



Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Unidade Curricular de Inteligência Artificial

Ano Letivo de 2022/2023

Trabalho Prático - Fase 2

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

16 de Dezembro de 2022



Índice

| | | |
|----------|--|-----------|
| 1 | Introdução | 1 |
| 1.1 | Estrutura do Relatório | 1 |
| 2 | Formulação do problema como um problema de pesquisa | 2 |
| 3 | Geração de circuitos | 3 |
| 3.1 | Tipos de Mapas | 4 |
| 3.1.1 | Exemplos de Mapas Criados | 4 |
| 4 | Gerar Grafos | 5 |
| 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 |
| 5 | Implementação de Vários Jogadores | 9 |
| 5.1 | Planeamento Inicial | 9 |
| 5.2 | Melhoramento do Sistema de Expansão | 9 |
| 5.3 | Reconsideração da Estratégia | 10 |
| 5.4 | Implementação Final | 11 |
| 5.4.1 | Estratégia de Verificação de Colisões | 11 |
| 6 | Estratégia de procura | 13 |
| 6.1 | Não informadas | 13 |
| 6.1.1 | Breadth-first(BFS) | 13 |
| 6.1.2 | Depth-first(DFS) | 14 |
| 6.2 | Informadas: | 14 |
| 6.2.1 | Algoritmo A* | 14 |
| 6.2.2 | Greedy-search | 15 |
| 6.2.3 | Dijkstra | 15 |
| 7 | Representação do método de procura graficamente | 16 |
| 7.1 | Estratégia | 16 |
| 7.2 | Interações | 16 |

| | | |
|-----------|---|-----------|
| 8 | Funcionamento | 18 |
| 8.1 | Run Algos | 19 |
| 8.2 | Build Graph | 22 |
| 8.3 | Build All | 23 |
| 8.4 | Show Map | 23 |
| 8.5 | Show Graph | 23 |
| 8.6 | Run Tests | 24 |
| 9 | Observação de Resultados | 25 |
| 9.0.1 | Resultados da Execução dos Algoritmos | 26 |
| 9.0.2 | Resultados de Geração dos Grafos | 27 |
| 10 | Conclusões | 28 |

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 |
| 8.1 | Comando para execução do programa | 18 |
| 8.2 | Menu | 19 |
| 8.3 | Seleção do tipo de mapa | 19 |
| 8.4 | Mapas | 20 |
| 8.5 | Heurísticas | 20 |
| 8.6 | Algoritmos | 21 |
| 8.7 | Solução | 21 |
| 8.8 | Representação dos movimentos | 22 |
| 8.9 | Representação dos steps | 22 |
| 8.10 | Construção do grafo | 22 |
| 8.11 | Construção do grafo para todos os mapas | 23 |
| 8.12 | Representação gráfica de um mapa | 23 |
| 8.13 | Representação gráfica do grafo do mapa 1 | 24 |
| 8.14 | Execução de todos os algoritmos para todos os mapas | 24 |

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 9 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;
- **Gerar grafos:** Explicação das heurísticas utilizadas e estrutura dos grafos;
- **Representação do método de procura graficamente:** Representação dos circuitos gerados representados por grafos;
- **Estratégia de procura:** Especificação da estratégia de procura e alguns resultados dessa mesma estratégia;
- **Funcionamento:** Descrição da utilização do programa e das suas diversas funcionalidades.
- **resultados:** Análise dos resultados obtidos nos testes ao programa.
- **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 os nossos agentes serão os carros dos jogadores ($0...n$), 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 os agentes completarem o seu objetivo, chegar à meta. O estado inicial dos agentes será representado como ponto P_n enquanto as suas ações os moverão do estado inicial para outro estado até ao seu estado objetivo.

- **Estado inicial:** Ponto inicial do jogador no mapa, representado por P ;
- **Estado Objetivo:** Ponto final do jogador no mapa, representado por F ;
- **Estado:** As várias coordenadas dos jogadores em conjunto com a velocidade dos mesmos nessa coordenada;
- **Operações:** Aplicar uma aceleração sobre o estado atual *por jogador*
- **Solução:** Sequência de coordenadas que levam ao objetivo, tendo em conta o algoritmo utilizado;
- **Custo da solução:** O custo da solução é calculado segundo o número de movimentos entre o estado inicial e o estado objetivo. Estes têm o custo de 1, 10, no caso de um jogador precisar de voltar para trás para evitar uma colisão, ou 25 se existir uma colisão com a parede. O custo final é a soma de todos os custos dos jogadores.

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

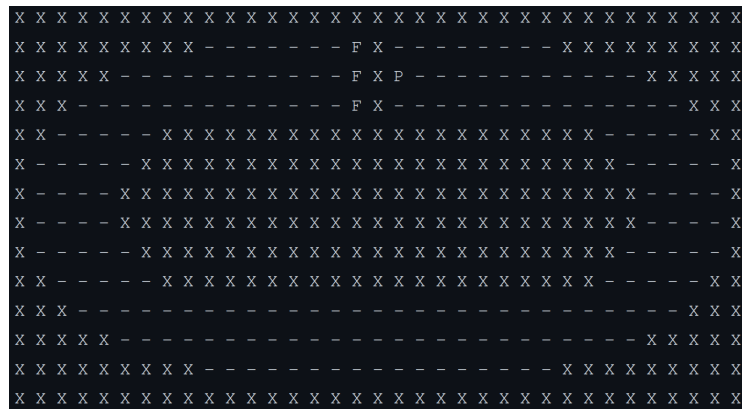


Figura 3.2: Circuito : maps/02.rmap

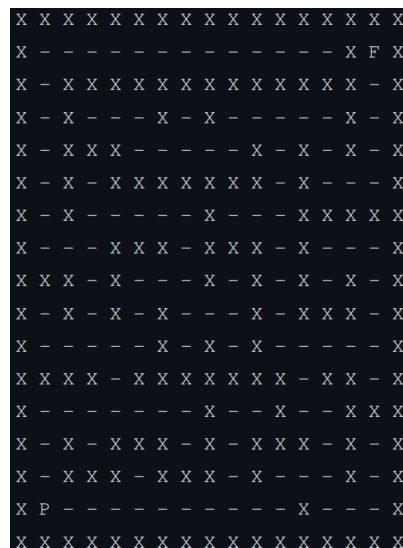


Figura 3.3: Labirinto: maps/14.rmap

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

O projeto dispõe 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, sendo 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 definida 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 pretende, não só, penalizar a paragem do jogador (uma vez que o jogo é referente a corridas), mas principalmente, melhorar a procura dos 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:

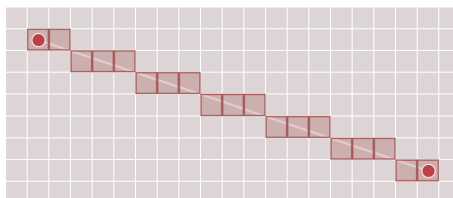


Figura 4.2: Representação da Supercover

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 os algoritmos de procura sem a necessidade de gerar uma nova expansão 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, os grafos correspondentes e as localizações iniciais do jogadores e das metas.

5 Implementação de Vários Jogadores

Na segunda fase de realização do projeto foi requisitada a implementação de vários jogadores sobre o mesmo mapa. Para tal é necessário, para além da aplicação das estratégias de procura, realizar uma verificação adicional capaz de evitar colisões entre os vários jogadores.

5.1 Planeamento Inicial

Com a introdução de vários *players* deu-se uma necessidade de uma nova maneira de geração do grafo onde as procuras seriam realizadas.

Inicialmente a estratégia definida pelo grupo deu-se pela expansão dos estado em cada nodo, i.e., cada nodo passaria a conter a informação das coordenadas de todos os jogadores, bem como as suas velocidades. Esta estratégia, apesar de levar a um grafo totalmente exato à realidade da simulação, iria necessitar de um poder de processamento enorme. Tal deve-se ao facto das expansões do grafo serem exponenciais ao número de jogadores.

Por outras palavras, um jogador tem, quando expandindo um grafo individual, um total de 9 expansões possíveis. No entanto, uma expansão de um grafo abrangente a dois jogadores necessita de expandir ambos em todas as possíveis combinações de movimentos. Sendo que cada um deles tem 9 movimentos possíveis, o total de combinações será 9×9 o que gera um total de 81 expansões. O mesmo aplica-se com um número crescente de jogadores provocando uma expansão na ordem de 9^n , sendo 'n' o número de jogadores presentes no mapa.

5.2 Melhoramento do Sistema de Expansão

Rapidamente o grupo decidiu que esta estratégia não seria viável dado os tempos de geração de cada grafo. De modo a tentar preservar a geração do grafo mais exato possível à simulação foi implementada a utilização de *threads* para a expansão do mesmo. A quando da expansão cada *thread* é responsável por adquirir um dos possíveis estados por expandir e aplicar sobre

o mesmo os possíveis movimentos. De modo a garantir um controlo de concorrência aos recursos, procuramos aplicar conhecimentos obtidos ao longo da Licenciatura, implementando um sistema de *ReadWrite Locks*. Estas 'fechaduras' impedem que várias *threads* escrevam ao mesmo tempo mas várias possam ter acesso de leitura aos recursos.

Esta mudança permitiu uma expansão bastante mais rápida dos grafos. No entanto, a complexidade dos mesmos impediu que a mudança fosse suficiente para tornar a estratégia em algo viável. Levando a uma inevitável reconsideração da estratégia para este problema.

5.3 Reconsideração da Estratégia

De volta ao início, no que toca à implementação viável de vários jogadores, surgiram duas estratégias possíveis:

- 1) Diminuir a precisão do programa;
- 2) Dividir os vários jogadores.

Na primeira opção a estratégia passaria por cada nodo apenas conter a informação relativa às coordenadas dos jogadores. Isto provocaria uma simplificação enorme do grafo, uma vez que as velocidades passariam a estar ocultas, passando toda essa responsabilidade para os algoritmos. No entanto, quer visto ao enunciado, às regras do jogo e ao conteúdo lecionado na cadeira, é da nossa opinião que não seria uma boa estratégia.

Esta causaria dois possíveis problemas: os algoritmos de procura seriam capazes de realizar jogadas ilegais, isto é, não teriam em conta a velocidade *umadasregrasdojogo*; Ou, de modo a aplicar as regras, iria ser necessária uma alteração completa às bases originais dos mesmos. I.e., um algoritmo de procura não só realizaria a procura sobre o grafo, como também necessitaria de calcular novas jogadas sempre que encontrava um nodo novo, uma vez que só assim saberia os nodos sobre os quais poderia aplicar a procura. Esta estratégia implicaria assim uma espécie de expansão extra por parte dos algoritmos de procura, o que iria implicar uma maior carga sobre os mesmos e, de certa forma, ignorar o conceito fundamental de expansão do grafo.

Dado todos estes fatores o grupo decidiu optar pela segunda estratégia. Esta tem por base dividir os diferentes jogadores nos seus próprios grafos. Esta implementação segue o funcionamento original do programa, onde cada mapa possui apenas um jogador, verificando no entanto, a cada jogada, possíveis colisões entre diferentes jogadores.

5.4 Implementação Final

Após todas as considerações e discussões de estratégias possíveis, o grupo decidiu implementar uma estratégia onde o *"workload"* é dividido pelos vários jogadores.

Para tal, a quando do início da execução, o mapa é analisado, guardando as informações do mesmo (coordenadas iniciais dos jogadores, metas, etc.). De seguida, para cada um dos jogadores é gerado um grafo onde ele é o único jogador (estratégia original). Este grafo é gerado, no entanto, com recurso a *threads* (como referido a cima), de modo a melhorar a performance para mapas maiores e/ou mais complexos, aproveitando as estratégias referidas acima.

Uma vez que todos os jogadores apresentam o seu próprio grafo podem ser aplicados os algoritmos de procura sobre os mesmos. Estes retornarão uma proposta de solução para o problema, no entanto, vários jogadores poderão ter coordenadas iguais no mesmo instante, algo que queremos evitar. Para tal é aplicada uma verificação a cada jogada de modo a evitar tais situações.

5.4.1 Estratégia de Verificação de Colisões

Após os caminhos de todos os jogadores serem gerados pelos algoritmos de procura, todos eles são comparados *'step-by-step'*, ou seja, a cada jogado o programa verifica se existe mais do que um jogador na mesma coordenada. Para isso recolhe todas as coordenadas de jogadores num dado instante e verifica iterativamente as posições dos mesmos. O primeiro jogador a encontrar-se numa coordenada onde exista, pelo menos, mais um jogador necessita de recalculer a sua rota para a meta.

No entanto, desta vez, este recebe como input para o algoritmo, uma lista de coordenadas por onde não pode passar. Estas coordenadas são assim as coordenadas atuais de todos os jogadores. A procura é, por sua vez, realizada a partir da coordenada do jogador no movimento anterior, i.e., se o jogador 1 detetar uma colisão na jogada 3, o mesmo irá voltar a realizar o algoritmo de procura, excluindo as coordenadas ocupadas, a partir da coordenada que tinha na jogada anterior. O novo caminho será, então, as jogadas até à posição anterior à deteção, seguidas das novas coordenadas indicadas pelo algoritmo de procura.

Ainda assim, recalculer a rota através da jogada anterior pode não ser o suficiente, uma vez que, o jogador pode, quer por motivos de velocidade, quer por outros jogadores ocuparem os espaços disponíveis, não conseguir encontrar um novo caminho retrocedendo uma jogada e recalculando. Caso tal aconteça existe uma penalização ao jogador. O mesmo deve retornar

até à última coordenada por onde passou, tendo, no entanto, a sua velocidade em ambos os eixos igual a zero. Isto expande a ideia de colisão original onde o jogador retorna à posição anterior com velocidade nula, necessitando, agora, de verificar se outro jogador já tomou o seu lugar e, no caso de isso se verificar, retornar, ainda mais, para trás.

Neste segundo caso o novo caminho passará a ser os movimentos até então, seguido da coordenada para onde retornou e por fim o caminho gerado pelo algoritmo de procura desde essa coordenada até à meta.

Este processo é então repetido para a mesma jogada quando uma alteração ocorre e termina quando todas as jogadas forem verificadas, todos os jogadores terminarem e não existir jogadas repetidas.

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 e a Greedy-search.

Os algoritmos de procura podem ainda receber como input uma lista de coordenadas "ocultas". Por outras palavras, coordenadas que não podem ser escolhidas no primeiro movimento da solução. Isto permite um ambiente dinâmico de procura onde obstáculos dinâmicos podem aparecer e impedir um jogador de avançar para a coordenada desejada num dado movimento. Para além destas as coordenadas podem ainda ser removidas totalmente, ou seja, coordenadas que não podem aparecer no caminho da solução final. Para estas últimas deve ser passado como input uma lista de coordenadas "removed".

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 resultado em alguns casos, uma vez que a cada colisão o resultado 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 este 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.ª 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:

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

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

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

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

Este algoritmo é dos que obtém os melhores resultados no requisito resultado, 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 $h(n) = \text{custo estimado do caminho mais curto do estado } n \text{ para o objetivo (função heurística)}$.

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 resultado 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^* , resultados bastante positivos, por outro lado este perde um pouco na eficiência, pois parte sempre do nodo com o menor custo.

7 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.

7.1 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.

7.2 Interações

Um carácter **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 carácter 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.

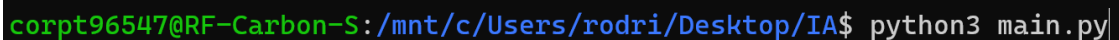
O nodo **vermelho** é a coordenada que está a ser verificada. Esta visualização dos algoritmos de procura foi desenvolvida dado a possibilidade de vários nodos estarem presentes na mesma coordenada, uma vez que no decorrer do jogo o jogador, pode alcançar, por exemplo, a coordenada (x,y) com uma velocidade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

8 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 `'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 8.1: 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 8.2: Menu

8.1 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 8.3: 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/16.rmap
0 ) Exit
-> |
```

Figura 8.4: 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 8.5: 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.

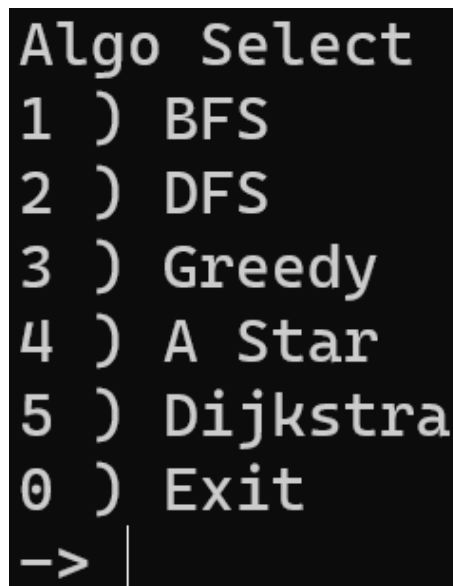


Figura 8.6: Algoritmos

Com todas as configurações selecionadas o programa apresenta então a solução do algoritmo para o mapa escolhido. Para além da representação gráfica da solução no gráfico, é mostrado também o tempo de execução em milissegundos, o resultado da solução apresentada pelo algoritmo, bem como, todos os movimentos feitos pelos players para chegar ao objetivo. (os movimentos apresentados são tendo em conta a rosa dos ventos, por exemplo, no caso de 'NE' significa North East)

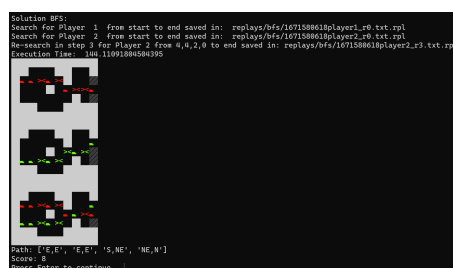


Figura 8.7: Solução

Após a apresentação da solução, podemos ver ainda uma animação de todos os movimentos dos players do estado inicial ao objetivo e uma representação gráfica de todos os passos que o algoritmo fez para chegar á solução, esta representação apenas se encontra disponível nos mapas que contêm apenas um player. (este método foi explicado no capítulo 7)

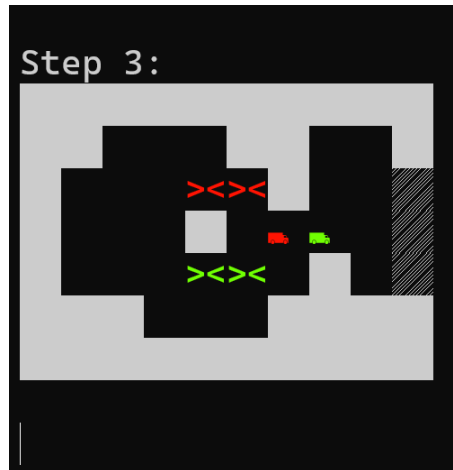


Figura 8.8: Representação dos movimentos

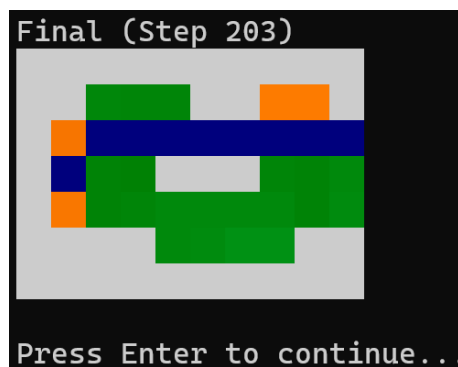


Figura 8.9: Representação dos steps

8.2 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 8.10: Construção do grafo

8.3 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 8.11: Construção do grafo para todos os mapas

8.4 Show Map

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

```
-> 2
maps/02.rmap:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXX-----FXP-----XXXXXXXX
XXXXX-----FXP-----XXXXX
XXX-----FXP-----XXX
XX-----XXXXXXXXXXXXXXXXXXXXXX-----XX
X-----XXXXXXXXXXXXXXXXXXXXXX-----X
X-----XXXXXXXXXXXXXXXXXXXXXX-----X
X-----XXXXXXXXXXXXXXXXXXXXXX-----X
X-----XXXXXXXXXXXXXXXXXXXXXX-----X
XX-----XXXXXXXXXXXXXXXXXXXXXX-----XX
XXX-----XXX
XXXXX-----XXXXX
XXXXXXXXXX-----XXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Press Enter to continue...
```

Figura 8.12: Representação gráfica de um mapa

8.5 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.

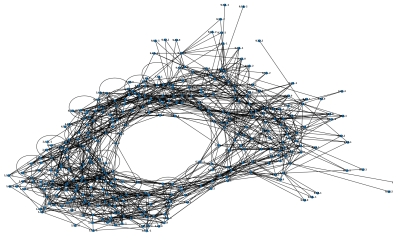


Figura 8.13: Representação gráfica do grafo do mapa 1

8.6 Run Tests

Por fim na opção *Run tests*, todos os mapas são executados para todos os algoritmos, mostrando os seus resultados 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 8.14: Execução de todos os algoritmos para todos os mapas

9 Observação de Resultados

Nesta parte final do projeto, para obtenção de resultados, foram usados os mapas anteriormente criados, contudo, os mapas 2, 4, 6, 10, 12, 13, 15 e 16 foram alterados de modo a existirem vários jogadores. Tal como já anteriormente analisado, as críticas em relação aos algoritmos mantém-se. Um destaque para os algoritmos de A* e Dijkstra em termos de resultados, tendo este ultimo apresentado ainda resultados mais surpreendentes em ambientes dinâmicos.

No que toca em comparações diretas entre um e vários jogadores são de notar os tempos de execução. Neste parâmetro o algoritmo que mais foi penalizado foi o DFS, este que, para um jogador, tende a ser consistentemente o mais rápido, num ambiente dinâmico sofre um aumento significativo no seu tempo de execução. Este aumento deve-se ao numero extenso de posições que cada uma das suas soluções apresenta, estas, num ambiente dinâmico, necessitam de ser verificadas para evitar colisões. Esta verificação leva, no caso de detetar a possibilidade de colisão, ao calculo de uma nova solução a partir desse passo e, consecutivamente, a um aumento do tempo necessário para executar uma procura com sucesso.

Apesar deste fator estar presente em todos os algoritmos, os seus efeitos escalam com o numero de passos de cada solução, bem como o tempo que cada uma demora a executar. Assim sendo, os algoritmos como o BFS ou A* apresentam um tempo de execução linearmente relacionado ao numero de jogadores, adicionando a isso o tempo de recalculer trajetórias para evitar possíveis colisões.

9.0.1 Resultados da Execução dos Algoritmos

| Mapa | Teste | BFS | DFS | Greedy | A* | Dijkstra |
|------|-----------|-----------|------------|-----------|-----------|-----------|
| 1 | T.Exec | 0.8935 | 0.0507 | 0.0524 | 1.2049 | 3.0443 |
| | Resultado | 4 | 59 | 4 | 4 | 4 |
| 2 | T.Exec | 4038.8772 | 9404.7513 | 2479.8750 | 3869.0936 | 4400.1998 |
| | Resultado | 192 | 9081 | 273 | 120 | 81 |
| 3 | T.Exec | 2.6056 | 0.5340 | 1.4219 | 2.0203 | 1.7218 |
| | Resultado | 13 | 203 | 92 | 13 | 13 |
| 4 | T.Exec | 573.5969 | 152.4543 | 45.5732 | 535.3188 | 1235.8746 |
| | Resultado | 498 | 2127 | 513 | 141 | 111 |
| 5 | T.Exec | 2.7668 | 1.0859 | 1.8129 | 2.2904 | 2.1860 |
| | Resultado | 183 | 342 | 161 | 43 | 43 |
| 6 | T.Exec | 2458.1880 | 182.2562 | 498.0914 | 1804.6824 | 1768.8224 |
| | Resultado | 120 | 3012 | 405 | 111 | 75 |
| 7 | T.Exec | 0.5986 | 0.0379 | 0.1392 | 0.4281 | 0.5030 |
| | Resultado | 31 | 10 | 7 | 7 | 7 |
| 8 | T.Exec | 0.4546 | 0.2067 | 0.0932 | 0.3879 | 0.4420 |
| | Resultado | 12 | 172 | 13 | 12 | 12 |
| 9 | T.Exec | 1.2767 | 0.5447 | 0.3409 | 1.5020 | 2.4435 |
| | Resultado | 8 | 175 | 9 | 8 | 8 |
| 10 | T.Exec | 536.0360 | 540.0195 | 5.1748 | 313.9214 | 692.6724 |
| | Resultado | 374 | 24187 | 288 | 324 | 324 |
| 11 | T.Exec | 79.3461 | 1.9471 | 0.1399 | 63.6391 | 143.3815 |
| | Resultado | 7 | 398 | 31 | 7 | 7 |
| 12 | T.Exec | 140.0561 | 10.4174 | 3.4663 | 152.3597 | 919.9829 |
| | Resultado | 69 | 1347 | 90 | 57 | 57 |
| 13 | T.Exec | 68.3636 | 3588.0901 | 5.2602 | 54.1906 | 369.3921 |
| | Resultado | 308 | 28746 | 152 | 80 | 88 |
| 14 | T.Exec | 18.3320 | 10.4622 | 6.0682 | 12.2780 | 15.2804 |
| | resRltado | 66 | 477 | 44 | 19 | 19 |
| 15 | T.Exec | 3242.1183 | 17879.6627 | 1.2519 | 1953.1815 | 6222.0852 |
| | Resultado | 54 | 23946 | 196 | 52 | 54 |
| 16 | T.Exec | 2.0887 | 0.6031 | 0.22053 | 2.2187 | 5.7518 |
| | Resultado | 8 | 179 | 38 | 8 | 8 |

Tabela 9.1: Tabela de Tempos e resultados dos testes

9.0.2 Resultados de Geração dos Grafos

| Mapa | Nodos | Arestas | Numero de Jogadores |
|------|-------|---------|---------------------|
| 1 | 345 | 1330 | 1 |
| 2 | 11754 | 58041 | 3 |
| 3 | 323 | 967 | 1 |
| 4 | 4092 | 16551 | 3 |
| 5 | 309 | 860 | 1 |
| 6 | 7509 | 34104 | 3 |
| 7 | 149 | 508 | 1 |
| 8 | 121 | 330 | 1 |
| 9 | 355 | 1330 | 1 |
| 10 | 10584 | 35800 | 4 |
| 11 | 2361 | 11877 | 1 |
| 12 | 9720 | 43950 | 3 |
| 13 | 26952 | 155464 | 8 |
| 14 | 794 | 2433 | 1 |
| 15 | 18830 | 91212 | 2 |
| 16 | 660 | 2490 | 2 |

Tabela 9.2: Tabela da criação dos grafos

Ao analisar a Tabela 9.0.2, podemos observar um aumento significativo do número de nodos e arestas. Estes ocorrem nos mapas que têm mais que um jogador e deve-se ao facto de cada *player* apresentar um grafo individual, para uma análise de resultados mais realista, todos os nodos e arestas dos grafos são somados. Esta decisão tem como objetivo demonstrar a diferença de complexidade entre mapas com vários jogadores e mapas "*single player*".

10 Conclusões

Com a realização deste projeto podemos observar o funcionamento de várias estratégias de procura, quer para ambientes estáticos, quer para ambientes dinâmicos, envolvendo vários jogadores. Fomos capazes de observar vantagens e desvantagens nos algoritmos testados. Conseguimos ainda implementar um programa capaz de interpretar vários mapas e aplicando vários algoritmos fosse possível encontrar soluções para o problema de minimização da pontuação do jogo aplicado a um ou vários jogadores em simultâneo, guardando, se desejado, *replays* de como foi realizada a procura no mapa por parte dos algoritmos.

Em conformidade com o indicado no último relatório, nesta segunda fase, cumprimos as metas que definimos para o projeto, implementando um ambiente dinâmico com vários jogadores em simultâneo. Com isto observamos as diferenças e dificuldades de garantir que não existem colisões entre jogadores e de minimizar a pontuação global do jogo.

Concluimos este projeto com grande satisfação, não só quanto ao resultado que apresentamos, mas também quanto ao progresso feito e conhecimento obtido ao longo desta Unidade Curricular.