**Inteligência Artificial**

Jogo dos 15

(Relatório)

Carlos Santos e Márcia Dias

9 Março 2019

ÍNDICE

[**1 Introdução** 2](#_Toc3052728)

[**2 Estratégias de Procura** 3](#_Toc3052729)

[**2.1 Procura não Guiada:** 3](#_Toc3052730)

[**2.1.1 Busca em profundidade (DFS - Depth-First Search)** 3](#_Toc3052731)

[**2.1.2 Busca em largura (BFS – Breadth-First Search)** 4](#_Toc3052732)

[**2.1.3 Busca iterativa limitada em profundidade (IDFS - Iterative Depth-First Search)** 4](#_Toc3052733)

[**2.2 Procura Guiada:** 5](#_Toc3052734)

[**2.2.1 O que é uma heurística?** 5](#_Toc3052735)

[**2.2.2 Busca gulosa (greedy)** 5](#_Toc3052736)

[**2.2.3 Busca A\*** 6](#_Toc3052737)

[**3 Descrição do problema de procura estudado e dois espaços de estados** 7](#_Toc3052738)

[**4 Descrição da implementação** 7](#_Toc3052739)

[**5 RESULTADOS** 8](#_Toc3052740)

[**6 Conclusão** 10](#_Toc3052741)

[**7 Bibliografia** 11](#_Toc3052742)

# **1 Introdução**

Um problema de busca/procura é a representação de uma relação binária na área da computação. Neste problema começamos com estado inicial e um final, onde o objetivo é chegar ao estado final a partir do estado inicial com a ajuda de uma função que vai mapear os estados.

O espaço de estados pode ser representado por uma árvore de procura onde cada nó é uma estrutura com, pelo menos, cinco componentes: a configuração (ou estado), o nó pai, a sua profundidade, o custo do caminho desde a raiz até ao próprio nó e o próprio caminho.

Neste trabalho implementámos um algoritmo para resolver o jogo dos 15 e, para isso, usámos algumas estratégias de busca, tais como:

* Busca em profundidade;
* Busca em Largura;
* Busca iterativa em profundidade;
* A\*;
* Busca gulosa com heurística;

Tanto a busca em profundidade como a busca em largura têm uma complexidade O(). Já a busca gulosa tem complexidade. Quanto a busca A\*, a sua complexidade depende do número de nós expandidos.

# **2 Estratégias de Procura**

Nesta secção abordamos cada uma das pesquisas utilizadas na implementação do jogo dos 15 de uma forma mais detalhada.

## **2.1 Procura não Guiada:**

### **2.1.1 Busca em profundidade (DFS - Depth-First Search)**

Este algoritmo realiza buscas mais focadas. Em vez de visitar um nível inteiro e depois passar ao próximo nível, este algoritmo faz ações sucessivas até alcançar o objetivo, não ter ações a serem executadas ou chegar a um estado que já foi visitado antes. Casa não tenha ações a serem executadas e tenha chegada a um estado já visitado, o algoritmo volta ao estado anterior e tenta outro caminho, ou seja, tenta uma ação alternativa.

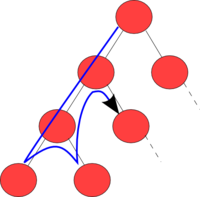


Figura 1-Busca em profundidade

Basicamente, este algoritmo progride através da expansão do primeiro nó filho da árvore de busca e, vai aprofundando cada vez mais até que a solução seja encontrada e ou então até o nó não ter mais filhos. Quando isso acontece a busca retrocede ao nível anterior e começa a partir do próximo nó filho e assim sucessivamente. Para isto, neste algoritmo são usadas stacks.

Este algoritmo é aplicado quando queremos encontrar componentes conectados, componentes fortemente conectados, quando queremos fazer uma ordenação topológica, ou resolver problemas de busca. Para problemas que têm múltiplas soluções a busca em profundidade pode ser boa, pelo facto de ter boa chance de encontrar uma solução depois de explorar um espaço pequeno do espaço total de busca. Entretanto ser for usada para pesquisar numa árvore com uma profundidade muito grande, pode demorar muito tempo ”de um lado da árvore ” quando na verdade que a solução pode estar ”do outra lado da árvore”. Caso a árvore seja infinita, pode não encontrar a solução.

A complexidade espacial deste algoritmo é O(bxd) e a complexidade temporal é . Onde d=profundidade, m=profundidade máxima e b=fator de ramificação.

### **2.1.2 Busca em largura (BFS – Breadth-First Search)**

Este algoritmo começa num nó definido pelo usuário e visita todos os nós vizinhos deste, ou seja, passa por todas as arestas e vértices de um grafo. Este algoritmo deve garantir que passa uma única vez por cada vértice e, para isto utiliza-se as filas, para garantir uma certa ordem, ao contrário da busca em profundidade que usa stacks. Caso o vértice já tenha sido visitado não entra na fila.

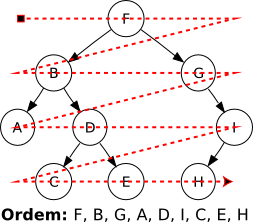


Figura 2- Busca em largura

Este algoritmo aplica-se quando queremos encontrar componente conectados, ou quando queremos todos os nós conectados a um só componente, ou quando queremos achas o menor caminho entre o nó raiz e outros nós do grafo e quando queremos testar bipartições.  
Este algoritmo garante que, se um problema tiver solução, é encontrada e, caso tenha múltiplas soluções, é encontrada a solução mais “raza”, ou seja, a solução ótima. Entretanto, ao ser usado numa árvore cuja solução esteja numa profundidade relativamente grande, acaba por consumir muita memória, visto que a quantidade de nós gerados é exponencial.

A complexidade espacial e temporal deste algoritmo é igual, . Onde d=profundidade e b=fator de ramificação.

### **2.1.3 Busca iterativa limitada em profundidade (IDFS - Iterative Depth-First Search)**

Esta busca visa combinar a busca em profundidade e a busca em largura, tendo um limite de profundidade. Este algoritmo começa com limite igual a um, caso a solução não seja encontrada a árvore anterior é eliminada e é feita uma nova com limite igual a dois. Se ainda não for encontrada a solução a árvore é novamente eliminada e é criada uma nova com profundidade três e este processo é repetido até ser encontrada a solução ou até a arvore ser toda gerada.

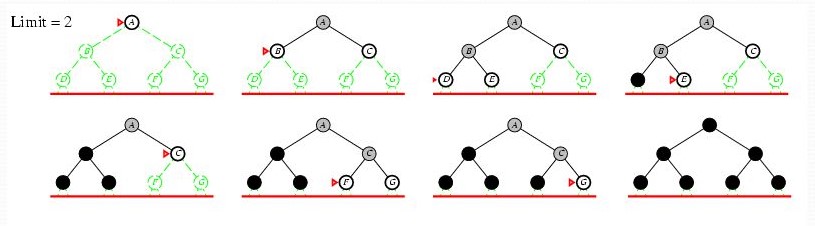


Figura 3- Busca iterativa limitada em profundidade

Este algoritmo tem as mesmas aplicações que a busca em profundidade. É um algorítmo completo.

A complexidade espacial deste algoritmo é O() e a complexidade temporal

## **2.2 Procura Guiada:**

### **2.2.1 O que é uma heurística?**

A heurística é um conjunto de regras e métodos que pretendem levar a inventar, descobrir ou resolver problemas.

### **2.2.2 Busca gulosa (greedy)**

A busca gulosa tenta expandir o nó que parece mais próximo do nó final com base na estimativa feita usando uma heurística. Para além disto tenta minimizar o custo estimado para chegar à solução.

Este tipo de busca é aplica quando queremos o custo mínimo numa árvore, ou quando queremos o caminho mais curto.

Exemplos de algoritmos gulosos são o de Prim e o de Kruskal.

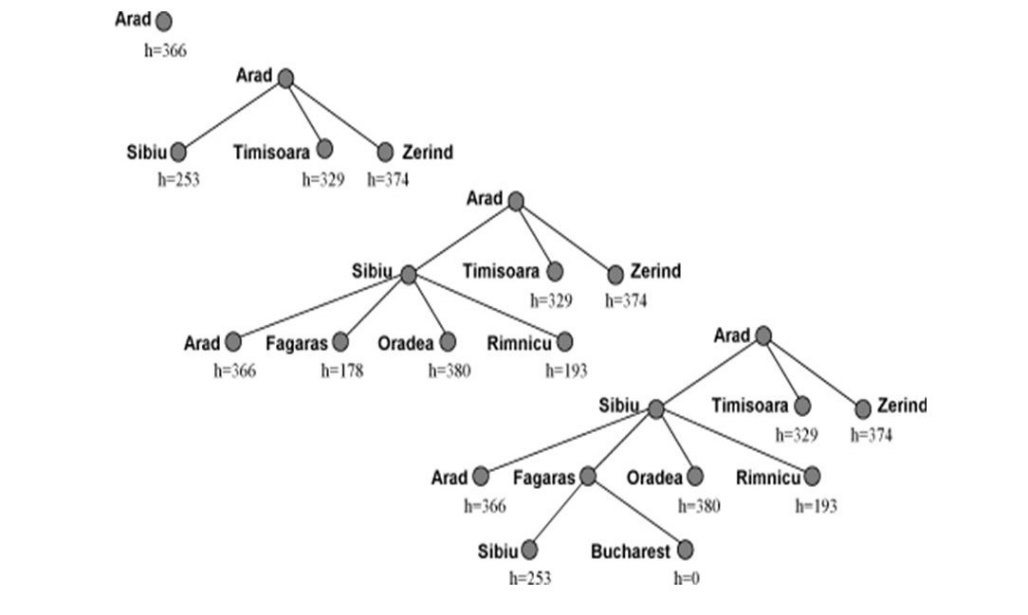


Figura 4- Busca Greedy

### **2.2.3 Busca A\***

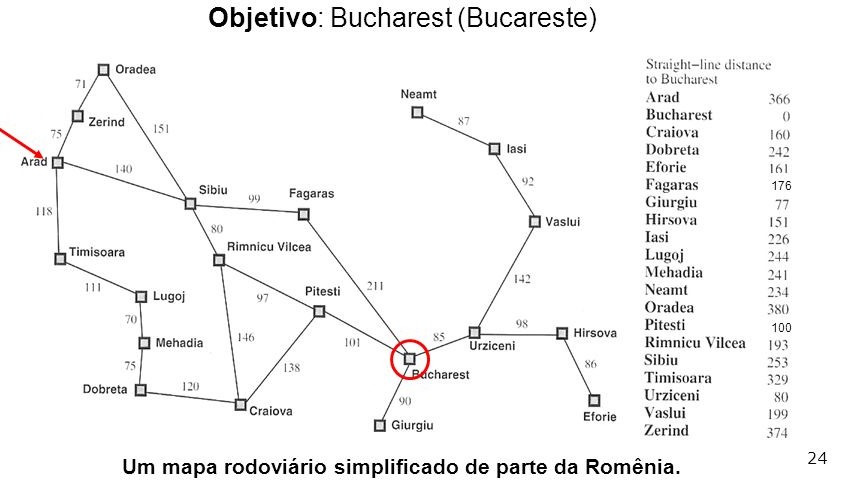
Este algoritmo faz a busca desde um nó inicial até um nó final. Este algoritmo é a combinação de aproximações heurísticas com a busca em largura. Aqui o objetivo é que o custo do próximo nó não seja maior que o anterior. Esta é uma busca ótima, completa e eficiente.

Figura 5- Busca A\*

# **3 Descrição do problema de procura estudado e dois espaços de estados**

O jogo dos 15 é um quebra-cabeças formado por um tabuleiro (matriz) 4x4. Esta é constituída por 16 células das quais 15 estão numeradas, por uma certa ordem e uma das células está em branco (normalmente representado por 0).

O problema consiste em partir de uma configuração inicial e chegar à configuração final dada. A célula em branco permite-nos mover todos as outras, uma a uma. Apenas podemos fazer movimentos para cima, para baixo, para a esquerda e para a direita.  
Neste jogo, nem sempre podemos chegar à configuração final a partir da configuração inicial.

# **4 Descrição da implementação**

Decidimos implementar o jogo em java. Escolhemos esta linguagem uma vez que tem muitas funções já implementadas e por ser uma linguagem multiplataforma, o que nos facilitou na construção do trabalho. Como é uma linguagem de programação orientada a objetos, permite-nos fazer a reutilização do código.

Com base no problema que nos foi apresentado e nos algorítmos implementados, usámos como estruturas de dados Matriz, Array, Stack, LinkedList, PriorityQueue, HashSet, e uma classe Node.

Os Arrays e as Matrizes para representar as configurações inicial e final. O uso das listas, filas e listas de prioridade acaba por estar diretamente ligado às estratégias de busca implementadas. Sempre que era criado um nó, era preciso verificar se o mesmo já existia, evitando assim que os nós fossem visitados mais que uma vez. Por isso, usámos uma HashSet para guardar os nós já visitados, uma vez que essa estrutura não permite a repetição de elementos. A classe Node por sua vez, foi usada para representar os estados do jogo.

Optámos por dividir o nosso código em três ficheiros:

* **game.java**, contendo a função main, onde é verificado se a configuração dada tem ou não solução. Caso tenha solução, é chamado o tipo de busca a ser efetuado para econtrar a solução.
* **Tabuleiro.java**, contendo todas as buscas implementadas (DFS, BFS, IDFS Greedy e AStar). Cada uma dessas buscas chama uma função auxiliar que gera os descendentes e caso encontre uma solução, é chamado a função encontrou. As funções que geram descendentes, por sua vez chama a função mover. No caso das funções MakeDescendetsG e MakeDescendentsA, é chamado uma heurística (heuristic e heuristicA) que calculam o custo de cada nó descendente usando a distância de Manhattan.
* **Node.java**, contendo a classe Node referido anteriormente.

# **5 RESULTADOS**

Teste 1 - Configuração Inicial

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

Teste 1 - Configuração Final

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégias | Tempo(segundos) | Encontrou solução? | Espaço | Profundidade/Custo |
| DFS |  | Não |  |  |
| BFS | 0.404s | Sim | 205.833MB | 12 |
| IDFS | 0.707s | Sim | 193.314MB | 16 |
| Greedy | 0.045s | Sim | 192.362MB | 16 |
| A\* | 0.012s | Sim | 192.363MB | 16 |

Teste 2 - Configuração Inicial

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

Teste 2 - Configuração Final

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégias | Tempo(segundos) | Encontrou solução? | Espaço | Profundidade/Custo |
| DFS |  | Não |  |  |
| BFS | 0.145s | Sim | 195.224MB | 10 |
| IDFS | 0.127s | Sim | 192.584MB | 10 |
| Greedy | 0.005s | Sim | 192.359MB | 10 |
| A\* | 0.006s | Sim | 192.357MB | 10 |

Teste 3 - Configuração Inicial

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

Teste 3 - Configuração Final

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégias | Tempo(segundos) | Encontrou solução? | Espaço | Profundidade/Custo |
| DFS | 0.001s | Sim | 192.344MB | 3 |
| BFS | 0.002s | Sim | 192.360MB | 3 |
| IDFS | 0.003s | Sim | 192.353MB | 3 |
| Greedy | 0.003s | Sim | 192.362MB | 3 |
| A\* | 0.005s | Sim | 192.342MB | 3 |

Teste 4 - Configuração Inicial

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

Teste 4 - Configuração Final

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Estratégias | Tempo(segundos) | Encontrou solução? | Espaço | Profundidade/Custo |
| DFS |  | Não |  |  |
| BFS |  | Não |  |  |
| IDFS |  | Não |  |  |
| Greedy | 0.082s | Sim | 193.170MB | 149 |
| A\* | 0.111s | Sim | 192.821MB | 149 |

# **6 Conclusão**

Analisando os resultados, podemos concluir que a procura em profundidade não é o ideal para resolver este tipo de problema, porque embora consiga encontrar a solução em pouco tempo, quando temos uma árvore com profundidade relativamente grande (como é o caso do jogo dos 15) não encontra a solução, para além do fato de consumir muitos recursos (tempo e memória). Limitando a profundidade até a qual pode ir, torna-se mais eficiente, entretanto quando a profundidade da solução é maior que o limite predefinido, a mesma não é encontrada.

Já a pesquisa em largura encontra sempre a melhor solução, entretanto quando a árvore de pesquisa tem uma profundidade relativamente grande, gasta muitos recursos, uma vez que a quantidade de nós gerados é exponencial à profundidade da árvore.

Podemos ainda concluir que quando o objetivo é ter o melhor desempenho a nível de tempo de execução e memória gasta, as estratégias guiadas (Greedy e A\*) são as melhores, apesar de nem sempre encontrarem a melhor solução.

# **7 Bibliografia**

<http://www.dcc.fc.up.pt/~ines/aulas/1819/IA/buscas_nao_informadas.pdf>

<https://pt.wikipedia.org/wiki/Busca_em_profundidade>

<https://pt.wikipedia.org/wiki/Busca_em_largura>

<http://www.dcc.fc.up.pt/~ines/aulas/1819/IA/buscas_informadas_new.pdf>

https://dicionario.priberam.org/

[https://en.wikipedia.org/wiki/Greedy\_algorithm#Applications](https://en.wikipedia.org/wiki/Greedy_algorithm)

<https://pt.wikipedia.org/wiki/Heur%C3%ADstica_(computa%C3%A7%C3%A3o)>