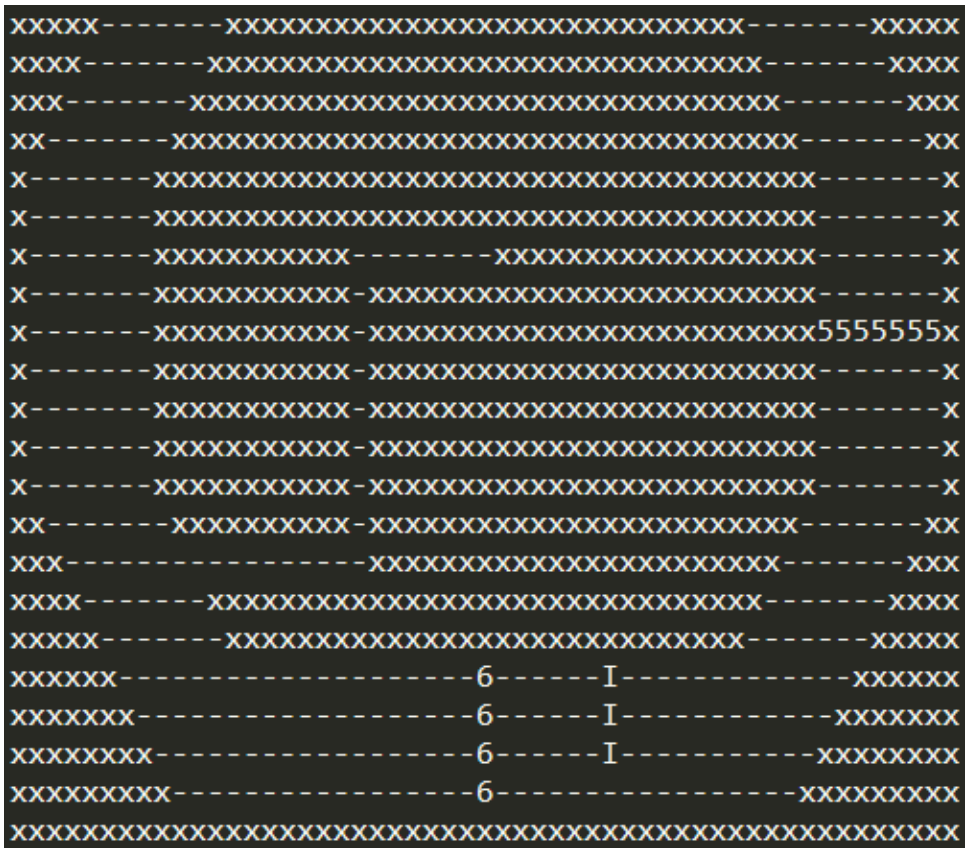


Figura 13 – Circuito Hexa Parte 3

Todos eles possuem estratégias diferentes. No último tópico serão apresentados os testes realizados com cada um dos algoritmos de pesquisa nos circuitos referidos.

Definimos três regras para construir circuitos:

- → Não existe um limite definido para o tamanho do circuito.
- → Não existe forma definida para o circuito. O número de linhas pode ser superior, igual ou inferior ao número de colunas e vice-versa.
- → Existem seis caracteres permitidos para a criação de um circuito VectorRace:
  - O caracter ‘X’ representa os limites do circuito.
  - → O caracter ‘-’ representa uma coordenada vazia que pode ser utilizada para a travessia.
  - → O caracter ‘P’ representa a única posição de partida, tal como no jogo original.
  - → O caracter ‘F’ representa as posições de chegada. Todos os blocos de posições de chegada devem estar na mesma linha ou na mesma coluna e sempre delimitados por um “x”.

→ Os caracteres ‘1...9’ representam os checkpoints do circuito. Tal como os blocos de posições de chegada, os checkpoints devem estar na mesma linha ou na mesma coluna e sempre delimitados por um x. Optamos por utilizar algarismos. Se o circuito tiver três checkpoints, então só é possível terminar o circuito depois de cruzar o checkpoint um, dois e três. Da mesma forma, se o circuito tiver três checkpoints então esses três checkpoints devem ser representados como 1, 2 e 3.
→ O caracter ‘I’ representa um obstáculo.

Classe Main

A classe main resume-se a um simples match que invoca os algoritmos de procura conforme o circuito escolhido pelo utilizador.

```
VECTORRACE!
ESCOLHER CIRCUITO:
PRIMA 1 PARA ESCOLHER O CIRCUITO ABERTO.
PRIMA 2 PARA ESCOLHER O CIRCUITO CIRCULAR.
PRIMA 3 PARA ESCOLHER O CIRCUITO LABIRINTO.
PRIMA 4 PARA ESCOLHER O CIRCUITO OVAL.
PRIMA 5 PARA ESCOLHER O CIRCUITO RETA.
PRIMA 6 PARA ESCOLHER O CIRCUITO BURACO.
PRIMA 7 PARA ESCOLHER O CIRCUITO ESTRADA.
PRIMA 8 PARA ESCOLHER O CIRCUITO CAMINHO APERTADO.
PRIMA 9 PARA ESCOLHER O CIRCUITO ESPIRAL.
PRIMA 10 PARA ESCOLHER O CIRCUITO HEXA.
PRIMA 11 PARA ESCOLHER O CIRCUITO RETA PERIGOSA.
PRIMA 0 PARA SAIR DO VECTORRACE.
```

Figura 14 – VectorRace!

Depois de escolhido o circuito, quatro jogadores executam a travessia com os quatro algoritmos de pesquisa disponíveis. É apresentado o caminho feito por cada um dos jogadores, o tempo total da travessia e o tempo de execução do algoritmo.

```
JOGADOR 3. CARRO SEGURO:

O CHECKPOINT NÚMERO 1 FOI CRUZADO NO NODO 107!
O CHECKPOINT NÚMERO 2 FOI CRUZADO NO NODO 526!
O CHECKPOINT NÚMERO 3 FOI CRUZADO NO NODO 137!
O CHECKPOINT NÚMERO 4 FOI CRUZADO NO NODO 556!
CIRCUITO TERMINADO NO NODO 401!
O JOGADOR 3 DEMOROU 37 SEGUNDOS PARA FINALIZAR O CIRCUITO!
O CAMINHO É CONSTITUÍDO POR: (LINHA, COLUNA, VELOCIDADELINHA, VELOCIDADECOLUNA)
CAMINHO: [(12, 16, 0, 0), (11, 15, -1, -1), (9, 13, -2, -2), (6, 10, -3, -3), (4, 7, -2, -3), (3, 4, -1, -3), (3, 2, 0, -2), (4, 1, 1, -1), (6, 1, 2, 0),
O ALGORITMO SEGURO DEMOROU CERCA DE 0.0200653076171875 SEGUNDOS!
```

Figura 15 - Execução do Algoritmo Seguro Para o Circuito Aberto

```
JOGADOR 4. CARRO AESTRELA:

O CHECKPOINT NÚMERO 1 FOI CRUZADO NO NODO 107!
O CHECKPOINT NÚMERO 2 FOI CRUZADO NO NODO 526!
O CHECKPOINT NÚMERO 3 FOI CRUZADO NO NODO 137!
O CHECKPOINT NÚMERO 4 FOI CRUZADO NO NODO 556!
CIRCUITO TERMINADO NO NODO 401!
O JOGADOR 4 DEMOROU 34 SEGUNDOS PARA FINALIZAR O CIRCUITO!
O CAMINHO É CONSTITUÍDO POR: (LINHA, COLUNA, VELOCIDADELINHA, VELOCIDADECOLUNA)
CAMINHO: [(12, 16, 0, 0), (11, 15, -1, -1), (9, 13, -2, -2), (6, 10, -3, -3), (4, 7, -2, -3), (3, 4, -1, -3), (3, 2, 0, -2), (4, 1, 1, -1), (6, 1, 2, 0),
O ALGORITMO AESTRELA DEMOROU CERCA DE 0.02221202850341797 SEGUNDOS!
```

Figura 16 - Execução do Algoritmo A* Para o Circuito Aberto

Classe Grid

```python
class Grid:
    WALL = 'x'
    OBSTACLE = 'I'
    EMPTY = '-'
    START = 'P'
    END = 'F'
```

Figura 17 – Classe Grid

A classe Grid resume o que foi dito anteriormente. Os checkpoints não podem ser incluídos nesta classe porque o número de checkpoints é sempre variável.

Classe Node

```python
class Node:
    id = int
    linha = int
    coluna = int
    type = Grid
    h1 = []

    def __init__(self):
        self.id = -1
        self.linha = 0
        self.coluna = 0
        self.type = None
        self.h1 = []

    def __str__(self):
        out = "Node: ID: " + str(self.id) + " Com Coordenadas: -> (" + str(self.linha) \
              + "," + str(self.coluna) + ') e Tipo: ' + str(self.type)
        return out
```

Figura 18 – Classe Node

A classe Node é composta por:

→ **id**: O identificador do Node no circuito.
→ **linha**: A abscissa da coordenada.
→ **coluna**: A ordenada da coordenada.
→ **type**: O tipo de caracter
→ **h1**: Lista das Heurísticas

Podíamos ter optado por uma solução mais compacta, como por exemplo, criar uma classe Coordenada que seria um tuplo constituído pela linha e pela coluna. No entanto, depois de avaliar as vantagens e desvantagens achamos que seria mais prático manter as variáveis separadas devido à facilidade de acesso.

A classe Node dispõe ainda do método init e str. É importante mencionar que o identificador do Node deve ser inicializado com o valor –1 porque o valor 0 é, na verdade, utilizado durante a criação do grafo.

Classe Graph

```python
class Graph:
    # Construtor da Classe
    def __init__(self, directed=True):
        self.nodes = []  # Lista de Nodos do Graph.
        self.partida = -1
        self.chegadas = []
        self.checkPoints = {}
        self.directed = directed  # Se o Graph é ou não Direcionado.
        self.graph = {}  # Dicionário Para Armazenar as Arestas (Não Pesadas)
        self.tamanhoCircuitoAtual = tuple()
```

Figura 19 – Classe Graph

A classe Graph é a classe principal do projeto. É composta por:

- → **nodes**: A lista de Nodes do grafo. Note-se que é uma lista de dados do tipo Node. É nesta lista que ficarão guardados todos os Nodes do circuito.
- → **partida**: Um inteiro que corresponde ao identificador do Node de partida. Como definimos que só existe um ponto de partida então faz sentido destacar esse identificador para os algoritmos de pesquisa saberem onde começar a travessia.
- → **chegadas**: A lista de Nodes de chegada. É importante mencionar que esta lista contém apenas os identificadores dos Nodes. Não há necessidade de guardar o Node visto que este já se encontra armazenado na lista nodes. No entanto é útil destacar estes identificadores para os algoritmos de pesquisa saberem quando terminar a travessia.
- → **checkPoints**: Um dicionário de checkPoints. A key é o número do checkpoint e a value é uma lista com os identificadores dos Nodes checkpoint. Exemplificando: Um circuito com três checkpoints, 1 2 e 3. Os checkpoint do tipo 1 estão nos Node cujo identificador é o 1030, 1031, 1032 e 1033. Os checkpoints do tipo 2 estão nos Node cujo identificador é o 2045, 2046, 2047 e 2048. Os checkpoint do tipo 3 estão nos Node cujo identificador é o 3045, 3046, 3047 e 3048. O dicionário seria populado da seguinte forma: {1: [1030, 1031, 1032, 1033], 2: [2045, 2046, 2047, 2048], 3: [3045, 3046, 3047, 3048]} Note-se que são apenas valores ilustrativos.
- → **directed**: Um boolean que indica se o grafo é ou não direcionado. Faz sentido pensar num grafo direcionado no contexto deste

problema visto que o circuito possui uma direção associada. No entanto, existem circuitos em a solução ótima implica voltar atrás e por esta razão optamos por um grafo não direcionado.

→ **graph**: O dicionário das arestas do grafo. É nesta estrutura que será armazenado o circuito. A key é um inteiro que corresponde ao identificador do Node e a value é um set() de identificadores de Node. Mais uma vez não há necessidade de armazenar o Node visto que através do identificador conseguimos aceder a todas as informações do Node guardadas nodes. As arestas não são pesadas porque o peso de uma travessia entre dois Nodes é sempre 1 segundo.

→ **tamanhoCircuitoAtual**: Um tuple que contém o número de linhas e o número de colunas do circuito.

Classe Carro

```
class Carro:
    linha = int
    coluna = int
    velocidadeLinha = int
    velocidadeColuna = int
    # Lista de ID NODES Disponiveis Tendo em Conta os Atributos Anteriores e o Circuito.
    listaMovimentos = []

    def __init__(self):
        self.linha = 0
        self.coluna = 0
        self.velocidadeLinha = 0
        self.velocidadeColuna = 0
        self.listaMovimentos = []
```

Figura 20 – Classe Carro

A classe carro é composta por:

→ **linha**: A linha onde o carro se encontra.

→ **coluna**: A coluna onde o carro se encontra.

→ **velocidadeLinha**: A velocidade na linha num determinado momento da travessia.

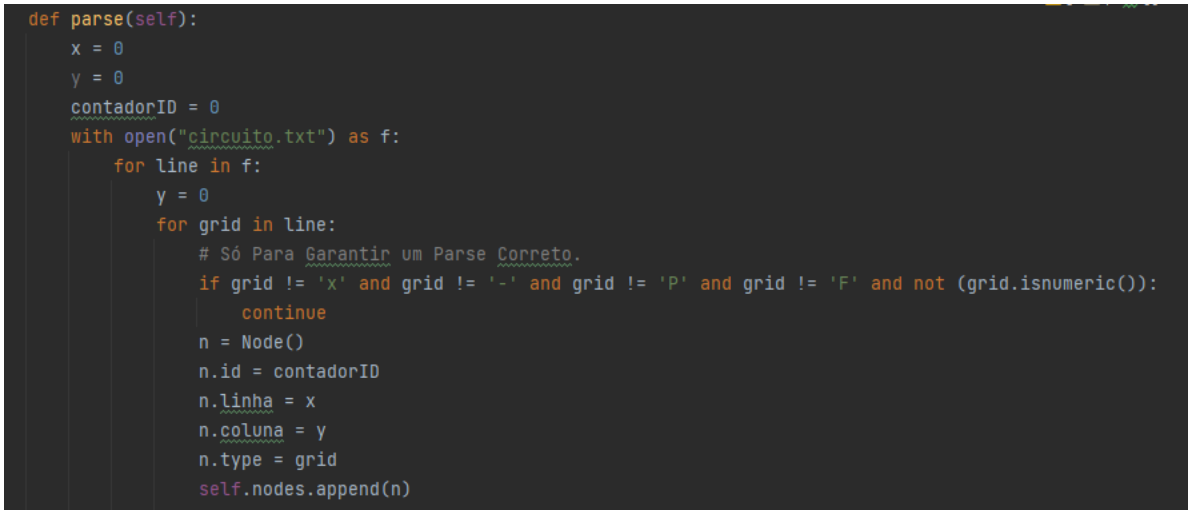
→ **velocidadeColuna**: A velocidade na coluna num determinado momento da travessia.

→ **listaMovimentos**: A lista de movimentos disponíveis do carro num determinado momento da travessia.

É essencial existir a classe carro para a travessia devido à necessidade de transporte da informação existente até ao momento.

Método Parse

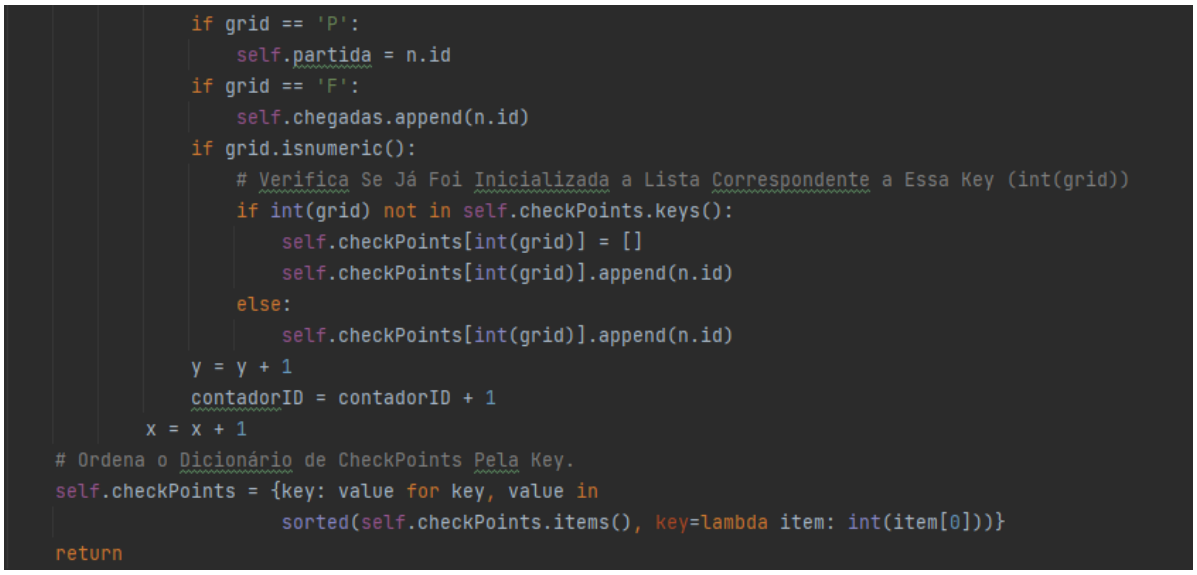
É neste método da classe Graph que é feito o parse do circuito através de um ficheiro de texto. Para uma explicação mais detalhada iremos dividir o método parse em dois:



```
def parse(self):
    x = 0
    y = 0
    contadorID = 0
    with open("circuito.txt") as f:
        for line in f:
            y = 0
            for grid in line:
                # Só Para Garantir um Parse Correto.
                if grid != 'x' and grid != '-' and grid != 'P' and grid != 'F' and not (grid.isnumeric()):
                    continue
                n = Node()
                n.id = contadorID
                n.linha = x
                n.coluna = y
                n.type = grid
                self.nodes.append(n)
```

Figura 21 – Método Parse Parte 1

Para o parse utilizamos a função pré-definida open. A estratégia utilizada é muito simples. Através de dois ciclos aninhados, ler cada caracter e criar um objeto Node para esse caracter. As variáveis x e y serão utilizadas para armazenar temporariamente a linha e coluna do caracter lido. O contadorID será incrementado em cada iteração permitindo que no final da execução do parse o último identificador corresponda ao número total de Nodes criados. Desta forma, os Nodes criados serão adicionados ordenadamente à lista de nodes. É também importante reduzir a probabilidade de falha no parse através de estruturas condicionais para garantir que não é guardado nenhum caracter errado.



```
                if grid == 'P':
                    self.partida = n.id
                if grid == 'F':
                    self.chegadas.append(n.id)
                if grid.isnumeric():
                    # Verifica Se Já Foi Inicializada a Lista Correspondente a Essa Key (int(grid))
                    if int(grid) not in self.checkPoints.keys():
                        self.checkPoints[int(grid)] = []
                        self.checkPoints[int(grid)].append(n.id)
                    else:
                        self.checkPoints[int(grid)].append(n.id)
                y = y + 1
                contadorID = contadorID + 1
            x = x + 1
    # Ordena o Dicionário de CheckPoints Pela Key.
    self.checkPoints = {key: value for key, value in
                        sorted(self.checkPoints.items(), key=lambda item: int(item[0]))}
    return
```

Figura 22 – Método Parse Parte 2

Depois de criado o Node é necessário verificar se é algum Node de partida, chegada ou checkpoint. Caso seja o Node de partida, colocamos o identificador na variável partida. Se for um Node de chegada, adicionamos o identificador à lista de identificadores de chegada. No caso de ser um checkpoint (algarismo) é só uma questão de verificar qual é o número do checkpoint e adicionar esse identificador à lista correspondente. Por fim, optamos por ordenar o dicionário de checkpoints pela key. Visto que a procura num dicionário é constante não seria necessário este último passo. No entanto, achamos que melhora a compreensão do código.

Método criaGrafo

Após o parse, falta apenas criar as arestas entre os Nodes processados. É desde já importante mencionar que apesar de existir o fator aceleração e velocidade, o grafo será sempre ligado por uma unidade.

```
def criaGrafo(self):
    linha = 0
    coluna = 0
    linhaBase = 0
    colunaBase = 0
    linhaCandidata = 0
    colunaCandidata = 0
    directions = [(-1, -1), (-1, 0), (-1, 1), (0, 0), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
    for node in self.nodes:
        self.graph[node.id] = set()
        linha = node.linha
        coluna = node.coluna
        for x, y in directions:
            linhaBase = linha + x
            colunaBase = coluna + y
            for nodeCandidato in self.nodes:
                if nodeCandidato.type != 'x':
                    linhaCandidata = nodeCandidato.linha
                    colunaCandidata = nodeCandidato.coluna
                    if linhaBase == linhaCandidata and colunaBase == colunaCandidata:
                        self.graph[node.id].add(nodeCandidato.id)
    return
```

Figura 23 - Método criaGrafo

A lista directions contém tuplos com todas as direções possíveis. Optamos por incluir a direção (0,0) pois é essencial a opção de manter a velocidade atual. A estratégia utilizada é extremamente simples e intuitiva. Fazer a travessia da lista de Nodes do grafo e verificar quais os seus adjacentes. Sendo assim, todos os Nodes à distância de uma unidade serão adjacentes, exceto aqueles que possuam um tipo x. Faz sentido descartar desde já essas arestas, visto que a travessia deve ser realizada dentro do circuito, evitando colisões. Tal como referimos no tópico anterior, todas as arestas serão criadas entre identificadores. No final da execução do método criaGrafo todos os Nodes do grafo estão ligados. Note-se também que haverá vários Nodes sem adjacentes. Na prática isso significa um caminho sem saída cabendo aos algoritmos de pesquisa descartar esses Nodes.

Algoritmos de Procura não Informada

Depth-First Search (DFS)

Optamos por implementar uma versão recursiva do algoritmo de pesquisa não informada Depth-First Search. Adicionamos algumas otimizações que irão permitir melhorar a eficiência da travessia.

```
def procura_DFS(self, start=int, checkPointsCruzados=int, end=[], path=[], visited=set()):
    totalCheckPoints = len(self.checkPoints)
    path.append(start)
    visited.add(start)
    if checkPointsCruzados > totalCheckPoints:
        for final in end:
            if start == final:
                print("Circuito Terminado No Nodo " + str(start) + "!")
                return path
    if checkPointsCruzados <= totalCheckPoints:
        for checkPoint in self.checkPoints[checkPointsCruzados]:
            if start == checkPoint:
                print("O CheckPoint Número " + str(checkPointsCruzados) + " Foi Cruzado no Nodo "
                      + str(start))
                checkPointsCruzados = checkPointsCruzados + 1
                # Essencial!
                visited.clear()
    for adjacent in self.graph[start]:
        if adjacent not in visited:
            resultado = self.procura_DFS(adjacent, checkPointsCruzados, end, path, visited)
            if resultado is not None:
                return resultado
    path.pop()
    return None
```

Figura 24 – Algoritmo Depth-First Search

Parâmetros:

- → **start**: Identificador do Node de partida.
- → **checkPointsCruzados**: Número de checkPoints cruzados no ínicio de cada invocação da função.
- → **end**: Lista de identificadores de Nodes de chegada.
- → **path**: Lista de identificadores dos Nodes que fazem parte da travessia.
- → **visited**: Set de identificadores dos Nodes visitados até ao momento.

O totalCheckPoints é essencial na travessia para o algoritmo saber quando é que pode cruzar os Nodes de chegada. Note-se que só pode terminar a travessia quando cruzar todos os checkPoints do circuito.

O algoritmo divide-se em dois pontos fundamentais:

- → O número de checkPoints cruzados é inferior ou igual ao número de checkPoints do circuito o que significa que ainda não é possível cruzar a meta.
- → O número de checkPoints cruzados é maior do que o número de checkPoints do circuito o que significa que já é possível cruzar a meta.

Quando um checkpoint é cruzado é necessário executar visited.clear() porque há uma grande probabilidade de a travessia já ter percorrido grande parte dos nodes.

Breadth-First Search (BFS)

Optamos por implementar uma versão iterativa do algoritmo de pesquisa não informada Breadth-First Search. Adicionamos algumas otimizações que irão permitir melhorar a eficiência da travessia.

```
def procura_BFS(self, start=int, end=[], queue=[], visited=set()):
    checkPointsCruzados = 1
    totalCheckPoints = len(self.checkPoints)
    path = []
    parents = {start: start}
    queue.append(start)
    visited.add(start)
    while queue:
        primeiroDaFila = queue.pop(0)
        path.append(primeiroDaFila)
        if checkPointsCruzados > totalCheckPoints:
            for final in end:
                if primeiroDaFila == final:
                    print("CIRCUITO TERMINADO NO NODO " + str(primeiroDaFila) + "!")
                    return path
        if checkPointsCruzados <= totalCheckPoints:
            for checkPoint in self.checkPoints[checkPointsCruzados]:
                if primeiroDaFila == checkPoint:
                    print("O CHECKPOINT NÚMERO " + str(checkPointsCruzados) + " FOI CRUZADO NO NODO "
                          + str(primeiroDaFila) + "!")
                    checkPointsCruzados = checkPointsCruzados + 1
```

Figura 25 – Algoritmo Breadth-First Search Parte 1

```
                    # Essencial!
                    visited.clear()
        for adjacent in self.graph[primeiroDaFila]:
            if adjacent not in visited:
                parents[adjacent] = primeiroDaFila
                queue.append(adjacent)
                visited.add(adjacent)
    return None
```

Figura 26 – Algoritmo Breadth-First Search Parte 2

Parâmetros:

- → **start**: Identificador do Node de partida.
- → **end**: Lista de identificadores de Nodes de chegada.
- → **queue**: Lista de espera com os identificadores a serem avaliados. É esta queue que controla a execução do algoritmo.
- → **visited**: Set de identificadores dos Nodes visitados até ao momento.

O BFS funciona da mesma forma que o DFS: Percorrer o grafo enquanto houver checkpoints para cruzar. Quando esta condição for satisfeita, o algoritmo procura um dos caracteres finais.

Note-se que tanto o BFS como o DFS são algoritmos de procura não informada, portanto é normal que a travessia seja extensa, mas completa.

Quando um checkpoint é cruzado é necessário executar visited.clear() porque há uma grande probabilidade de a travessia já ter percorrido grande parte dos nodes, assim como acontece no DFS.

Ao contrário do DFS, o BFS faz uma travessia em largura, ou seja, avalia todos os Nodes adjacentes e adiciona-os à queue. Como veremos nos testes realizados, esta estratégia é mais demorada e menos eficiente.

Algoritmos de Procura Informada

Optamos por utilizar dois algoritmos de procura informada: O Algoritmo Seguro e o Algoritmo A*. São os dois muito semelhantes em termos estratégicos. Ambos utilizam a mesma heurística que será explicada já no próximo tópico.

Heurística da Distância

Como o próprio nome diz, a heurística da distância associa a cada um dos nodos a distância real a cada um dos checkpoints/chegadas.

A distância real entre dois pontos é dada pela seguinte fórmula:

$$\rightarrow \quad dAB^2 = (xB - xA)^2 + (yB - yA)^2$$

Sendo (xA, xB) as coordenadas do ponto de origem e (yA, yB) as coordenadas do ponto de destino.

Cada Node tem associado uma lista h1 na qual são inseridas as distâncias a cada um dos checkpoints.

```python
def heuristicaCheckPoint(self):
    contadorCheckPoint = 1
    numeroCheckPoint = len(self.checkPoints)
    listaFinais = self.chegadas
    for idNode in self.nodes:
        linhaOrigem = idNode.linha
        colunaOrigem = idNode.coluna
        while contadorCheckPoint <= numeroCheckPoint:
            # Utilizamos a Distância ao Primeiro CheckPoint Dessa Lista de CheckPoints
            listaCheckPoint = self.checkPoints[contadorCheckPoint].copy()
            linhaDestino = self.nodes[listaCheckPoint[0]].linha
            colunaDestino = self.nodes[listaCheckPoint[0]].coluna
            distReal = math.sqrt((math.pow(linhaDestino - linhaOrigem, 2)) +
                                 (math.pow(colunaDestino - colunaOrigem, 2)))
            self.nodes[idNode.id].h1.append(distReal)
            contadorCheckPoint = contadorCheckPoint + 1
        contadorCheckPoint = 1
        # Da Mesma Forma Utilizamos a Distância à Primeira Chegada Dessa Lista de Chegadas
        linhaDestino = self.nodes[listaFinais[0]].linha
        colunaDestino = self.nodes[listaFinais[0]].coluna
        distReal = math.sqrt((math.pow(linhaDestino - linhaOrigem, 2)) +
                             (math.pow(colunaDestino - colunaOrigem, 2)))
        self.nodes[idNode.id].h1.append(distReal)
    return
```

Figura 27 - Função heuristicaCheckPoint

Optamos por esta heurística por ser a mais simples e lógica para a resolução deste problema. Através da distância aos Node destino, o algoritmo de pesquisa pode sempre tomar uma boa decisão em termos de custo do caminho.

Limite de Velocidade

É necessário evitar colisões devido aos vinte e cinco segundos de penalização. Iria compensar colidir apenas em circuitos grandes (100x100). Tirando esses casos, concluímos que seria mais vantajoso evitar ao máximo as colisões com a parede.

A heurística da distância aliada à imposição de um limite de velocidade resulta na descoberta de um caminho quase ótimo.

Como tal, implementamos duas funções que a partir das nove direções possíveis do carro, calculam a racionalidade de cada uma dessas nove jogadas possíveis.

Função jogadaValida

```python
def jogadaValida(self, linha, coluna, velocidadeLinha, velocidadeColuna, directionLinhas, directionColunas):
    iterator = 0
    conditionLinha = False
    # Lista Dos Caracteres Seguinte Numa Determinada Direção.
    typeSeguinte = []
    t = tuple()
    if directionLinhas > 0 and directionColunas > 0:
        for idNode in self.nodes:
            if idNode.linha > linha and idNode.coluna == coluna:
                # Precisamos de Guardar a Linha Para Ordenar.
                t = (idNode.linha, idNode.type)
                typeSeguinte.append(t)
        # Ordenar a Lista de Tuplos Pelo Primeiro Item. Neste Caso, Pelas Linhas.
        typeSeguinte.sort()
        velocidadeLimite = self.quadradosParaTravar(abs(velocidadeLinha))
        for linhaCandidata, typeCandidato in typeSeguinte:
            if typeCandidato == 'x':
                if iterator >= velocidadeLimite:
                    conditionLinha = True
                else:
                    return False
            else:
                iterator = iterator + 1
```

Figura 28 - Função jogadaValida

A função jogadaValida calcula se uma determinada jogada é válida. Para testar a validade de uma jogada utilizamos a sugestão dada pelo VectorRace:

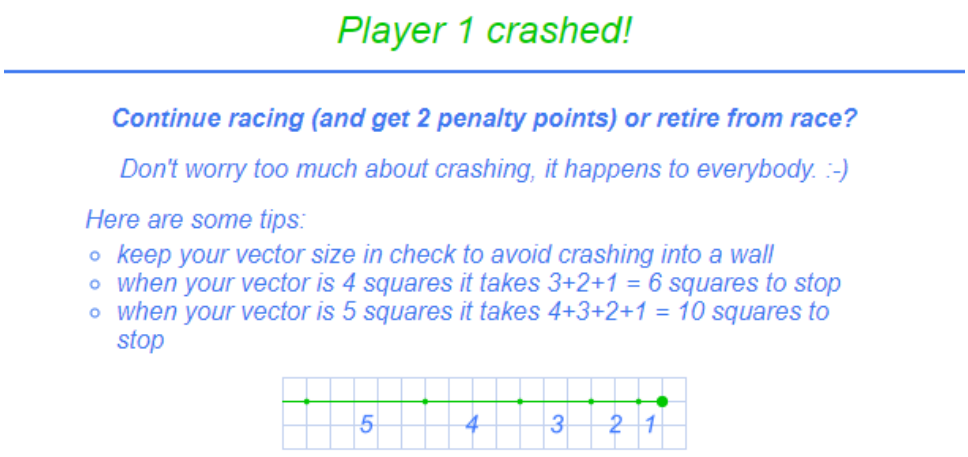


Figura 29 - Sugestão VectorRace

Exemplicando: Numa determinada fase da travessia o carro encontra-se na linha quatro e na coluna sete com velocidade seis na linha e velocidade três na coluna. Então a velocidade máxima nessa linha é o

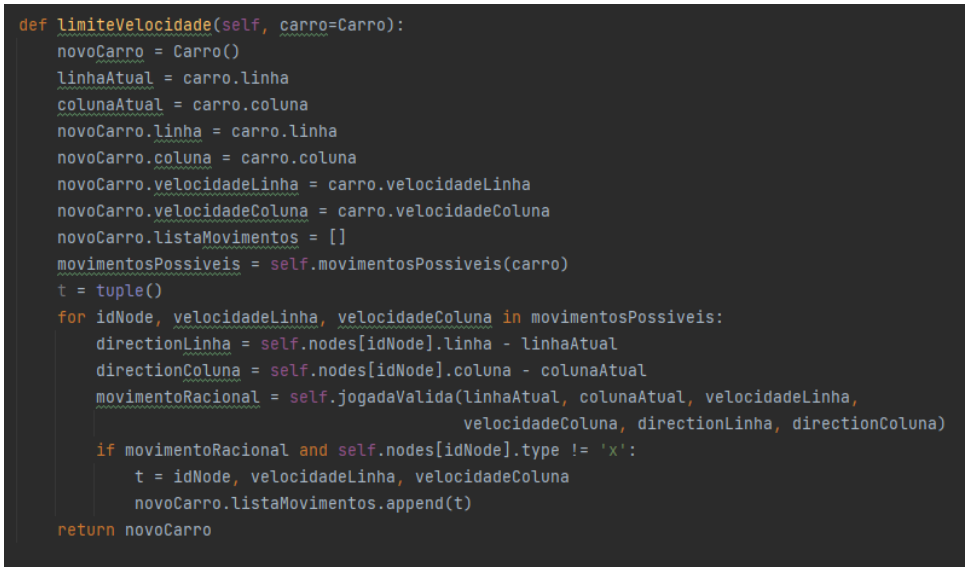
somatório de seis e a velocidade máxima nessa coluna é o somatório de três.

Sabendo o número de quadrados necessários para travar podemos evitar sempre colisões.

A função jogadaValida calcula se uma jogada é válida tendo em conta a linha de Origem, a coluna de Origem, a velocidade na linha e a velocidade na coluna. Se a distância à proxima parede nessa direção for menor do que a distância mínima para travar então a jogada não é válida.

Função limiteVelocidade

A função limiteVelocidade filtra a lista de movimentos disponíveis de um carro através da aplicação da jogadaValida a cada um dos movimentos. Após a aplicação da função limiteVelocidade o carro terá a lista de movimentos disponíveis avaliada e atualizada. Sabendo que os movimentos são racionais resta apenas escolher o melhor Node conforme o valor da heurística.



```
def limiteVelocidade(self, carro=Carro):
    novoCarro = Carro()
    linhaAtual = carro.linha
    colunaAtual = carro.coluna
    novoCarro.linha = carro.linha
    novoCarro.coluna = carro.coluna
    novoCarro.velocidadeLinha = carro.velocidadeLinha
    novoCarro.velocidadeColuna = carro.velocidadeColuna
    novoCarro.listaMovimentos = []
    movimentosPossiveis = self.movimentosPossiveis(carro)
    t = tuple()
    for idNode, velocidadeLinha, velocidadeColuna in movimentosPossiveis:
        directionLinha = self.nodes[idNode].linha - linhaAtual
        directionColuna = self.nodes[idNode].coluna - colunaAtual
        movimentoRacional = self.jogadaValida(linhaAtual, colunaAtual, velocidadeLinha,
                                              velocidadeColuna, directionLinha, directionColuna)
        if movimentoRacional and self.nodes[idNode].type != 'x':
            t = idNode, velocidadeLinha, velocidadeColuna
            novoCarro.listaMovimentos.append(t)
    return novoCarro
```

Figura 30 - Função limiteVelocidade

Algoritmo A*

```
def aestrela(self):
    checkPointsCruzados = 1
    totalCheckPoints = len(self.checkPoints)
    listaDistCheckPoint = []
    carro = Carro()
    carro.linha = self.nodes[self.partida].linha
    carro.coluna = self.nodes[self.partida].coluna
    carro.velocidadeLinha = 0
    carro.velocidadeColuna = 0
    t = tuple()
    t = self.nodes[self.partida].linha, self.nodes[self.partida].coluna, 0, 0
    outroT = tuple()
    parents = {t: t}
    path = [t]
    visited = set()
    visited.add(self.partida)
    existeCaminho = True
```

Figura 31 – Algoritmo A*, Inicializações.

Não existem parâmetros porque toda a informação necessária está guardada no circuito. É necessário criar o carro para a travessia e o dicionário parents para armazenar o Node pai correspondente. Além disso é também preciso criar um set() visited que será extremamente importante para controlar caminhos sem saída.

```
while checkPointsCruzados <= totalCheckPoints:
    linhaAtual = carro.linha
    colunaAtual = carro.coluna
    velocidadeLinhaAtual = carro.velocidadeLinha
    velocidadeColunaAtual = carro.velocidadeColuna
    carro = self.limiteVelocidade(carro)
    listaDistCheckPoint = []
    t = ()
    for idNode, velocidadeLinha, velocidadeColuna in carro.listaMovimentos:
        t = self.nodes[idNode].h1[checkPointsCruzados - 1], idNode, velocidadeLinha, velocidadeColuna
        if idNode not in visited:
            listaDistCheckPoint.append(t)
    if len(listaDistCheckPoint) == 0:
        for idNode, velocidadeLinha, velocidadeColuna in carro.listaMovimentos:
            t = self.nodes[idNode].h1[checkPointsCruzados - 1], idNode, velocidadeLinha, velocidadeColuna
            if idNode in visited:
                visited.remove(idNode)
                listaDistCheckPoint.append(t)
            else:
                listaDistCheckPoint.append(t)
    listaDistCheckPoint.sort()
    distH, idNode, velocidadeLinhaTemp, velocidadeColunaTemp = listaDistCheckPoint[0]
    visited.add(idNode)
    carro.linha = self.nodes[idNode].linha
    carro.coluna = self.nodes[idNode].coluna
```

Figura 32 – Algoritmo A* Parte 1

```
    carro.coluna = self.nodes[idNode].coluna
    carro.velocidadeLinha = velocidadeLinhaTemp
    carro.velocidadeColuna = velocidadeColunaTemp
    carro.listaMovimentos = []
    t = ()
    t = linhaAtual, colunaAtual, velocidadeLinhaAtual, velocidadeColunaAtual
    outroT = carro.linha, carro.coluna, carro.velocidadeLinha, carro.velocidadeColuna
    parents[outroT] = t
    path.append(outroT)
    if self.cruzouNodesImportantes(linhaAtual, colunaAtual, carro.linha, carro.coluna, checkPointsCruzados):
        checkPointsCruzados = checkPointsCruzados + 1
        visited.clear()
        print("O CHECKPOINT NÚMERO " + str((checkPointsCruzados) - 1) + " FOI CRUZADO NO NODO "
                                    + str(idNode) + "!")
# Os CheckPoints Foram Todos Cruzados. Pode Finalizar.
while existeCaminho:
    linhaAtual = carro.linha
    colunaAtual = carro.coluna
    velocidadeLinhaAtual = carro.velocidadeLinha
    velocidadeColunaAtual = carro.velocidadeColuna
    carro = self.limiteVelocidade(carro)
    listaDistCheckPoint = []
    t = ()
    for idNode, velocidadeLinha, velocidadeColuna in carro.listaMovimentos:
        t = self.nodes[idNode].h1[checkPointsCruzados - 1], idNode, velocidadeLinha, velocidadeColuna
```

Figura 33 – Algoritmo A* Parte 2

Figura 34 – Algoritmo A* Parte 3

O algoritmo A* divide-se em dois pontos fundamentais:

→ O número de checkPoints cruzados é inferior ou igual ao número de checkPoints do circuito o que significa que ainda não é possível cruzar a meta.
→ O número de checkPoints cruzados é maior do que o número de checkPoints do circuito o que significa que já é possível cruzar a meta.

A cada iteração a lista de movimentos disponíveis do carro é atualizada e filtrada pela função limiteVelocidade. Sabendo que todas as jogadas resultantes são racionais, basta escolher a melhor em termos de distância ao checkpoint.

Note-se que o carro pode ter cruzado um checkpoint durante a travessia de um Node para o Node. Caso isso tenha acontecido, atualiza-se o contador checkPointsCruzados e incrementa-se o índice da heurística a avaliar.

É também importante mencionar que todo o caminho é guardado num tuplo que contém a linha, a coluna, a velocidade na linha e a velocidade na coluna.

Após os checkpoints terem sido todos cruzados a travessia pode terminar.

Secções Críticas

```
for idNode, velocidadeLinha, velocidadeColuna in carro.listaMovimentos:
    t = self.nodes[idNode].h1[checkPointsCruzados - 1], idNode, velocidadeLinha, velocidadeColuna
    if idNode not in visited:
        listaDistCheckPoint.append(t)
if len(listaDistCheckPoint) == 0:
    for idNode, velocidadeLinha, velocidadeColuna in carro.listaMovimentos:
        t = self.nodes[idNode].h1[checkPointsCruzados - 1], idNode, velocidadeLinha, velocidadeColuna
        if idNode in visited:
            visited.remove(idNode)
            listaDistCheckPoint.append(t)
        else:
            listaDistCheckPoint.append(t)
```

Figura 35 - Secção Crítica A*

Quando é que a lista de movimentos é vazia? Quando a travessia está num caminho sem saída. Isto acontece quando o único Node disponível é o anterior e já foi visitado. Então é necessário forçar o algoritmo a voltar para trás, mesmo que esse Node já tenho sido visitado.

Algoritmo Seguro

O algoritmo Seguro funciona exatamente da mesma forma que o A*. A única diferença é que este não acelera mais do que três unidades. É uma alternativa ao A* e há casos em que este apresenta uma performance igual ou melhor. Por exemplo, quando o circuito é fechado.

```
def seguro(self):
    checkPointsCruzados = 1
    totalCheckPoints = len(self.checkPoints)
    listaDistCheckPoint = []
    carro = Carro()
    carro.linha = self.nodes[self.partida].linha
    carro.coluna = self.nodes[self.partida].coluna
    carro.velocidadeLinha = 0
    carro.velocidadeColuna = 0
    t = tuple()
    t = self.nodes[self.partida].linha, self.nodes[self.partida].coluna, 0, 0
    outroT = tuple()
    parents = {t: t}
    path = [t]
    visited = set()
    visited.add(self.partida)
    existeCaminho = True
    while checkPointsCruzados <= totalCheckPoints:
        linhaAtual = carro.linha
        colunaAtual = carro.coluna
        velocidadeLinhaAtual = carro.velocidadeLinha
        velocidadeColunaAtual = carro.velocidadeColuna
        carro = self.limiteSeguro(carro)
        listaDistCheckPoint = []
        t = ()
```

Figura 36 – Algoritmo Seguro

A única diferença está na função limiteSeguro. Ao contrário da limiteVelocidade, a função limiteSeguro só permite alcançar a velocidade máxima de três.

Função drawPath

Criamos uma função que representa num ficheiro de texto a travessia do A*. O caminho é apresentado através do caracter 'A'.

```python
def drawPath(self, path=[], fileToRead="", fileToWrite=""):
    x = 0
    y = 0
    data = []
    with open(fileToRead, 'r') as originalFile:
        for line in originalFile:
            data.append(line)
    for linha, coluna, velocidadeLinha, velocidadeColuna in path:
        x = 0
        for line in data:
            if x == linha:
                data[x] = self.replace_char_at_index(line, coluna, 'A')
            x = x + 1
    with open(fileToWrite, 'w+') as newFile:
        newFile.writelines(data)
    return
```

Figura 37 - Função drawPath

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xPF-----------------------------------------------111x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figura 38 – Circuito Reta

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xAAAAA-AA--AA---AA----AA-----AA----AA---AA--AA-AAAAAx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figura 39 – Path do Circuito Reta

Testes Realizados

Neste tópico iremos mostrar a performance dos quatro algoritmos para os dez circuitos.

Circuito 1 – Circuito Aberto

DFS – Tamanho do Path: 915

BFS – Tamanho do Path: 966

Algoritmo Seguro – Tamanho do Path: 37

A* - Tamanho do Path: 34

Circuito 2 – Circuito Circular

DFS – Tamanho do Path: 40

BFS – Tamanho do Path: 1351

Algoritmo Seguro – Tamanho do Path: 15

A* - Tamanho do Path: 12

Circuito 3 – Circuito Labirinto

DFS – Tamanho do Path: 93

BFS – Tamanho do Path: 78

Algoritmo Seguro – Tamanho do Path: 20

A* - Tamanho do Path: 20

Circuito 4 – Circuito Oval

DFS – Tamanho do Path: 772

BFS – Tamanho do Path: 1585

Algoritmo Seguro – Tamanho do Path: 49

A* - Tamanho do Path: 43

Circuito 5 – Circuito Reta

DFS – Tamanho do Path: 98

BFS – Tamanho do Path: 101

Algoritmo Seguro – Tamanho do Path: 18

A* - Tamanho do Path: 16

Circuito 6 – Circuito Buraco

DFS – Tamanho do Path: 156

BFS – Tamanho do Path: 247

Algoritmo Seguro – Tamanho do Path: 84

A* - Tamanho do Path: 77

Circuito 7 – Circuito Estrada

DFS – Tamanho do Path: 299

BFS – Tamanho do Path: 699

Algoritmo Seguro – Tamanho do Path: 57

A* - Tamanho do Path: 53

Circuito 8 – Circuito Caminho Apertado

DFS – Tamanho do Path: 242

BFS – Tamanho do Path: 841

Algoritmo Seguro – Tamanho do Path: 107

A* - Tamanho do Path: 89

Circuito 9 – Circuito Espiral

DFS – Tamanho do Path: 120

BFS – Tamanho do Path: 224

Algoritmo Seguro – Tamanho do Path: 59

A* - Tamanho do Path: 53

Circuito 10 – Circuito Hexa

Algoritmo Seguro – Tamanho do Path: 190

A* - Tamanho do Path: 177

Conclusão

Iremos utilizar a conclusão para falar um pouco do que fizemos e do que não fizemos.

Vantagens:

O A* encontra sempre um caminho quase ótimo, o que por si já é muito bom.

Os circuitos são todos diferentes e com estratégias distintas.

Os algoritmos sabem lidar com caminhos sem saída.

Desvantagens:

Apesar do A* encontrar sempre um caminho quase ótimo há situações em que ele não atinge a velocidade máxima permitida devido a precauções que não estão bem calculadas.

O DFS e o BFS andam sempre com velocidade um.

A heurística da distância é aplicada para o primeiro checkpoint da lista. O primeiro checkpoint da lista ou está no limite esquerdo ou no limite direito da lista de checkpoints e isso implica que o algoritmo vá fazer a travessia próximo da parede.