



**Universidade do Minho**

Escola de Engenharia

Licenciatura em Engenharia Informática

## **Unidade Curricular de Inteligência Artificial**

Ano Letivo de 2022/2023

### **Trabalho Prático - Fase 1**

Filipa Gomes(A96556)   Pedro Oliveira(A95076)   Ricardo Oliveira(A96794)  
Rodrigo Freitas(A96547)

25 de Novembro de 2022



# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Estrutura do Relatório . . . . .	1
<b>2</b>	<b>Formulação do problema como um problema de pesquisa</b>	<b>2</b>
<b>3</b>	<b>Geração de circuitos</b>	<b>3</b>
3.1	Tipos de Mapas . . . . .	4
3.1.1	Exemplos de Mapas Criados . . . . .	4
<b>4</b>	<b>Gerar Grafos</b>	<b>5</b>
4.1	Estrutura de um Grafo . . . . .	5
4.2	Heurísticas . . . . .	6
4.2.1	Implementação Global . . . . .	6
4.2.2	Euclidean . . . . .	7
4.2.3	Manhattan . . . . .	7
4.2.4	Supercover . . . . .	8
4.3	Load Save de Estados . . . . .	8
<b>5</b>	<b>Representação do método de procura graficamente</b>	<b>9</b>
5.1	Estratégia . . . . .	9
5.2	Interações . . . . .	9
<b>6</b>	<b>Estratégia de procura</b>	<b>11</b>
6.1	Não informadas . . . . .	11
6.1.1	Breadth-first(BFS) . . . . .	11
6.1.2	Depth-first(DFS) . . . . .	12
6.2	Informadas: . . . . .	12
6.2.1	Algoritmo A* . . . . .	12
6.2.2	Greedy-search . . . . .	13
6.2.3	Dijkstra . . . . .	13
<b>7</b>	<b>Funcionamento</b>	<b>14</b>
7.1	Run Algos . . . . .	15
7.2	Build Graph . . . . .	18
7.3	Build All . . . . .	19
7.4	Show Map . . . . .	19
7.5	Show Graph . . . . .	20

7.6 Run Tests . . . . .	20
<b>8 Observação de Resultados</b>	<b>22</b>
<b>9 Conclusões e Trabalho Futuro</b>	<b>25</b>

# Lista de Figuras

3.1	Exemplo de ficheiro *.rmap . . . . .	3
3.2	Circuito : maps/02.rmap . . . . .	4
3.3	Labirinto: maps/14.rmap . . . . .	4
4.1	Diferença entre a heurística de Manhattan e a Euclidian . . . . .	7
4.2	Representação da Supercover . . . . .	8
7.1	Comando para execução do programa . . . . .	14
7.2	Menu . . . . .	15
7.3	Seleção do tipo de mapa . . . . .	15
7.4	Mapas . . . . .	16
7.5	Heurísticas . . . . .	16
7.6	Algoritmos . . . . .	17
7.7	Solução . . . . .	17
7.8	Representação dos movimentos . . . . .	18
7.9	Representação dos steps . . . . .	18
7.10	Construção do grafo . . . . .	19
7.11	Construção do grafo para todos os mapas . . . . .	19
7.12	Representação gráfica de um mapa . . . . .	20
7.13	Representação gráfica do grafo do mapa 1 . . . . .	20
7.14	Execução de todos os algoritmos para todos os mapas . . . . .	21

# 1 Introdução

Para este trabalho é pretendido o desenvolvimento de diversos algoritmos de procura para a resolução de um jogo, nomeadamente o VectorRace, também conhecido como RaceTrack. Este é um jogo de simulação de carros simplificado, contém um conjunto de movimentos e regras associadas.

## 1.1 Estrutura do Relatório

O presente relatório é constituído por 7 capítulos:

- **Introdução:** Pequena introdução do trabalho realizado e alguma contextualização;
- **Formulação do problema como um problema de pesquisa:** Onde é formulado e especificado todo o processo de pesquisa
- **Geração de circuitos:** Processo de construção de alguns circuitos;
- **Representação da pista em forma de Grafo:** Representação dos circuitos gerados representados por grafos;
- **Estratégia de procura:** Especificação da estratégia de procura e alguns resultados dessa mesma estratégia;
- **Conclusões e Trabalho Futuro:** Conclusões e considerações finais do grupo após realização deste projeto.

## 2 Formulação do problema como um problema de pesquisa

Nesta fase é nos pedido que criemos um problema de pesquisa conforme o enunciado dado. Neste caso o nosso agente será o carro do jogador, o objetivo será chegar á meta final (ao ponto F), enquanto o nosso problema será qual será o melhor caminho ou quais serão todos os possíveis caminhos para o agente completar o seu objetivo, chegar à meta. O estado inicial do agente será representado como ponto P enquanto as suas ações o moverão do estado inicial para outro estado até ao seu estado objetivo.

- **Estado inicial:** Ponto inicial do mapa, representado por P
- **Estado Objetivo:** Ponto final do mapa, representado por F
- **Estado:** As várias coordenadas do circuito em conjunto com a velocidade do jogador nessa coordenada
- **Operações:** Aplicar uma aceleração sobre o estado atual
- **Solução:** Sequência de coordenadas que levam ao objetivo, tendo em conta o algoritmo utilizado.
- **Custo da solução:** O custo da solução e calculado segundo o numero de movimentos entre o estado inicial e o estado objetivo. Estes têm o custo de 1 ou 25, no caso de existir uma colisão.

### 3 Geração de circuitos

Para a criação de mapas usamos uma extensão específica para os mesmos '.rmap'. Na sua construção é necessário garantir que todas as linhas tenham o mesmo comprimento. Como notação optamos por:

- **X** - Paredes
- **P** - Player
- **F** - Final
- **-** - Caminho

O ficheiro deverá então ser guardado na pasta 'maps/' e o nome deverá ser do tipo '[nome].rmap'. A representação textual do mapa deverá começar na primeira linha do ficheiro e, quando o mesmo é *parsed*, todas as linhas com um tamanho menor à maior linha do mapa, a mesma é completa com paredes entre o final da mesma e o máximo.

```
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
```

Figura 3.1: Exemplo de ficheiro \*.rmap

## 3.1 Tipos de Mapas

Ao longo da realização da primeira fase deste projeto depara-mo-nos com as diversas possibilidades de procura e as diferentes soluções obtidas por cada uma delas.

Para conseguir testar, quer os diferentes algoritmos, quer a geração dos grafos, o grupo optou pela criação de vários mapas. Os mesmos serão utilizados no capítulo relativo aos testes e resultados.

### 3.1.1 Exemplos de Mapas Criados

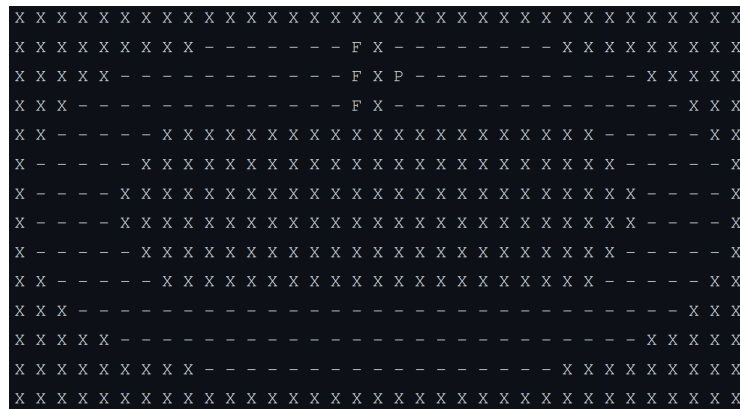


Figura 3.2: Circuito : maps/02.rmap

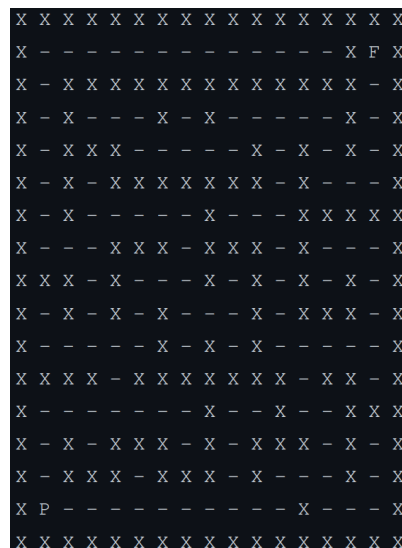


Figura 3.3: Labirinto: maps/14.rmap



## 4 Gerar Grafos

Para este projeto, foram implementadas algumas funcionalidades que achamos importantes para um funcionamento mais rápido e intuitivo do programa, bem como para uma maior diversidade de resultados.

Assim sendo, ao longo deste capítulo exploraremos as estratégias implementadas para a geração de grafos para a futura implementação.

### 4.1 Estrutura de um Grafo

Um grafo é uma estrutura composta por nodos e arestas. Ambos, através de diferentes condições, têm um peso atribuído. O peso de um dado nodo depende da heurística utilizada e representa a distância expectada entre o nodo atual e o nodo de destino. Cada nodo é representado pelo seu nome, em formato de string, sendo caracterizado da seguinte forma: ' $x,y,v_x,v_y$ ', onde  $x,y$  é a sua posição atual e  $v_x,v_y$  a sua velocidade atual no nodo.

Esta decisão foi tomada uma vez que, para a mesma coordenada, o jogador pode possuir diferentes velocidades, resultando em diferentes estados possíveis. I.e., um jogador que se encontre no nodo ' $1,1,0,0$ ' (coordenadas  $x,y=1,1$  e velocidades  $v_x,v_y=0,0$ ) poderá mover-se para todas as posições adjacentes à coordenada ' $1,1$ '. No entanto, um jogador que se encontre no nodo ' $1,1,1,0$ ' se poderá apenas mover para as coordenadas adjacentes à coordenada ' $2,1$ ', devido ao facto de já apresentar uma velocidade positiva no eixo horizontal.

Por sua vez, o peso de uma aresta reflete, no sistema do jogo, a pontuação adicionada ao jogador a cada movimento. Isto é, um movimento normal entre duas coordenadas apresenta um peso 1, enquanto um movimento que origine colisão apresenta um peso de 25.

O grafo para ser criado necessita de um mapa, construindo assim, segundo uma dada heurística, a expansão dos possíveis movimentos e posições do jogador. Os movimentos podem ser uma combinação de duas velocidades ( $v_x,v_y$ ), valores que variam entre -1 e 1 tomando apenas valores inteiros.

Assim sendo, para cada nodo existem, no máximo, 9 expansões possíveis, respetivamente:

'NW' : (-1, -1)	'N' : (0 , -1)	'NE' : ( 1, -1)
'W' : (-1, 0)	'K' : (0 , 0)	'E' : ( 1, 0)
'SW' : (-1, 1)	'S' : (0 , 1)	'SE' : ( 1, 1)

Do tipo ' $m:(ax,ay)$ ', onde ' $m$ ' é o nome do movimento, e o par  $(ax,ay)$  é a aceleração aplicada pelo movimento.

Assim, após ser expandido para todos os movimentos é calculado o peso da aresta, verificando se existiu colisão após aplicar a aceleração ao nodo atual. O novo nodo é então a coordenada original com adição da velocidade atual mais a nova aceleração.

## 4.2 Heurísticas

Nesta fase do projeto dispomos de três opções para geração de grafos, i.e., três diferentes heurísticas de modo a estimar o peso de um nodo. Este peso simboliza a estimativa da distância entre o mesmo e a meta e é utilizado por algoritmos, como por exemplo o A\*, para o cálculo do melhor caminho entre dois pontos.

### 4.2.1 Implementação Global

As heurísticas são aplicadas a quando da criação do grafo, no processo de expansão. Quando um novo nodo é criado, este vai utilizar a heurística para estimar a distância entre si e o nodo objetivo, passando isso a ser o seu peso.

Uma vez que se trata de um problema de minimização e existe a possibilidade de existirem várias coordenadas finais, a estratégia para a aplicação da heurística passou por calcular os valores entre a coordenada atual e cada uma das possíveis '*metas*'. Após ter uma lista de todos os valores é calculado o mínimo e usado para peso do nodo.

É importante notar que, após testes e ponderações por parte do grupo, foi adicionada uma penalização para nodos onde, mesmo não batendo, não ocorre um movimento. Esta penalização é de 5 unidades de distância e tem como objetivo, não só, penalizar a paragem do jogador (uma vez que o jogo é referente a corridas), mas principalmente, melhorar a procura do algoritmos de pesquisa informada.

### 4.2.2 Euclidean

A heurística Euclidiana, vinda da definição matemática, estima a distância mínima entre dois pontos calculando a linha reta que os conecta. Para obter o valor dessa distância recorreremos à definição:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (4.1)$$

Onde  $(x_1, y_1)$  são as coordenadas do nodo atual e  $(x_2, y_2)$  as coordenadas do destino.

### 4.2.3 Manhattan

A heurística de Manhattan, ao contrário da euclidiana, é obtida através da soma das diferenças absolutas de suas coordenadas. O nome Manhattan alude à estrutura quadriculada da cidade, uma vez que, para o cálculo da distância esta tem mais em conta a geometria, como se calcula-se a distância numa matriz apenas movendo vertical ou horizontalmente.

O seu cálculo é bastante mais simples, passando apenas pela seguinte fórmula.

$$|x_2 - x_1| + |y_2 - y_1| \quad (4.2)$$

A diferença da anterior em termos de valor total pode ser visualizada na imagem a baixo:



Figura 4.1: Diferença entre a heurística de Manhattan e a Euclidian

### 4.2.4 Supercover

Uma '*Supercover Line*' é um conjunto de todos os pontos pelo qual a linha reta entre duas coordenadas passa. I.e., ao calcular a reta entre ambos, como na heurística euclidiana, são calculadas as coordenadas pelas quais o jogador precisaria de passar para fazer o caminho entre ambos (dado estar num ambiente bi-dimensional). Um exemplo do funcionamento deste algoritmo pode ser visualizado na imagem seguinte:

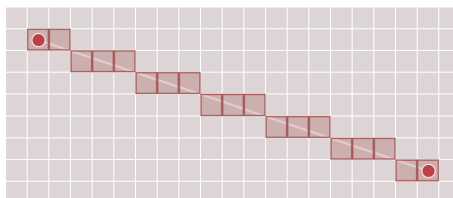


Figura 4.2: Representação da Supercover

Após o programa possuir a lista de coordenadas que o jogador precisaria de passar para ir do ponto A ao ponto B é então estimada a distância, calculando o tamanho da lista.

```
lista_coordenadas=get_supercover_line(x1,y1,x2,y2)
distancia_estimada=len(lista_coordenadas)
```

Onde '*get\_supercover\_line()*' é a função que retorna a lista de coordenadas entre dois pontos. Esta é também a função utilizada para calcular o movimento de um jogador quando aplicado um movimento.

## 4.3 Load|Save de Estados

Durante a realização do trabalho, sentimos a necessidade de melhorar a performance do mesmo. Para isso decidimos implementar a possibilidade de guardar os mapas previamente carregados. Isto permite a sua utilização para que se possa utilizar os algoritmos de procura sem um novo '*load*' do grafo.

No que toca à sua implementação, sempre que executamos os algoritmos sobre um mapa ou escolhemos carregar o grafo é gravado um ficheiro do tipo '\*.stt', localizado na pasta 'cache/'. Estes ficheiros contêm a informação do estado do programa, isto é, qual o mapa, o grafo correspondente e a localização inicial do jogador e das metas.

## 5 Representação do método de procura graficamente

Foi criada uma estratégia de fácil compreensão e de visualização da aplicação dos algoritmos de procura sobre os grafos.

### 5.1 Estratégia

Quando um algoritmo de procura é utilizado com a opção de replay é criado um ficheiro do tipo '\*.rpl', presente na pasta 'replays/[nome do algoritmo]/'. Este ficheiro contém o mapa original sobre o qual a procura foi realizada na linha inicial de modo a poder ser gerada uma representação gráfica.

Após a linha inicial, cada entrada do ficheiro indica qual o nodo de procura atual, todos os nodos abertos para procura atualmente e todos os nodos que já foram analisados.

Com tudo isto, podemos recorrer ao *script* fornecido no repositório com o nome 'replay.py', fornecendo como argumento do programa o nome do ficheiro de replay. Em alternativa o mesmo pode ser visualizado no final da execução do algoritmo no módulo principal do programa.

### 5.2 Interações

Um caracter **verde** significa que existe pelo menos um nodo que pode ser verificado naquela coordenada do mapa, sendo que, mais nodos na mesma coordenada, dá origem a uma representação de um caracter com um tom verde mais escuro.

Por outro lado, uma coordenada representada a **laranja** indica não haver nenhum nodo aberto, porém já verificado anteriormente. Mais uma vez, um tom mais escuro indica uma maior quantidade de vezes que a coordenada já foi verificada.

O nodo **vermelho** é a coordenada que está a ser verificada. Esta visualização dos algoritmos de procura foi desenvolvida dado a possibilidade de vários nodos estarem presentes na mesma coordenada, uma vez que no decorrer do jogo o jogador, pode alcançar, por exemplo, a coordenada  $(x,y)$  com uma velocidade  $(1,1)$ ,  $(1,0)$ , etc.

Assim sendo, a expansão do grafo é feita em concordância não só, com as coordenadas, mas também com a velocidade do jogador nesse nodo, já que esta afeta as possibilidades do próximo movimento do jogador.

## 6 Estratégia de procura

Neste capítulo apresentaremos a estratégia escolhida pelo grupo, alguns motivos da escolha e a sua justificação. Para a implementação da estratégia de procura usamos duas estratégias não informadas e duas informadas, sendo elas, Breadth-first e Depth-first como não informadas e informadas temos o Algoritmo A\*, Algoritmo de Dijkstra (Custo Uniforme) e a Greedy-search.

### 6.1 Não informadas

#### 6.1.1 Breadth-first(BFS)

Todos os nós de menor profundidade são expandidos primeiro, o bom desta estratégia é de ser uma procura muito sistemática, apesar de ser muito “time-consuming” e ocupar muito espaço.

Como o fator de ramificação (coordenadas possíveis no circuito são finitas, torna-se uma procura completa tendo um tempo em que supondo fator de ramificação  $b$ ,  $n = 1 + b + b^2 + b^3 + \dots + b^n = O(b^d)$ , exponencial em  $d$ . Guardando cada nó em memória,  $O(b^d)$  ocupando assim muito espaço. Tal como nos é dito no enunciado o custo de mudança de coordenada é 1 se permanecer na pista logo poderemos considerar ser ótima.

- **b** : o máximo fator de ramificação (o número máximo de sucessores de um nó) da árvore de procura;
- **d** : a profundidade da melhor solução;
- **m** : a máxima profundidade do espaço de estados;

Este algoritmo, consegue na maior parte das vezes, obter ou chegar próximo daquela que é a solução ótima para o problema. Contudo uma das desvantagens é o facto de este não detetar colisões, uma vez que não é informado. Isto provoca assim uma disparidade do score em alguns casos, uma vez que a cada colisão o score aumenta 25 pontos.

## 6.1.2 Depth-first(DFS)

Os nós mais profundos da árvore serão sempre os expandidos, o bom desta estratégia é que necessita de muito pouca memória e ótimo para problemas com muitas soluções como o que temos. Apesar disso não pode ser usada em árvores com profundidade infinita, pode ficar presa em ramos errados, mas como a nossa árvore é finita não tem existe problema. Não sendo uma procura completa, o seu tempo é de  $O(b^m)$ , mau se  $m > d$ , e não é ótima, pois devolve sempre em princípio a 1.<sup>a</sup> solução que encontra.

Este algoritmo, não se trata dos melhores no que toca a atingir uma solução ótima, uma vez que, a sua pesquisa em profundidade dos nodos o prejudica. Por outro lado, a sua execução é bastante rápida.

## 6.2 Informadas:

Procura informada em que se utiliza informação sobre o problema para evitar que o algoritmo de procura fique perdido e sem rumo.

### 6.2.1 Algoritmo A\*

Evita a expansão de caminhos que são dispendiosos, o algoritmo A\* combina a procura greedy com a uniforme, minimizando a soma do caminho já efetuado com o mínimo previsto do que falta até a solução. Usa a função:

$$f(n) = g(n) + h(n);$$

$$g(n) = \text{custototal}, \text{ até agora, para chegar ao estado } n \text{ (custo do percurso)};$$

$$h(n) = \text{custoestimado para chegar ao objetivo (não deve sobrestimar o custo para chegar à solução (heurística))};$$

$$f(n) = \text{custoestimado da solução menos dispendiosa que passa pelo nó } n.$$

Este algoritmo é dos que obtém os melhores resultados no requisito score, sendo que este tem em consideração, todos os pontos essenciais para a obtenção de uma solução ótima, tratando-se do custo da aresta para o próximo nodo, o custo do nodo para o qual se vai deslocar e o



custo até ao nodo em que se encontra. Tendo em conta que este algoritmo considera vários aspetos para apresentar uma solução, este sofre uma perda de eficiência.

### 6.2.2 Greedy-search

Expansão do nó que parece estar mais perto da solução, onde o  $h(n)$  = custo estimado do caminho mais curto do estado  $n$  para o objetivo (função heurística).

Não é uma estratégia de procura completa, pois pode entrar em ciclo, e é suscetível a falsos arranques. quanto à complexidade no tempo é de  $O(b^m)$ , mas com uma boa função heurística pode diminuir consideravelmente, quanto à complexidade no espaço  $O(b^m)$ , mantendo sempre todos os nós em memória. Não é ótima porque não encontra sempre a solução ótima, sendo necessário detetar estados repetidos.

Neste trabalho, este algoritmo analisa no máximo 9 nodos de cada vez escolhendo o mais barato. Caso não encontre uma solução este volta para traz. Relativamente ao score este apresenta soluções satisfatórias e uma eficiência melhor que o  $A^*$ .

### 6.2.3 Dijkstra

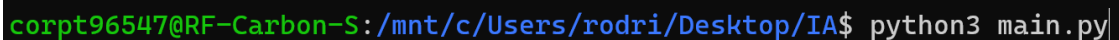
O algoritmo de Dijkstra, resolve o problema do caminho mais curto num grafo onde os peso/custos são não negativos, como no caso atual, com um tempo de  $O(E + V \log(V))$  onde  $V$  é o número de vértices e  $E$  é o número de arestas. Considera um conjunto  $S$  de caminhos menores, com um vértice inicial  $I$ . A cada passo do algoritmo procura-se nas adjacências dos vértices pertencentes a  $S$  o vértice com menor distância relativa a  $I$  e adiciona-o a  $S$  e, então, repetindo os passos até que todos os vértices alcançáveis por  $I$  estejam em  $S$ . Arestas que ligam vértices que já pertencem a  $S$  não serão consideradas possíveis candidatas.

Este algoritmo, é muito semelhante ao  $A^*$ , diferindo no facto de este apenas ter em conta o peso atual e o peso da aresta. Este apresenta, assim como o  $A^*$ , scores bastante positivos, por outro lado este perde um pouco na eficiência, pois parte sempre do nodo com o menor custo.

## 7 Funcionamento

Para que seja iniciado o programa é necessário instalar as bibliotecas necessárias para a sua execução, este processo pode ser feito através do comando `pip install requirements.txt`, que instalará todas as bibliotecas presentes no arquivo `'requirements.txt'` que se encontra no nosso repositório.

Após garantidas as condições necessárias para a execução do programa, este pode ser feito através da compilação do arquivo `'main.py'`.



```
corpt96547@RF-Carbon-S:/mnt/c/Users/rodri/Desktop/IA$ python3 main.py|
```

Figura 7.1: Comando para execução do programa

Com o programa em execução um menu é apresentado, onde é possível escolher entre 6 opções diferentes, sendo estas:

- **Run Algos** - Permite obter a solução para os diversos mapas, tendo em conta os algoritmos implementados
- **Build Graph** - Proporciona a criação do grafo de um mapa escolhido
- **Build All** - Efetua a mesma operação que o *Build Graph*, contudo agora, para todos os mapas
- **Show Map** - Representação gráfica dos mapas
- **Show Graph** - Representação gráfica dos grafos, anteriormente já criados com a opção *Build Graph* (Esta ação pode ser um pouco demorada, dada a dimensão dos mapas e consequentemente o seu número elevado de nodos)
- **Run Tests** - Permite a obtenção dos tempos de execução, bem como, a pontuação da execução de todos os algoritmos para os mapas existentes.

```
Main Menu
1 ) Run Algos
2 ) Build Graph
3 ) Build All
4 ) Show Map
5 ) Show Graph
6 ) Run Tests
0 ) Exit
-> |
```

Figura 7.2: Menu

## 7.1 Run Algos

Na opção *Run Algos*, inicialmente é dada a opção de usarmos um mapa novo ou que já se encontre em cache, isto é, se pretendemos executar um mapa da pasta dos mapas, ou usar um que já foi previamente carregado no programa.

```
Type
1 ) New Map
2 ) Cached Map
0 ) Cancel
-> |
```

Figura 7.3: Seleção do tipo de mapa

```
Map Select
1 ) maps/01.rmap
2 ) maps/02.rmap
3 ) maps/03.rmap
4 ) maps/04.rmap
5 ) maps/05.rmap
6 ) maps/06.rmap
7 ) maps/07.rmap
8 ) maps/08.rmap
9 ) maps/09.rmap
10 ) maps/10.rmap
11 ) maps/11.rmap
12 ) maps/12.rmap
13 ) maps/13.rmap
14 ) maps/14.rmap
15 ) maps/15.rmap
16 ) maps/drag_n_back.rmap
17 ) maps/portimao.rmap
0 ) Exit
```

Figura 7.4: Mapas

Exclusivamente no caso de ser selecionada a opção de usarmos um mapa novo, será também dada a opção de escolher a heurística. Neste momento dispomos de 3 tipos de heurísticas, sendo estas a de Manhattan, a Euclidiana e a Superclover. A Superclover será utilizada como heurística default no nosso programa.

```
Heuristic Select
1 ) Manhattan
2 ) Euclidean
3 ) Superclover
0 ) Default
-> |
```

Figura 7.5: Heurísticas

Em seguida é necessário escolher um dos algoritmos dos quais dispomos, sendo eles, BFS, DFS, Greedy, A Star e Dijkstra. Opcionalmente, é também possível escolher a opção de replay que será abordada mais á frente.



Após a apresentação da solução, podemos ver ainda uma animação de todos os movimentos do player do estado inicial ao objetivo e uma representação gráfica de todos os passos que o algoritmo fez para chegar á solução. (este método foi explicado no capítulo 5)

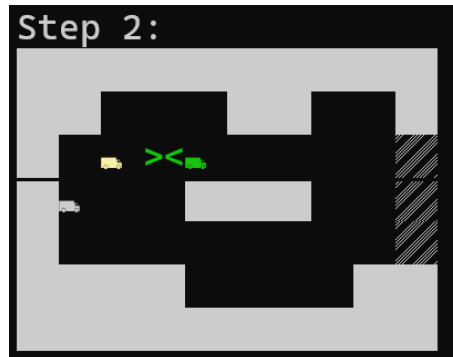


Figura 7.8: Representação dos movimentos



Figura 7.9: Representação dos steps

## 7.2 Build Graph

Na opção *Build Graph*, obtemos o tempo de construção o número de nodos e o número de arestas, para um determinado mapa e heurística.

```

Heuristic Select
1 ) Manhattan
2 ) Euclidean
3 ) Supercover
0 ) Default
-> 3
Build Time: 0.025461673736572266
Number of Nodes:345
Number of Edges:1330
Press Enter to continue...|

```

Figura 7.10: Construção do grafo

## 7.3 Build All

Na opção *Build All*, obtemos o mesmo conteúdo da opção *Build Graph*, mas agora, para todos os mapas.

```

-> 3
-----
Map: maps/01.rmap
Build Time: 0.026189088821411133
Number of Nodes:345
Number of Edges:1330
-----
Map: maps/02.rmap
Build Time: 0.9849903583526611
Number of Nodes:3918
Number of Edges:19347
-----
Map: maps/03.rmap
Build Time: 0.019711971282958984
Number of Nodes:323
Number of Edges:967
-----
Map: maps/04.rmap
Build Time: 0.23585128784179688
Number of Nodes:1364
Number of Edges:5517
-----
Map: maps/05.rmap
Build Time: 0.0191800594329834
Number of Nodes:309
Number of Edges:860

```

Figura 7.11: Construção do grafo para todos os mapas

## 7.4 Show Map

Na opção *Show Map*, é possível executar a representação gráfica de qualquer mapa.

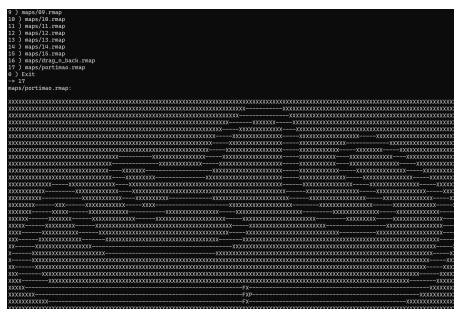


Figura 7.12: Representação gráfica de um mapa

## 7.5 Show Graph

Na opção *Show Graph*, é possível representar um grafo já em cache. Este método por vezes é custoso no requisito tempo, devido ao elevado número de nodos e arestas que um mapa contém.

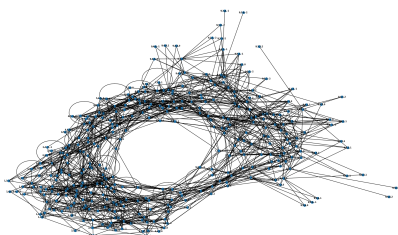


Figura 7.13: Representação gráfica do grafo do mapa 1

## 7.6 Run Tests

Por fim na opção *Run tests*, todos os mapas são executados para todos os algoritmos, mostrando os seus scores e os, respetivos, tempo de execução.



```
-> 6
-----
Map: maps/01.rmap
BFS:
Execution Time: 0.6048679351806641
Score: 4
DFS:
Execution Time: 0.07653236389160156
Score: 59
Greedy:
Execution Time: 0.04076957702636719
Score: 4
A Star:
Execution Time: 0.6799697875976562
Score: 4
Dijkstra:
Execution Time: 2.7527809143066406
Score: 4
-----
Map: maps/02.rmap
BFS:
Execution Time: 449.57828521728516
Score: 68
DFS:
Execution Time: 165.14873504638672
Score: 1741
Greedy:
Execution Time: 139.68420028686523
Score: 47
A Star:
Execution Time: 396.7578411102295
Score: 23
```

Figura 7.14: Execução de todos os algoritmos para todos os mapas

## 8 Observação de Resultados

Após colocarmos o projeto operacional, decidimos comparar os resultados obtidos.

Mapa	Teste	BFS	DFS	Greedy	A*	Dijkstra
1	T.Exec Score	1.0018 4	0.0431 59	0.0839 4	1.7325 4	4.9860 4
2	T.Exec Score	1037.3530 68	313.7462 1741	476.1452 49	843.6634 23	902.4348 23
3	T.Exec Score	4.2271 13	0.7026 203	2.2683 92	3.4167 13	2.9067 13
4	T.Exec Score	102.8938 152	34.1837 445	6.1285 156	91.6998 35	89.3738 35
5	T.Exec Score	8.7554 183	2.4812 342	6.4072 161	8.4319 43	6.7813 43
6	T.Exec Score	368.4484 22	13.816 799	120.7809 130	416.8715 22	354.4754 23
7	T.Exec Score	0.8752 31	0.0638 10	0.2837 7	0.7762 7	0.8211 7
8	T.Exec Score	0.6895 12	0.2915 172	0.1831 13	1.1010 12	1.2161 12
9	T.Exec Score	2.4380 8	0.8566 175	0.5998 9	2.6264 8	5.7022 8
10	T.Exec Score	46.9398 36	4.2009 696	0.8251 88	20.6260 12	56.2005 12
11	T.Exec Score	130.1610 7	5.2001 398	0.4322 31	99.5755 7	210.5181 7
12	T.Exec Score	242.0911 8	0.5600 88	0.2415 8	106.3249 8	373.7978 8
13	T.Exec Score	16.8919 4	0.0300 4	0.1180 4	37.9304 4	692.2821 4
14	T.Exec Score	27.3005 66	25.3245 477	10.7243 44	27.7132 19	29.6034 19
15	T.Exec Score	2001.0380 11	2093.6007 5777	0.4315 67	1527.4116 11	4215.4626 11

Tabela 8.1: Tabela de Tempos e Scores dos testes

Mapa	Teste	Valores
1	Nodos	345
	Arestas	1330
2	Nodos	3918
	Arestas	19347
3	Nodos	323
	Arestas	967
4	Nodos	1364
	Arestas	5517
5	Nodos	309
	Arestas	860
6	Nodos	2499
	Arestas	11365
7	Nodos	149
	Arestas	508
8	Nodos	121
	Arestas	330
9	Nodos	355
	Arestas	1330
10	Nodos	2646
	Arestas	8950
11	Nodos	2361
	Arestas	11877
12	Nodos	3220
	Arestas	14414
13	Nodos	9049
	Arestas	55245
14	Nodos	794
	Arestas	2433
15	Nodos	9413
	Arestas	45647

Tabela 8.2: Tabela da criação dos grafos

Tendo em conta a Tabela 8, podemos verificar que no que toca ao requisito score os algoritmos que se destacam são o de Dijkstra e o A\*, obtendo sempre os scores mais baixos e semelhantes. Relativamente ao tempo, o DFS destaca-se demonstrando que se trata de um algoritmo bastante rápido, contudo não tanto eficiente. Os algoritmos BFS e Greedy, apresentam resultados satisfatórios, sendo que existe uma variação da aproximação do score ótimo, tendo em conta o mapa em questão.

Por fim, com a Tabela 8 observamos que existe uma grande quantidade de nodos, uma vez que estes variam, tendo em conta as coordenadas e a velocidade. Sendo assim e dado o número de nodos, o número de arestas aumenta, bem como, a complexidade dos grafos.

## 9 Conclusões e Trabalho Futuro

Com a realização deste projeto podemos observar o funcionamento de várias estratégias de procura, tais como a Breadth-first, a Depth-first, o Algoritmo A\*, Greedy-search e Dijkstra, sendo capaz de identificar diferenças, vantagens e desvantagens do mesmo. Conseguimos implementar um programa capaz de interpretar vários mapas e aplicando vários algoritmos fosse possível encontrar uma soluções excelentes para o problema de minimização da pontuação do jogo.

Futuramente pretendemos implementar um ambiente dinâmico, capaz de lidar com dois ou mais jogadores e/ou obstáculos.