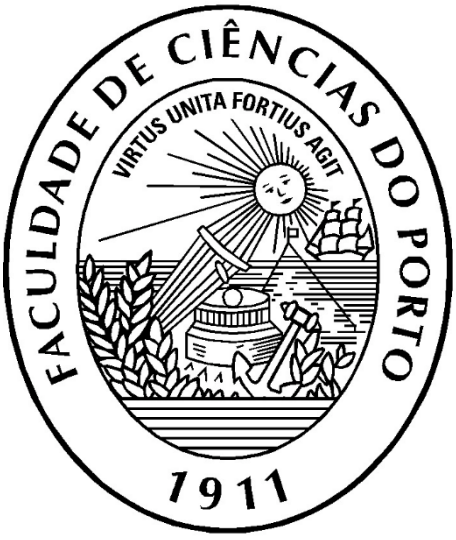
RELATÓRIO

Inteligência Artificial



JOGO DOS 15

André Cirne - 201505860
José Sousa - 201503443

Índice

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 à essência do problema nem todos os pares de configuração inicial e configuração final, são solucionáveis.

Uma das abordagens a este problema, é tentar 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 de encontrar a configuração final. Neste sentido, de forma experimental, implementamos alguns, dos vários algoritmos que tentam responder a estes problemas e analisamos os resultados obtidos.

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:

$$(Inv_f\%2 == 0) == (blankRow_f\%2 == 1) == (Inv_i\%2 == 0) == (blankRow_i\%2 == 1)$$

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 encontrem 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 memória, já que só é necessário guardar os nós do caminho percorrido da raiz, ao nó folha.

Complexidade Temporal: $O(b^d)$

Complexidade Espacial: $O(b\ x\ m)$

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 nós do mesmo nível, que ainda faltam visitar e os 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: $O(b^d)$

Complexidade Espacial: $O(b^d)$

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

Busca Iterativa Limitada em Profundidade

A busca em profundidade iterativa(IDFS) procura combinar as virtudes da busca em profundidade e em largura. Usa menos de memória que o BFS e 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 e 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: $O(b^L)$

Complexidade Espacial: $O(b\ x\ L)$

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 (3), minimizando assim o custo estimado de chegar à solução.

Ao utilizar este tipo métodos de pesquisa tentamos atingir um número 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 distância de Manhattan como heurística. O cálculo desta distância é efetuado através da soma da distância horizontal e vertical (4), de cada peça na posição atual em relação à posição final que pretendemos. Este cálculo faz com que esta heurística seja admissível, já que não ultrapassa o custo real da solução.

Como a função heurística não é monótona este algoritmo não completo nem ótimo. A procura gulosa apresenta uma complexidade temporal e espacial de $O(b^m)$.

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 distância 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. (5) (6)

Devido à utilização da distancia à raiz, em conjunção com a heurística da pesquisa gulosa como função heurística, a busca A* consegue ser ótima, completa e eficiente.

A procura A* apresenta uma complexidade temporal e espacial de $O(b^m)$.

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

Descrição da Implementação

Linguagem utilizada

Desde do início do problema, devido à forma como se encontra formulado, a existência de tabuleiros, considerámos importante que a linguagem utiliza-se o paradigma de orientação ao objeto. Inicialmente começamos por tentar implementar 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 à conclusão de um problema, por Python é uma linguagem interpretada levando a que a execução seja mais lenta quando comparada com outras linguagens. Por isso acabamos por optar por Java que continua a ser uma linguagem orientada a objeto, não interpretada, e que apresenta um bom equilíbrio entre rapidez de execução.

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 dados que consegue guardar um número de elementos, que só se encontra limitado pela memória física da máquina onde está a correr. Além disso é muito dinâmica, já que permite utilizar qualquer tipo de dados para formar uma LinkedList. As operações de remoção e de adição no início da estrutura com tempo constante, contribuem assim para um algoritmo mais rápido.

PriorityQueue e Queue

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, também se aplicam à PriorityQueue e Queue. A utilização da Queue foi mais para garantir proteção, já que esta estrutura de dados, só permite inserir no fim e retirar pelo inicio dados. A PriorityQueue permite manter uma lista automaticamente ordenada e garantido uma complexidade na ordenação/adição de $O(log\ n)$ (7). 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 são utilizadas as configurações dos tabuleiros 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 todos 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. Por isso utilizamos simplesmente apontadores para o nó pai, mantendo o número mínimo de nós em memória, para que conseguíssemos voltar a gerar o caminho.

Resultados

Foram efetuados 3 testes, com um limite de 60 segundos para algoritmos que não garantem completude e sempre com a seguinte configuração final:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Os resultados, encontram-se a seguir.

Tabela 1- Teste 1

Estratégia	Tempo (s)	Espaço		Optimabilidade	Custo
		Gerado	Visitado		
DFS	60	3056826	1567626	-	-
BFS	10,328	1312072	615947	Sim	17
IDFS	0,621	1789269	1789255	Sim	17
Gulosa	60	2806232	1401170	-	-
A*	0,024	228	110	Sim	17

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

Tabela 2-Teste 2

Estratégia	Tempo (s)	Espaço		Optimabilidade	Custo
		Gerado	Visitado		
DFS	60	3051314	1564798	-	-
BFS	-	-	-	-	-
IDFS	1181,206	-	-	Sim	27
Gulosa	60	2806232	1401170	-	-
A*	0,185	7142	3480	Sim	27

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

O BFS ficou sem memória.
O IDFS efetuou *overflow*, ao contador de nós.

Tabela 3-Teste3

Estratégia	Tempo (s)	Espaço		Optimabilidade	Custo
		Gerado	Visitado		
DFS	60	3091872	1585587	-	-
BFS	0,206	45347	21238	Sim	12
IDFS	0,066	38172	38162	Sim	12
Gulosa	0,012	46	23	Sim	12
A*	0,012	62	31	Sim	12

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

Conclusão

Como era de esperar ao nível experimental existiu grande diferenciação entre os métodos de procura guiada e não guiada.

Nas estratégias de procura não guiada destaca-se o BFS, já que consegue garantir um bom equilíbrio entre ser ótimo/completo e a rapidez na procura da solução. No entanto como efetua uma pesquisa exaustiva, o seu gasto de memória é maior quando comparado com todos os outros. Por outro lado, o IDFS consegue ter um menor gasto de memória com uma complexidade equivalente, mas é claro que não garante completude. A situação do BFS ter um gasto de memória superior ao IDFS foi evidente no teste 3, onde não consegui-o responder ao problema depois de ter consumido toda a memória.

Ao longo dos vários testes apercebemo-nos que o DFS nunca apresenta uma solução e a pesquisa gulosa, só num dos testes, é que apresentou uma resposta.

O A* em 2 dos 3 testes foi aquele que teve o menor número de nós gerados, apresentando sempre uma solução ótima e em menor tempo. A única vez em que isto não ocorreu, foi no caso onde a procura gulosa apresentou solução.

Como tal, podemos concluir que o método A* foi o melhor método de pesquisa, nestes ensaios.

Referências

1 O jogo do 15. [Online].; 2015 [cited 2017 Fevereiro 28. Available from: https://pt.wikipedia.org/wiki/O_jogo_do_15.

2 Ryan M. Tiles Game. [Online].; 2004 [cited 2017 Março 1. Available from: https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html.

3 Wikipedia. [Online].; 2017. Available from: https://en.wikipedia.org/wiki/Heuristic_(computer_science)#Newell_and_Simon:_heuristic_search_hypothesis.

4 Black PE. Manhattan distance. [Online].; 2006 [cited 2017 Fevereiro 25. Available from: https://xlinux.nist.gov/dads/HTML/manhattanDistance.html.

5 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.

6 Eranki R. Pathfinding using A* (A-Star). [Online].; 2002 [cited 2017 Fevereiro 25. Available from: http://web.mit.edu/eranki/www/tutorials/search/.

7 Oracle. PriorityQueue. [Online].; 2016 [cited 2017 Fevereiro 25. Available from: http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html.