1. **Estruturas de dados para representar tabela e realizar operações**

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 . Este documento, tem como objetivo encontrar a melhor forma para representar tabela para o problema.

* 1. **Operações suportadas pela classe Table:**

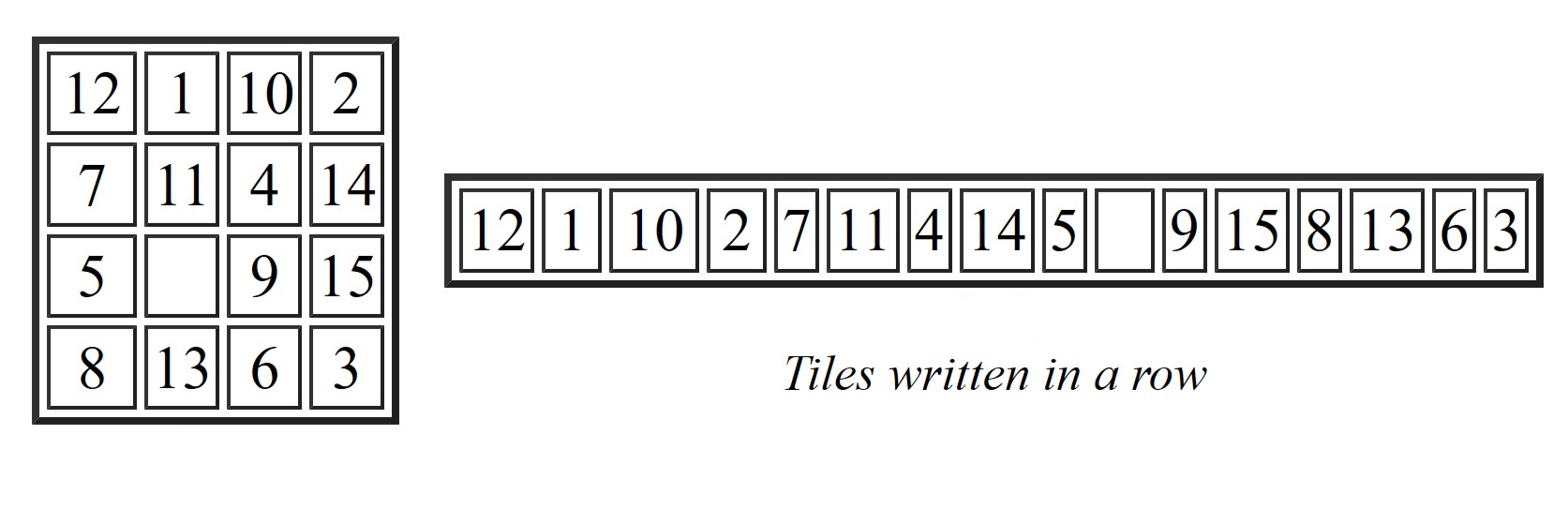
A classe deverá ter capacidades de comparar, manipular e iterar objetos. Deverá ter um construtor capaz de converter os dados fornecidos (em princípio uma matriz a gravar e valores inteiros para a linha e coluna da peça branca) em valores de classe.

* + 1. **A Questão/Verificação de ser Solucionável**

Deverá possuir um método **isSolvable()** que retorna um valor booleano verdadeiro caso esta tabela esteja em condições de chegar a método final. Este método irá basear-se na aplicação do teste de inversões. Uma **inversão** é o número total de células que, após a célula atual, têm valor inferior ao da célula em causa.

Para poder aplicar este teste, é necessário que a tabela representativa do puzzle seja expressa em uma única fila, um *array,* a *Figura 1* representa essa operação:

Figura 1-Conversão de tabela em array



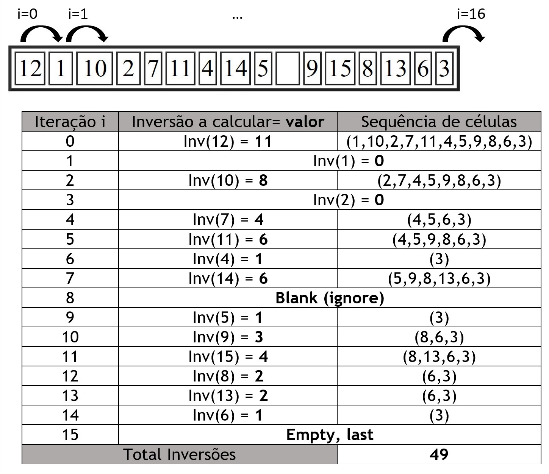
[](https://user-images.githubusercontent.com/39220282/53185801-a43a7700-35f7-11e9-984a-4a39b1c27687.png)Seguidamente, deverá ser calculado o número de inversões de cada célula, posteriormente, esses valores deverão ser todos somados, obtendo assim a **inversão total** da disposição atual, tal como a *Figura 2* demostra.

Figura 2-Cálculo de inversão total

Posteriormente, tendo já calculado a inversão total, sendo esta 49 para o caso específico, encontramo-nos em condições de aplicar a fórmula. Sendo este problema representado por uma tabela de **largura par**, apenas interessa a segunda parte da fórmula:

* Se largura da tabela for **ímpar**:

Sistema **tem solução** se inversão total for **par**

* Se largura da tabela for **par:**
  + - * Se a célula vazia estiver em **linhas pares** a contar de baixo, o sistema **tem solução** se inversão total for **ímpar**
      * Se a célula vazia estiver em **linhas ímpares** a contar de baixo, o sistema **tem solução**¸ se inversão total for **par.**

A *Figura 3* é o esquema desta fórmula:

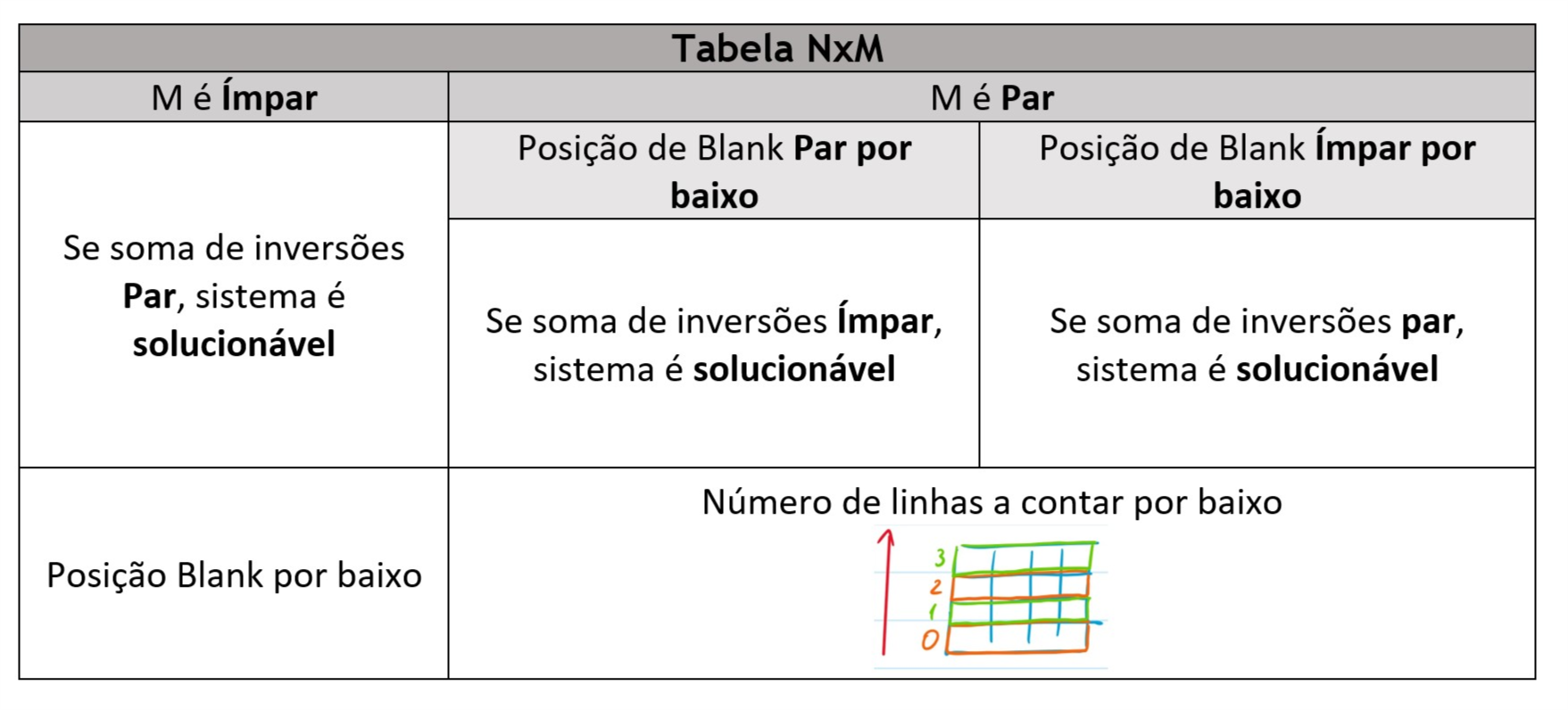


Figura 3-Fórmula do teste de inversões

* + 1. **A Verificação de ser Estado Final**

Deverá possuir um método **isSolution()** que retorna valor booleano verdadeiro, caso a configuração atual seja a configuração desejada, pelo que não será necessário continuar a pesquisa.

Uma das maneiras de aplicar este método é simplesmente converter a configuração atual em formato de array, tal como representado na *Figura 1* e comparar com a tabela objetivo, esta também convertida em array.

* + 1. **A Posição da Célula Vazia/Branca**

Deverá possuir um método *whiteLocation()* que retorna a posição, linha e coluna, da célula vazia na matriz. O ideal é a própria classe ter variáveis ***row*** e ***col*** cujos valores sejam os pretendidos (linha e coluna da célula vazia), caso seja este o caso (recomendado) terão que ser implementados métodos ***getRow()***,***getCol()***,***setRow()***e ***setCol()***, que retornem e definam os valores de linha e coluna da célula vazia respetivamente.

* + 1. **As Heurísticas**

É importante referir que os métodos referidos nesta secção apenas deverão ser utilizados para métodos de pesquisa informados, tais como, o método *greedy* e o método A\*.Uma heurística é um método de pesquisa que quantifica o quão próximo estamos do nosso objetivo. Neste problema, iremos considerar duas heurísticas, podendo haver outras.

A primeira é o método da **distância Hamming** entre a tabela atual e a tabela objetivo. Este método deverá constituir uma função **distHamming()** que retorna um valor inteiro, sendo esse valor, a distância Hamming da configuração atual. Notar que a distância Hamming é o número de células em posições incorretas na tabela, ignorando a célula vazia.

A segunda é o método da **distancia Manhattan** entre a tabela atual e a tabela objetivo. Este método deverá constituir uma função **distManhattan()** que retorna um valor inteiro, sendo esse valor, a soma de todas as distâncias Manhattan de cada célula individual, ignorando a célula vazia. Notar que a distância Manhattan de uma célula individual é a soma da distância horizontal e vertical da posição da célula atual até à sua posição correta.

Por posições incorretas referimo-nos às posições que não são as posições na tabela objetivo. A *Figura 4* representa o objetivo destes métodos.

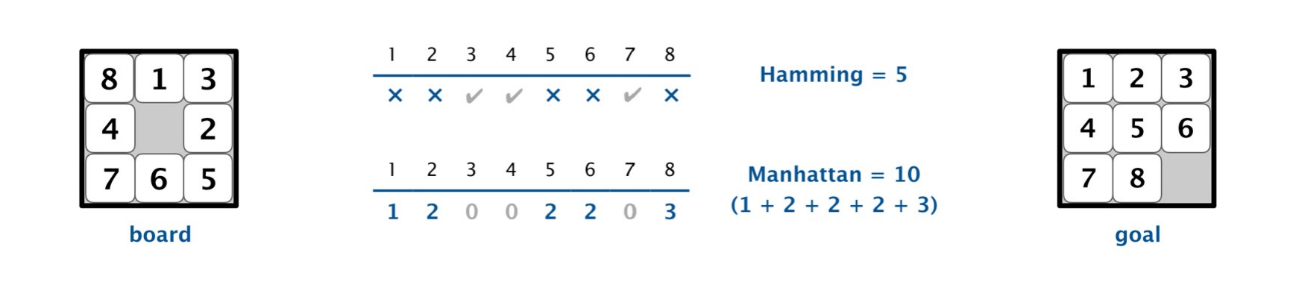
[](https://user-images.githubusercontent.com/39220282/53172205-acd18400-35dc-11e9-8edf-6760e47ba182.png)

Figura 4-Distância Hamming e Manhattan

* + 1. **Verificar Igualdades**

Independentemente do método de pesquisa que iremos aplicar, teremos sempre que verificar se a configuração obtida é igual à configuração objetivo (caso necessário para a terminação do programa) tendo sido este método já referido, no entanto, também é necessário verificar se a nossa configuração é igual a outra qualquer. Isto é especialmente útil em métodos de pesquisa como o BFS e DFS pois impede criação de ciclos. Como tal, deverá ser implementado um método **equals()** que retorne o valor booleano da comparação entre as duas tabelas.

Notar o método referido em 1.1.2 e nesta secção poderiam ser fundidos num só. Uma observação interessante é que, em vez de andar a comparar item a item poderíamos transformar as matrizes em *array* e por sua vez em *strings* e depois utilizar o método *built-in* do Java *equals()*, esta ideia será especialmente importante numa próxima secção, o que irá obrigar à implementação de um método **toString()**.

* + 1. **Uma Opção de Descendência**

Para os métodos de pesquisa utilizados, iremos estar sucessivamente a aplicar deslocações da célula vazia segundo um conjunto de movimentos legais: cima, *baixo*, *direita* e *esquerda*. Cada movimento irá alterar a configuração atual, gerando configurações descendentes/vizinhas sendo cada uma delas, formada por cada um dos movimentos legais quando aplicados à célula vazia.

Sendo assim, por questões futuras no cálculo de descendentes, a classe poderá implementar um método **legalDescendents()** onde irá retornar uma “lista” (idealmente poderá ser uma *LinkedList<Table>*) contendo todos os descendentes legais da configuração atual. Notar que ao serem legais, só poderão ser no máximo três descendentes (só três movimentos) isto porque, ao chegar a uma configuração por um movimento x, se, na criação dos seus descendentes, aplicar um movimento inverso/simétrico a x irei regressar à configuração que me levou à origem da configuração atual, levando à criação de um ciclo.

* 1. **Nodes, Arrays, Matrizes:**

Depois da ideia geral dos métodos a implementar da classe Table, resta definir a melhor forma de a representar, a melhor organização de memória e de **abstração**. Como iremos ver, a melhor forma de implementar com objetivo de reduzir o custo de memória é utilizando um *array*, no entanto, a forma mais ideal, sem complicações em cálculos de posições é uma matriz.

* + 1. **Vendo Tabelas como Nós interligados**

Uma primeira ideia seria utilizar a classe Table como duas classes, uma com o mesmo nome e uma classe **Node** sendo que, na classe Table estariam implementados todos os métodos apresentados anteriormente, com modificações para acomodar a nova classe e incluindo um array *Node[] board = new Node[16]*, sendo este um array de 16 nós (um para cada célula)

O objetivo da classe Node seria representar uma célula, guardar as suas posições na tabela (linha e coluna) bem como possuir ligações para todos os seu quatro vizinhos, a célula acima, a célula abaixo, a célula à direita e a célula à esquerda, tendo em conta que, se a célula vizinha estivesse fora do tabuleiro, teria valor *Null*. A *Figura 5* é a representação gráfica desta ideia.

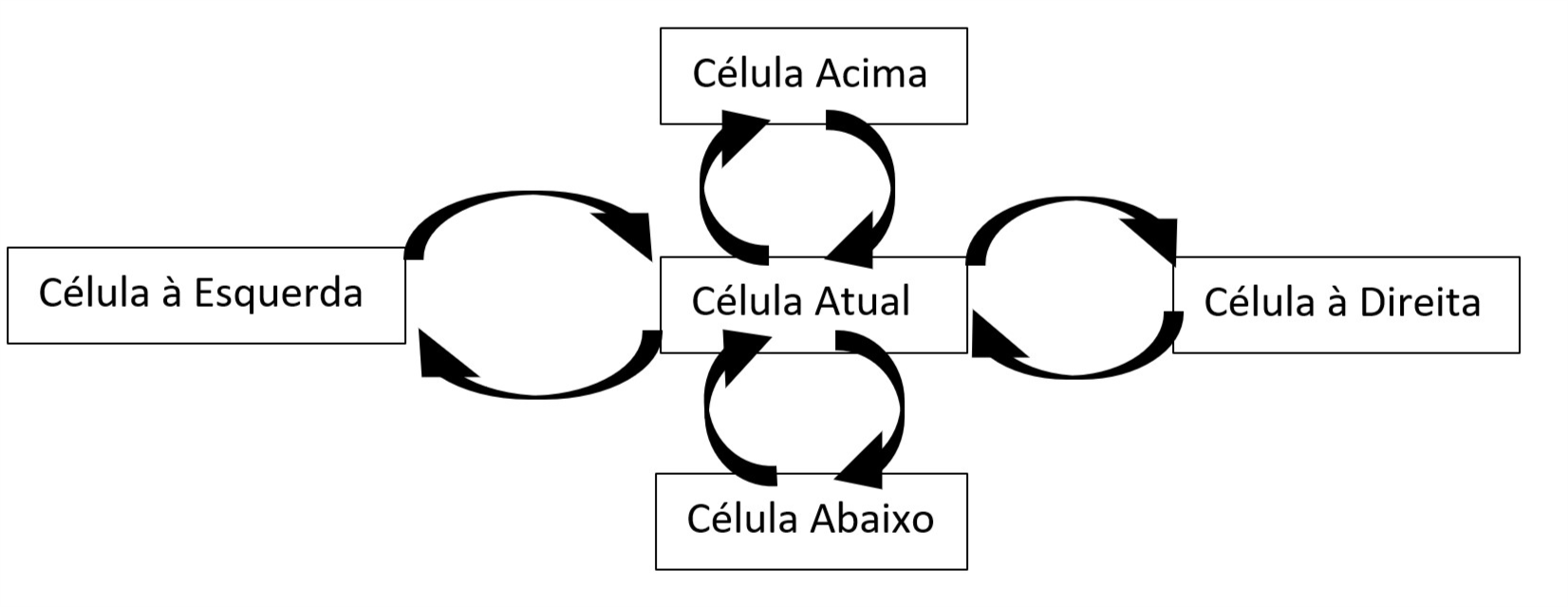


Figura 5-Representação gráfica da classe Node

Uma representação geral da classe Node seria a seguinte:

public class Node{

Node up,down,left,right; *//The neighbor nodes/cells*

int row,col,val; *//the positions in the matrix of this cell and its value*

public Node(int row,int col,int val) *//the constructor, takes the matrix positons and the cell value and creates the Node.*

public boolean isBlank()

public int getRow() | public int getCol() | public int getVal()

public Node getUp() | public Node getDown()

public Node getLeft() | public Node getRight()

private void setRow(int row) | private void setCol(int col)

private void setVal(int val)

private void setUp(Node cell) | private void setDown(Node cell)

private void setRight(Node cell) | private void setLeft(Node cell)

private void moveUp() | private void moveDown()

private void moveRight() | private void moveLeft()

*// | is only to delimt the methods*

}

Esta representação, apesar de facilitar a deslocação na tabela e comparação com outras tabelas, não deverá ser tomada não só pelo facto de ter uma implementação mais avançada mas também pelo facto de ocupar demasiado espaço.

Assumindo que o sistema operativo a rodar o programa é um sistema de 32 bits, e que, cada variável tem atribuído um registo tendo este 4 *bytes*, a classe Node necessitaria de 28 *bytes* para poder armazenar os seus valores.

Considerando a implementação da classe Table adaptada para a classe Node, esta iria ter um *array* *Node[] board=new Node[16]***,** em termos de memória um *array* de Nodes necessitará de 28 *bytes* para cada posição, 16\*28=448 *bytes* apenas para armazenar valores, mais 12 *bytes* como *header*do *array* e ficaríamos com 460 *bytes* necessários apenas para implementar a classe Node, fora outras variáveis necessárias à classe Table, o que, quando são aplicados métodos de pesquisa exponenciais de tal forma que aumentam drasticamente o número de tabelas/configurações, a quantidade de memória requerida para esta implementação faz com que esta ideia não seja viável.

* + 1. **Arrays Representando Tabelas**

Após a tentativa anterior e a necessidade para uma estrutura fácil de consultar e com pouco impacto na memória, surge a hipótese de visualizar toda a tabela como um *array*, a *Figura 1* representa esta ideia.

A vantagem deste processo vem do baixo consumo de memória, considerando um sistema operativo de 32 *bits* criar um *array* de 16 posições ocuparia um espaço na memória de 16\*4 + 12= 76 *bytes* sendo este o menor valor em qualquer um dos processos apresentados neste trabalho.

No entanto, utilizar uma *array* impõe novas dificuldades, nomeadamente a perda da informação de linhas e colunas para cada célula o que dificulta todos os métodos a implementar, requerendo novas verificações. Uma abordagem para tentar solucionar este problema seria ver o *array* como uma matriz 4\*4 de dimensão 1 (o que na verdade é) isto significaria o seguinte:

* Todos os valores em posições de índice entre 0 e 3 estariam na linha 0
* Todos os valores em posições de índice entre 4 e 7 estariam na linha 1
* Todos os valores em posições de índice entre 8 e 11 estariam na linha 2
* Todos os valores em posições de índice entre 12 e 15 estariam na linha 3

Para contabilizar as colunas bastava, num ciclo que percorresse o *array* todo, manter uma variável iniciada por 0 (coluna 0) e incrementada em cada ciclo, até a um valor máximo de 3 (coluna 0) e facilmente se encontrava os valores da coluna para cada célula

* + 1. **Uma Matriz para uma Tabela**

A nossa última e terceira opção explorada seria na classe Table manter uma matriz 4x4, em termos de uso de memória, considerando um OS de 32 *bits*, criar a matriz seria o mesmo que criar 4 *arrays* de tamanho 4, logo ocupariam (4\*4 +12)\*4 = 112 *bytes* mais 4 registos de matriz para endereçamento do *array* mais os 12 *bytes* do *header* da matriz, sendo o total, ((4\*4+12)\*4 +4\*4 +12)= 140 *bytes*

Esta implementação, apesar de não gastar muito espaço, rapidamente pode consumir demasiada memória que o método de pesquisa for em grande profundidade, como tal, podemos colocar medidas de diminuição de espaço ocupado, uma delas é operar em *arrays* e só se a configuração for válida (solucionável e não repetida) é que passamos à criação de um novo objeto Table.

1. **Os Métodos de Pesquisa**

O objetivo desta secção é apresentar os vários métodos e cuidados a ter quando implementar os métodos de pesquisa.

* 1. **Uma Comparação Eficiente e Um Caminho Contínuo**

Um dos grandes problemas em eficiência temporal é a comparação de matrizes, para tal, basta, aquando a criação de objetos de Table, criar uma *string* que represente a configuração, convertendo a tabela para *array* como demonstra a *Figura 1* e por sua vez converter para *string*, assumindo que o mesmo acontece para todos os objetos de Table, comparar duas tabelas é tão simples e **linear** como str\_*table1.equals(str\_table2)* daí a importância de um método *toString()* referido na secção 1.1.5.

Abordando agora um método de pesquisa, BFS, em cada iteração do problema é necessário verificar não só se a tabela obtida é solucionável mas também se não é repetida, uma solução linear para o problema é guardar uma ***HashTable*** onde, como chaves temos as *strings* de cada tabela, comparar fica então linear, basta encontrar uma chave ou percorrer a lista de chaves e comparar. O facto que torna este método verdadeiro é a inclusão da célula vazia na *string* garantindo que, por exemplo, [1,0,2], [0,1,2] e [1,2,0] sejam respetivamente “102”, ”012” e “120” e não apenas “12”.

A ideia final, suportada em utilização de *strings* e *HashTables* é o caminho para a solução correta. O caminho é uma série de movimentos “up”, “down”, “right” e “left”. Uma vez que vamos adicionando tabelas **válidas** à *HashTable* podemos adicionar a comfiguração inicial também, sendo o primeiro indíce, de chave a *string* da configuração e valores a *string* vazia, “”, e o valor *Null* em cada iteração, para cada tabela válida, podemos, se esta ainda não existir, adicioná-la à *HashTable* com chave a sua configuração e valores, a *string* caminhoPai() concatenada com o movimento realizado e uma outra *string* que representa o índice para o “nó” pai.

Quando chegarmos a uma solução, basta imprimir o seu valor.caminho() para obter todos os movimentos que levam à solução em tempo ótimo para o método utilizado.