Introdução:

O jogo do 15 é um enigma que tem como objetivo a ordenação numérica de 15 peças, pudendo o jogador movimentar uma peça de cada vez através de um único espaço vazio. O praticante apenas pode movimentar em quatro direções (cima, baixo, esquerda, direita). Devido ao facto de ser um enigma complexo, nem todas as configurações iniciais conseguem ser resolvidas.

A interpelação para a resolução do problema pode ser feita de forma algorítmica, e quanto mais rápido e otimizado for esse algoritmo, melhor será para chegar à configuração final do quebra-cabeças.

Neste caso, iremos implementar diversas operações e cálculos computacionais tais como: *Depth first search* (busca em profundidade), *Breadth-first search (*busca em largura), *Iterative deepening depth-first search (*busca iterativa em profundidade), *Greedy search (*busca gulosa) e *A-star search (*busca A estrela).

Métodos de pesquisa

Os métodos que iremos abordar podem ser divididos em duas categorias, a pesquisa não-informada e a pesquisa informada. Os pilares da nossa procura são as árvores de pesquisa, pois ao longo da busca, os nós serão criados para pudermos atingir o objetivo final.

*Depth First Search (DFS)*

Comecemos por um tipo de pesquisa não informada denominada de DFS ou busca em profundidade, que consiste na expansão do primeiro nó até ser encontrado o seu alvo ou então até a um nó que não tenha filhos, isto é, um nó folha. Nesse caso o algoritmo irá fazer *backtrack,* ou seja, irá retornar até ao nó anterior que percorra um caminho que ainda não tenha sido visitado anteriormente. Como vemos, este algoritmo não é muito otimizável, pelo facto de puder percorrer um caminho muito profundo em que não haja solução, havendo noutro nó um caminho menos profundo e que nos leve até à resposta, perdendo assim tempo de execução e levando ao gasto de memória.

O DFS, em termos de complexidade espacial, fica abaixo do BFS. Já no caso de complexidade temporal, depende do número de nós e caminhos que atravessa.

# Complexidade temporal:

Complexidade espacial: O(b x p)

b = fator de ramificação e p = profundidade máxima



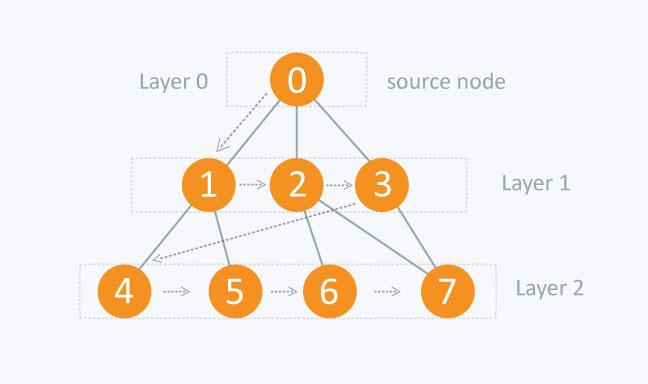
*Breadth First Search* (BFS)

O BFS, também denominada de busca em largura, é um tipo de busca não informada em que ao contrário da DFS, faz a sua pesquisa a partir de níveis, isto é, começando pela raiz, vai procurando em cada nível de profundidade todos os nós lá pertencentes , caso não encontre nenhum passa para o nível seguinte, e continua este processo até encontrar o nó almejado. Através deste algoritmo podemos garantir que qualquer nó será visitado, pudendo encontrar assim mais que uma solução, dando prioridade sempre ao nó que se encontra a menor profundidade da raiz. Como tinha dito anteriormente, o BFS terá uma complexidade espacial superior à do DFS caso haja um elevado grau de ramificação de nós num determinado nível profundidade, levando ao gasto acentuado da memória.

# Complexidade temporal:

# Complexidade espacial:

b = fator de ramificação



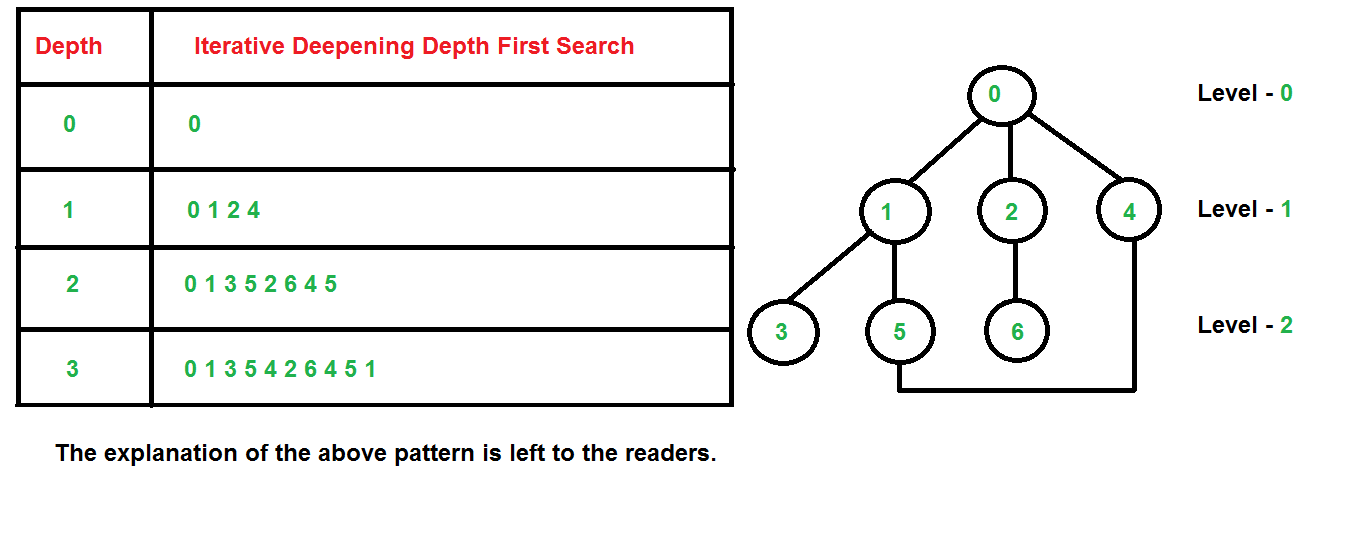
*Iterative Deepening Depth First Search* (IDFS)

A IDFS, chamada também de busca em profundidade iterativa, é um algoritmo na qual junta as qualidades da DFS e da BFS, que são, o pouco gasto de memória e a habilidade de pesquisar todos os nós da árvore até encontrar a solução mais otimizável. A IDFS corre as diversas instâncias de um DFS, mas progressivamente limitado pela profundidade, isto é, no primeiro nível, o limite será um, no segundo nível, o limite já será dois, e assim sucessivamente até atingir o seu limite máximo. A estratégia é sublime, mas não perfeita, pelo facto de depender do limite da profundidade, ou seja, caso o limite seja menor que a profundidade, o algoritmo poderá não encontrar solução.

# Complexidade temporal:

# Complexidade espacial: O(b x l)

# b = fator de ramificação e l = limite



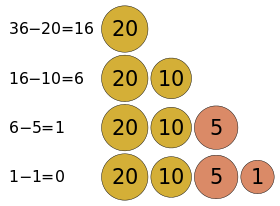
*Greedy Search*

Agora iremos abordar os tipos informados, isto é, tem acesso a conhecimentos específicos do problema que será tratado, tendo como objetivo principal o caminho mais próximo da solução.

Primeiramente, iremos falar da busca gulosa, que consiste numa procura que utiliza uma heurística para minimizar o custo da resposta. O algoritmo vai se movimentando ao longo da árvore, escolhendo sempre aqueles nós que não foram ainda visitados e que tenham uma menor heurística. Como heurística utilizaremos a distância de *Manhattan*, esta tem como seu propósito, o cálculo deste intervalo é feito através da soma da distância horizontal e vertical de cada peça na sua posição atual relativamente àquela que queremos chegar. A heurística não é uma função monótona fazendo com que este algoritmo não seja completo nem ótimo.

# Complexidade temporal:

b = fator de ramificação e p = profundidade máxima

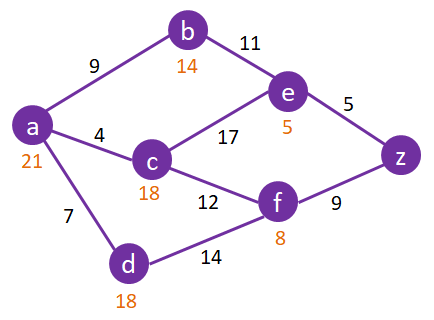


*A-Star Search*

Por último, iremos abordar o algoritmo de busca A estrela, esta pesquisa traduz-se pela combinação de outros algoritmos de pesquisa informada, o *Greedy* e do algoritmo de *Dijkstra*, este último tem como objetivo descobrir o caminho mais curto para um determinado nó, usando como heurística a distância entre, o nó onde se encontra e a raiz. Este algoritmo é ótimo e completo, mas pelo facto de ter que visitar todos os nós da árvore, torna-se ineficiente. Já no caso da *A-Star,* usando a distância até à raiz juntamente com *Greedy* como função de heurística, o algoritmo consegue tornar-se ótimo, completo e efetivo.

# Complexidade temporal:

b = fator de ramificação e p = profundidade máxima



Linguagem Implementada

A linguagem de programação utilizada para a realização deste projeto foi Java, pelo facto de esta ser uma linguagem orientada a objetos, o que permite fatorar em partes relativamente independentes para serem viáveis, tendo sido feito esse processo através de classes e objetos.

Estruturas utilizadas

*Linked List*

A *linked list*, à qual também podemos chamar de lista ligada, é uma estrutura de dados linear e dinâmica, que foi usada por nós, pelo facto de não ser necessário definir, ao momento da criação da mesma, o número máximo de elementos que esta poderá conter. Devido ao seu dinamismo, a inserção e remoção de elementos da lista não implica a mudança de lugar dos outros elementos, o que permite com que o algoritmo seja mais eficiente para o caso.

*Queue e PriorityQueue*

A *PriorityQueue* em java provém da *Queue*, que por sua vez, provém da *Linked List*, logo todas as vantagens também se aplicam ás *Queues. N*ós usamos este tipo de lista, pelo facto de esta receber pela ordem FIFO(*first in, first out*) os nós da árvore. No caso da *PriorityQueue*, esta foi usada para nós pudermos organizar os nós por custo crescente, ou seja, na 1 posição da *Queue* se encontra o objeto com menor custo. A técnica utilizada nesse processo foi uma *heap* de mínimos.

HashMap

Solvabilidade

Para testar a solvabilidade temos primeiro que transformar a tabela num vetor, ou seja, colocar os números em linha.

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

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

Primeiramente verificamos se o *input* tem 16 valores e que não são repetidos. Após isto temos que verificar, para cada número, quantas inversões existem. Por exemplo, se o número 12 estiver no canto superior esquerdo terá 11 inversões porque todos os números seguintes serão menores que ele.

Sendo a nossa matriz uma matriz com largura par, temos que ver onde se encontra o 0, ou espaço em branco.

* Se o quadrado em branco se encontrar numa linha par a contar do fim, o número de inversões da solvabilidade terá de ser ímpar
* Se o quadrado em branco se encontrar numa linha ímpar a contar do fim, o número de inversões da solvabilidade terá de ser par

Porque é que esta fórmula funciona?

Pegaremos, por exemplo, na tabela a baixo. O número de inversões é 49 e o quadrado branco está numa linha par a contar do fim. Ao mudarmos o 11 de lugar o número de inversões passa a ser 48, par, mas o quadrado em branco passar a estar numa linha ímpar a contar do fim, logo a nossa fórmula passa a estar correta.

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

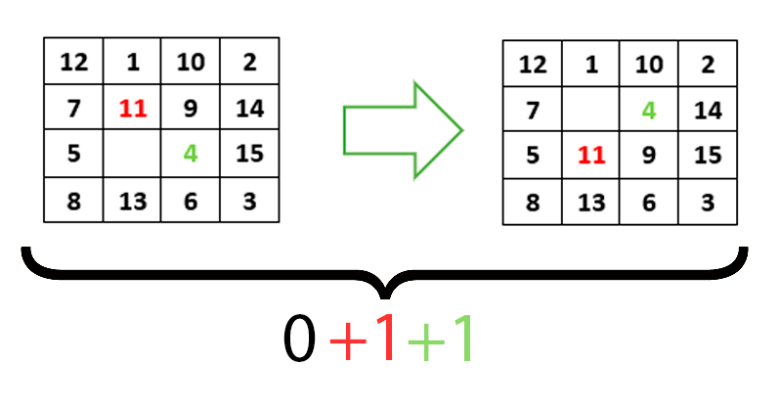
|  |  |  |  |
| --- | --- | --- | --- |
| **12** | **1** | **10** | **2** |
| **7** |  | **4** | **14** |
| **5** | **11** | **9** | **15** |
| **8** | **13** | **6** | **3** |

Heurísticas

As heurísticas servem para calcular os pesos das tabelas. Estes pesos servem de guia para os algoritmos de busca guiada (A\* e *greedy*).

Somatório

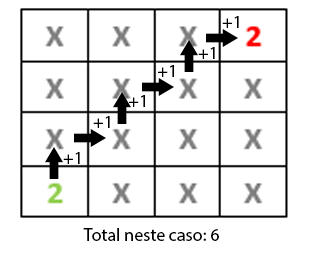
Esta heurística é criada a partir da quantidade de valores fora do sítio em relação à tabela final, como está representado na figura.



Manhattan distance

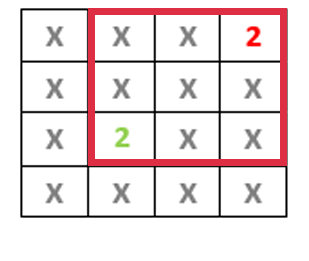
Esta heurística irá devolver o **somatório das distâncias** de cada valor para a sua casa final.

Por exemplo, se o número 1 estiver na posição (2,2) na tabela dada mas a sua configuração final é (1,1) precisará de 2 movimentos para a alcançar. Exemplo:



O cálculo da distância é feito da seguinte forma:

* Assumiremos a diferença das posições do valor em questão como um quadrado criado a partir de dois pontos (essas posições)



* Se x1>=x2, o valor total terá como valor x1-x2, se não, terá x2-x1 como valor.
* Se y1>=y2, ao valor total será adicionado y1-y2, se não, será adicionado y2-y1.