

Implementação do Jogo dos 15

FCUP

Inteligência Artificial (CC2006) 2018/2019

Trabalho de Grupo AK

Eduardo Morgado – 201706894 – MIERSI

Simão Cardoso – 201604595 – MIERSI

Sónia Rocha – 201704679 – MIERSI

1. Introdução:

1.1. Um problema de Pesquisa:

Podemos considerar um problema de pesquisa como sendo uma sequência de ações que permitem a chegada a um estado objetivo/final a partir de um estado inicial [5]. Cada uma destas ações irá alterar um estado, o conjunto de todos os estados possíveis e os operadores que relacionam os estados entre si formam o espaço de pesquisa [5,6].

Sendo assim um problema de pesquisa corretamente formulado pode ser descrito por um estado inicial do problema (ponto de partida), um conjunto de operadores que levam à criação de novos estados a partir de um dado estado, um espaço de estados acessíveis a partir do inicial por uma sequência qualquer de ações (aplicações de operadores), um caminho (uma sequência de ações) associado a um custo, dependendo este do custo individual de cada ação e por um teste de verificação da chegada ao estado objetivo/final [1,5].

1.2. Métodos de Busca:

De entre os vários métodos de pesquisa, os mais usados entram nas categorias de pesquisa não informada/força bruta e pesquisa informada/pesquisa com heurística¹.

1.2.1. Pesquisa Não Informada

Uma pesquisa não informada é uma pesquisa geral, mais fraca, uma vez que não pressupõe propriedades sobre o problema a tratar. De entre os métodos de pesquisa não informada encontramos:

- Pesquisa em Largura (BFS) – Análise exaustiva onde, a partir de um nó origem/raiz, todos os nós são gerados por ordem crescente de distância para a raiz [3]. O primeiro caminho para o nó objetivo será o de menor comprimento, normalmente, a sua implementação envolve a utilização de uma *queue*. Tem como complexidades temporais/espaciais $O(b^d)$;

¹ Uma heurística são regras com possibilidade de solucionar um problema, no entanto, não garantem a existência de uma solução. A partir do conhecimento de propriedades do problema, uma heurística facilita a pesquisa da solução [3].

- Pesquisa em Profundidade (DFS) – A partir de um nó raiz, apenas explora o próximo filho depois de acabar a exploração do primeiro, ou seja, explora um caminho até encontrar o nó objetivo ou ser forçado a regressar, normalmente, pode ser implementado por métodos, iterativos, utilizando *stacks*, ou por métodos recursivos, utilizando a *stack* do sistema. Apresenta complexidades temporal e espacial, respetivamente, $O(b^d)$ e $O(b * d)$ [1];
- Pesquisa Limitada em Profundidade (LDFS) – Método suportado por DFS, no entanto, é estipulado *a priori* um limite para a profundidade do caminho a ser percorrido. Apresenta complexidades temporal e espacial, respetivamente, $O(b^l)$ e $O(b * l)$, onde l é o limite de profundidade [1];
- Pesquisa em Profundidade Limitada (IDFS) – Combina vantagens de BFS com DFS, é suportada por uma LDFS onde vamos aumentando o limite fixo em LDFS, garantindo assim o teste de todos os possíveis limites de profundidade, reduzindo o espaço utilizado. Apresenta complexidades temporal e espacial, respetivamente, $O(b^d)$ e $O(b * d)$ [1];

1.2.2. Pesquisa Informada

Uma pesquisa informada recorre a funções heurísticas para estimar o quão próxima da solução um nó está, essa estimativa será utilizada para conduzir melhor a pesquisa, tornando-a muito mais eficiente que uma pesquisa não informada.

Este método utiliza uma função $f(n)$ para avaliar um nó, retornando um valor que pode servir como incentivador para a escolha do nó em questão [3]. De entre os vários métodos de pesquisa informada encontramos:

- Best-First Search – Método que expande o nó com o melhor valor para função de busca utilizada, não garante encontrar um caminho ótimo do estado inicial para o estado objetivo [3]. A partir deste método, podem-se gerar outros métodos de pesquisa (BFS/A*/Greedy), dependendo de como o valor da função de busca é calculado: apenas o custo entre nós(BFS), apenas o valor heurístico do nó para a solução (Greedy) ou uma junção dos dois (A*) [3]. A complexidade temporal/espacial deste método varia consoante a função de pesquisa utilizada, podendo ter complexidades iguais a BFS, A* ou Greedy. ;
- Pesquisa A* – Método baseado em Best-First Search onde a função de pesquisa, $f(n)$, é a soma ($f(n)=g(n)+h(n)$) entre o custo entre dois nós, $g(n)$, e a estimativa do custo do estado atual para o estado objetivo, uma função heurística, $h(n)$. Este método garante sempre encontrar a solução ótima caso a escolha da função heurística, $h(n)$, nunca sobrestime o custo do nó atual para o nó objetivo. Nenhum outro algoritmo garante expandir menos nós que o A*. Apresenta complexidades temporal e espacial de $O(b^d)$;

- Pesquisa *Greedy* – Método baseado em Best-First Search onde a função de pesquisa, $f(n)$, é apenas o valor heurístico do nó atual para o nó objetivo. Este algoritmo irá sempre escolher os nós com menor valor, mas não garante uma solução ótima. Apresenta complexidades temporal e espacial de $O(b^d)$;

2. Estratégias de Procura:

Nesta secção iremos ver com mais pormenor os métodos de pesquisa que foram implementados no trabalho prático.

2.1. Pesquisa Não Informada:

Já vimos anteriormente que, a pesquisa não informada é uma pesquisa mais exaustiva, como tal, menos eficiente quando comparada com métodos informados.

2.1.1. Pesquisa em Largura (BFS)

Este método é inicializado na raiz da árvore de pesquisa e apenas explora nós de nível seguinte após explorar todos os nós do nível atual. É um método completo (garante, caso exista, encontrar solução), tem possibilidade de ser ótimo (a solução encontrada é a primeira, com o menor número de ações), caso os operadores para o problema tenham todos o mesmo custo uniforme. Apesar de ser ótimo e completo, requer muita memória e tempo [1], uma vez que, é necessário armazenar todos os nós, apresentando uma complexidade temporal e espacial exponencial, sendo a quantidade de memória disponível para resolver o problema o maior limitador deste método [1], como tal, não é recomendado ser utilizado para problemas com instâncias muito grandes de estrada.

É usual implementar este método com uma *queue* FIFO, onde, quando visitamos um nó, expandimo-lo e adicionamos os seus nós ainda não visitados, marcando-os como visitados, ao final da fila. A *Figura 1* apresenta o pseudocódigo para este método.

```
BFS(inicial,final):
1   Q <- MK_QUEUE(inicial)
2   Enquanto Q_IS_EMPTY() != False fazer
3       no <- DEQUEUE(Q)
4       Se no == final então retorna Sucesso
5       Para cada no_filho fazer
6           Se no_filho não visitado então
7               visita_no_filho
8               ENQUEUE(no_filho,Q)
```

Figura 1-Pseudocódigo BFS

Este método tem como complexidade espacial e temporal $O(b^d)$, uma vez que, para cada nível d , terá que guardar todos os estados gerados por todas as operações válidas dos nós de

profundidade $d-1$. Uma vez que, apresenta altas complexidades (exponenciais), não deve ser aplicado a problemas de grandes instâncias.

2.1.2. Pesquisa em Profundidade (DFS)

À semelhança da pesquisa em largura, a pesquisa em profundidade (DFS) é uma estratégia genérica de pesquisa exaustiva. Quando aplicada a partir de um nó origem s , começa por explorar um ramo enquanto não haja *backtrack*² [3]. A Figura 2 apresenta um esquema do funcionamento do método de pesquisa DFS.

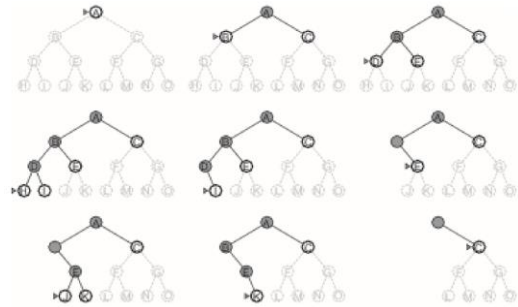


Figura 2-Esquema de DFS (Imagem retirada dos slides de IA, Uninformed Search, slide 16)

Este método de pesquisa é relativamente simples, gastando muito menos tempo que o método de pesquisa anterior. Este método não garante ser completo, uma vez que, pode expandir um caminho que nunca levará a uma solução, sem limitadores (de tempo, profundidade ou memória) poderá nunca acabar, é um método que, dependendo da complexidade da árvore de pesquisa, gasta pouca memória, uma vez que apenas precisa de guardar o caminho a ser percorrido juntamente com os nós ainda não expandidos, no entanto não é ótimo, uma vez que, como expande apenas um caminho de cada vez, poderá encontrar uma solução a uma profundidade muito maior que a solução ótima. Se o problema tiver várias soluções (vários caminhos de chegada a um nó objetivo) pode ser mais rápido que o BFS, pois apresenta uma maior probabilidade de encontrar uma solução, com um pequeno espaço de busca explorado/percorrido. No entanto, não é um método com muita eficiência temporal, a sua complexidade temporal depende da complexidade/profundidade da árvore de pesquisa, caso esta seja muito extensa ou mesmo infinita, corremos o risco de nunca encontrar uma solução. A sua complexidade temporal também depende muito da ordem em que os operadores são aplicados a um nó [1], ao definir uma certa ordem de geração de nós, podemos estar a deslocar soluções para um extremo oposto da árvore.

É usual utilizar como estrutura de dados de armazenamento de nós para este método uma *stack*, podendo essa ser a *stack* do sistema (para DFS recursivo) ou uma (LIFO) implementada na função (para DFS iterativo). A Figura 3 apresenta o pseudocódigo iterativo para este método.

```
DFS(inicial,final):
1   S <- MK_STACK(inicial)
2   Enquanto S_IS_EMPTY() != False fazer
3       no <- POP(S)
4       Se no == final então retorna Sucesso
5       Para cada no_filho fazer
6           Se no_filho não visitado então
7               visita_no_filho
8               PUSH(no_filho,S)
```

Figura 3-Pseudocódigo DFS

² A partir de um algoritmo de pesquisa de soluções para um dado problema, cada vez que, na lista de candidatos para a solução por explorar, seja impossível de o explorar um dado nó candidato, devido a limitadores de tempo/memória/profundidade na árvore de pesquisa, *backtracking* é terminar a busca nesse candidato, não o expandindo, continuando a busca noutro candidato anterior, sendo esse próximo candidato escolhido tendo em conta o algoritmo de busca implementado [7].

As complexidades temporais e espaciais para este método são, respetivamente, $O(b^d)$ e $O(b * d)$ [1]. Resumindo, este método reduz o uso de memória, tem a capacidade de encontrar rapidamente uma solução em certas condições, não garantindo ser ótimo nem completo, no entanto, não deve ser utilizado com árvores de elevada profundidade, pois pode nunca vir a encontrar uma solução ou mesmo acabar.

2.1.3. Pesquisa Iterativa Limitada em Profundidade(IDFS)

A pesquisa iterativa limitada em profundidade, parte da necessidade de criar métodos de pesquisa mais eficientes tanto em tempo como uso de memória. Anteriormente vimos os métodos BFS e DFS, para o BFS, este era completo e ótimo mas tinha como o maior limitador, o consumo de memória, para o DFS este minimizava o consumo de memória, sendo assim, o IDFS é uma junção entre os dois métodos de pesquisa.

O algoritmo funciona aplicando um DFS limitado em profundidade, LDFS³, onde vamos consecutivamente, em cada iteração do algoritmo, incrementando a profundidade limite para o DFS, testando assim, todos os possíveis níveis de profundidade. Sendo assim, uma vez que, junta as vantagens dos métodos BFS e DFS, é um método completo e ótimo, uma vez que, apenas iremos expandir árvores de profundidade superior à limite caso não seja encontrada a solução. Apresenta complexidades temporais e espaciais, respetivamente, $O(b^l)$ e $O(b * l)$, onde b é o nível de ramificação e l , o limite de profundidade [3]. Devido a estas características este método é ideal para quando, em pesquisas exaustivas/não informadas, a árvore de pesquisa é muito grande e é desconhecida a profundidade da solução.

A Figura 4 apresenta o pseudocódigo para este algoritmo.

```
IDFS(inicial,final,limite):
1   Para i<-0 até i==limite fazer
2       S <- MK_STACK(inicial)
3       Enquanto S_IS_EMPTY() != False fazer
4           no <- POP(S)
5           Se no == final então retorna Sucesso
6           Se PROFUNDIDADE(no) == i então retorna
7           Para cada no_filho fazer
8               Se no_filho não visitado então
9                   visita_no_filho
```

Figura 4-Pseudocódigo IDFS

³ Para um LDFS, este faz a mesma pesquisa que um DFS, no entanto, é imposto *a priori* um limite de profundidade, ou seja, se, durante a exploração de um dado caminho, este atinge o limite de profundidade sem produzir solução, a busca para e avança para outro caminho. Uma nota importante para este método é a sua completude. Caso o limite seja d e a solução para o problema se encontrar a uma profundidade $d+k$, o método não é capaz de produzir solução. É um método que não é ótimo pois estamos a aplicar um DFS.

2.2. Pesquisa Informada:

Uma pesquisa informada, utiliza conhecimento de propriedades do problema para encontrar uma solução, ao contrário dos métodos não informados/exaustivos que apenas utilizam inteligência quando inserem um nó na função de enfileiramento, os métodos informados, utilizam funções para descrever com que prioridade os nós devem ser expandidos.

2.2.1. O que é uma Heurística?

Como referido anteriormente, uma heurística é uma função/método que quantifica o quão próximo estamos do nosso objetivo. Podem existir várias heurísticas, no entanto, para o problema que iremos apresentar em 3, iremos considerar 2, a distância Manhattan e a distância Hamming [2].

A distância Manhattan é a distância entre dois pontos, neste caso, entre a posição correta e a posição errada, para dois pontos $p1(x1,y1)$ e $p2(x2,y2)$, a distância Manhattan é $|x1-x2|+|y1-y2|$, enquanto que a distância Hamming é o número de peças fora do lugar. A Figura 5 apresenta um esquema das duas distâncias aplicado ao puzzle dos 8.

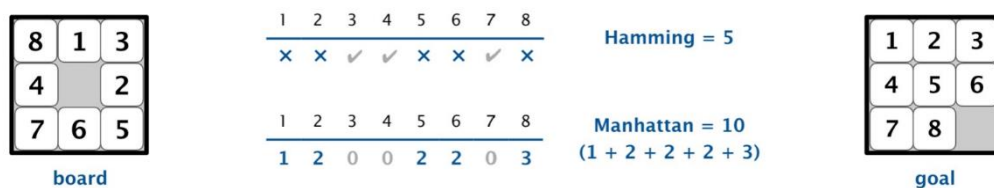


Figura 5-Distância Manhattan vs Hamming (imagem retirada de [2])

2.2.2. Busca Greedy

A Busca Gulosa (Greedy), vai sempre na mesma direção num caminho da árvore para procurar a solução, tentando minimizar o custo estimado para chegar a esta, pode mudar de direção dependendo do custo dos nós ainda não explorados da árvore de procura. Na presença de *dead-ends* pode ter que escolher o caminho de maior custo. Esta pesquisa não é ótima e é incompleta (não verifica nós repetidos no caminho, embora na nossa implementação haja essa verificação).

As heurísticas escolhidas para este método foi a distância de Hamming e de Manhattan. A Figura 6 apresenta o pseudocódigo para este algoritmo.

```

Greedy(inicial,final,limite):
1      PQ <- MIN_MK_PQUEUE()
2      ADD(initial,h,PQ)
3      Enquanto PQ_IS_EMPTY() != False fazer
4          no <- POLL(PQ)
5          Se no == final então retorna Sucesso
6          Para cada no_filho fazer
7              Se no_filho não visitado então
8                  visita_no_filho
9                  ADD(no_filho,h,PQ)

```

Figura 6-Pseudocódigo Greedy

2.2.3. Busca A*

Esta técnica requer que a estimação do custo restante no próximo nó não seja nunca maior que custo restante do nó anterior. Sob esta hipótese, sempre é possível encontrar a solução ótima com a busca A*.

A busca A* degenera para a busca em amplitude quando a heurística não aponta preferência por nenhum caminho ao longo da busca. Esta pesquisa é completa somente para grafos com fator de ramificação finito.

Nenhum outro algoritmo ótimo garante expandir menos nós do que o A*. A complexidade espacial: número de nós expandidos para chegar a um estado final cresce exponencialmente com o tamanho da entrada.

A *Figura 7* apresenta o pseudocódigo deste algoritmo.

```
A*(inicial,final,limite):
1      PQ <- MIN_MK_PQUEUE()
2      ADD(initial,g+h,PQ)
3      Enquanto PQ_IS_EMPTY() != False fazer
4          no <- POLL(PQ)
5          Se no == final então retorna Sucesso
6          Para cada no_filho fazer
7              Se no_filho não visitado então
8                  visita_no_filho
9                  ADD(no_filho,g+h,PQ)
```

*Figura 7-Pseudocódigo A**

3. O problema dos 15 (15 Puzzle):

O problema com que fomos apresentados foi o jogo dos 15, um problema representado por uma matriz 4x4 onde há 15 células numeradas e uma célula em branco. Variações deste jogo podem conter parte de uma imagem em cada célula. O problema consiste em partir de uma configuração inicial embaralhada das células e chegar a uma configuração final com uma ordenação determinada de algarismos. Os movimentos/operadores possíveis para se chegar de uma configuração a outra são: 1) mover a célula em branco para cima, 2) mover a célula em branco para baixo, 3) mover a célula em branco para a direita e 4) mover a célula em branco para a esquerda. A *Figura 8* apresenta exemplos de puzzles.

1	2	3	4
5	6	8	12
13	9		7
14	11	10	15

1	2	3	4
13	6	8	12
5	9		7
14	11	10	15

Figura 8-Exemplos de Configurações para Jogo dos 15 (imagem retirada do enunciado do problema)

3.1. A Questão de ser uma Configuração solúvel:

Para alguns conjuntos de configurações iniciais e finais, este problema, não tem solução, como tal é necessário verificar a solvabilidade do puzzle.

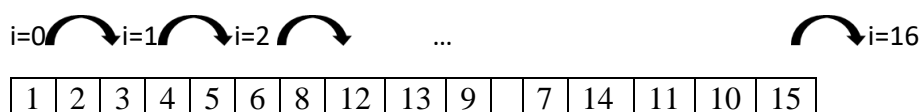
Primeiramente, é necessário representar a tabela em formato de linha [4], a *Figura 9* exemplifica este processo.

1	2	3	4
5	6	8	12
13	9		7
14	11	10	15

1	2	3	4	5	6	8	12	13	9		7	14	11	10	15
---	---	---	---	---	---	---	----	----	---	--	---	----	----	----	----

Figura 9-Tabela convertida em fila única

De seguida deverá ser calculado o número de inversões⁴ de cada célula, sendo depois somados para obter a **inversão total** da configuração atual. A *Figura 10* apresenta o cálculos de inversões.



Iteração i	Inversão a calcular= valor	Sequência de células
0	Inv(1) = 0	
1	Inv(2) = 0	
2	Inv(3) = 0	
3	Inv(4) = 0	
4	Inv(5) = 0	
5	Inv(6) = 0	
6	Inv(8) = 1	{7}
7	Inv(12) = 4	{9,7,11,10}
8	Inv(13) = 4	{9,7,11,10}
9	Inv(9) = 1	{7}
10	Blank (ignore)	
11	Inv(7) = 0	
12	Inv(14) = 2	{11,10}
13	Inv(11) = 1	{10}
14	Inv(10) = 0	
15	Inv(15) = 0	
Total Inversões		13

Figura 10-Cálculos de inversões

Posteriormente, tendo já calculado a inversão total, sendo esta 13, para a configuração atual, encontramos-nos em condições de aplicar a fórmula. Sendo este problema representado por uma tabela NxN, onde N é **par**, basta aplicar a segunda parte da fórmula, esta fórmula foi retirada de [4]:

- Se sistema tem N **ímpar**:
 - Sistema **tem solução** se inversão total for **par**;
- Se sistema tem N **par**:

⁴ O número de inversões de uma dada célula é o número de células não vazias na fila após a atual com valor inferior à atual [4]. O estado *standart*, representado na *Figura 12*, não tem inversões [4].

- Se célula vazia estiver em **linhas pares** a contar de baixo, o sistema **tem solução** se inversão total for **ímpar**;
- Se célula vazia estiver em **linhas ímpares** a contar de baixo, o sistema **tem solução** se inversão total for **par**;

A *Figura 11* apresenta um esquema da fórmula.

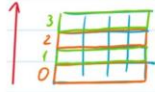
Tabela NxM		
M é Ímpar	M é Par	
	Posição de Blank Par por baixo	Posição de Blank Ímpar por baixo
Se soma de inversões Par , sistema é solucionável	Se soma de inversões Ímpar , sistema é solucionável	Se soma de inversões par , sistema é solucionável
Posição Blank por baixo	Número de linhas a contar por baixo 	

Figura 11-Fórmula para teste de solvabilidade do puzzle

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figura 12-Estado standart do puzzle(imagem retirada do enunciado do problema)

Este teste permite verificar se, uma dada configuração, esta consegue chegar ao estado *standart*, *Figura 12*. Para o nosso problema este teste é importante, uma vez que, permite verificar a qual **espaço de estados**⁵ as nossas configurações iniciais e objetivo pertencem. Para o problema são nos dadas duas configurações, uma como o ponto de partida da busca e outra como o ponto de chegada, no entanto, o ponto de chegada poderá não ser o estado *standart*, ou seja, para duas quaisquer configurações *a* e *b*, se *a* chega a *standart* (se passar no teste de solvabilidade) e *b* chega a *standart* então *a* chega a *b*, se *a* não chega a *standart* e *b* não chega a *standart* então *a* chega a *b*. Esta condição é passada aos nós gerados de *a* e *b* por aplicação dos operadores (cima/baixo/esquerda/direita), pelo que basta verificar apenas uma vez, antes de aplicar o algoritmo de busca.

⁵ Para este problema, estamos a considerar como espaço de estados, chegar a *standart* ou não chegar a *standart*

4. Descrição de Implementação:

Para esta implementação de resolução do problema utilizamos como linguagem o Java, dos fatores que levaram a esta escolha, estão a experiência anterior com esta linguagem, o facto de ser uma linguagem de programação orientada a objetos, facilitando na abstração do problema e da representação da tabela, bem como a existência de várias bibliotecas com estruturas de dados que utilizamos, facilitando assim a implementação do problema. Quando comparada a linguagens como C, a nossa escolha apresenta mais vantagens, no entanto, uma outra linguagem que poderia ter sido utilizada, seria Python, acabando por não ser escolhida por falta de experiência de alguns membros do grupo com a linguagem.

4.1. Estruturas de Dados Escolhidas

Nesta implementação, utilizamos 4 estruturas de dados implementadas em bibliotecas do java para enfileirar configurações de tabelas, verificar se tabelas já foram vistas e para as organizar por prioridades de expansão (para A* e Greedy) essas estruturas foram Queue, HashMap, LinkedList e PriorityQueue. É importante referir que para algumas destas estruturas, utilizamos como objetos, classes criadas por nós, a sua implementação não veio aumentar as complexidades temporais das estruturas, iremos ver mais à frente quais e onde foram estas classes utilizadas.

Para a implementação da Queue, esta foi escolhida, apenas para a implementação de BFS, uma vez que tem uma estrutura FIFO para armazenar-mos os dados. Em termos de complexidades, para inserir/remover a queue apresenta uma complexidade $O(n)$ uma vez que para cada inserção/remoção terá que atualizar as posições de todos os seus dados.

Para o método LDFS e IDFS poderíamos ter utilizado uma Stack do java, no entanto, utilizamos a *stack* do sistema (recursivamente).

A LinkedList foi utilizada para armazenar os nós descendentes de um nó produzidos por aplicação de operadores a um nó (para esta implementação, consideramos um nó como sendo uma configuração da tabela). Como, apenas a utilizamos para inserir/remover nós (não há necessidade de consultar a lista) e, como na inserção, estamos a adicionar um elemento ao topo da lista (`addFirst`) e na remoção, a retirar a cabeça da lista (`removeFirst()`), ambas estas complexidades são $O(1)$ o que não influencia a complexidade (temporal) do programa, como a lista só é gerada quando chamada, não aumenta a complexidade dos programas.

Para a utilização da PriorityQueue, utilizada tanto no A* como no Greedy, esta tem complexidade de inserção/remoção $O(n)$, podendo ser reduzida para $O(\log(n))$ se utilizarmos uma HashTable para guardar a posição dos elementos na PriorityQueue (na nossa implementação não utilizamos esta opção).

Em qualquer algoritmo, é necessário guardar se uma dada configuração já foi obtida, para isso utilizamos um HashMap, sendo aqui onde todas as verificações de *contains* são feitas, a sua complexidade espacial é $O(n)$ e temporal para verificar se um elemento pertence à hash, para

inserir/remover um elemento da hash é $O(1)$, na secção a seguir explicamos o uso/implementação desta estrutura de dados.

4.2. Estrutura do código

4.2.1. Uma Classe Table

Independentemente do método de pesquisa que irá ser utilizado, terá sempre que existir uma estrutura de dados geral que represente a tabela, uma classe **Table** igual para todas as implementações. A classe deverá ter capacidades de comparar, manipular e gerar outras tabelas(descendentes). Deverá ter um construtor capaz de converter os dados fornecidos (uma matriz) em valores de classe. A *Figura 13* apresenta a interface da classe.

```
class Table{ //classe para armazenar uma configuração da tabela
    String str_table; //string com a configuração da tabela
    int table[]; //matrix representando a tabela
    int row,col; //valores das posições, linha coluna, da célula vazia na tabela
    int last_move; //movimento para gerar esta configuração (1->Up/-1->Down/-2->Left/2->Right)

    Table(int[][]table,int last_move); //construtor da tabela a partir de matriz
    Table(int[]table,int last_moves,String s); //construtor de tabela a partir de array

    boolean isValid(int r,int c); //retorna true caso valores estejam dentro da tabela
    int getBlankRow(); //retorna a linha da célula vazia
    int getBlanckCol(); //retorna a coluna da célula vazia
    int getLastMove(); //retorna o operador utilizado para gerar esta tabela
    String getString(); //retorna a string da configuração da tabela
    int[][] getTable(); //retorna uma matriz cópia da tabela
    LinkedList<Table> getDescendents(); //retorna a lista das tabelas produzidas a partir desta
}
```

Figura 13-Classe Table

4.2.2. Uma Classe PriorityNodes

Para a implementação do método A* e Greedy, é necessário utilizar uma PriorityQueue, no entanto, esta apenas toma um objeto, uma solução seria introduzir um valor inteiro para a distância de heurística de cada tabela, no entanto, como apenas é necessário para este método, decidimos implementar uma classe PriorityNodes, a *Figura 14* apresenta a interface da classe.

```
class PriorityNodes implements Comparable<PriorityNodes>{
    //classe para guardar nós na PriorityQueue
    Table t; //armazena a configuração para uma tabela
    int heuristic_dist; //armazena a distância heurística para a tabela
    int steps; //armazena o número de passos para chegar à configuração t
    String moves; //string com a sequência de movimentos para chegar a t

    PriorityNodes(Table t,int heuristic_dist,int steps,int moves); //construtor

    Table getTable(); //retorna a tabela t
    int getDist(); //retorna distância heurística
    int getSteps(); //retorna o número de passos para chegar a t
    String getMoves(); //retorna a string da sequência de movimentos para chegar a t
    int compareTo(PriorityNodes pn); //implementação do comparador a ser utilizado em PriorityQueue
}
```

Figura 14-Classe PriorityNodes

4.2.3. Uma Classe HashNode

Para todos os métodos de pesquisa, utilizamos uma HashMap para guardar nós visitados, uma HashMap guarda valores V para chaves K, como tal, consideramos uma chave uma String, sendo a configuração de uma tabela, e como valores V, queremos guardar a sequência de movimentos para a chegada à tabela de chave K, juntamente com o seu número de passos. Inicialmente, guardávamos um array String[] com dois valores, o primeiro, a sequência de movimentos e o segundo o número de movimentos, no entanto, o programa estava a perder tempo quando criava um novo nó, pois tinha que ir buscar à hash o número de movimentos do pai, pai[1], converter para inteiro (Integer.parseInt(pai[1])), depois incrementar e converter de novo para string, adicionando à hash do filho. Para simplificar este processo (a conversão demorava muito tempo) criámos uma classe HashNode que contém os dois valores. A *Figura 15* apresenta a interface da classe.

```
class HashNode{//classe para guardar valor de table na HashMap
    int steps; //armazena o número de passos para chegar à configuração t guardada na HashMap
    String moves; //string com a sequência de movimentos para chegar a t guardada na HashMap

    HashNode(String moves,int steps);//construtor

    int getSteps(); //retorna o número de passos para chegar a t
    String getMoves(); //retorna a string da sequência de movimentos para chegar a t
}
```

Figura 15-Classe HashNode

4.2.4. O Programa Puzzle

A *Figura 16* descreve os métodos mais importantes do programa de resolução do problema proposto, a descrição não é uma descrição completa, para aceder ao código e ver a implementação completa poderá consultar a página do GitHub da equipa, <https://github.com/thejoblessducks/the-15-that-was-the-puzzle.git> ou <https://github.com/eamorgado/ProjetoIA-15Puzzle.git> .

```

public class Puzzle{
    static int size; //variável para guardar tamanho de HashMap em DFS e IDFS
    static int[][] final_state=new int[4][4]; //guarda configuração objetivo
    static long start_time,end_time; //variáveis para contar tempo de execução
    static String solution_str; //string com configuração objetivo

    static boolean isGoal(String table); //retorna true se a string introduzida for igual à objetivo
    static boolean isSolvable(int[]table,int row); //dada uma tabela em array, e a linha da célula
    vazia, retorna true se chega a standart (pela fórmula)
    static String putPath(String s,int p); //dada seq de movimentos s e último movimento p, cria uma
    nova sequência

    static int distHamming(Table tb); //calcula a distância Hamming para uma dada tabela
    static int distManhattan(Table tb); // calcula a distância Manhattan para dada tabela

    static void tableBFS(Table tb); //aplica BFS a tabela
    static void dfsVisit(Table t, HashMap<String,HashNode> h,int limit, String s, HashNode hn);
//método recursivo de DFS
    static void tableDFS(Table tb,int limit); //aplica LDFS a tabela
    static void tableIDFS(Tablet b); //aplica IDFS a tabela, iterando limit em DFS

    static void tableAHamming(Table tb); //aplica A* a table com heurística Hamming
    static void tableAManhattan(Table tb); //aplica A* a table com heurística Manhattan
    static void tableGreedyHamming/ Table tb); // aplica Greedy a table com Hamming
    static void tableGreedyManhattan(Table tb); // aplica Greedy a table com Manhattan
}

```

Figura 16-Implementação geral do programa Puzzle

5. Resultados:

A seguir, são apresentados os resultados de vários testes para o programa desenvolvido. É de referir que, para o IDFS, utilizamos como limite máximo de iteração 80, uma vez que qualquer puzzle com solução se resolve em no máximo 80 movimentos [8]. No entanto, perante vários testes, deparámo-nos com uma complicação no DFS/LDSF, onde ele nunca mais saía de um caminho e o programa eventualmente ficaria sem memória. Esta complicação poderia ser resolvida, se, ao gerar os filhos de um nó, estes fossem gerados aleatoriamente (o código tem essa opção em comentário), no entanto, viria a danificar os outros métodos, como tal, para os próximos testes, utilizamos como profundidade limite para LDFS com 20 (quando utilizado sem chamar IDFS). Para cada método, corremos o programa 5 vezes para obter um valor médio do tempo. Assumimos como espaço os nós guardados ao mesmo tempo na HashMap.

- Para configuração 1

Estratégia	Tempo(segundos)	Espaço	Encontra?	Custo
BFS	0.3500829	27676	S	12
LDFS(20)	1.3964016	21	S	20
IDFS	0.2554804	13	S	12
A* Hamming	0.0700111	70	S	12
A* Manhattan	0.0502611	28	S	12
Greedy Hamming	0.0266049	19	S	12
Greedy Manhattan	0.0246417	17	S	12

Inicial:

1	2	3	4
5	6	8	12
13	9	0	7
14	11	10	15

Final:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

- Para configuração 2

Estratégia	Tempo(segundos)	Espaço	Encontra?	Custo
BFS	0.2252488	3784	S	13
LDFS(20)	0.6350354	21	S	20
IDFS	0.2211609	13	S	13
A* Hamming	0.0268596	46	S	13
A* Manhattan	0.0204986	19	S	13
Greedy Hamming	0.2127478	19	S	13
Greedy Manhattan	0.0223825	17	S	13

Inicial:

0	9	12	7
14	5	13	2
6	1	4	8
10	15	3	11

Final:

9	5	12	7
14	13	0	8
1	3	2	4
6	10	15	11

- Para configuração 3

Estratégia	Tempo(segundos)	Espaço	Encontra?	Custo
BFS	0.1393498	9567	S	10
LDFS(20)	0.9263232	21	S	20
IDFS	0.1018146	11	S	10
A* Hamming	0.0182352	12	S	10
A* Manhattan	0.0102835	11	S	10
Greedy Hamming	0.0190581	11	S	10
Greedy Manhattan	0.0265747	11	S	10

Inicial:

6	12	0	9
14	2	5	11
7	8	4	13
3	10	1	15

Final:

14	6	12	9
7	2	5	11
8	4	13	15
3	0	10	1

- Para configuração 4

Estratégia	Tempo(segundos)	Espaço	Encontra?	Custo
BFS	0.2430367	45102	S	12
LDFS(20)	0.6463010	20	S	20
IDFS	0.1957684	13	S	12
A* Hamming	0.0307578	16	S	12
A* Manhattan	0.0237424	12	S	12
Greedy Hamming	0.0622905	505	S	52
Greedy Manhattan	0.0218456 0	25	S	14

Inicial:

13	11	15	4
8	9	1	5
12	14	0	2
7	10	3	6

Final:

14	11	4	5
8	0	14	15
12	1	3	2
7	9	10	6

6. Comentários Finais

Uma das primeiras observações que verificamos, é a complexidade exponencial do método BFS, de todos, é aquele que expande o maior número de nós. É importante referir que, em todos os casos, as profundidades das soluções ótimas não ultrapassam o valor de 20, como têm uma profundidade relativamente pequena, todos os métodos são completos (embora, caso o limite de LDFS seja alterado de 20 para 80, este deixa de ser completo, e corre o risco de não acabar).

Conforme previsto, o método DFS utiliza uma quantidade muito menor de espaço que o método BFS, no entanto é o método que apresenta uma maior complexidade temporal, não sendo um método ótimo.

Verifica-se pela experiência que tanto o IDFS como o A* produzem soluções ótimas, embora o A* tenha uma menor complexidade temporal. A complexidade espacial de IDFS é relativamente menor quando comparada com o A*, no entanto, isto poderá ter a ver com a forma como consideramos o espaço (o tamanho máximo da HasHMap, para o DFS e IDFS é natural ter tamanho $l+1$, uma vez que, em cada iteração apenas tem guardado os nós de um caminho, como o caminho só pode ter no máximo l nós, o seu tamanho será $l+1$).

Prova-se pela configuração 4 que o método Greedy não leva a uma solução ótima e que o método A* é o método que garante o menor número de nós expandidos em menor tempo para chegar à solução ótima, é de notar que em configurações anteriores isto não seria tão aparente.

Finalmente, não podemos deixar de notar a clara diferença temporal e espacial entre as implementações com heurísticas Hamming e Manhattan, por observação, concluímos que uma implementação com heurística Manhattan irá levar à chega de um resultado, mais rapidamente e com menor número de nós expandidos. Isto acontece devido ao facto de, para uma heurística ser admissível, esta nunca poderá sobrestimar o valor real de uma distância, e por experiência a Hamming revela sobrestimar esse valor, ou seja, a heurística Hamming apenas considera o facto de uma peça estar fora do seu lugar ou não, não considerando o quão distante essa peça está na

realidade da sua posição correta. Podemos ter uma célula à distância 1 da sua posição correta e outra à distância 7, por exemplo, da sua posição correta e ambas são tratadas/valorizadas da mesma forma, ora isto cria uma grande gama de nós com prioridades iguais, o que leva à necessidade de expandir mais nós e por sua vez, gastar mais tempo. Já a heurística Manhattan, tem em conta essa informação, ou seja, irá “tornar mais difícil” de encontrar nós com valores iguais, sendo assim mais precisa.

Sendo assim, após analisar os dados, podemos concluir que, a melhor estratégia para este problema, foi a implementação A*, uma vez que, produz uma solução ótima com o menor espaço ocupado e na menor quantidade de tempo. Entre as heurísticas, podemos concluir que a de Manhattan, por ter uma maior precisão de prioridade, é a melhor heurística a utilizar para resolver o problemas dos 15.

Nota Adicional: Este relatório, bem como o programa utilizado para obtenção dos dados, pode ser acedido no repositório da equipa, <https://github.com/thejoblessducks/the-15-that-was-the-puzzle.git>, ou em <https://github.com/eamorgado/ProjetoIA-15Puzzle.git>

7. Referências:

[1] École Polytechnique Montréal 6 de dezembro de 2000, [Michel Gagnon](#), consultado pela última vez a 4 de março de 2019, <http://www.professeurs.polymtl.ca/michel.gagnon/Disiplinas/Bac/IA/ResolProb/resproblema.html>

[2] Princeton University Department of Computer Science n.d., *8 Puzzle*, consultado pela última vez a 6 de março de 2019, <http://www.cs.princeton.edu/courses/archive/spr18/cos226/assignments/8puzzle/index.html>

[3] Stanford University Computer Science n.d., *Search Methods*, [Bruce Donald](#) and [Patrick Doyle](#), consultado pela última vez a 4 de março de 2019, <https://users.cs.duke.edu/~brd/Teaching/Previous/AI/Lectures/Summaries/search.html#Brute-Force/Blind Search Methods>

[4] The University of Birmingham School of Computer Science 26 de novembro de 2014, *Software Workshop Java Solvability of the Tiles Game*, [MarkRyan](#), consultado pela última vez a 3 de março de 2019, <https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html>

[5] University of Cambridge Department of Engineering Multimedia Group n.d., *Problem Solving by Search*, Rod Goodyer, consultado pela última vez a 4 de março de 2019, <http://www.g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/nonflash/problemframenf.htm>

[6] University of Cambridge Department of Engineering Multimedia Group n.d., *What is Search?*, Rod Goodyer, consultado pela última vez a 4 de março de 2019, <http://www.g.eng.cam.ac.uk/mmg/teaching/artificialintelligence/nonflash/problemframenf.htm>

[7] Wikipedia 7 de dezembro de 2018, *Backtracking*, consultado pela última vez a 6 de março de 2019, <https://en.wikipedia.org/wiki/Backtracking>

[8] A. Br ünger, A. Marzetta, K. Fukuda, and J. Nievergelt, “The Parrallel Search Bench ZRAM and Its Applications”, *Annals of Operations Research*, vol. 90, pp. 45-63, 1999