Jogo dos 15

Análise e criação de algoritmos

Inteligência Artificial

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:

* Pesquisa não-informada;
* Pesquisa informada;

Pesquisas Não-Informadas

Breve definição:

Pesquisas Não Informadas, também conhecidas como *blind search* ou *unguided search*, é uma classe de algoritmos de pesquisa padrão que opera através de força bruta. A pesquisa é não informada, pois não é dado ao algoritmo nenhum tipo de informação adicional sobre os estados do nó/objeto atual em relação ao nó/objeto final.

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** |

Depth First Search (DFS)

O DFS ou busca em profundidade, 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 *backtracking,* 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 eficiente, pelo facto de poder percorrer um caminho muito profundo em que não haja solução , apesar da existência de nós menos profundos que nos levam até à resposta, perdendo assim tempo de execução e levando a um gasto de memória excessivo.

O DFS, em termos de complexidade espacial, fica abaixo do BFS. Já a complexidade temporal estará diretamente depende do número de nós e caminhos que o algoritmo atravessa.

Complexidade do Algoritmo:

# Complexidade temporal:

Complexidade espacial:

**b** = fator de ramificação e **p** = profundidade máxima



Fig.1

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 múltiplas soluções , dando prioridade sempre ao nó que se encontra na menor profundidade.

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 do Algoritmo:

# Complexidade temporal:

# Complexidade espacial:

**b** = fator de ramificação

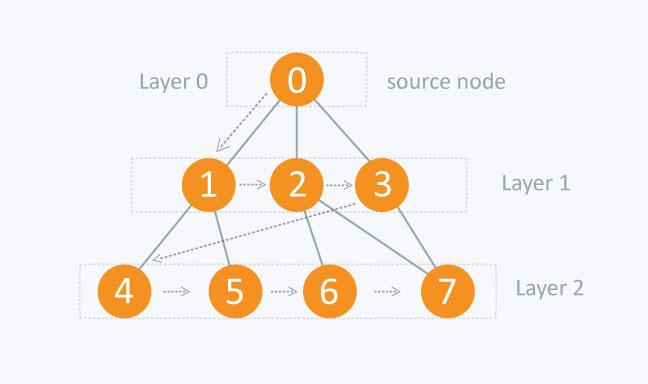
Exemplo da ordem de pesquisa do BFS:

Fig.2

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 do Algoritmo:

# Complexidade temporal:

# Complexidade espacial:

# b = fator de ramificação e l = limite

Exemplo do funcionamento do IDFS:

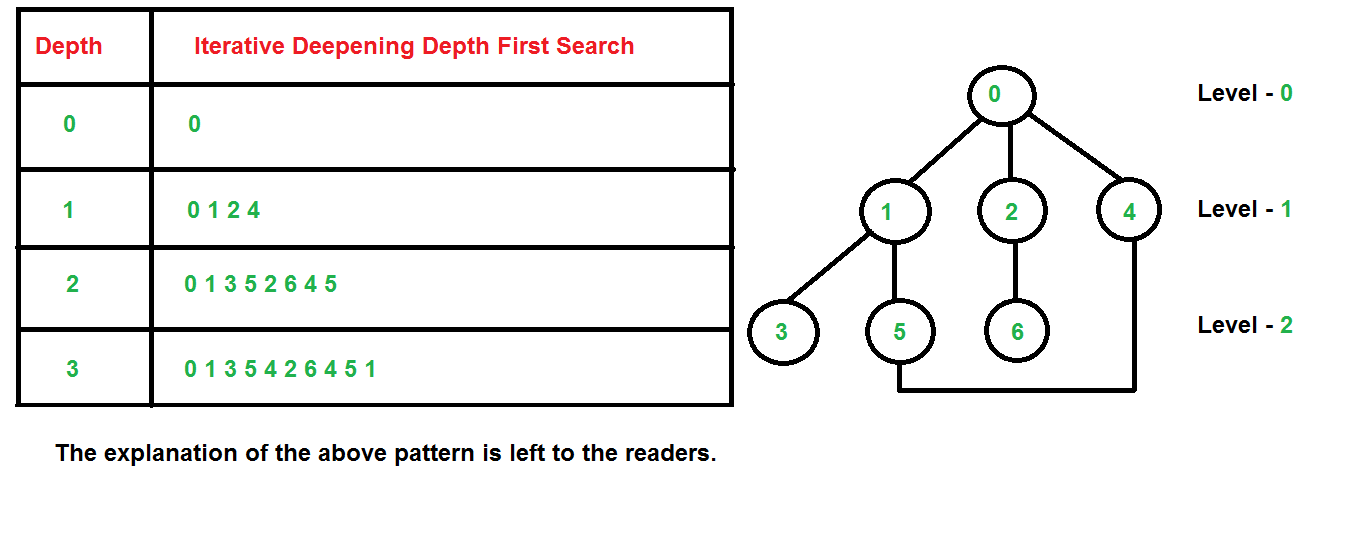


Fig.3

Buscas Informadas

Breve definição:

A Busca Informada é um tipo de pesquisa , que , ao contrário das Buscas Não-Informada, utiliza heurísticas ,para se guiar e chegar ao objetivo final . Neste trabalho utilizamos dois tipos de heurísticas: Manhattan Distance e Soma de Peças.

Greedy Search

A busca Greedy, 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 do Algoritmo:

# 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 do Algoritmo:

# Complexidade temporal:

b = fator de ramificação e p = profundidade máxima

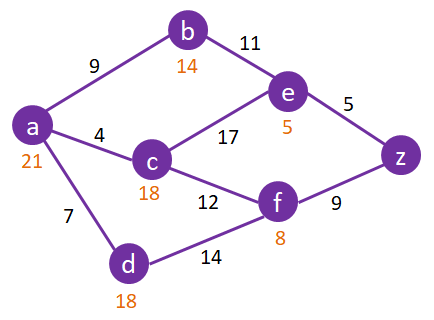


Fig.5

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*

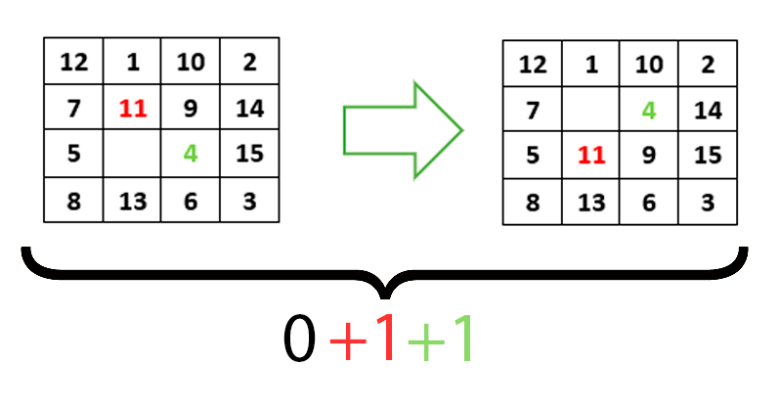
Em ciência de computação, uma *hashmap*, à qual também podemos denominar de “tabela de dispersão”, é uma estrutura de dados que associa chaves de pesquisa (*keys*) e valores. Este sistema tem como objetivo, a partir de uma chave simples, fazer uma pesquisa rápida e obter o valor almejado. Um dos motivos pela qual utilizamos esta estrutura foi, pelo facto, desta ter a capacidade de indexar grandes volumes de informação, outro dos motivos foi, devido à possibilidade de guardar um valor baseado numa chave que pode ser usada mais tarde para retirar esse mesmo valor.

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.

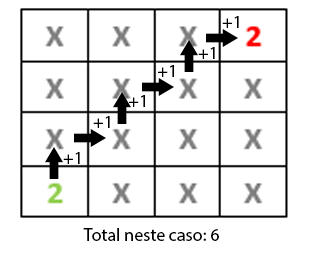


Neste exemplo apenas estão 2 valores trocados em relação à tabela final, por isso o peso que a tabela teria nesta heurística seria 2.

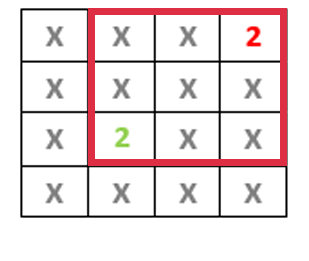
Manhattan distance

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.

Resultados

Testamos o nosso programa com vários *inputs* mas apresentaremos aqui três deles.

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

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Estratégia | | Tempo (ms) | Espaço | Profundidade da solução | Nº Nós visitados |
| BFS | | 73 | 9Mb | 12 | 5783 |
| DFS (Depth 80) | | 200 | 31Mb | 78 | 910269 |
| IDFS | | 92 | 1Mb | 12 | 5552 |
| GREEDY | Somatório | 47 | 1Mb | 50 | 683 |
| Manhattan | 46 | 5Mb | 16 | 108 |
| A\* | Somatório | 22 | 296Kb | 12 | 48 |
| Manhattan | 39 | 1MB | 12 | 39 |

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

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Estratégia | | Tempo (ms) | Espaço | Profundidade da solução | Nº Nós visitados |
| BFS | | 107 | 8MB | 12 | 10525 |
| DFS (Depth 80) | | 1370 | 137Mb | 76 | 11522632 |
| IDFS | | 123 | 10Mb | 12 | 9105 |
| GREEDY | Somatório | 55 | 590Kb | 22 | 93 |
| Manhattan | 51 | 1Mb | 14 | 20 |
| A\* | Somatório | 31 | 589Kb | 12 | 115 |
| Manhattan | 58 | 1MB | 12 | 58 |

|  |  |  |  |
| --- | --- | --- | --- |
| **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** | **5** | **11** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Estratégia | | Tempo (ms) | Espaço | Profundidade da solução | Nº Nós visitados |
| BFS | | 158 | 9Mb | 13 | 13790 |
| DFS (Depth 80) | | 408 | 113MB | 79 | 2260561 |
| IDFS | | 139 | 14Mb | 13 | 8718 |
| GREEDY | Somatório | 51 | 295Kb | 13 | 16 |
| Manhattan | 33 | 2Mb | 19 | 47 |
| A\* | Somatório | 51 | 296Kb | 13 | 45 |
| Manhattan | 52 | 295Kb | 13 | 16 |

(DFS retirado no Teste 2 no Gráfico 1 por ser um valor demasiado alto, afetando assim a visualização das restantes informações)

Conclusão

Nos tipos de busca não-informados, o BFS é o mais rápido, mas não fica muito longe do IDFS. Isto acontece porque a forma de busca por ambos é igualmente por camada, mas o IDFS acaba por gastar mais memória porque volta a ler os nós que visitou anteriormente. O DFS é de longe o mais pesado destas pesquisas e o que visita mais nós, devido à sua profundidade.

Nos tipos de busca informados a procura pelo resultado é incrivelmente mais linear e mais rápida. Isto acontece porque estes tipos utilizam heurísticas, que servem para atribuir pesos nas operações feitas pelo programa, ou seja, o programa não complicará o jogo pois não lhe interessa ficar mais longe do resultado final, a menos que perceba que não há alternativa.

Quanto às heurísticas usadas nestes últimos tipos, têm algum impacto dependendo se formos a utilizar Greedy ou A-star. Em quase todos os casos que utilizamos, a heurística de somatório na busca Greedy teve uma maior profundidade da solução do que a de Manhattan, mas gastou menos memória, ou seja, com Manhattan temos quase sempre o resultado com menos jogadas, mas sempre mais rapidame

nte do que com o somatório. Na busca A-star costuma demorar menos tempo, usar menos memória mas encontra a mesma profundidade e visita mais nós.

Concluindo, as buscas informadas são mais complexas mas muito mais funcionais que as não-informadas. Neste jogo, se utilizarmos a busca Greedy, devemos optar pela heurística se Manhattan pois é mais precisa, mas se utilizarmos A-star, devemos de usar a heurística de somatório.

Referências

Solvability of the Tiles Game, *by Mark Ryan*

https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html

Breadth First Search, *by Prateek Garg*

https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS), *Rachit Belwariar*

https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/