15 PUZZLE

Francisco Ribeiro, Matheus Bissacot, Sérgio Coelho

Universidade do Porto

1 Introdução

1.1 O que é um problema de procura?

Um problema de procura é constituído por um estado inicial e um estado final e uma função que mapeia "todos" os estados que um objeto terá de passar para que a partir desse estado inicial chegue ao estado objetivo.
Ou seja, um problema de busca visa encontrar a partir do estado inicial e das funções sucessor um conjunto de estados no qual seja possível chegar no estado final.

1.2 Quais são os métodos utilizados para resolver problemas de procura?

Um algoritmo que resolve esse tipo de problema é chamado de Algoritmo de Busca. As propriedades essenciais para esse tipo de algoritmo são a completude, ou seja, um programa deve retornar uma solução para qualquer dado input quando existir uma solução para o input, a otimização, pois o programa deve achar as melhores soluções com o menor custo de caminhos e ter uma complexidade espacial e temporal reduzida para garantir que qualquer máquina consiga executar o algoritmo.
Existem dois métodos principais para se resolver um problema de busca, estes são:

- **Busca não informada (ou busca cega)**

  Na busca não informada, não temos qualquer tipo de informação auxiliar que permita o algoritmo encontrar qual é o melhor caminho para

se chegar no estado final. Exemplos de pesquisas cegas são o DFS (do inglês Depth First Search, ou seja, a Busca em Profundidade) e o BFS (do inglês Breadth First Search, ou seja, a Busca em Largura)

- **Busca informada**

  Na busca informada, temos heurísticas que calculam o custo que cada caminho tem. Exemplos de algoritmos de busca informada são o Greedy e o A*. Iremos abordar este tópico com mais profundidade no 3º tópico do relatório.

Definido o que é um problema de busca, agora é necessário vermos qual tipo de problema de busca vamos abordar neste trabalho. Vamos descrever então o que é o Jogo do 15.

2 Descrição

O jogo do 15 é um jogo com 16 casas, dos quais 15 estão ocupados com peças numeradas de 1 a 15 e um espaço branco por norma é representado por 0. O objetivo do jogo é mover a peça vazia de forma que se chegue a uma configuração final desejada.
Este jogo é um problema de busca, que envolve através de um estado inicial, chegar a um estado final. Temos também uma função sucessora (que define os movimentos possíveis,, estes que no melhor caso são: cima, baixo, esquerda e direita) e por fim temos o conjunto de estados intermediários (os diferentes tabuleiros formados ao longo do caminho).
Para enteder melhor o jogo porque não jogá-lo 15 Puzzle Game.
É importante mencionar que nem todas as configurações iniciais têm solução, o que pode ser determinado pelas paridades dos movimentos e da distância do espaço vazio.
Quando há solução, podemos encontrar o caminho aplicando diversas estratégias de busca.

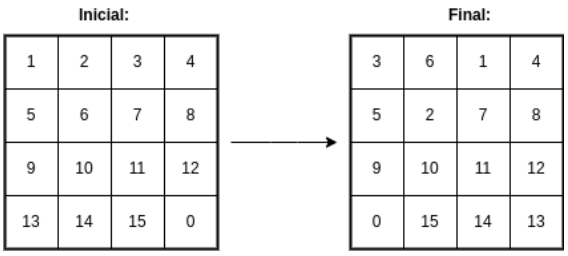


Figure 1: Representação de um jogo.

3 Estratégias de pesquisas não informadas

3.1 DFS (Pesquisa de Profundidade)

Começamos por explorar cada ramo o quanto possível. (Basicamente vamos da esquerda para a direita e de cima para baixo). O seu de depth só é contabilizado quando encontrarmos a solução, retornando cada nível percorrido. Quando chegamos ao limite de um nó, ou, quando essa pesquisa é sem sucesso, o mecanismo "Backtracking" retorna através do caminho percorrido até lá de modo a que tente encontrar possíveis soluções na possibilidade seguinte.
A complexidade espacial deste algoritmo é de b*m, sendo b o fator de ramificação e m a profundidade máxima. Já a complexidade temporal é de b^m.
Este algoritmo é usado quando o uso de memória eficiente é importante e quando qualquer solução para o problema, independemente do seu tamanho, é o suficiente para se resolver o problema. Para falarmos melhor sobre o Pesquisa de Profundidade, é interessante fazermos uma comparação com outro algoritmo de busca não informada: A Pesquisa de Largura.

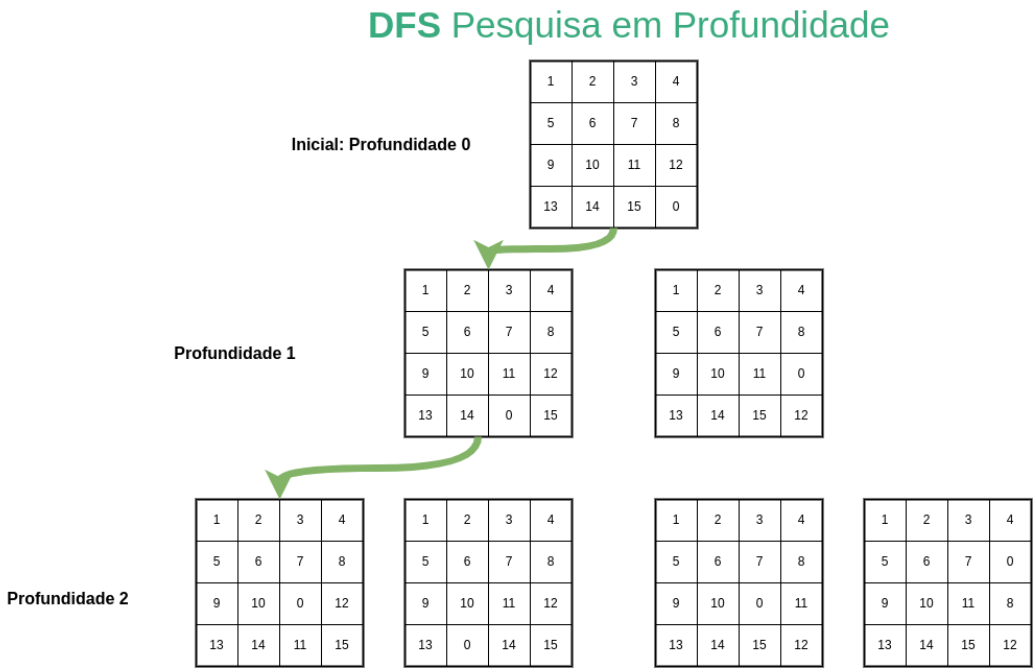


Figure 2: Pesquisa em profundidade.

3.2 BFS (Pesquisa de Largura)

Consiste em não expandir os nós de nível n+1, enquanto os nós do nível n forem expandidos, ou seja, primeiro fazemos as buscas em todos os vizinhos de um nó e só depois passamos para o próximo nível. Isso garante que nenhum vértice ou aresta será visitado mais de uma vez. Para que isso aconteça, utilizamos as filas para garantir a ordem correta de chegada dos vértices. A complexidade espacial e temporal do BFS são iguais: b^d, sendo d a profundidade onde se há a solução. Isso acontece por conta da necessidade do algoritmo ter que armazenar todas as folhas da árvore ao mesmo tempo. Por conta do processo de armazenamento, o grande problema deste algoritmo é justamente o seu uso de memória. Além disso, a procura por cada folha da árvore torna o código menos eficiente comparado ao DFS que, em comparação, só vê quando a folha é solução ou não. Apesar desses problemas, o BFS evita que o algoritmo entre num loop infinito, diferentemente do DFS que, se implementado mal e não analisa os nós visitados, faz uma procura infinita e prejudicial para a máquina.

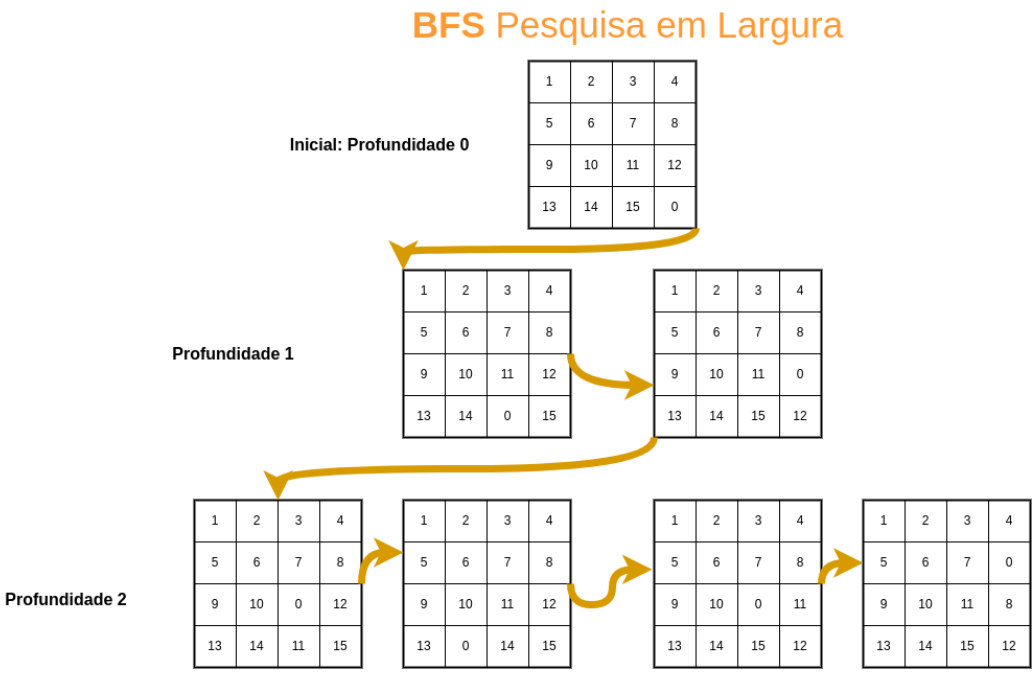


Figure 3: Pesquisa em largura.

3.3 IDFS (Pesquisa de profundidade iterativa)

Seria uma combinação entre as duas buscas citadas anteriormente. A busca iterativa limitada em profundidade não precisa de armazenar toda a árvore de busca na memória, assim como a busca em profundidade. Além disso, sua busca é completa como a busca em largura, assim garantindo que encontrará a solução se houver uma. Sua complexidade temporal é de b^l, sendo l o limite de profundidade imposto pelo algoritmo. Já o complexidade espacial é de b*l. Apesar de ser uma boa opção para situações com espaço de memória limitado, esta busca pode ser muito lenta em problemas que envolvem muitas camadas ou níveis de profundidade. Isso acontece por conta da sua redundância extrema e checkagem constante de nós que já foram vistos. Analisado todas as buscas não-guiadas, veremos as buscas guiadas.

4 Heurísticas

Para entendermos como funcionam as pesquisas guiadas, temos antes de entender o conceito de heurística.
A heurística (que por termos de curiosidade vem do grego e significa "eu encontro, descubro") na computação é uma função que determina o custo das alternativas nos algoritmos de pesquisa. Ou seja, são funções que calculam o quão ótimo um certo nó é para se atingir o estado final.
Imaginemos por exemplo que queremos descobrir o caminho com menor distância entre uma cidade A até uma cidade B. Uma heurística seria a distância em linha reta da cidade A até a cidade B. No caso do Jogo do 15, iremos utlizar duas heurísticas: o somatório de peças fora do lugar do estado final e o somatório do Manhattan Distance. Para o primeiro caso, basta calcular a soma das peças que estão fora de posição (Por exemplo, numa sequência 1 2 3 3 1 2, o custo seria 3, pois todas as peças estão fora da posição). No segundo caso, o Manhattan Distance é calculado pela expressão $|x1 - x2| + |y1 - y2|$ na qual (x1, y1) é a posição de uma peça no tabuleiro 4x4 e (x2, y2) é a posição dessa mesma peça no tabuleiro final 4x4.
Esclarecido o conceito de heurística, vamos ao Algoritmos Greedy e A*.

5 Estratégias de pesquisas informadas

5.1 Greedy/gulosa

Basicamente faz a melhor escolha local e espera que essa escolha leve para a melhor escolha global. É similar ao DFS, mas com uma diferença crucial: o Greedy pode mudar de direção dependendo do custo dos nós ainda não explorados da árvore de procura. Apesar de possuir uma heurística e teoricamente calcular qual é a melhor rota, o algoritmo nem sempre chega ao resultado esperado. Normalmente, a complexidade temporal e espacial é de b^m, contudo isso depende muito da qualidade da heurísitca utlizada e do tipo de problema. Em suma, este algoritmo depende muito do tipo de problema apresentado para ser eficiente.

5.2 A*

O A* tem algumas semelhanças com o Greedy, mas possui um fator diferenciador importante: a combinação do custo entre o nó raiz e o nó atual

junto com a heurística que calcula o custo entre o nó atual e o nó final (Uma representação disso seria f(n) = g(n) + h(n)). Com este cálculo, conseguimos calcular um custo mais eficiente em direção ao objetivo. Para o A* ser ótimo e completo, o h(n) é preciso ser admissível, ou seja, nunca super-estimar o custo real da melhor solução. Para além disso, a função f(n) tem de ser monótona. Isso significa que f(n) nunca decresce.

A sua complexidade espacial e exponencial no pior caso é exponencial, mais especificamente b^d. Este algoritmo é extremamente popular para achar caminhos em jogos de estratégia para computador.

6 Descrição da Implementação

6.1 Linguagem utilizada

A linguagem escolhida para implementarmos estas pesquisas foi java, escolhemos esta linguagem porque para além de ter uma sintaxe clara e concisa o que facilita a sua leitura e implementação dos algoritmos, dispões também de uma vasta biblioteca e estruturas de dados o que nos facilitar a focar no que realmente importa, estratégias de busca.

6.2 Requisitos

Java 8 ou superior.

6.3 Execução

Para executar o programa, basta correr o seguinte comando:

```
$ javac Estrategia_a_utilizar.java && java  Estrategia_a_utilizar
```

Como é improvável partilharmos a mesma versão do jdk, é por isso que recomendamos compilar o programa na vossa máquina.
Introduzir a configuração inicial e final e apenas se for realizável é que será possível utilizar a busca pretendida tabuleiros.

Pode optar por intruduzir um teste de um documento .txt, para isso correr o seguinte comando:

```
$ java  Estrategia_a_utilizar<teste.txt
```

No final será possível ver dois testes que pode utilizar.

6.4 Implementação

Para estas diversas pesquisas, foram utilizadas diversas estruturas de dados, mas focando mais no tabuleiro em si, utilizamos um array de inteiros para representar as posições dos numeros no tabuleiro. Construimos uma classe própria para armazenar esse array e aí implementamos também outros métodos que facilitam mais á frente nas procuras.

6.4.1 Classe Tabuleiro

```
1  import java.util.*;
2
3  class Tabuleiro{
4      int[] tabuleiro; //representacao do tabuleiro
5      int posEspaco; // posicao do espaco vazio
6
7      //construtor
8      Tabuleiro(){
9          tabuleiro = new int[16];
10     }
11
12     //le o tabuleiro
13     public void read(Scanner in){
14         for(int i = 0; i < 16; i++) tabuleiro[i] = in.nextInt();
15         posEspaco = findIndex(tabuleiro,0);
16     }
17
18     //imprime o tabuleiro
19     public String toString() {
20         String res = "";
21         for(int i = 0; i < 16; i++) {
22             if(i%4 == 0) {
23                 if (i==0) res = "\n";
24                 else res += "|\n"; res += "+----------------+\n";}
25             if (tabuleiro[i]<10){res += "|"+ tabuleiro[i] + "  ";}
26             else{res += "|"+tabuleiro[i] + " ";}
27         }
28         res+="|"+"\n+----------------+";
29         res += "\n";
30         return res;
31     }
32
33     //metodo que retorna uma copia do tabuleiro atual
```

```
34      public Tabuleiro clone(){
35          Tabuleiro novo = new Tabuleiro();
36          for (int i = 0; i < 16; i++) novo.tabuleiro[i] = tabuleiro[i];
37          novo.posEspaco = posEspaco;
38          return novo;
39      }
40
41      // move o espa o vazio para cima
42      public boolean moveUp() {
43          if (posEspaco >= 4) {
44              int temp = tabuleiro[posEspaco-4];
45              tabuleiro[posEspaco-4] = 0;
46              tabuleiro[posEspaco] = temp;
47              posEspaco -= 4;
48              return true;
49          }
50          return false;
51      }
52
53      // move o espa o vazio para baixo
54      public boolean moveDown() {
55          if (posEspaco <= 11) {
56              int temp = tabuleiro[posEspaco+4];
57              tabuleiro[posEspaco+4] = 0;
58              tabuleiro[posEspaco] = temp;
59              posEspaco += 4;
60              return true;
61          }
62          return false;
63      }
64
65      // move o espa o vazio para a esquerda
66      public boolean moveLeft() {
67          if (posEspaco % 4 != 0) {
68              int temp = tabuleiro[posEspaco-1];
69              tabuleiro[posEspaco-1] = 0;
70              tabuleiro[posEspaco] = temp;
71              posEspaco--;
72              return true;
73          }
74          return false;
75      }
76
77      // move o espa o vazio para a direita
78      public boolean moveRight() {
79          if (posEspaco % 4 != 3) {
80              int temp = tabuleiro[posEspaco+1];
81              tabuleiro[posEspaco+1] = 0;
82              tabuleiro[posEspaco] = temp;
83              posEspaco++;
84              return true;
85          }
86          return false;
87      }
88
89      // verifica se o tabuleiro atual    igual ao tabuleiro do parametro
90      public boolean equals(Tabuleiro t) {
```

```
91          if (t == null) return false;
92          for (int i = 0; i < 16; i++) {
93              if (tabuleiro[i] != t.tabuleiro[i]) return false;
94          }
95          return true;
96      }
97
98      // compara a distancia do 0 da sua posicao com o 0 de outro tabuleiro
99      public int compareDis(Tabuleiro t,int num) {
100         int pos0 = 0;
101         int pos0t = 0;
102         for (int i = 0; i < 16; i++) {
103             if (tabuleiro[i] == num) pos0 = i;
104             if (t.tabuleiro[i] == num) pos0t = i;
105         }
106         int dist = Math.abs(pos0/4 - pos0t/4) + Math.abs(pos0%4 - pos0t%4);
107         return dist;
108     }
109
110     //verifica se o tabuleiro tem solucao a comparar com outro tabuleiro
111     public boolean hasSolution(Tabuleiro t) {
112         int[] temp=this.clone().tabuleiro;
113         int inv = 0;
114         int dist_brancos = compareDis(t,0);
115         for (int i = 0; i < 16; i++) {
116             if(temp[i] != t.tabuleiro[i]) {
117                 for(int j = findIndex(temp, t.tabuleiro[i]); j > i; j--) {
118                     int temporario = temp[j - 1];
119                     temp[j - 1] = temp[j];
120                     temp[j] = temporario;
121                     inv++;
122                 }
123
124             }
125         }
126         if (inv%2 == dist_brancos%2) return true;
127         return false;
128     }
129
130     //contar as pe as fora de posicao
131     public int countMisplaced(Tabuleiro t) {
132         int count = 0;
133         for (int i = 0; i < 16; i++) {
134             if (tabuleiro[i] != t.tabuleiro[i]) count++;
135         }
136         return count;
137     }
138
139     //contar a distancia de manhattan
140     public int countManhattan(Tabuleiro t) {
141         int count = 0;
142         for (int i = 0; i < 16; i++) {
143             if (tabuleiro[i] != t.tabuleiro[i]) {
144                 int pos = findIndex(t.tabuleiro, tabuleiro[i]);
145                 count += Math.abs(pos/4 - i/4) + Math.abs(pos%4 - i%4);
146             }
147         }
```

```
148         return count;
149     }
150 
151     //encontra o indice da lista de um numero num tabuleiro
152     public int findIndex(int[] t,int num){
153         for (int i = 0; i < 16; i++) {
154             if (t[i] == num) return i;
155         }
156         return -1;
157     }
158 
159     // para hashing
160     @Override
161     public int hashCode() {
162         return Arrays.hashCode(this.tabuleiro);
163     }
164 
165     @Override
166     public boolean equals(Object obj) {
167         if (this == obj) {
168             return true;
169         }
170         if (!(obj instanceof Tabuleiro)) {
171             return false;
172         }
173         Tabuleiro other = (Tabuleiro) obj;
174         return Arrays.equals(this.tabuleiro, other.tabuleiro);
175     }
176 
177 }
```

6.4.2 Classe Node

Para representar por separado, um tabuleiro com o seu custo, profundidade, caminho e o seu nó anterior, criamos a classe Node para facilitar isso, também aplicamos alguns métodos auxiliares que mais tarde vão ser mais úteis.

```
1  class Node{
2      Tabuleiro tab; // tabuleiro do n
3      Node pai; // n  pai
4      int profundidade; // profundidade do n
5      int custo; // custo do n  para o objetivo, utilizado apenas nas
       pesquisas orientadas
6      String movimento; // movimento que gerou o n
7  
8      // construtor
9      Node(Tabuleiro tab, Node pai, int profundidade, int custo, String
       movimento){
10         this.tab = tab;
11         this.pai = pai;
12         this.profundidade = profundidade;
13         this.custo = custo;
14         this.movimento = movimento;
15     }
```

```
16
17          //imprime o n
18          public String toString() {
19              String res = "";
20              res += "Movimento: " + movimento + "\n";
21              res += "profundidade: " + profundidade + "\n";
22              return res;
23          }
24
25          //metodo que retorna uma copia do n   atual
26          public Node clone(){
27              Node novo = new Node(tab.clone(), pai, profundidade, custo,
    movimento);
28              return novo;
29          }
30
31          // move o espa o vazio para cima
32          public Node moveUp(Tabuleiro objetivo, int opcao, Tabuleiro raiz){
33              Tabuleiro tab1=tab.clone();
34              if (tab1.moveUp()) {
35                  Node novo = new Node(tab1, this, profundidade+1, custo,
    movimento+" cima");
36                  novo.custo(objetivo, opcao,raiz);
37                  return novo;
38              }
39              return null;
40          }
41
42          // move o espa o vazio para baixo
43          public Node moveDown(Tabuleiro objetivo, int opcao, Tabuleiro raiz){
44              Tabuleiro tab1=tab.clone();
45              if (tab1.moveDown()) {
46                  Node novo = new Node(tab1, this, profundidade+1, custo,
    movimento+" baixo");
47                  novo.custo(objetivo, opcao,raiz);
48                  return novo;
49              }
50              return null;
51          }
52
53          // move o espa o vazio para a esquerda
54          public Node moveLeft(Tabuleiro objetivo, int opcao, Tabuleiro raiz){
55              Tabuleiro tab1=tab.clone();
56              if (tab1.moveLeft()) {
57                  Node novo = new Node(tab1, this, profundidade+1, custo,
    movimento+" esquerda");
58                  novo.custo(objetivo, opcao,raiz);
59                  return novo;
60              }
61              return null;
62          }
63
64          // move o espa o vazio para a direita
65          public Node moveRight(Tabuleiro objetivo, int opcao, Tabuleiro raiz)
    {
66              Tabuleiro tab1=tab.clone();
67              if (tab1.moveRight()) {
```

```
68                  Node novo = new Node(tab1, this, profundidade+1, custo,
    movimento+" direira");
69                  novo.custo(objetivo, opcao,raiz);
70                  return novo;
71              }
72              return null;
73          }
74
75          // calcula e altera custo do no consoante as opcoes
76          // a raiz so vai ser utilizada para a pesquisa Astar
77          public void custo(Tabuleiro objetivo, int opcao, Tabuleiro raiz){
78              switch (opcao){
79                  //para Greedy
80                  case 1: custo = tab.countManhattan(objetivo);break;
81                  case 2: custo = tab.countMisplaced(objetivo);break;
82                  //para Astar
83                  case 3: custo = tab.countManhattan(objetivo) + profundidade;
84                          break;
85                  case 4: custo = tab.countMisplaced(objetivo) + profundidade;
86                          break;
87                  case 5: custo = (tab.countManhattan(objetivo) + tab.
    countManhattan(raiz));
88                          break;
89                  case 6: custo = (tab.countMisplaced(objetivo) + tab.
    countMisplaced(raiz));
90                          break;
91                  case 0: custo = 0;
92                          break; // custo nulo para busca em largura e
    profundidade
93              }
94          }
95  }
```

6.4.3 BFS

Passando agora ás pesquisas, em cada pesquisa utilizamos uma main, porque assim foi pedido previamente haver um ficheiro para cada pesquisa. Na pesquisa em largura, utilizamos uma fila *Queue*, e um hashmap para guardarmos os nos visitados, impedindo assim de andar em loop entre os mesmos nós. Utilizamos uma fila pois assim vai se analisando os nós do mais antigo para o mais recente cumprindo assim as regras do BFS

```
1  import java.util.*;
2
3  class BFS{
4      static Queue<Node> fila;
5      static HashMap<Tabuleiro, Integer> visitados;
6
7      public static void main(String[] args) {
8          Scanner in = new Scanner(System.in);
9          Tabuleiro atual=new Tabuleiro();
10         Tabuleiro objetivo=new Tabuleiro();
11
```

```
12          atual.read(in);
13          objetivo.read(in);
14
15          if (!atual.hasSolution(objetivo)){
16              System.out.println("O tabuleiro inicial nao tem solucao para o
    tabuleiro objetivo!");
17              return;
18          }
19          System.out.println("O tabuleiro inicial tem solucao para o tabuleiro
     objetivo!");
20          System.out.println();
21          System.out.println("----------------BFS----------------");
22          solve(atual,objetivo);
23
24      }
25
26      public static void solve(Tabuleiro atual, Tabuleiro objetivo){
27          // gravar o tempo inicial do programa
28          long tempoInicial = System.currentTimeMillis();
29
30          // gravar a quantidade de memoria usada
31          long usedMemory = Runtime.getRuntime().totalMemory() -Runtime.
    getRuntime().freeMemory();
32
33          //contar o numero de nos gerados
34          int nos=0;
35
36          //criar o hashmap para armazenar os nos visitados
37          visitados = new HashMap<>();
38
39          fila = new LinkedList<Node>(); // fila de nos a serem visitados
40          Node no = new Node(atual, null,0,0,""); // cria o no inicial
41          fila.add(no); // adiciona o no inicial na fila
42          System.out.println("Iniciando a busca...");
43          while(!fila.isEmpty()){
44              no = fila.poll(); // retira o primeiro no da fila
45              if(no.tab.equals(objetivo)){ // verifica se o no atual e o
    objetivo
46                  // gravar o tempo final do programa
47                  long tempoFinal = (long) (System.currentTimeMillis());
48                  // gravar a quantidade de memoria usada
49                  long finalMemory = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime().freeMemory();
50
51                  // imprimir os resultados
52                  System.out.println("Encontrou a solucao!");
53                  System.out.println(no);
54                  System.out.printf("Tempo de execucao: %.3f s%n", (tempoFinal
     - tempoInicial) / 1000d);
55                  System.out.println("Espaco de memoria: " + ((finalMemory-
    usedMemory)/1000000) + "  MB");
56                  System.out.println("Nos gerados: " + nos);
57                  return;
58              }
59              if (!visitados.containsKey(no.tab)){
60                  visitados.put(no.tab, no.tab.hashCode());
61                  Node up = no.moveUp(objetivo,0,null);nos++;
```

```
                Node down = no.moveDown(objetivo,0,null);nos++;
                Node left = no.moveLeft(objetivo,0,null);nos++;
                Node right = no.moveRight(objetivo,0,null);nos++;

                if (up!=null) fila.add(up);
                if (down!=null) fila.add(down);
                if (left!=null) fila.add(left);
                if (right!=null) fila.add(right);
            }
        }
    }
}
```

6.4.4 DFS

Á semelhança do Bfs, criamos um main tambem a unica diferença foi termos utilizado um stack, o que permite analisar uma ramo de cada vez até ao final e nao utilizamos nenhuma estrutura de dados para armazenar os nos visitados. Infelimente não conseguimos implementar um DFS sem limite de profundidade, e com isso utilizamos uma profundidade que se altera consoante o puzzle apresentado.

```
import java.util.*;
class DFS{
    static Stack<Node> fila;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Tabuleiro atual=new Tabuleiro();
        Tabuleiro objetivo=new Tabuleiro();

        atual.read(in);
        objetivo.read(in);

        if (!atual.hasSolution(objetivo)){
            System.out.println("O tabuleiro inicial nao tem solucao para o tabuleiro objetivo!");
            return;
        }
        System.out.println("O tabuleiro inicial tem solucao para o tabuleiro objetivo!");
        System.out.println();
        System.out.println("----------------DFS----------------");
        solve(atual,objetivo);

    }

    public static void solve(Tabuleiro atual, Tabuleiro objetivo){
        int MAX_DEPTH=atual.countMisplaced(objetivo)+ atual.countManhattan(objetivo);
        // gravar o tempo inicial do programa
        long tempoInicial = System.currentTimeMillis();

```

```
29          // gravar a quantidade de memoria usada
30          long usedMemory = Runtime.getRuntime().totalMemory() -Runtime.
    getRuntime().freeMemory();
31
32          //contar o numero de nos gerados
33          int nos=0;
34
35
36          fila = new Stack<Node>(); // fila de nos a serem visitados
37          Node no = new Node(atual, null,0,0,""); // cria o no inicial
38          fila.add(no); // adiciona o no inicial na fila
39          System.out.println("Iniciando a busca com MAX_DEPTH a " + MAX_DEPTH
    + " ...");
40          while(!fila.isEmpty()){
41              no = fila.pop(); // retira o primeiro no da fila
42              if(no.tab.equals(objetivo)){ // verifica se o no atual e o
    objetivo
43                  // gravar o tempo final do programa
44                  long tempoFinal = (long) (System.currentTimeMillis());
45                  // gravar a quantidade de memoria usada
46                  long finalMemory = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime().freeMemory();
47
48                  // imprimir os resultados
49                  System.out.println("Encontrou a solucao!");
50                  System.out.println(no);
51                  System.out.printf("Tempo de execucao: %.3f s%n", (tempoFinal
     - tempoInicial) / 1000d);
52                  System.out.println("Espaco de memoria: " + ((finalMemory-
    usedMemory)/1000000) + "  MB");
53                  System.out.println("Nos gerados: " + nos);
54                  return;
55              }
56              if (no.profundidade<MAX_DEPTH){
57                  Node up = no.moveUp(objetivo,0,null);nos++;
58                  Node left = no.moveLeft(objetivo,0,null);nos++;
59                  Node down = no.moveDown(objetivo,0,null);nos++;
60                  Node right = no.moveRight(objetivo,0,null);nos++;
61
62                  if (up!=null)   fila.add(up);
63                  if (down!=null) fila.add(down);
64                  if (left!=null) fila.add(left);
65                  if (right!=null) fila.add(right);
66              }
67          }
68          System.out.println("Nao encontrou solucao :(");
69      }
70
71
72  }
```

6.4.5 IDFS

Nesta busca basicamente utilizamos o código do DFS quase na totalidade, a unica diferença, é que a profundidade máxima é interativa, basicamente vai se incrementando e fazendo a pesquisa com a profundidade máxima igual ao valor anterior mais um. A profundidade começa no 1 e vai até á MaxDepth, que é o mesmo valor máximo de profundidade do DFS.

```
1  import java.util.*;
2  class IDFS{
3      static Stack<Node> fila;
4      static HashMap<Tabuleiro, Integer> visitados;
5
6      public static void main(String[] args) {
7          Scanner in = new Scanner(System.in);
8          Tabuleiro atual=new Tabuleiro();
9          Tabuleiro objetivo=new Tabuleiro();
10
11         atual.read(in);
12         objetivo.read(in);
13
14         if (!atual.hasSolution(objetivo)){
15             System.out.println("O tabuleiro inicial nao tem solucao para o
       tabuleiro objetivo!");
16             return;
17         }
18         System.out.println("O tabuleiro inicial tem solucao para o tabuleiro
        objetivo!");
19         System.out.println();
20         System.out.println("----------------IDFS----------------");
21         solve(atual,objetivo);
22
23     }
24
25     public static void solve(Tabuleiro atual, Tabuleiro objetivo){
26         int MAX_DEPTH=atual.countMisplaced(objetivo)+ atual.countManhattan(
       objetivo);
27
28         // gravar o tempo inicial do programa
29         long tempoInicial = System.currentTimeMillis();
30
31         // gravar a quantidade de memoria usada
32         long usedMemory = Runtime.getRuntime().totalMemory() -Runtime.
       getRuntime().freeMemory();
33
34         //contar o numero de nos gerados
35         int nos=0;
36
37         //criar o hashmap para armazenar os nos visitados
38         visitados = new HashMap<>();
39
40
41         fila = new Stack<Node>(); // fila de nos a serem visitados
42         Node inicio = new Node(atual, null,0,0,""); // cria o no inicial
43         for (int j=1; j<=MAX_DEPTH;j++){
```

```
44              System.out.println("Iniciando a busca com profundidade "+ j + "
    e MAX_DEPTH a " + MAX_DEPTH + " ...");
45              fila.add(inicio); // adiciona o no inicial na fila
46              nos++;
47
48              while(!fila.isEmpty()){
49                  Node no = fila.pop().clone(); // retira o primeiro no da
    fila
50
51                  if(no.tab.equals(objetivo)){ // verifica se o no atual    o
    objetivo
52                      // gravar o tempo final do programa
53                      long tempoFinal = (long) (System.currentTimeMillis());
54                      // gravar a quantidade de memoria usada
55                      long finalMemory = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime().freeMemory();
56
57                      // imprimir os resultados
58                      System.out.println("Encontrou a solu   o!");
59                      System.out.println(no);
60                      System.out.printf("Tempo de execu   o: %.3f s%n", (
    tempoFinal - tempoInicial) / 1000d);
61                      System.out.println("Espa o de mem ria: " + ((
    finalMemory-usedMemory)/1000000) + "  MB");
62                      System.out.println("N s gerados: " + nos);
63                      return;
64                  }
65
66                  if (!visitados.containsKey(no.tab) && no.profundidade<j){
67                      visitados.put(no.tab, no.tab.hashCode());
68                      Node up = no.moveUp(objetivo,0,null);nos++;
69                      Node left = no.moveLeft(objetivo,0,null);nos++;
70                      Node down = no.moveDown(objetivo,0,null);nos++;
71                      Node right = no.moveRight(objetivo,0,null);nos++;
72
73                      if (up!=null)   fila.add(up);
74                      if (down!=null) fila.add(down);
75                      if (left!=null) fila.add(left);
76                      if (right!=null) fila.add(right);
77                  }
78              }
79              visitados.clear();
80           }
81              System.out.println("N o encontrou solu   o :(");
82          }
83
84
85  }
```

6.4.6 Greedy/gulosa

Nesta busca, orientamos mais a pesquisa de forma a cumprir duas euristicas, a distancia de Manhanttan, ou as casas fora do sitio. Nesta implementação passamos a utilizar o atributo custo da class Node, para calcular o custo

é só basicamente utilizar os métodos que já foram apresentados na classe Tabuleiro e Node.
Utilizamos uma priority queue que compara os nós através do seu custo, utilizamos também um hashmap para os nós visitados, só para impedir de estar em loop.

```
import java.util.*;

class Greedy{
    static HashMap<Tabuleiro, Integer> visitados;
    static PriorityQueue<Node> fila;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Tabuleiro atual=new Tabuleiro();
        Tabuleiro objetivo=new Tabuleiro();

        atual.read(in);
        objetivo.read(in);

        if (!atual.hasSolution(objetivo)){
            System.out.println("O tabuleiro inicial nao tem solucao para o tabuleiro objetivo!");
            return;
        }
        System.out.println("O tabuleiro inicial tem solucao para o tabuleiro objetivo!");
        System.out.println();
        System.out.println("-----Greedy Utilizando a Euristica distacia de Manhattan-----");
        solve(atual,objetivo,1);
        System.out.println('\n');
        System.out.println("-----Greedy Utilizando a Euristica pecas Missplaced-----");
        solve(atual,objetivo,2);
    }

    public static void solve(Tabuleiro atual, Tabuleiro objetivo, int euristica){
        // gravar o tempo inicial do programa
        long tempoInicial = System.currentTimeMillis();

        // gravar a quantidade de memoria usada
        long usedMemory = Runtime.getRuntime().totalMemory() -Runtime.getRuntime().freeMemory();

        //contar o numero de nos gerados
        int nos=0;

        //criar o hashmap para armazenar os nos visitados
        visitados = new HashMap<>();

        //fila de prioridade para armazenar os nos a serem visitados
        //compara os custos dos nos de acordo com a euristica escolhida
        fila = new PriorityQueue<>(new Comparator<Node>() {
```

```
44              @Override
45              public int compare(Node o1, Node o2) {
46                  return o1.custo - o2.custo;
47              }
48          });
49
50          Node no = new Node(atual, null,0,0,""); // cria o no inicial
51          fila.add(no); // adiciona o no inicial na fila
52          System.out.println("Iniciando a busca ...");
53          while(!fila.isEmpty()){
54              no = fila.poll(); // retira o primeiro no da fila
55              if(no.tab.equals(objetivo)){ // verifica se o no atual e o
    objetivo
56                  // gravar o tempo final do programa
57                  long tempoFinal = (long) (System.currentTimeMillis());
58                  // gravar a quantidade de memoria usada
59                  long finalMemory = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime().freeMemory();
60
61                  // imprimir os resultados
62                  System.out.println("Encontrou a solucao!");
63                  System.out.println(no);
64                  System.out.printf("Tempo de execucao: %.3f s%n", (tempoFinal
     - tempoInicial) / 1000d);
65                  System.out.println("Espa o de memoria: " + ((finalMemory-
    usedMemory)/1000000) + "  MB");
66                  System.out.println("Nos gerados: " + nos);
67                  return;
68              }
69              if (!visitados.containsKey(no.tab)){
70                  visitados.put(no.tab, no.tab.hashCode());
71                  Node up = no.moveUp(objetivo,euristica,null);nos++;
72                  Node left = no.moveLeft(objetivo,euristica,null);nos++;
73                  Node right = no.moveRight(objetivo,euristica,null);nos++;
74                  Node down = no.moveDown(objetivo,euristica,null);nos++;
75
76                  if (up!=null) fila.add(up);
77                  if (down!=null) fila.add(down);
78                  if (left!=null) fila.add(left);
79                  if (right!=null) fila.add(right);
80              }
81          }
82      }
83  }
```

6.4.7 A*

A unica diferença que fizemos neste foi utilizar euristicas diferentes, para a de manhanttan somamos essa distância do objetivo mais a respetiva profundidade do nó, para a missplaced somamos essa distancia do no atual para o inicio, e do no atual para o objetivo. Utilizamos isto porque os resultados foram melhores assim. As euristicas disponíveis encontram se no método

custo na classe Node, na classe Astar é possível alterar as euristicas que queremos utilizar.

```
import java.util.*;

class Astar{
    static PriorityQueue<Node> fila;
    static HashMap<Tabuleiro, Integer> visitados;

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Tabuleiro atual=new Tabuleiro();
        Tabuleiro objetivo=new Tabuleiro();

        atual.read(in);
        objetivo.read(in);

        if (!atual.hasSolution(objetivo)){
            System.out.println("O tabuleiro inicial nao tem solucao para o tabuleiro objetivo!");
            return;
        }
        System.out.println("O tabuleiro inicial tem solucao para o tabuleiro objetivo!");
        System.out.println();
        System.out.println("-----A* Utilizando a Euristica distacia de Manhattan-----");
        solve(atual,objetivo,3);
        System.out.println('\n');
        System.out.println("-----A* Utilizando a Euristica pecas Missplaced-----");
        solve(atual,objetivo,6);
    }

    public static void solve(Tabuleiro atual, Tabuleiro objetivo, int euristica){
        // gravar o tempo inicial do programa
        long tempoInicial = System.currentTimeMillis();

        // gravar a quantidade de memoria usada
        long usedMemory = Runtime.getRuntime().totalMemory() -Runtime.getRuntime().freeMemory();

        //contar o numero de nos gerados
        int nos=0;

        //criar o hashmap para armazenar os nos visitados
        visitados = new HashMap<>();

        //fila de prioridade para armazenar os nos a serem visitados
        //compara os custos dos nos de acordo com a euristica escolhida
        fila = new PriorityQueue<>(new Comparator<Node>() {
            @Override
            public int compare(Node o1, Node o2) {
                return o1.custo - o2.custo;
            }
        });
```

```
49
50          // guardar raiz para euristica
51          Tabuleiro root=atual.clone();
52
53          Node no = new Node(atual, null,0,0,""); // cria o no inicial
54          fila.add(no); // adiciona o no inicial na fila
55          System.out.println("Iniciando a busca ...");
56
57          while(!fila.isEmpty()){
58              no = fila.poll(); // retira o primeiro no da fila
59              if(no.tab.equals(objetivo)){ // verifica se o no atual    o
    objetivo
60                  // gravar o tempo final do programa
61                  long tempoFinal = (long) (System.currentTimeMillis());
62                  // gravar a quantidade de memoria usada
63                  long finalMemory = Runtime.getRuntime().totalMemory() -
    Runtime.getRuntime().freeMemory();
64
65                  // imprimir os resultados
66                  System.out.println("Encontrou a solucao!");
67                  System.out.println(no);
68                  System.out.printf("Tempo de execucao: %.3f s%n", (tempoFinal
     - tempoInicial) / 1000d);
69                  System.out.println("Espaco de memoria: " + ((finalMemory-
    usedMemory)/1000000) + "  MB");
70                  System.out.println("Nos gerados: " + nos);
71                  return;
72              }
73              if (!visitados.containsKey(no.tab)){ // verifica se o no atual
    j   foi visitado
74                  visitados.put(no.tab, no.hashCode()); // adiciona o no atual
     na lista de visitados
75                  Node up = no.moveUp(objetivo,euristica,root);nos++;
76                  Node left = no.moveLeft(objetivo,euristica,root);nos++;
77                  Node down = no.moveDown(objetivo,euristica,root);nos++;
78                  Node right = no.moveRight(objetivo,euristica,root);nos++;
79
80                  if (up!=null && !up.equals(up.pai) && !visitados.containsKey
    (up.tab)) fila.add(up);
81                  if (down!=null && !down.equals(down.pai) && !visitados.
    containsKey(down.tab)) fila.add(down);
82                  if (left!=null && !left.equals(left.pai) && !visitados.
    containsKey(left.tab)) fila.add(left);
83                  if (right!=null && !right.equals(right.pai) && !visitados.
    containsKey(right.tab)) fila.add(right);
84              }
85
86          }
87      }
88
89  }
```

7 Resultados

7.1 Testes

Quando aumentada a profundidade dos testes, é aconselhado a não testar as pesquisas não orientadas, pois estas vão correr num tempo mais alongado com a possibilidade de nem dar resposta. Os nos gerados nas próximas tabelas de teste, está a incluir todos os nós criados incluindo os null, estes que são resultado de movimentos ilegais no tabuleiro.

7.1.1 Teste 1

Para o seguinte teste de profundidade 12, os resultados foram:

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

Estratégia	Tempo (s)	Nos gerados	Encontrou	Profundidade
DFS	0.075	491150	Sim	24
BFS	0.031	71552	Sim	12
IDFS	0.027	38256	Sim	12
Gulosa Manhattan	0.002	72	Sim	12
Gulosa Missplaced	0.000	60	Sim	12
A* Manhattan	0.002	88	Sim	12
A* Missplaced	0.001	240	Sim	12

7.1.2 Teste 2

Para o seguinte teste de profundidade 15, os resultados foram:

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

Estratégia	Tempo (s)	Nos gerados	Encontrou	Profundidade
DFS	28.034	398886276	Sim	27
BFS	0.233	587148	Sim	15
IDFS	0.162	539322	Sim	15
Gulosa Manhattan	0.003	332	Sim	33
Gulosa Missplaced	0.004	2900	Sim	91
A* Manhattan	0.003	612	Sim	15
A* Missplaced	0.003	3336	Sim	15

7.1.3 Teste 3

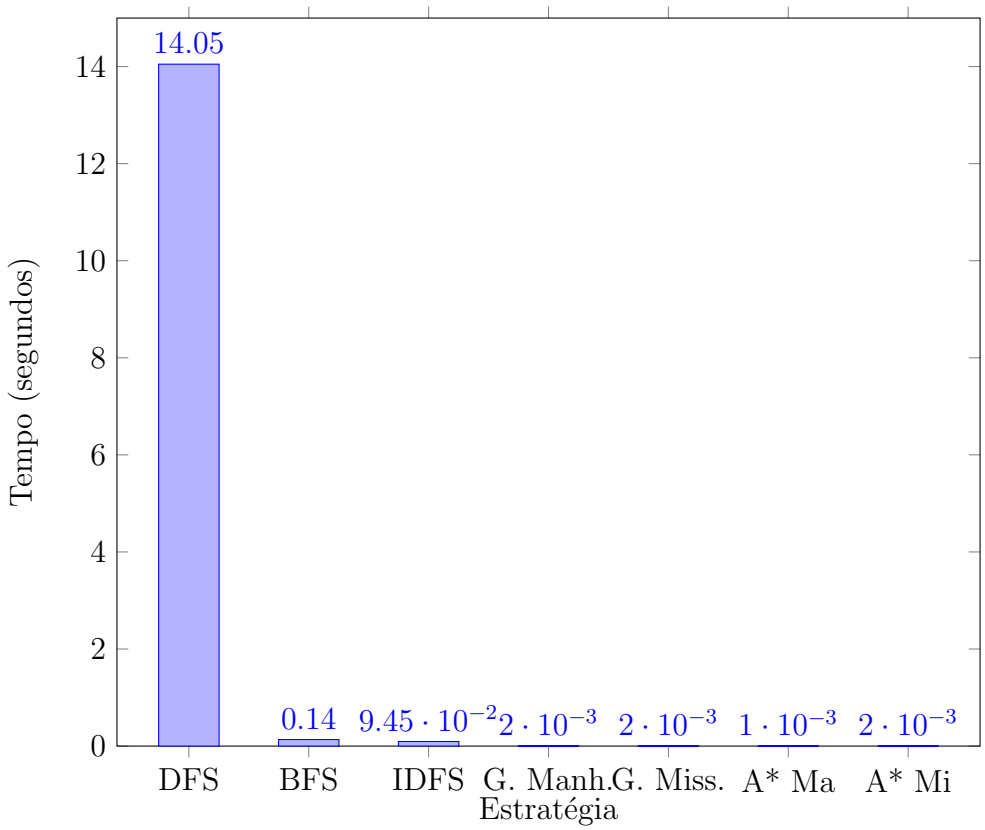
Para o seguinte teste de profundidade 32 não foi possível correr no BFS e DFS, os resultados foram:

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

Estratégia	Tempo (s)	Nos gerados	Encontrou	Profundidade
DFS	-	-	Nao	-
BFS	-	-	Nao	-
IDFS	-	-	Nao	-
Gulosa Manhattan	0.017	14592	Sim	108
Gulosa Missplaced	0.036	41864	Sim	250
A* Manhattan	0.047	58636	Sim	32
A* Missplaced	0.034	64804	Sim	80

7.1.4 Média:

Estes números são apenas dos dois primeiros testes pois temos resultados de todas as pesquisas.



Com isto percebemos que as pesquisas guiadas são mais eficientes, a euristica de Manhanttan também é mais eficiente do que a missplaced.

8 Conclusão

De acordo com a pesquisa feita sobre os algoritmos e visto as complexidades temporais e espaciais, nós concluímos que os resultados dos algoritmos menos eficiente para os mais eficientes é nesta ordem: Pesquisa de profundidade, Pesquisa de largura, Pesquisa de Profundidade Iterativa, Greedy e A.
Como as três primeiras (DFS, BFS e IDFS) são buscas não guiadas, estas são menos eficientes na hora de escolher em comparação às pesquisas guiadas (Guloso e A*). A diferença entre estas duas, é que a A* para além de calcular o custo entre o nó atual e o nó final, como o Greedy, também calcula o custo do nó atual até ao nó raiz.

Contudo, na implementação que fizemos, o resultado final foi Depth-First Search, Breadth-First Search, Pesquisa de Profundidade Iterativa, A* e por fim Greedy, isto num nível de memória gasta, o que traduz para alguns testes para tempo gasto. Porém através da A* conseguimos sempre caminhos com profundidades menores que a greedy, logo a pesquisa a utilizar será de acordo com o que for prioridade para o utilizador, ou chegar a uma resposta rapidamente ou chegar ao caminho mais curto.

9 Biliografia

Geeksforgeeks article, 15 Puzzle solvability
Mediatum document, "Solving the 15-Puzzle Game Using Local Value-Iteration"
Kociemba - "fifteensolver"
Solvability of the Tiles Game,the university of Birmingham
The University of British Columbia-15 puzzle
Wolfram-15Puzzle
Lukago GitHub repository, of a possible 15 Puzzle implementation
Section.io "understanding search algorithms in ai"
Link do vídeo que ajuda a entender melhor como conseguimos ver se um jogo é fazivél.