Inteligência Artificial

Vector Race

Licenciatura em Engenharia Informática

Primeira Fase

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)

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 para 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.

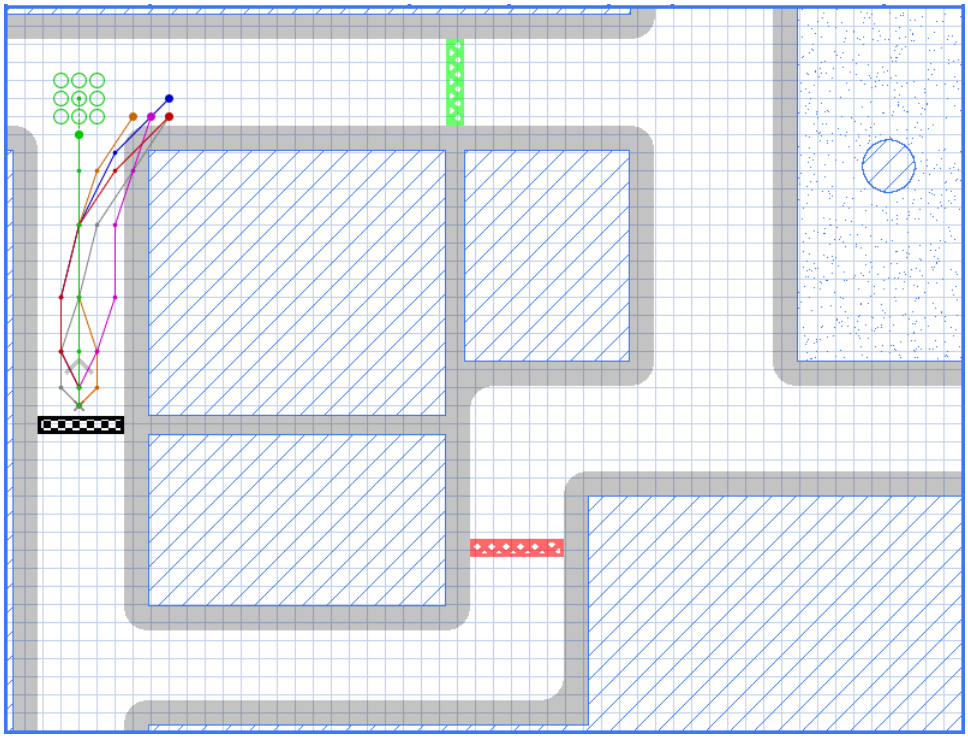


Figura 1 – Exemplo de um Estado VectorRace.

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) e algoritmo Breadth-First Search (BFS)

Estratégia Utilizada

Neste tópico iremos explicar detalhadamente a estratégia utilizada.

Construção de Circuitos

Criamos cinco circuitos para testar os algoritmos de procura não informada. O circuito Oval, o circuito Labirinto, o circuito Reta, o circuito Aberto e o circuito Circular. Todos eles possuem estratégias diferentes. No último tópico serão apresentados os testes realizados com cada um dos algoritmos de pesquisa no circuitos referidos.

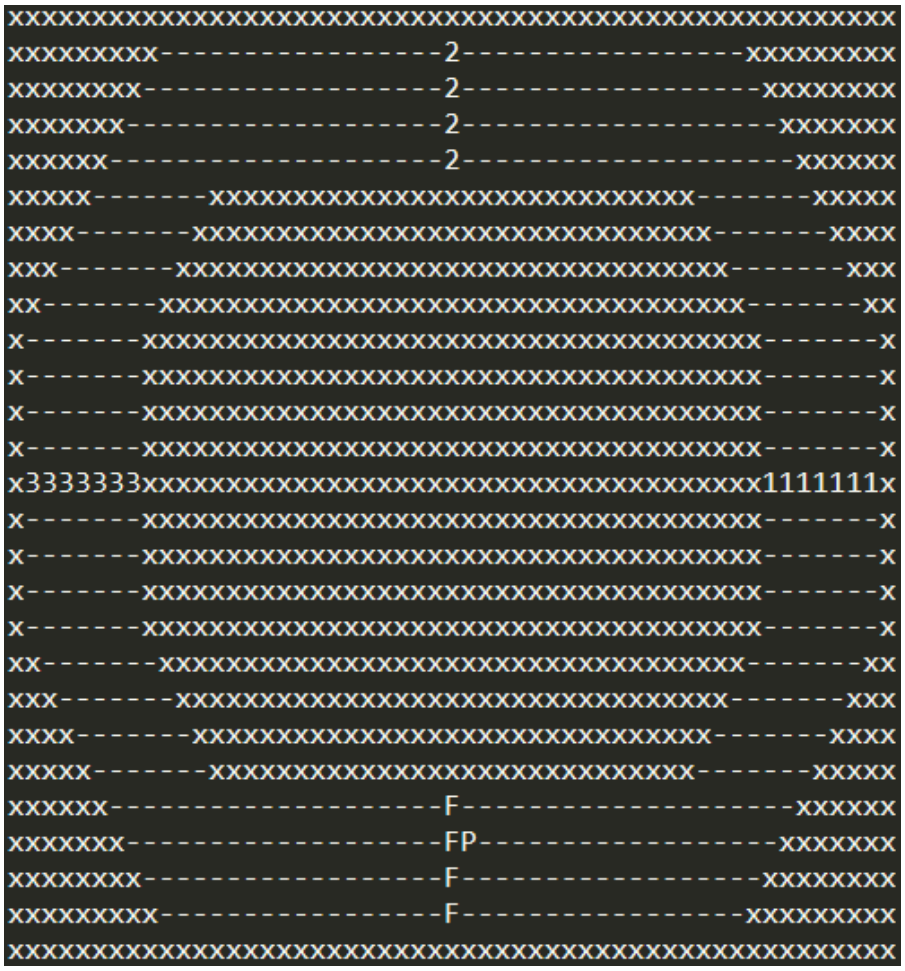


Figura 2 – Circuito Oval.

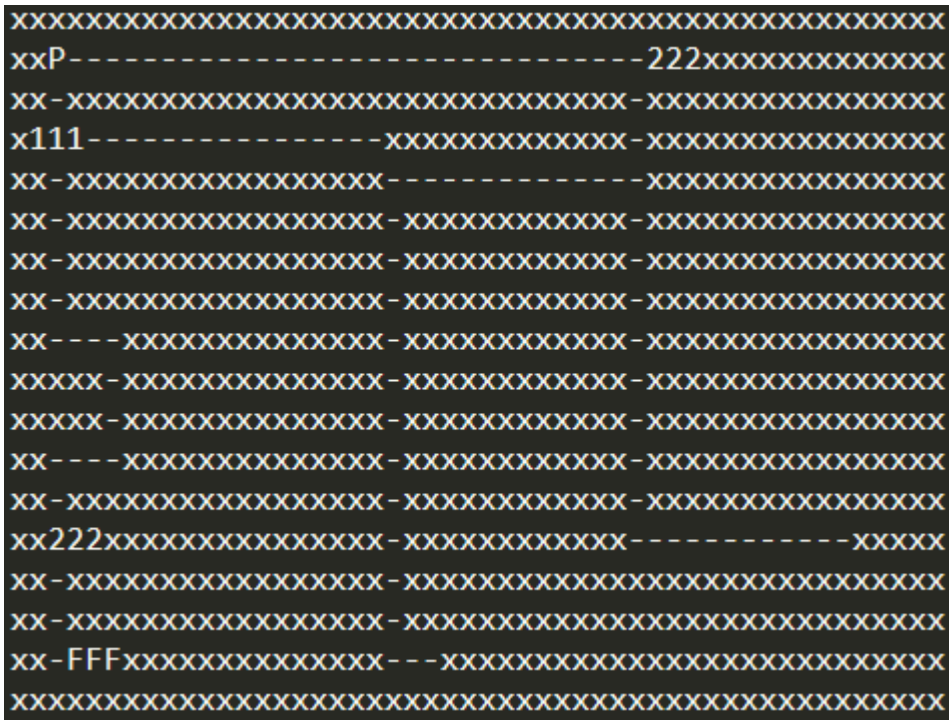


Figura 3 – Circuito Labirinto.

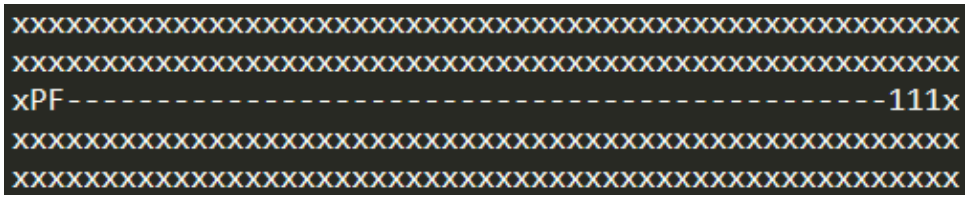


Figura 4 – Circuito Reta.

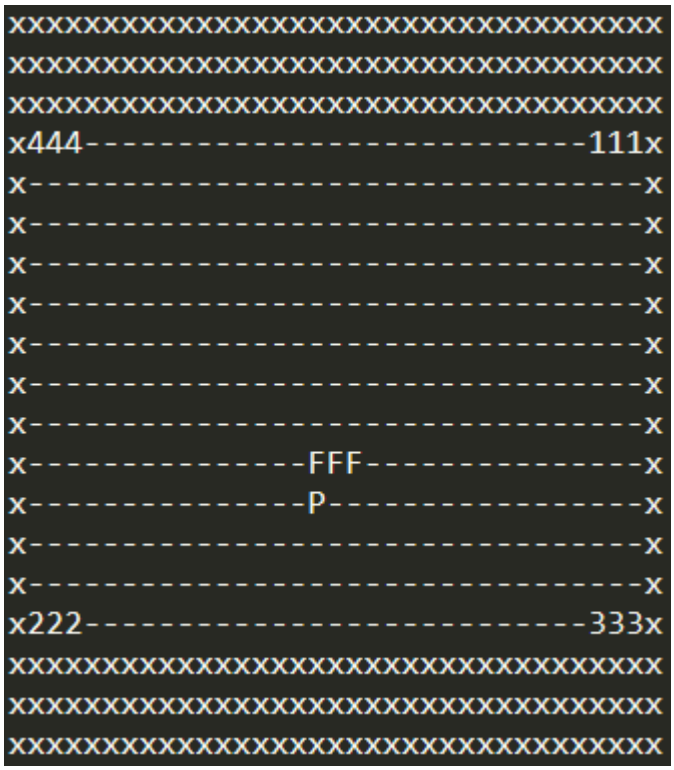


Figura 5 – Circuito Aberto.

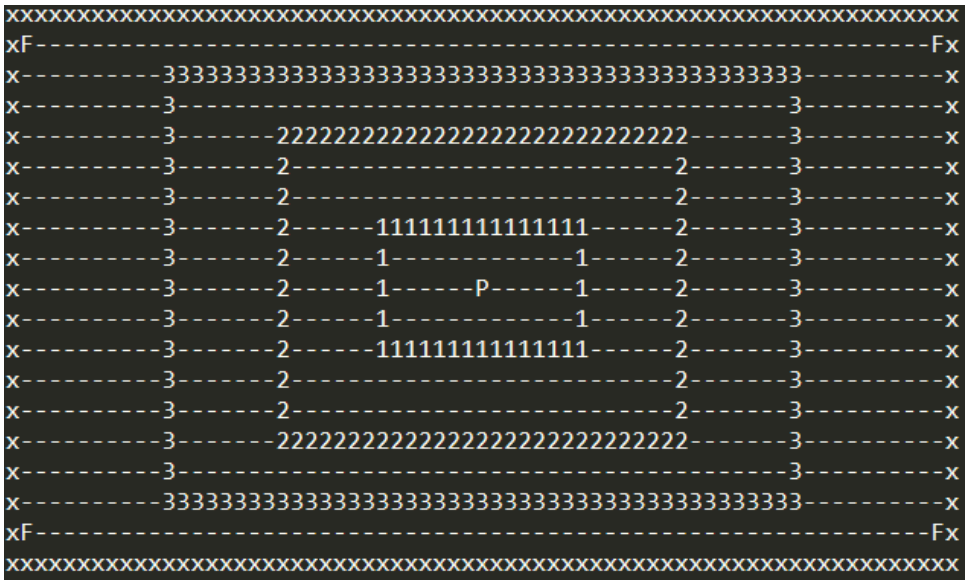


Figura 6 – Circuito Circular.

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 cinco caracteres permitidos para criar um circuito VectorRace:
  - O carater "X" representa os limites do circuito.
  - → O carater "-" representa uma coordenada vazia que pode ser utilizada para a travessia.
  - → O carater "P" representa a única posição de partida, tal como no jogo original.
  - → O carater "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".
  - → Algarismos "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 algarismo. 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.

Classe Grid

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

Figura 7 – 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 incerto.

Classe Node

```python
from Grid import Grid


class Node:
    id = int
    linha = int
    coluna = int
    type = Grid

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

    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 8 – Classe Node

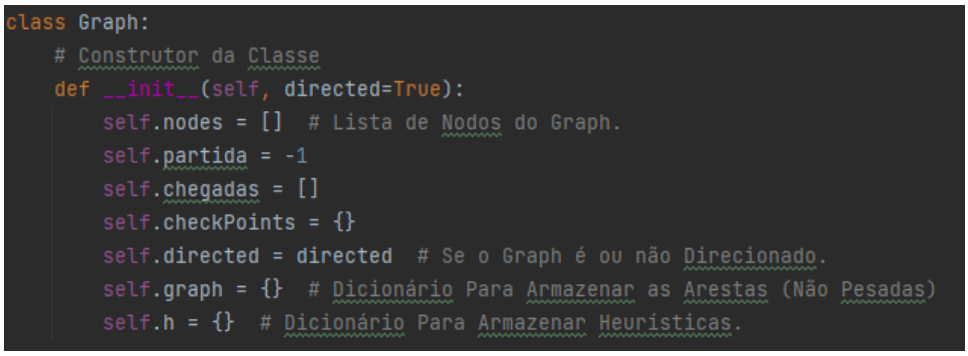
A classe Node é composta por:

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

Podíamos ter optado por uma solução mais compacta, 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.h = {}  # Dicionário Para Armazenar Heurísticas.
```

Figura 9 – 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, 0 1 e 2. Os checkpoint do tipo 0 estão nos Node cujo identificador é o 1030, 1031, 1032 e 1033. Os checkpoints do tipo 1 estão nos Node cujo identificador é o 2045, 2046, 2047 e 2048. Os checkpoint do tipo 2 estão nos Node cujo identificador é o 3045, 3046, 3047 e 3048. O dicionário seria populado da seguinte forma: {0: [1030, 1031, 1032, 1033], 1: [2045, 2046, 2047, 2048], 2:

[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.

→ **h**: O dicionário das heurísticas. A key é um inteiro que corresponde ao identificador do Node e a value é um set() de heurísticas ainda não definidas.

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 10 – 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 11 – 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 12 - Método criaGrafo

A lista directions contém tuplos com todas as direções possíveis. Optamos por incluir a direção (0,0) pois aquando implementadas as velocidades será 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 13 – 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.

As próximas imagens mostram os resultados do DFS para os nossos circuitos:

```
ID NODE: 1415. ID DOS NODES ADJACENTES: 1361 1362 1363
ID NODE: 1416. ID DOS NODES ADJACENTES: 1362 1363 1364
ID NODE: 1417. ID DOS NODES ADJACENTES: 1363 1364 1365
ID NODE: 1418. ID DOS NODES ADJACENTES: 1364 1365 1366
ID NODE: 1419. ID DOS NODES ADJACENTES: 1365 1366 1367
ID NODE: 1420. ID DOS NODES ADJACENTES: 1368 1366 1367
ID NODE: 1421. ID DOS NODES ADJACENTES: 1368 1367
ID NODE: 1422. ID DOS NODES ADJACENTES: 1368
ID NODE: 1423. ID DOS NODES ADJACENTES:
ID NODE: 1424. ID DOS NODES ADJACENTES:
ID NODE: 1425. ID DOS NODES ADJACENTES:
ID NODE: 1426. ID DOS NODES ADJACENTES:
ID NODE: 1427. ID DOS NODES ADJACENTES:
ID NODE: 1428. ID DOS NODES ADJACENTES:
ID NODE: 1429. ID DOS NODES ADJACENTES:
ID NODE: 1430. ID DOS NODES ADJACENTES:
ID NODO PARTIDA: 1246
ID NODOS CHEGADA: 1192 1245 1298 1351
ID NODOS CHECKPOINT:
ID NODOS CHECKPOINT NÚMERO 1: 734 735 736 737 738 739 740
ID NODOS CHECKPOINT NÚMERO 2: 79 132 185 238
ID NODOS CHECKPOINT NÚMERO 3: 690 691 692 693 694 695 696
```

Figura 14 – Excerto de Criação do Grafo para o Circuito Oval

```
Circuito Criado!

Algoritmo DFS:

O CheckPoint Número 1 Foi Cruzado no Nodo 736
O CheckPoint Número 2 Foi Cruzado no Nodo 132
O CheckPoint Número 3 Foi Cruzado no Nodo 690
Circuito Terminado No Nodo 1192!
O Caminho Encontrado é Constituído Por 805 Nodos!
Caminho: [1246, 1192, 1191, 1190, 1189, 1188, 1187, 1186, 1185, 1184, 1236, 1288, 1287, 1286, 1285, 1284, 1283, 1282, 1281, 1280, 1226, 1227, 1228, 1229, 123
O Algoritmo DFS Demorou Cerca de 0.0010004043579101562 Segundos!

Algoritmo BFS:

O CheckPoint Número 1 Foi Cruzado no Nodo 734
O CheckPoint Número 2 Foi Cruzado no Nodo 132
O CheckPoint Número 3 Foi Cruzado no Nodo 692
Circuito Terminado No Nodo 1351!
O Caminho Encontrado é Constituído Por 1594 Nodos!
Caminho: [1246, 1192, 1193, 1194, 1298, 1299, 1300, 1245, 1247, 1191, 1244, 1248, 1195, 1350, 1351, 1352, 1297, 1353, 1354, 1301, 1190, 1243, 1296, 1249, 119
O Algoritmo BFS Demorou Cerca de 0.0011034011840820312 Segundos!

Process finished with exit code 0
```

Figura 15 – Execução do DFS e do BFS para o Circuito Oval

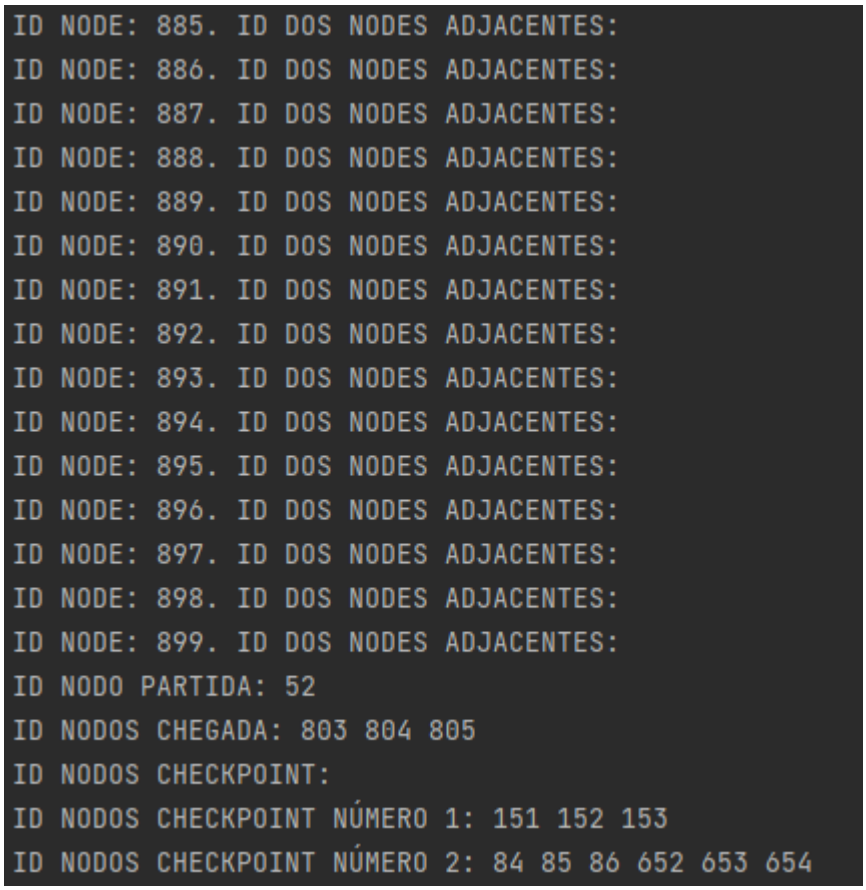


Figura 16 – Excerto de Criação do Grafo para o Circuito Labirinto

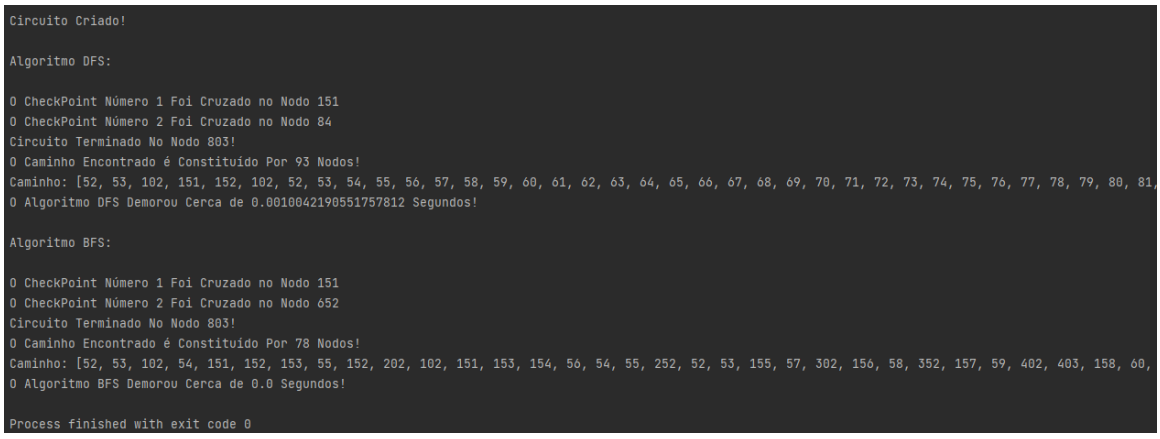


Figura 17 · Execução do DFS e do BFS para o Circuito Labirinto

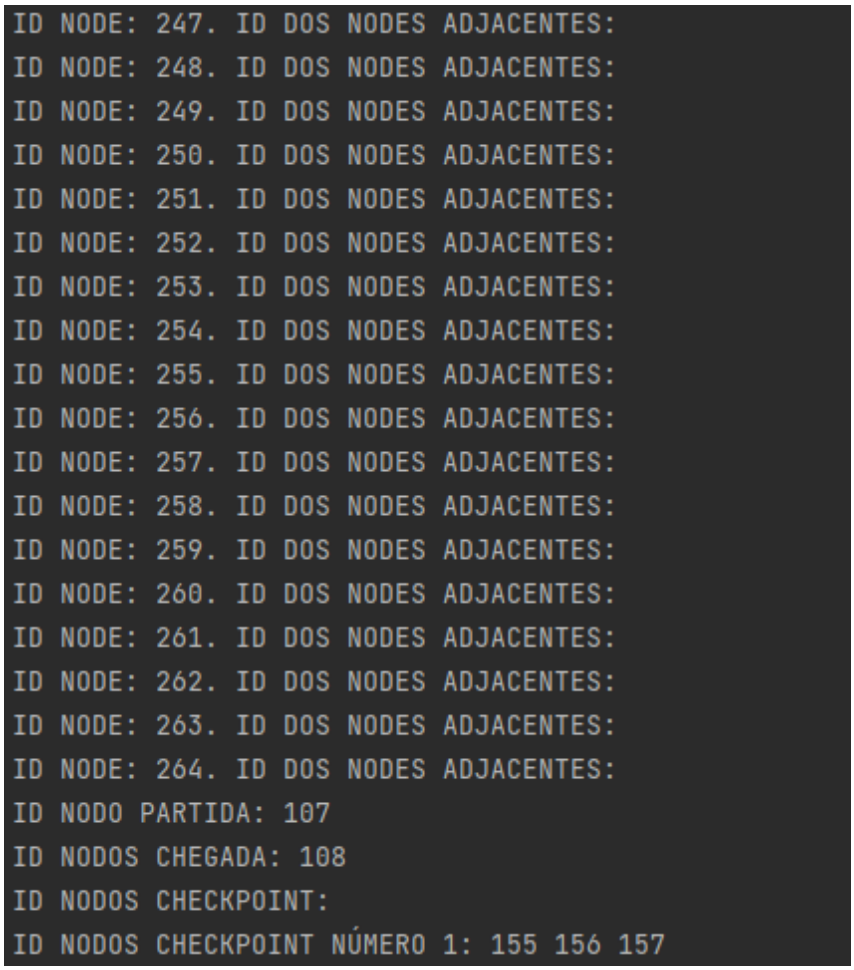


Figura 18 - Excerto de Criação do Grafo para o Circuito Reta

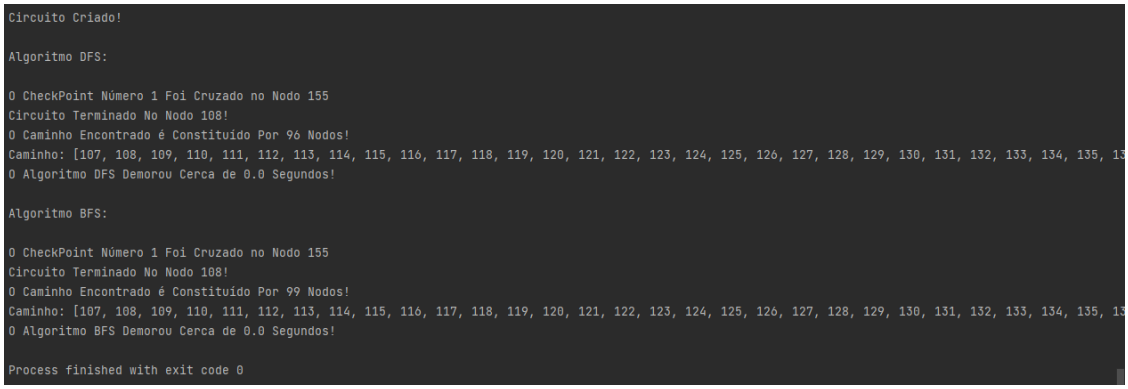


Figura 19 - Execução do DFS e do BFS para o Circuito Reta

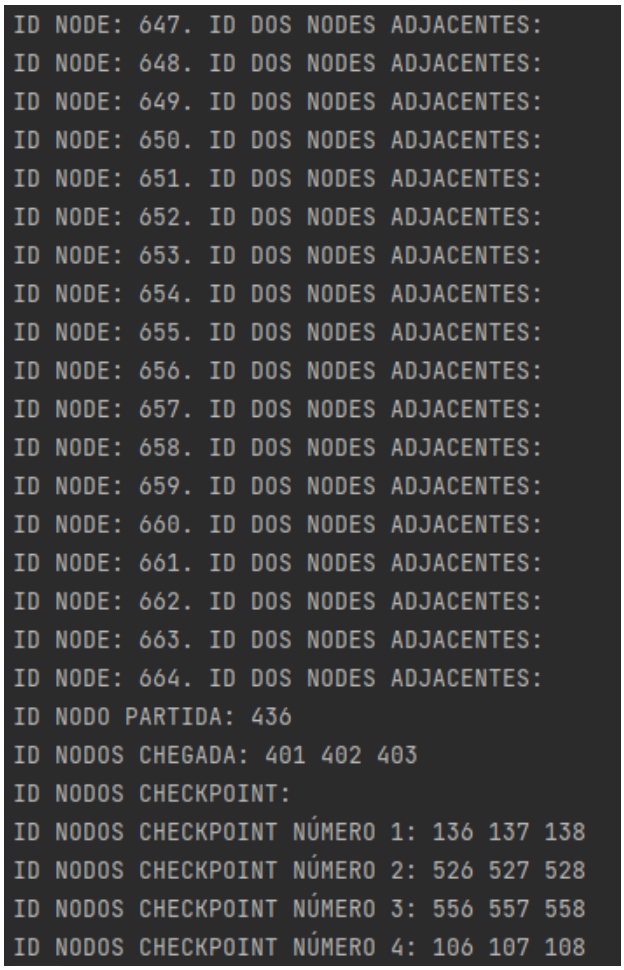


Figura 20 - Excerto de Criação do Grafo para o Circuito Aberto

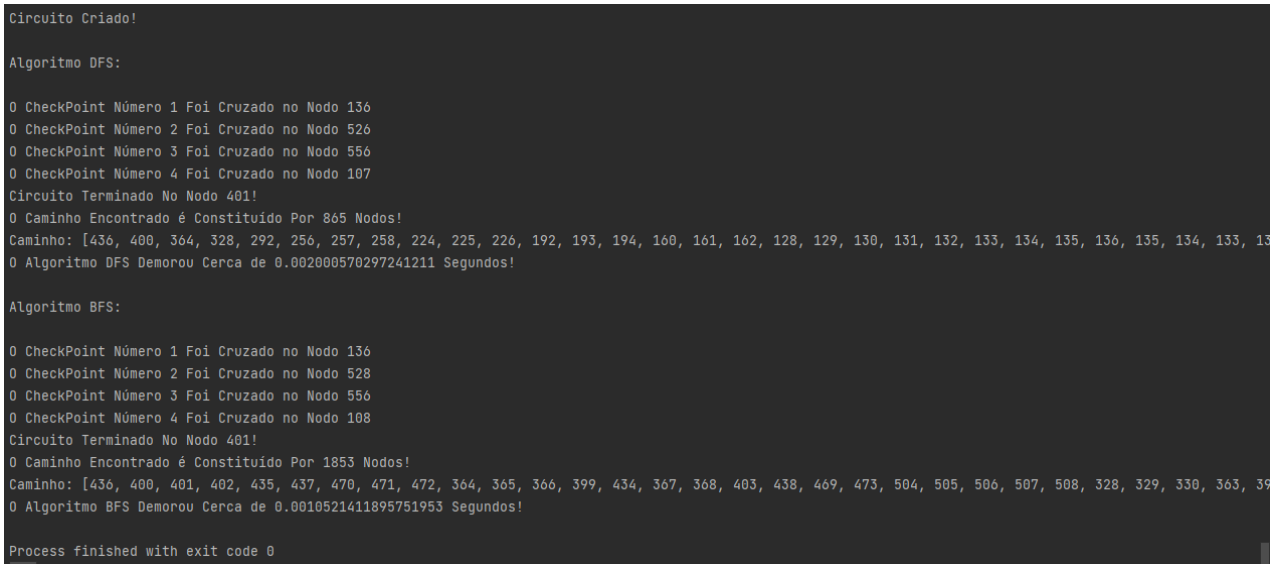


Figura 21 - Execução do DFS e do BFS para o Circuito Aberto

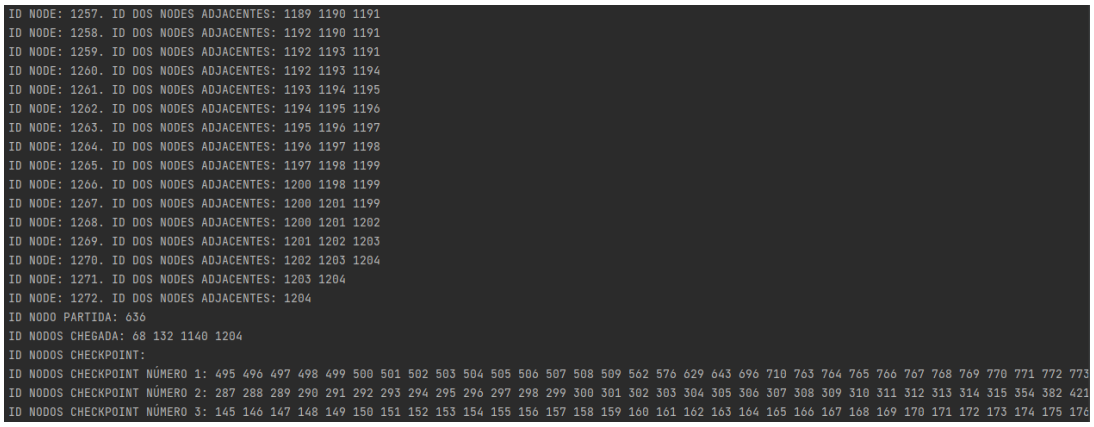


Figura 22 - Excerto de Criação do Grafo para o Circuito Circular

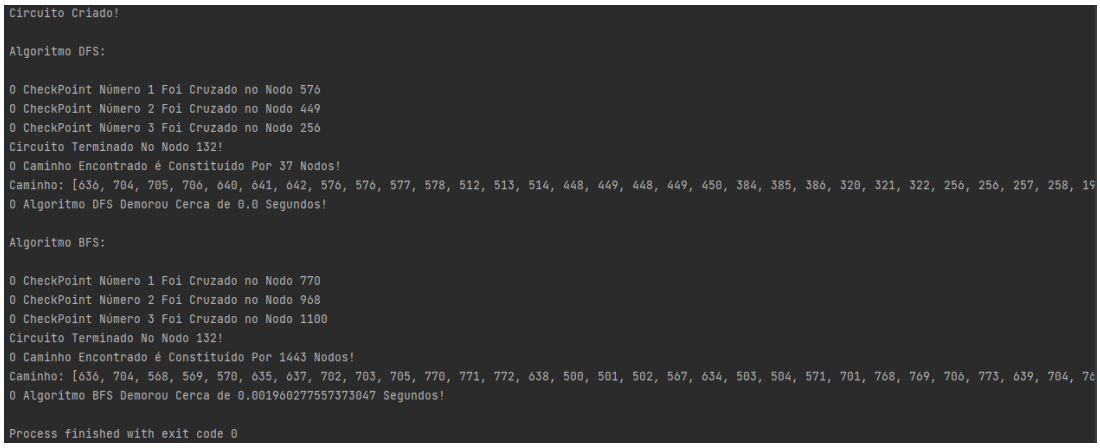


Figura 23 - Execução do DFS e do BFS para o Circuito Circular

Conclusão

Através dos testes feitos nos nossos mapas criados, ficamos a perceber que o algoritmo de pesquisa DFS costuma ser uma solução mais otimizada do que o algoritmo BFS.

Dito isto, o custo de cada operando será diretamente proporcional às jogadas utilizadas até se encontrar o estado final, pois, como referido no enunciado, cada jogada tem o custo de um segundo.

Resta também mencionar que ainda não implementamos aceleração e velocidade.