**RELATÓRIO**

**Inteligência Artificial**

****

****

**JOGO DOS 15**

**André Cirne - 201505860**

**José Sousa - 201503443**

Índice

[Introdução 3](#_Toc476081297)

[Estratégias de Procura 3](#_Toc476081298)

[Procura não guiada 3](#_Toc476081299)

[Profundidade 3](#_Toc476081300)

[Largura 4](#_Toc476081301)

[Busca Iterativa Limitada em Profundidade 4](#_Toc476081302)

[Procura guiada 4](#_Toc476081303)

[Gulosa 5](#_Toc476081304)

[Busca A\* 5](#_Toc476081305)

[Descrição da Implementação 5](#_Toc476081306)

[Linguagem utilizada 5](#_Toc476081307)

[Estruturas de dados utilizadas 6](#_Toc476081308)

[LinkedList 6](#_Toc476081309)

[PriorityQueue 6](#_Toc476081310)

[HashSet 6](#_Toc476081311)

[Considerações de implementação 6](#_Toc476081312)

[Resultados 6](#_Toc476081313)

[Conclusão 6](#_Toc476081314)

[Referências 8](#_Toc476081315)

# Introdução

O jogo dos 15 é um quebra-cabeças com objetivo de dadas duas configurações, inicial e final, movimentar a peça vazia até obter a configuração final, utilizando unicamente os movimentos permitidos (cima, baixo, direita e esquerda) (1). Devido a essência do problema nem todos os pares de configuração inicial e configuração final são solucionáveis.

Uma das abordagens que podemos tomar a este problema, é tentarmos obter a configuração final de forma algorítmica a partir da inicial. Num caso ótimo obter essa configuração no menor custo possível. Para a resolução deste problema podemos utilizar algoritmos de pesquisa, os quais vão percorrer todo o espaço de procura na tentativa d encontrar a configuração final. Neste sentido, de forma experimental implementamos alguns, dos vários algoritmos que tentam responder a estes problemas. Analisando e comparando a complexidade temporal e espacial teóricas com os resultados amostrais. Os algoritmos que iremos a abordar são os seguintes: busca em profundidade, busca em largura, busca iterativa em profundidade, busca gulosa e busca A estrela.

# Estratégias de Procura

Na base dos algoritmos abordados, encontra-se a estrutura de uma árvore de pesquisa subentendida, onde os nós são gerados à medida que o método de pesquisa avança.

Como anteriormente foi referido, nem todas as configurações são solucionáveis logo antes de correr qualquer algoritmo tivemos que o verificar. Utilizando o calculo das inversões do tabuleiro e analisando a sua paridade quando comparado à paridade da posição em branco dá-nos a solvabilidade, do tabuleiro em relação a um determinado tabuleiro padrão. Se efetuarmos estes métodos para a configuração final e inicial e dos dois resultarem o mesmo valor de verdade, o problema é solucionável. (2) Do qual resulta a seguinte condição:

Podemos então a partir da configuração inicial, conseguir obter a final.

## Procura não guiada

### Profundidade

A busca em profundidade (DFS) consiste a partir da raiz expandir os nós mais profundos, até que o alvo da busca seja encontrado ou até que ele se depare com um nó folha, que no caso especifico deste problema será quando todos os novos nós que podemos gerar, já se encontram no caminho de busca. Perante esta situação dá-se o *backtracking*, para o nó mais profundo ainda não visitado.

O algoritmo não é ótimo, já que como dá prioridade aos nós mais profundos poderá encontrar uma solução num nível mais profundo, quando a solução ótima se encontra num nó ainda não explorado numa profundidade menor. Em relação à completude podemos entender que o algoritmo não é completo, já que em determinadas configurações, o algoritmo não nos consegue dar uma resposta em tempo útil.

Em contrapartida, para a impressão dos nós intermediários entre o nó inicial e folha, a busca em profundidade consegue utilizar menos memórias já que só é necessário guardar os nós do caminho percorrido da raiz ao nó folha.

Complexidade Temporal

Complexidade Espacial:

Onde: b = fator de ramificação e m = profundidade máxima

### Largura

A pesquisa em Largura(BFS) em comparação ao DFS muda a ordem de visita dos nós, ou seja, começando pela raiz explora todos os nós da mesma profundidade, até que não haja mais nenhum e avance para profundidade seguinte, repetindo o processo até encontrar o nó pretendido.

Podemos afirmar que o algoritmo realiza uma busca exaustiva, garantindo assim que todos os nós serão visitados, encontrando sempre uma resposta(completo). e como é dada prioridade aos nós da mesma profundidade a solução é sempre ótima.

Em termos de utilização de memória num determinado estado da pesquisa é necessário guardar todos os nos do mesmo nível, que ainda faltam visitar e os nós que já foram expandidos da profundidade seguinte. Dependendo do fator de ramificação e da profundidade da solução, o BFS pode ultrapassar o gasto de memória do DFS.

Complexidade Temporal:

Complexidade Espacial:

Onde: b = fator de ramificação e m = profundidade máxima

### Busca Iterativa Limitada em Profundidade

A busca em profundidade (IDFS) iterativa procura combinar as virtudes da busca em profundidade e em largura. Usa menos de memória que o BFS mas consegue examinar todo o espaço de busca. Este tipo de busca impõe um limite na profundidade máxima de um caminho.

O algoritmo consiste em correr várias instancias de um DFS progressivamente limitado em profundidade. Entende-se progressivamente limitado como na primeira instancia, o limite será um, na segunda dois assim sucessivamente até chegar ao limite máximo.

O IDFS é uma estratégia ótima, mas não completa já que dependendo do limite da profundidade, quando este é menor que a profundidade da solução, poderá não encontrar uma solução.

Complexidade Temporal:

Complexidade Espacial:

Onde: b = fator de ramificação e L = limite

## Procura guiada

As procuras guiadas, ao contrário dos métodos anteriormente apresentados, utilizam conhecimentos específicos ao problema a abordar, com o objetivo de ao longo da pesquisa avançar sempre por um caminho que seja mais próximo da solução (2), minimizando assim o custo estimado de chegar à solução.

Ao utilizar este tipo métodos de pesquisa tentamos atingir um numero menor de nós gerados e um tempo menor até chegar à solução.

### Gulosa

A procura gulosa é um tipo de procura guiada, que tenta minimizar o custo da solução, utilizado uma heurística.

O algoritmo movimenta-se no espaço de procura, escolhendo de todos os nós ainda não visitados aquele com a menor heurística. Na implementação deste algoritmo usamos a distancia de Manhattan como heurística. O calculo desta distancia é efetuado através da soma da distancia horizontal e vertical (3), de cada peça na posição atual em relação à posição final que pretendemos.

A procura gulosa apresenta uma complexidade temporal e espacial de .

Onde: b = fator de ramificação e m = profundidade máxima

### Busca A\*

A busca A\*, é a tentativa de combinar dois algoritmos a busca gulosa, que tenta estimar o custo para atingir a solução ótima, que na nossa abordagem usou distância de Manhattan, mas que tem como ponto fraco os anteriormente referidos e o algoritmo Dijkstra, algoritmo que descobre o caminho mais curto para um determinado nó, usando a distancia do nó atual à raiz como heurística. O *Dijkstra* é ótimo e completo, no entanto, ineficiente já que todos os nós têm de ser visitados. (4) (5)

Devido à utilização da função de heurística acima descrita, consegue ser ótimo, completo e eficiente.

A procura gulosa apresenta uma complexidade temporal e espacial de .

Onde: b = fator de ramificação e m = profundidade máxima

# Descrição da Implementação

## Linguagem utilizada

Desde do inicio do problema devido à forma como se encontra formulado o problema, a existência de tabuleiros, que consideramos importante que a linguagem utilizasse o paradigma de orientação ao objeto. Inicialmente começamos por tentar implementar o a solução ao problema do jogo dos 15 usando Python, já que é uma linguagem multiparadigma e assim poderíamos tirar também proveito de uma abordagem mais funcional. No entanto chegamos a conclusão de dois problemas, por um lado Python é uma linguagem interpretada “pouco lenta”, por isso acabamos por optar por Java, continua a ser uma linguagem orientada a objeto, não interpretada, e que apresenta um bom equilíbrio entre rapidez de código e alto nível.

## Estruturas de dados utilizadas

### LinkedList

Como não se sabe à partida qual será o número de elementos que vamos ter de guardar, esta é a estrutura de dado que consegue guardar um numero de elementos que só se encontra limitado pela memória física da máquina onde está a correr e que além disso é muito dinâmica, já que permite utilizar qualquer tipo de tipos de dados para formar uma LinkedList. As operações de remoção e de adição no inicio da estrutura com tempo contante, contribuindo assim para um algoritmo mais rápido.

### PriorityQueue

A PriorityQueue em java encontra-se construída com base na Queue, que por sua vez é construída com base na LinkedList, logo todas as características em relação a mais valias desta estrutura de dados, também se aplicam à PriorityQueue. Esta estrutura permite manter uma lista de forma automaticamente ordenada e garantido uma complexidade na ordenação/adição de (6),logo foi utilizada na busca A\* e busca Gulosa onde é essencial esta tarefa.

### HashSet

A utilização de HashSet’s, na nossa implementação deveu-se a dois factos. Por um lado, a complexidade de verificar se um determinado nó já se encontra na estrutura é de complexidade linear, o que é importante nos algoritmos onde é necessário verificar nós repetidos, além disto como não soa utilizados os números em si, mas sim as suas *hash’s*, possibilita um uso menor de memória.

## Considerações de implementação

Era pretendido neste trabalho, além de identificar o nível onde se encontra a solução também imprimir todo os nós percorridos para chegar a essa solução. Foi necessário encontrar um meio termo entre gasto de memória desnecessário e a capacidade de imprimir o caminho que deu origem à solução.

Utilizamos simplesmente apontadores para o nó pai, mantendo o número mínimo de nós em memória para que conseguíssemos voltar a gerar o caminho.

# Resultados



# Conclusão

Ao contrário do que originalmente se previa o algoritmo de pesquisa gulosa conseguiu gerando menos espaços e sobre sequentemente um menor tempo de execução, claro que este resultado deve ter acontecido devido ao input dado e o mais provável é que com outra configuração a pesquisa gulosa se tornar-se mais rápida.

No nosso código para inputs, mais profundos podia se ter efetuado uma otimização no a star, a utilização de de um HasMap auxiliar, para não voltar a gerar nos com o mesmo tabuleiro como base, mas com uma heurística maior.

# Referências

x

|  |  |
| --- | --- |
| 1. | O jogo do 15. [Online].; 2015 [cited 2017 Fevereiro 28. Available from: <https://pt.wikipedia.org/wiki/O_jogo_do_15>. |
| 2. | Wikipedia. [Online].; 2017. Available from: <https://en.wikipedia.org/wiki/Heuristic_(computer_science)#Newell_and_Simon:_heuristic_search_hypothesis>. |
| 3. | Black PE. Manhattan distance. [Online].; 2006 [cited 2017 Fevereiro 25. Available from: <https://xlinux.nist.gov/dads/HTML/manhattanDistance.html>. |
| 4. | Ribeiro P. DAA1617. [Online].; 2016 [cited 2017 Fevereiro 25. Available from: <http://www.dcc.fc.up.pt/~pribeiro/aulas/daa1617/slides/9_distancias_11122016.pdf>. |
| 5. | Eranki R. Pathfinding using A\* (A-Star). [Online].; 2002 [cited 2017 Fevereiro 25. Available from: <http://web.mit.edu/eranki/www/tutorials/search/>. |
| 6. | Oracle. PriorityQueue. [Online].; 2016 [cited 2017 Fevereiro 25. Available from: <http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html>. |

x