Inteligência Artificial

CC2006

Estratégias de Procura

Catarina Monteiro - 202105279

Lara Sousa - 202109782

Mariana Serrão - 202109927

2022 / 2023

Índice

Índice de Ilustrações

Índice de Tabelas

Índice de Gráficos

1. Introdução

Um problema de busca/procura consiste em dado um estado inicial chegar ao estado final (ou conjunto de estados) desejado através de ações ditas legais.

Este tipo de problemas pode ser formalmente definido através dos seguintes elementos:

- Espaço de estados: conjunto de possíveis estados, podendo ser representado através de uma árvore ou de um grafo gerados a cada instante;
- Estado inicial: estado em que o agente começa;
- Estado final/conjunto de estados finais: estado a que o agente pretende chegar, podendo este ser um conjunto deles ou uma propriedade que se aplica a vários estados;
- Função sucessor: mapeia um estado num conjunto de novos estados;
- Ações: conjunto de movimentos/ações disponíveis para o agente executar a cada instante;
- Função de custo: função que reflete o custo de cada ação exequível;
- Cada nó é uma estrutura com pelo menos 5 componentes/campos:
    - estado
    - nó pai
    - jogada/regra aplicada para gerar o nó
    - profundidade do nó na árvore
    - custo do caminho desde o nó raiz

Para um agente resolver um problema do tipo acima referido, existem vários métodos que subdividem em duas categorias: estratégias de busca não guiadas/não informadas e estratégias de busca guiadas/informadas.

Um método de busca não guiado (MBNG) não tem qualquer termo de comparação, ou seja não consegue saber quão perto da solução está, podendo não dar a solução ideal (com o menor custo). Em contrapartida, um método de busca guiado (MBG), através de uma heurística, consegue ter a noção de qual é a melhor alternativa, assim como o quão perto está do seu objetivo.

Alguns exemplos dos referidos métodos são:

- Breadth First Search - MBNG
- Depth First Search - MBNG
- Uniform Cost - MBNG
- Iterative Deepening - MBNG
- Bidirectional Search - MBNG
- Greedy Search - MBG
- A* Algorithm - MBG

2. Descrição do problema em estudo

Ao longo deste trabalho foram realizados vários algoritmos com o objetivo de resolver o jogo dos 15.

Este jogo consiste num quadrado dividido em dezasseis partes iguais movíveis (peças), sendo que quinze delas estão numeradas de 1 a 15 (ou imagens diferentes em cada peça). Através de movimentos legais (mover a peça em branco para cima, baixo, esquerda ou direita sem nunca deixar de ter a forma de um quadrado num todo) passar de uma dada disposição inicial (estado inicial) para um objetivo determinado à partida (estado final).

É de referir que no total existem 16! = 2.092279e+13 estados iniciais, no entanto apenas metade podem ser consideradas solucionáveis para um certo estado final.

3. Estratégias de Procura

3.1. Procura não guiada

3.1.1. Procura em Profundidade (DFS)

A pesquisa em profundidade (DFS) é um dos algoritmos utilizados na pesquisa de uma estrutura de dados em árvore ou grafo. O algoritmo começa no nó raiz (topo) de uma árvore e vai até onde pode, num determinado ramo. Depois retrocede até encontrar um percurso que não tenha sido visitado e, então, explora-o. Este processo é realizado até que toda a estrutura tenha sido visitada.

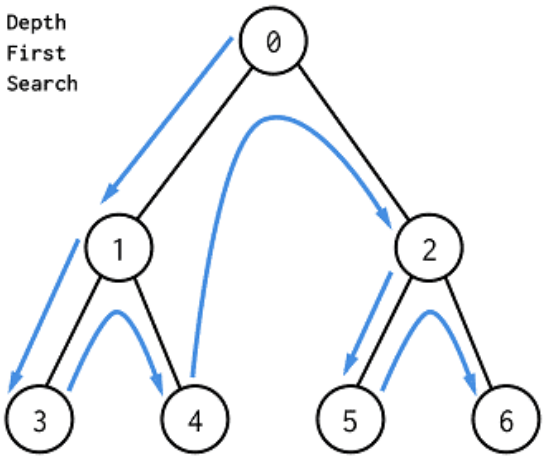

Figura 1 - Algoritmo DFS

As propriedades da procura em profundidade dependem vigorosamente do contexto em que são aplicadas, grafos ou árvores. A versão de pesquisa em grafo, que evita estados repetidos e percursos redundantes, é completa em espaços de estado finito, uma vez que eventualmente irá expandir todos os nós. Por outro lado, na versão de pesquisa em árvore, não está garantido que a pesquisa fique completa, o que pode originar que o algoritmo fique preso num loop. Este tipo de procura em árvore pode ser modificado sem custo extra de memória, de forma a que se compare novos estados com aqueles no percurso da raiz até ao nó atual. Esta abordagem evita loops infinitos em espaços de estado finito, mas não evita a propagação de percursos redundantes. Em espaços de estado infinito, ambas as versões falham quando um percurso infinito que não é o objetivo é encontrado.

Pesquisas em profundidade são frequentemente utilizadas como recurso em algoritmos mais complexos. Por exemplo, o algoritmo de correspondência Hopcroft-Karp, utiliza a procura DFS como parte de seu algoritmo para ajudar a encontrar uma correspondência num grafo. DFS é também utilizado em algoritmos de pesquisas em árvores, que têm aplicações em problemas como o do caixeiro viajante e em algoritmos, tais como o de Ford-Fulkerson.

A complexidade temporal deste algoritmo é O(b^d), sendo b o branching factor e d a profundidade da árvore. Já a complexidade espacial, depende apenas da profundidade: O(d).

3.1.2. Procura em Largura (BFS)

O algoritmo de busca em largura ou BFS é o método mais amplamente utilizado, tipicamente em árvores e grafos, com o objetivo de encontrar um elemento/estado concreto.

Neste método de busca inicia-se a pesquisa pelo nó raiz (0) e analisam-se os seus filhos no sentido esquerda->direita (1->2). De seguida, passa para a profundidade seguinte (3->4->5->6) até chegar aos nós folha.

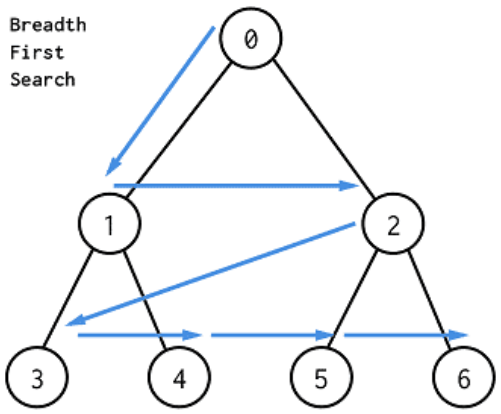

Figura 2 - Algoritmo BFS

No entanto, aquando da utilização de grafos, ao contrário de árvores de pesquisa, estes podem conter ciclos sendo necessário categorizar os nós em “visitados” e “não visitados” de modo a evitar que um nó seja analisado várias vezes, sem necessidade.

A complexidade temporal deste método de pesquisa é O(b^d), sendo b o branching factor e d a profundidade da árvore, assim como a complexidade espacial

3.1.3. Procura Iterativa Limitada em Profundidade (IDFS)

O método de procura iterativa limitada em profundidade pode ser considerado uma junção dos métodos DFS e BFS, uma vez que combina a eficiência espacial e a rapidez de pesquisa destes respetivamente.

Neste método de pesquisa não informada, atribuo um limite de profundidade, começando em zero e uso a estratégia de pesquisa em profundidade em cada limite para visitar os nós e procurar a solução. Se encontrar a solução paro, senão tento um novo limite de profundidade (1,2,3,4…), mas de cada vez que tentar um novo limite não mantendo espaço na memória para guardar os nós já visitados.

Tal como vemos na figura, tentamos para um limite 0, como não encontramos a solução, tentamos para o limite 1 e assim sucessivamente, até ao limite de profundidade 3.

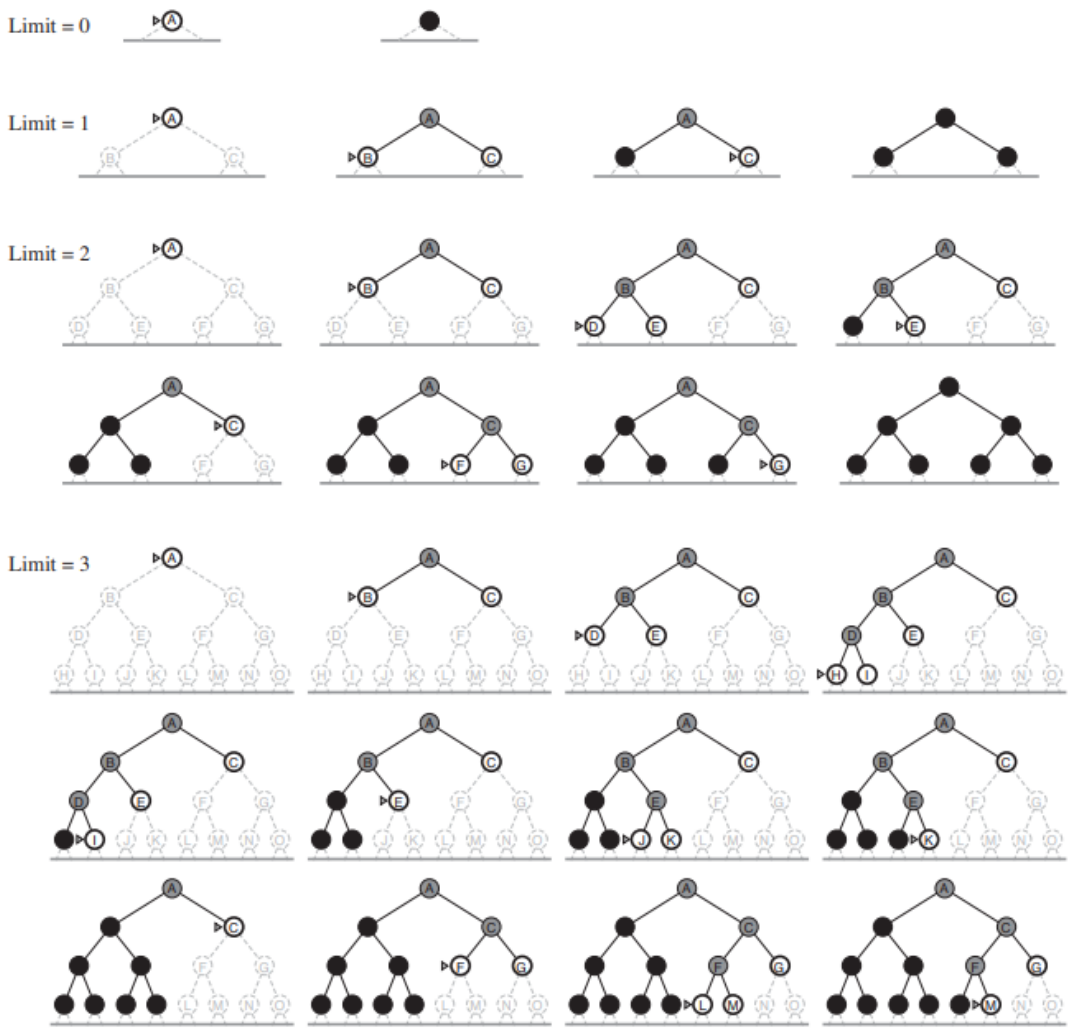

Figura 3 - Algoritmo IDFS

Sendo b o branching factor e d a profundidade da árvore podemos afirmar que a complexidade temporal deste método é b^d e a complexidade espacial é O(d). Este é, portanto, o melhor método de procura não informado.

3.2. Procura guiada

Uma heurística, h(n), surge como uma componente adicional à função de avaliação, f(n), onde n é o estado onde nos encontramos, e estima o caminho com um menor custo necessário para chegar até ao goal (estado final).

Por exemplo, se A é o estado inicial e B o estado final, posso seguir pelo estado C ou pelo estado D. A minha heurística vai ajudar também a responder à pergunta "Qual o melhor estado para chegar ao goal?". Até ao momento, as pesquisas não informadas como BFS e DFS não seriam capazes de responder a uma questão deste estatuto, apenas interpretavam como um estado e somente isso. Mas com a heurística, é possível estimar qual a melhor solução, entre escolher C ou D e assim sucessivamente, para chegar ao goal, B.

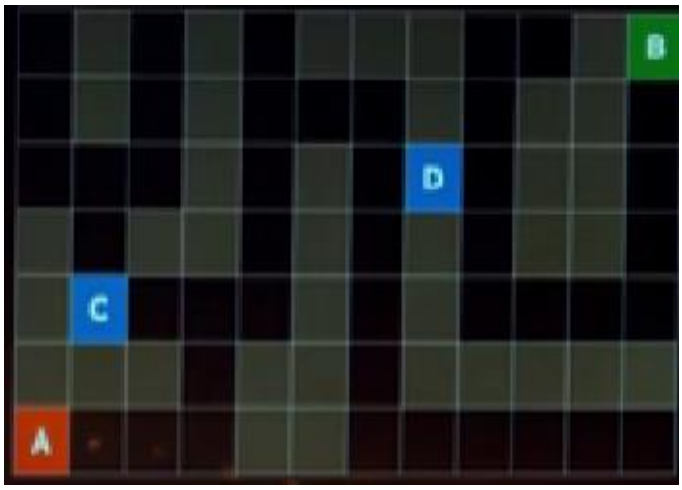

Figura 4 - Procura Guiada

Implementou-se duas heurísticas. A primeira heurística é responsável pela determinação do número de peças do puzzle deslocadas, peças cuja sua posição no puzzle inicial difere da sua posição no puzzle goal. Se todas as peças forem deslocadas, então o valor da heurística inicialmente é 15. Devido às diferenças encontradas nas posições das peças, é intuitivo que todas as peças terão de se mover, pelo menos uma vez, o que titula esta primeira heurística como admissível. A segunda heurística é responsável pelo cálculo da distância de Manhattan, isto é, a distância desde a posição atual até ao goal.~

3.2.1. Procura Gulosa

O método de procura greedy é considerado um algoritmo simples e intuitivo tipicamente utilizado em problemas de otimização. Ao contrário de outros algoritmos, este realiza a "escolha ótima" a cada instante sem ter em consideração estados anteriores nem estados futuros.

Pelo motivo acima referido, o greedy não admite backtrack, ou seja, não existe a possibilidade de voltar para um estado anterior de forma a tomar outra escolha que não a efetuada. Por esta e outras razões, este método de busca não produz a solução ótima, por exemplo no "problema do troco" , introduzido na cadeira de Desenho e Análise de Algoritmos. Este consiste em perfazer uma certa quantidade monetária em moedas tentando utilizar o menor número de moedas. Vejamos os seguintes exemplos:

- Quantidade desejada – 40
- Conjunto de moedas A = {1,2,10,20}
- Solução apresentada pelo algoritmo: 20+20 (2 moedas)
- Conjunto de moedas B = {1,2,10,20,25}
- Solução apresentada pelo algoritmo: 25+10+2+2+1(5 moedas)

Confirmamos, então, o acima referido.

O greedy avalia os nós usando apenas a função heurística, f(n) = h(n). Pelo que, neste trabalho foram utilizadas as duas heurísticas, referidas no subcapítulo anterior.

Vejamos o seguinte exemplo, onde A é o estado inicial e B o estado final (goal):

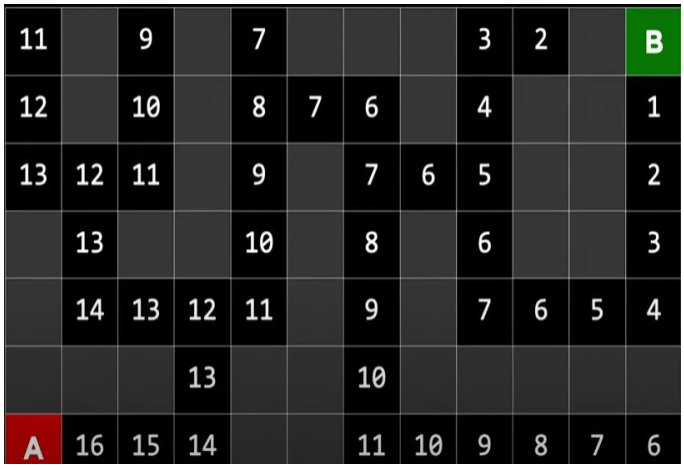

Figura 5 – Distância de Manhattan em Greedy

De acordo com a distância de Manhattan, ignora-se as fronteiras e calcula-se a distância para cada estado até ao goal, tal como podemos ver na figura acima. O pensamento do Greedy para obter o melhor caminho até ao goal seria iniciar em A e seguir até ter um momento de decisão. Neste caso, o primeiro momento de decisão seria entre escolher um estado que se encontra a uma distância de 11 passos de B e um outro estado que se encontra a uma distância de 13 passos de B, intuitivamente este algoritmo escolherá aquele cuja distância é menor, 11.

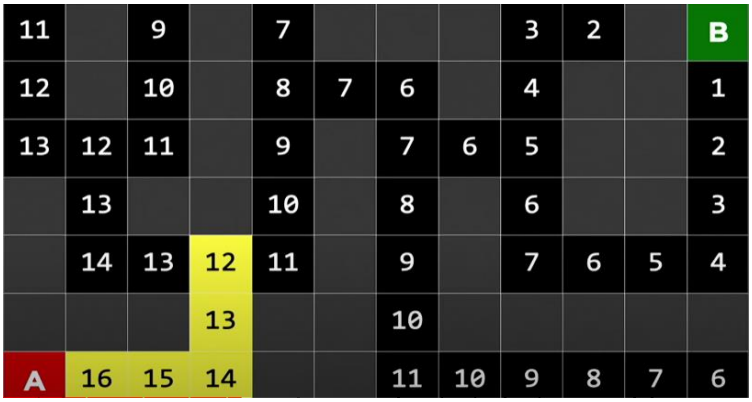

Figura 6 – Primeiro Momento de Decisão

Um segundo momento de decisão seria o representado a seguir:

Figura 7 -Segundo Momento de Decisão

Aqui ambos os estados, seguindo para cima ou para a direita, apresentam o mesmo valor de heurística e isto prova que, muitas das vezes, este algoritmo tem de escolher de forma random. E assim sucessivamente, escolhendo sempre aquele cuja distância de Manhattan é menor, o Greedy estima o caminho a seguir representado como o melhor para alcançar o goal.

Figura 8 - Comportamento do Greedy

Mas será esta a solução ótima? A resposta é não, o que iremos ver em 3.2.2..

Como anteriormente dito, existe a opção de utilizar outra heurística denominada de misplaced(). O funcionamento desta é um pouco diferente uma vez que a avaliação é feita através do número de peças "fora do sítio", ou seja, o número de peças que não se encontra na posição desejada (configuração final). À semelhança da heurística anterior, este algoritmo de pesquisa toma a melhor decisão momentânea, isto é, sem ter em conta qualquer estado anterior nem futuro.

3.2.2. Procura A*

O A* encontra sempre a solução ótima, isto é, estima sempre o caminho que implica um custo menor até ao goal. Esse custo corresponde ao somatório da distância de Manhattan, h(n) com a distância do estado inicial até ao estado em que nos encontramos, g(n), por isso, é um algoritmo muito mais completo.

Custo, f(n) = h(n) + g(n).

Vejamos o exemplo, com base na seguinte figura que usa a heurística referente à distância de Manhattan:

Figura 9 - Comportamento do A*

Neste caso onde A é o estado inicial e B o final, o A* não procuraria apenas aquele que tivesse um menor custo, pois este algoritmo, através do somatório equivalente ao custo (referido anteriormente), apercebe-se que o melhor é considerar recuar, pois há um outro caminho ainda disponível cujo custo não é só 13, mas sim 19 (13 + 6), que é inferior a 21(14+5). Logo, efetuaria backtrack e encontraria a solução ótima e, portanto, o melhor caminho para B, ao contrário do Greedy.

Relativamente à heurística misplaced, segue a mesma ordem de ideias referida em 3.2.1., no entanto, neste método de busca existe a possibilidade de efetuar bakctrack aquando de uma "decisão mal tomada". Também este algoritmo tem a particularidade de seguir certos caminhos tendo em conta os possíveis estados em que ficará assim como os em que já esteve.

4. Descrição da Implementação

Neste trabalho foi utilizada a linguagem python devido à maior familiaridade e exposição ao longo do curso, até ao momento. Através desta escolha foi nos possível utilizar MinHeaps como estrutura de dados assim como a não necessidade de definir limites quanto a listas, entre outras características desta linguagem. Para além disso, esta proporcionou uma ótima conduta para a representação da estrutura de dados escolhida, matrizes. É de salientar a sua rapidez de execução, eficiência e acessibilidade.

Para representar o tabuleiro do jogo dos 15 usou-se matrizes, de forma a que os dados de input, inseridos pelo utilizador, fossem colocados em matrizes quadradas de tamanho 4; uma para conter os valores iniciais do jogo, *m_inicial*, e outra correspondente à solução, à qual se pretende chegar, *m_final*.

O comando a efetuar, como explicado no ficheiro de apoio (readme file), inclui à partida o método de busca a utilizar ('BFS', 'DFS', 'IDFS', 'A*-misplaced', 'A*-Manhattan', 'Greedy-misplaced', 'Greedy-Manhattan') assim como o as configurações dos tabuleiros de jogo através do módulo argparse, que permitirá a invocação do código com o formato pedido (code <strategy> <config). Esta ligação é feita através do ficheiro *Estrategias.py* importada no início do ficheiro *Principal.py*. No entanto, nem todos os inputs dados pelo utilizador são coerentes no contexto de jogo, por isso, através da igualdade, ou não, da paridade da diferença das posições do espaço vazio (número 0) e da mesma quanto às trocas necessárias das peças em posições erradas, verificou-se se em quais casos se conseguiu, partindo da configuração inicial, chegar à configuração final. Para isso, criou-se um código que implementa esse algoritmo. Caso não seja possível chegar à configuração final, é impressa a mensagem "Não é possível chegar à configuração pretendida".

Quanto à possibilidade de chegar à configuração dita “standard” (de 1 a 15 ordenado na horizontal) implementou-se também um algoritmo semelhante ao anteriormente descrito que verifica se é possível chegar à configuração referida. Caso a resposta seja positiva é impressa no terminal a seguinte mensagem: "É possível chegar à configuração standard", caso contrário aparecerá "Não é possível chegar à configuração standard".

Como referido, optou-se pela utilização de uma classe MinhHeap com vários elementos: dados, tamanho, comparador, e o estado final, representados por *self.dados*, *self.tamanho*, *self.comparador*, *self.estado_final*. Implementou-se funções como *__len__(self), __contains__(self, item), __str__(self), compara(self, x, y), obtem_pos(self, elem),* responsáveis, respetivamente por: retornar o tamanho; verificar se um item está contido em dados; converter em string (de forma a que seja facilmente percetível); verificar se o jogo já terminou; e obter a posição de um dado elemento x. Adicionou-se funções para efetuar os movimentos e a inserção, remoção ou troca de valores, na MinHeap: *up(self), down(self), swap(self), pop(self) e push(self)*. Quanto à eficiência desta estrutura de dados concluiu-se que cumpre o propósito atribuído de forma rápida e eficiente, não tendo tido qualquer problema quanto a esta.

Foi criada também a class *Puzzle* que é inerente aos movimentos que serão feitos no jogo, à leitura dos valores provenientes de um ficheiro e respetiva disposição num tabuleiro. Primeiramente, criou-se o construtor da *classe Puzzle*, onde se inicializou os membros dessa classe: o estado, os filhos, o pai (recuar), a profundidade e uma função que encontra as coordenadas de um ponto representados, respetivamente, como *self.estado*, *self.filhos*, *self.recuar*, *self.profundidade*, *self.encontra__coord()*. Implementou-se também inúmeras funções: *__hash__(self)* coloca ‘ ‘ entre cada valor recebido no input e retorna o seu valor em hash; *__copy__(self)* guarda o estado do tabuleiro (fazendo uma cópia deste) assim como auxilia na representação do caminho que a estratégia de busca está a percorrer (impresso no terminal); *__str__(self)* responsável por imprimir dois tabuleiros portadores dos valores dados no input; *__eq__(self, other)* verifica se os atributos são iguais ou não, *encontra_coord(self)* e *coord(self)* que encontra as coordenadas do ponto e retorna o referido.

Para se transformar uma configuração de jogo na sua sucessora, implementou-se quatro funções que se destinam a movimentar as peças no tabuleiro. De forma a facilitar a movimentação destas, a peça a ser movimentada será a vazia, ou seja, a que contém o número zero. Assim estas funções movem o zero para a esquerda, para a direita, para cima e para baixo (*zero_esquerda(), zero_direita(), zero_cima(), zero_baixo(), respetivamente*), agregados numa só função denominada de *movimentos(self)*. Em cada uma delas é aplicada uma função, *mov_legal()*, que verifica se é possível deslocar a peça na direção pedida, ou seja, se a peça fica dentro dos limites do tabuleiro. Nos casos em que não é possível, a função retorna *False*. Para se descobrir a posição do zero criou-se

uma função *pos_zero()*, que retorna um tuplo com as coordenadas (linha, coluna) da peça vazia. Foi feita, também, a innclusão dos membros da classe e a importação do copy, uma funcionalidade do python de fulcral importância para guardar o estado do tabuleiro da jogada anterior.

No ficheiro Estrategias.py, fora da classe MinHeap, criou-se as funções das heurísticas, a distância de Manhattan para cada valor deslocado (fora da posição) e o contador das peças deslocadas, *manhattan(estado_inicial, estado_final) e misplaced(estado_inicial, estado_final).* Com as heurísticas desenvolvidas implementou-se então as estratégias de busca *bfs(estado_inicial, estado_final)*, *dfs(estado_inicial, estado_final, profundidade)*, *idfs(estado_inicial, estado_final, profundidade)*, *astar(estado_inicial, estado_final, heuristica)* e *greedy(estado_inicial, estado_final, heuristica)*, onde "profundidade" é a profundidade máxima até à qual se irá tentar encontrar a solução (introduzida pelo utilizador no input) e "heuristica" é um parâmetro que recebe uma das duas funções, manhattan ou misplaced, dependendo do caso.

Em todas as funções criou-se uma variável (*total_nos*), cujo objetivo é indicar o número de nós utilizados pela estratégia. Tanto a Greedy como a A* utilizam uma das heurísticas escolhidas durante a sua execução (distância de Manhattan e o número de peças fora do seu lugar pretendido). No caso das pesquisas de profundidade (DFS e IDFS), apenas são iteradas até se atingir a profundidade máxima indicada pelo utilizador no input.

Em todos os métodos de busca se consegue ver, no terminal, os dois tabuleiros indicativos dos estados contidos no seu input e se é possível chegar à configuração standard ("1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0"). De seguida aparecerá o nome da pesquisa selecionada assim como o tempo de execução, os nós gerados e armazenados e a percentagem do CPU e de memória RAM utilizados. Finalmente conseguirá visualizar as trocas efetuadas, ou seja, 'Cima' indica que o espaço em branco foi trocado com a peça acima, assim como os estados dos tabuleiros que levaram à configuração desejada.

5. Resultados

Após a análise de todas as estratégias para uma configuração inicial “1 2 3 4 5 6 8 12 13 9 0 7 14 11 10 15”, com o objetivo de chegar à configuração final “1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0”, organizaram-se os resultados numa tabela onde é indicado o tempo de execução, a percentagem de CPU utilizada, a percentagem de Memória RAM gasta, o resultado do problema (se a solução foi ou não encontrada) e o total de nós percorridos.

Estes testes foram realizados num computador portátil Lenovo, com uma RAM de 8,00 GB , um sistema operativo de 64 bits e um processador baseado em x64. Para a DFS e IDFS foram utilizadas uma profundidade limite 12, uma vez que é a primeira profundidade na qual se consegue encontrar uma solução. Uma vez que os valores vão variando, estas tratam-se apenas de aproximações.

Tabela 1 - Comparação de Resultados

Estratégia	Tempo (segundos)	Espaço CPU (%)	Espaço RAM (%)	Encontrou a solução?	Profundidade/Custo (nós)
DFS	3,7	9,4	60,1	Sim	24190
BFS	12,1	11,8	11,8	Sim	78259
IDFS	127,6	11,1	77,9	Sim	1216966
Greedy Manhattan	0.01	0	59,7	Sim	57
Greedy Misplaced	0.01	12,5	61,9	Sim	135
A* Manhattan	0,01	0	62,5	Sim	57
A* Misplaced	0.01	0	62,2	Sim	135

Com o intuito de se conseguir perceber a variação do tempo de execução no DFS nos diferentes tipos de profundidade, foi elaborado um gráfico que contém os valores do tempo gasto tendo em conta a profundidade máxima limitada.

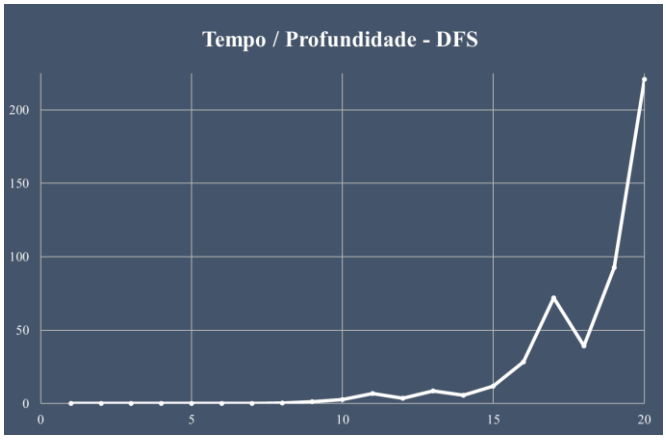

Gráfico 1 - Tempo/Profundidade (DFS)

6. Comentários Finais e Conclusões

Com a realização deste trabalho e, em especial, por análise do tópico 5, pode-se concluir aspetos comparativos entre as diversas estratégias de procura, pelo seu desempenho e eficácia a encontrar soluções.

Primeiramente, centralizando-nos nas estratégias de procura não informada, é notório que o DFS teve um melhor desempenho que o BFS e o IDFS, pois obteve a solução em menor tempo de execução, utilizando menos memória e percorrendo um menor número de nós. No entanto, ainda que o DFS tenha um melhor desempenho, todos foram eficazes na procura da solução.

Segundamente, em relação às estratégias de procura informada, tanto o Greedy como o A* tiveram o mesmo custo de profundidade e tempo de execução, nas duas heurísticas, distinguindo-se apenas no espaço de memória utilizado, no qual o Greedy obteve valores ligeiramente mais baixos. Ambos foram eficazes, pois encontraram a solução e mantiveram um bom desempenho, destacando-se das estratégias de procura não informada, por serem algoritmos mais rápidos e simultaneamente percorrerem menos nós, tal como esperado, visto que, utilizam heurísticas na sua implementação. Neste caso, não se verificou grandes diferenças, mas em diferentes configurações dadas como input, testou-se que o A* tem no geral um melhor desempenho. No entanto, para este problema, tanto o Greedy como o A* seriam os mais indicados, desde que a heurística manhattan fosse a eleita, visto que, apresenta um menor custo de profundidade do que a heurística misplaced, tal como esperado.

7. Referências Bibliográficas

https://www.freelancinggig.com/blog/2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms/ —---> imagens 1 e 2, dos gráficos

Estratégias não informadas de Procura/Busca, slide 2, Moodle, Inteligência Artificial

Artificial Intelligence: Foundations of Computational Agents Book by Alan Mackworth and David Lynton Poole, secção 3.1 - 3.1.1, pag. 83, secção 3.4.3, pag. 86, secção 3.5., pg 92, secção 3.5., pg 93, secção 3.6., pg 103, secções 3.4, 3.5, pgg. 94,102

https://personal.math.ubc.ca/~cass/courses/m308-02b/projects/grant/fifteen.html

Permutations, Interchages and Parity

https://brilliant.org/wiki/depth-first-search-dfs/

https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/