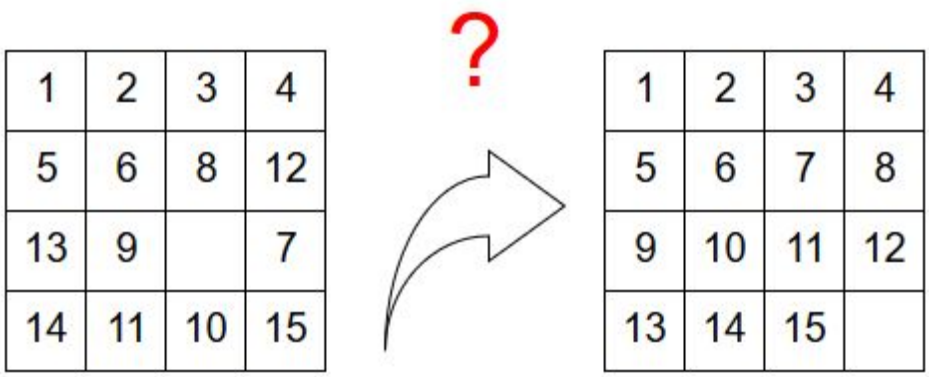
U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Jogo dos 15



Ana Coutinho up200303059
Ana Germano up201105083

Inteligência Artificial

Índice

1. Introdução

O jogo dos 15 é um jogo no qual temos um tabuleiro com 16 espaços, 15 deles estão numerados de 1 a 15 e o último é um espaço em branco, que no nosso trabalho nós representamos com um 0. Neste jogo o objetivo é ordenar as peças, ou seja, a partir de uma configuração inicial baralhada, chegar a uma configuração final, ordenada, segundo o nosso critério.

Assim sendo, com este trabalho pretendeu-se verificar se era possível, a partir de uma dada configuração inicial deste jogo, chegar a uma configuração final, também pré determinada pelo utilizador, bem como descobrir o tempo e espaço gastos por cada tipo diferente de pesquisa para descobrir esta solução. Para tal tivemos que ir desenvolvendo estratégias para implementarmos os métodos necessários.

Este problema é então classificado como um problema de pesquisa (ou busca/procura), sendo este tipo de problemas caracterizado pela necessidade de encontrar uma solução dentro de um grupo muito alargado, possivelmente infinito, de soluções possíveis. Isto contrasta com os problemas de decisão em que apenas é necessário decidir se uma determinada resposta é solução ou não. [1]

Para resolver este tipo de problemas existem vários métodos comumente utilizados, como por exemplo métodos de pesquisa não guiada ou cega, ou seja, que não utilizam uma heurística para fazer a procura e que simplesmente vão gerando as soluções possíveis até chegarmos à solução pretendida. Alguns dos exemplos deste tipo de pesquisa são a Pesquisa em Profundidade, também conhecida como *Depth First Search* (DFS), a Pesquisa em Largura ou *Breadth First Search* (BFS) e a Pesquisa Iterativa Limitada em Profundidade ou *Iterative Deepening Depth First Search* (IDDFS).

Podemos ainda utilizar métodos de pesquisa guiada, que tal como o nome indica vão ter uma heurística a guiar a procura da solução, ou seja, consoante o problema é decidida uma heurística que vai limitar o espaço de procura. A heurística vai ser uma espécie de guia para ser efetuada a procura, limitando o número de soluções geradas que não vão levar à solução pretendida. Alguns exemplos são a Pesquisa A* e a Pesquisa Gulosa ou *Greedy.*

No capítulo seguinte iremos descrever mais pormenorizadamente cada um destes métodos de pesquisa, realçando a forma como funcionam e em que situação deve ser utilizado cada um destes, de forma a resolver os diferentes problemas de pesquisa.

Vamos ainda discutir os resultados obtidos na resolução do problema do jogo dos 15 com cada um dos métodos indicados, dando especial enfoque às diferenças e semelhanças encontradas.

Mas antes de abordarmos estes assuntos mais aprofundadamente vamos pensar em como poderemos abordar um problema de pesquisa. Primeiramente podemos pensar no problema como sendo um objectivo, no caso presente o objectivo é, a partir de uma determinada configuração inicial, chegar a uma configuração final dada. [2]

Agora para podermos chegar à solução pretendida, ou seja o caminho da configuração inicial para a final, vamos utilizar uma estrutura de dados denominada árvore de busca. A configuração inicial vai representar o nó raiz e a configuração final representa o nó folha ao qual pretendemos chegar. O caminho propriamente dito vai ser obtido gerando os filhos do nó inicial, dados pelas jogadas admissíveis, de acordo com as regras do jogo, e assim sucessivamente até chegarmos ao nó folha final, isto caso seja possível chegar da nossa configuração inicial à configuração final. [2]

2. Estratégias de Pesquisa

2.1. Pesquisa não guiada

Uma estratégia de pesquisa não guiada ou cega, blind search no inglês, é uma estratégia de pesquisa que não tem qualquer informação sobre o domínio da pesquisa, ou seja, não tem nenhum dado adicional que ajude a limitar as opções. Neste tipo de pesquisa ao ser gerada a árvore de pesquisa, não há qualquer referência a qual o nó que deverá ser explorado primeiro, todos têm igual probabilidade de ser solução. [3]

Este tipo de estratégia de pesquisa é usado essencialmente quando não temos informação suficiente que nos ajude a guiar a pesquisa. Dentro deste tipo de pesquisa existem estratégias diferentes, que deverão ser usadas consoante as necessidades e especificações do nosso problema. [3]

2.1.1. Pesquisa em Profundidade - DFS

O algoritmo de DFS faz a procura atravessando o grafo em profundidade, ou seja, após escolher um nó segue sempre o mesmo ramo até chegar a uma folha e só depois volta atrás para verificar os restantes ramos. [3] [4]

Para fazermos este tipo de pesquisa podemos usar uma pilha ou stack, que é uma estrutura de dados que nos vai permitir adicionar e remover nós no início da pilha. O que vai acontecer é que sempre que expandimos os nós de um determinado nível estes vão ser colocados no início e quando quisermos ir buscar o

próximo nó a expandir vamos retirar o que está no início. Depois de expandirmos este nó vamos ficar com os respectivos filhos no início da pilha e assim sucessivamente. [3] [4]

Uma alternativa mais comum à pilha é fazer a pesquisa de forma recursiva, em que em vez da estrutura de dados explícita vamos utilizar uma pilha de recursão. [5]

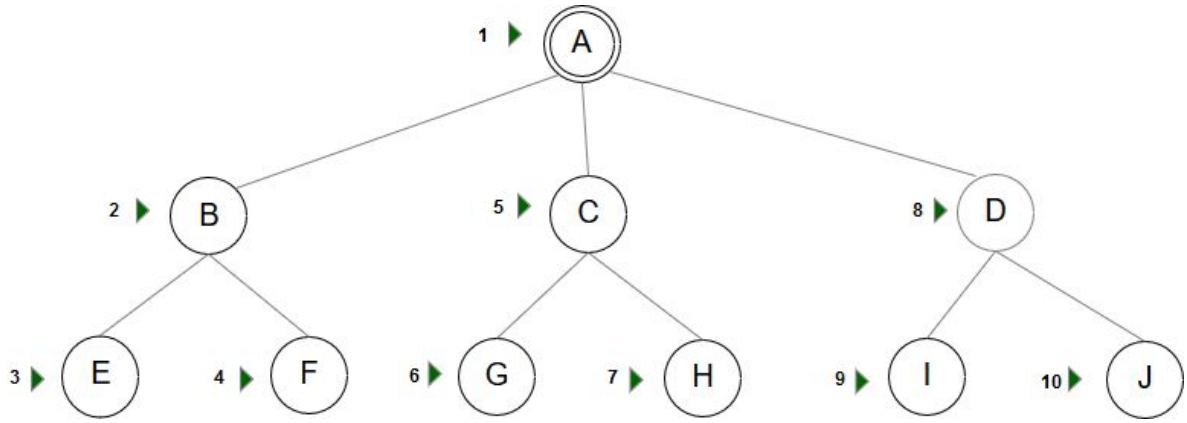


Fig. 1 - Ordem pela qual são visitados os nós da árvore de pesquisa no algoritmo de DFS.

Uma das vantagens da DFS em relação a outras metodologias é que a complexidade espacial é apenas linear e não exponencial, como no caso da BFS. Isto acontece porque usando este algoritmo apenas precisamos de guardar na pilha os nós que estão no caminho entre a raiz e o nó actual e não todos os caminhos desde a raiz até ao nível do nós actual. [5]

Tomando **d** como o nível em que é encontrada a solução e **b** como o factor de ramificação (número médio de nós gerados por cada nó expandido), concluímos que a complexidade temporal deste algoritmo é da ordem de $\mathbf{O(b^d)}$, uma vez que vão ser gerados todos os nós até chegarmos à solução, tal como acontece na BFS mas numa ordem diferente. [3] [5]

Assim podemos dizer que a DFS não é limitada espacialmente mas sim temporalmente, pelo que será uma boa aposta nos casos em que temos uma limitação espacial e não uma limitação temporal. [5]

Para contornar esta limitação temporal podemos não explorar todo o caminho entre a raiz e as folhas, diminuindo assim drasticamente a quantidade de espaço e tempo necessários para encontrar uma solução, caso esta exista. [5]

Ao utilizarmos este algoritmo temos que ter em atenção que existe a possibilidade que este entre num ciclo infinito e nunca mais termine de percorrer o caminho mais à esquerda. Mesmo que o nosso grafo seja finito existe sempre a

possibilidade de que ao efetuar a DFS seja gerada uma árvore de pesquisa infinita. [5]

Uma forma de resolver este problema poderá ser colocar um limite à profundidade que poderá ser alcançada. Idealmente este ponto seria o nível **d** onde se encontra a solução, no entanto este dado raramente é conhecido, uma vez que a DFS é utilizada em casos em que não temos informação adicional para guiar a pesquisa. Isto poderá levar a que seja escolhido um nível de paragem anterior à solução, resultando em que a mesma não seja encontrada. Em contrapartida se for escolhido um nível muito maior que d vamos estar a aumentar desnecessariamente o tempo de execução e a primeira solução a ser encontrada poderá não ser uma solução ótima. [5]

Daqui podemos concluir que com a DFS não é garantido que seja encontrada solução e mesmo que se encontre uma solução, a mesma pode não ser a solução ótima, uma vez que poderão existir várias soluções. [5]

2.1.2. Pesquisa em Largura - BFS

Numa pesquisa em largura vamos expandir a árvore de pesquisa em camadas. Começamos por expandir a raíz, verificamos se a raiz é solução, se não fôr passamos a expandir todos os filhos da raiz, ou seja todos os nós que estejam no nível 1. Para cada nó gerado verificamos se é solução e só vamos expandir os nós do nível 2 se não fôr encontrada solução no nível 1 e assim sucessivamente. [3] Ou seja, todos os nós do nível **d** são expandidos antes de qualquer nó do nível **d+1**. [3] [6]

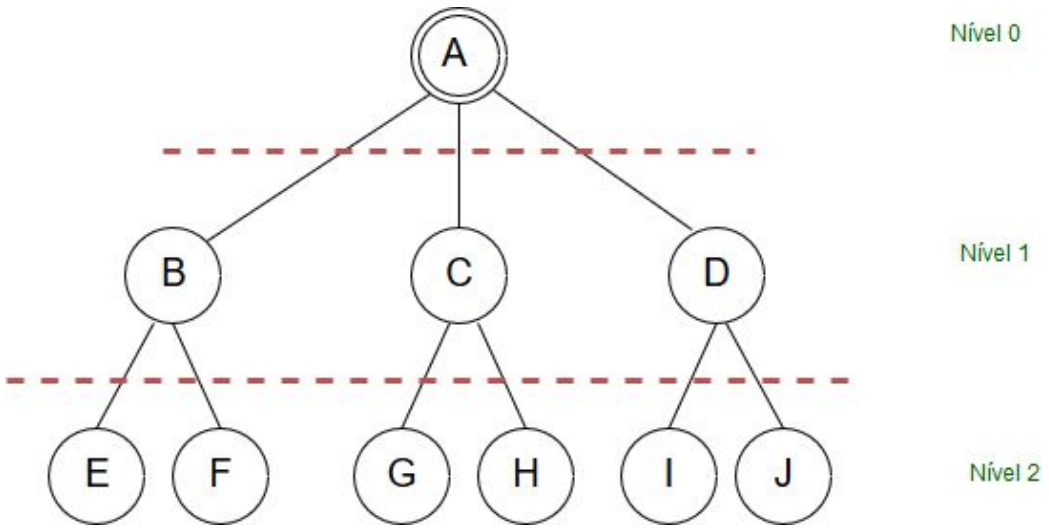


Fig. 2 - Ordem pela qual são visitados os nós na árvore de pesquisa com o algoritmo de BFS.

Para implementar esta pesquisa podemos utilizar uma estrutura de dados denominada fila, ou queue em inglês. Inicialmente a nossa fila vai conter apenas a raiz. Depois em cada iteração o nó que está à cabeça da fila vai ser extraído e expandido. Finalmente os filhos gerados são colocados no fim da fila. [3] [6]

Em contraste com a DFS usando a BFS se existir uma solução é garantido que a vamos encontrar e no caso de existirem diversas soluções vamos sempre encontrar a solução que está na profundidade mais baixa primeiro. Sendo que se o nosso problema seguir uma função não decrescente, então isso significa que a solução que vamos encontrar é a de custo menor, ou seja a solução ótima. [3] [6]

Uma outra clara vantagem em relação à DFS é que na BFS nunca vamos entrar num ciclo infinito.[6]

Para medirmos as complexidades temporal e espacial temos que ter em conta o factor de ramificação da nossa árvore, representado novamente por **b**. Isto significa que cada nó vai gerar em média **b** novos nós. [3] [6]

Assim sendo as complexidades vão ser exponenciais e da ordem $\mathbf{O(b^d)}$, em que **d** representa o nível em que é encontrada a solução.[3] [6]

Como podemos ver pela ordem de grandeza da complexidade espacial este método de pesquisa tem um grande inconveniente a nível de espaço requerido, uma vez que é necessário guardar todos os níveis que já foram expandidos em memória de forma a ser possível gerar o nível seguinte. Isto poderá levar a situações em que a memória do computador seja esgotada ao fim de algumas iterações. E num caso extremo, em que a solução esteja muito afastada da raiz o tempo consumido por esta metodologia vai aumentar muito. [6]

2.1.3. Pesquisa Iterativa Limitada em Profundidade - IDDFS

Do que foi discutido verificamos que com a BFS conseguimos obter uma solução ótima, mas temos problemas com a quantidade de espaço necessária e com a DFS já não temos o problema do espaço, mas podemos não encontrar a solução ótima, ou mesmo não encontrar solução nenhuma. O ideal seria talvez uma combinação de ambas as pesquisas, que é o que acontece na IDDFS. [7] [8]

A ideia chave da IDDFS é combinar a eficácia espacial da DFS e a eficácia temporal da BFS (para soluções em nós perto da raiz). Na sua execução a IDDFS vai chamar uma DFS limitada por altura, ou seja, vai fazer uma pesquisa em profundidade, mas apenas até ao nível de limite, como se fosse uma pesquisa em largura. Com um limite de profundidade 0 apenas vai fazer a procura na raiz, se a raiz não for solução, chama nova DFS desta vez com limite 1, recomeça nova pesquisa pela raíz e expande os nós até ao nível 1. E vai seguindo sempre este procedimento até encontrar uma solução, caso esta exista. [8] [9]

De notar que sempre que é iniciada uma nova pesquisa temos a possibilidade de apagar a pesquisa anterior pois a mesma não é mais necessária, o

que vai contribuir para uma diminuição do espaço gasto quando é utilizado este algoritmo.

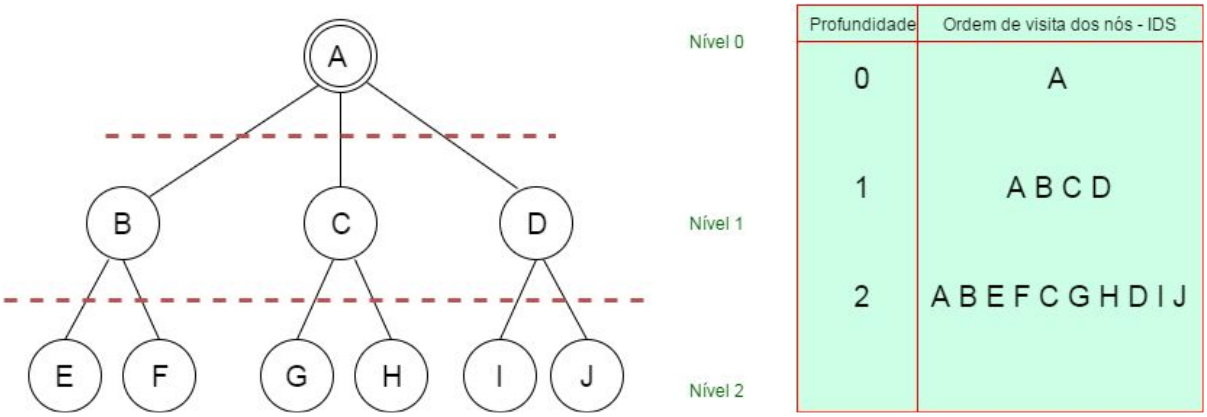


Fig. 3 - Ordem pela qual são visitados os nós na árvore de pesquisa com o algoritmo de IDDFS.

O que vai acontecer é que os níveis superiores vão ser visitados inúmeras vezes ao passo que o último nível (aquele onde esperamos que se encontre a solução) vai ser visitado apenas uma vez, o nível anterior 2 vezes e assim por diante. À primeira vista esta parece ser uma solução com um custo temporal e espacial demasiado elevado, o que na realidade não acontece. Tal como na DFS não precisamos de guardar todos os nós anteriores para fazermos a pesquisa, além de que de cada vez que terminamos uma chamada sem encontrar a solução, apagamos toda a pesquisa e recomeçamos de novo. Relativamente ao custo temporal, o mesmo acaba por não ser tão elevado pois os níveis com mais nós numa árvore são os níveis inferiores, que vão ser visitados menos vezes. [8]

Para calcularmos a complexidade temporal deste algoritmo, temos que ter novamente em conta um factor de expansão, representado por **b**, bem como a profundidade da solução representada por **d**. [9]

Assim sendo e tendo em conta que foi discutido anteriormente vamos chegar à seguinte expressão:

$$\mathbf{d*b + (d-1)*b_2 + \ldots + 3*b_{d-2} + 2*b_{d-1} + b_d}$$

Em que **d*b** representa a raiz e $\mathbf{b_d}$ o nível final em que é encontrada a solução. [9]

Depois de avaliarmos cuidadosamente esta expressão chegamos à conclusão que a IDDFS vai seguir assintoticamente o tempo da BFS e da DFS, ou seja a sua complexidade temporal vai ser da ordem de $\mathbf{O(b^d)}$. Contudo, como se pode verificar existe um factor constante nesta expressão que vai aumentar um pouco o tempo de execução deste algoritmo face aos demais. [9]

	Complexidade Temporal	Complexidade Espacial	Quando usar?
DFS	$O(b^d)$	O(bd)	• Não interessa se a solução está mais perto da raiz • Quando o grafo/árvore não é muito grande/infinito
BFS	$O(b^d)$	$O(b^d)$	• Quando o espaço e memória não é um problema • Quando queremos a resposta ótima (mais perto da raiz)
IDDFS	$O(b^d)$	O(bd)	• Quando precisamos de fazer uma BFS mas não temos espaço em memória suficiente e é aceitável um algoritmo mais lento

Quadro 1 - Resumo comparativo entre algoritmos de DFS, BFS e IDDFS. [9]

2.2. Pesquisa Guiada

Em certos problemas mais complexos os algoritmos de pesquisa cega poderão não dar os resultados de que necessitamos. Uma forma de contornar este problema é utilizar um algoritmo de pesquisa guiada. Para tal precisamos primeiro de escolher uma boa heurística que nos diga que parte da árvore devemos explorar de seguida. Com uma boa heurística, adequada ao problema em questão conseguimos encontrar soluções mesmo para os problemas mais complexos e extensos. [8]

Para resolver estes problemas mais complexos usando uma pesquisa guiada vamos ter que ter uma aproximação diferente ao problema, o mesmo passará a ser encarado como um problema de optimização em que cada jogada/movimento tem associado um custo e nós queremos arranjar formas de encontrar caminhos/soluções que minimizem o custo total. [8]

Existem vários algoritmos de pesquisa guiada diferentes, mas apenas iremos abordar dois, o greedy e o A*, pois serão estes dois que irão ser implementados no nosso programa e comparados com os algoritmos de pesquisa cega. Relativamente à heurística escolhida optamos pela Manhattan Distance, que é uma heurística admissível para este problema, apesar de ter as suas limitações. [10]

A maior limitação da Manhattan Distance quando aplicada ao jogo dos 15 é que nesta heurística cada quadrado é considerado independentemente, quando na realidade eles interferem uns com os outros. Contudo existe a possibilidade de combinar outras heurísticas para ter uma maior precisão. [10]

Um factor positivo relativamente a esta heurística é que nunca sobrestima o custo de alcançar o objectivo. Isto vai garantir que em algoritmos como o A* seja encontrada a solução ótima, num tempo mais curto. [11]

2.2.1. Pesquisa Greedy

A ideia chave deste algoritmo de pesquisa é em cada escolha que fazemos optarmos pelo nó que mais se aproxima do nosso objectivo e que possivelmente nos irá levar a uma solução mais rápido. Tomando h(n) como a nossa função heurística, este algoritmo vai avaliar em cada nó a sua função heurística e vamos ficar com f(n) = h(n). [12]

Este algoritmo tem uma implementação muito fácil, sendo normalmente utilizada uma priority queue. Expandimos o primeiro sucessor de um nó e comparamos a sua heurística com a do pai, se for melhor vai para a frente da queue (elemento de maior prioridade) senão vai para o meio, conforme o valor da sua função heurística. Este processo vai sendo repetido até encontrarmos a solução. [14]

Neste trabalho efetuamos a pesquisa greedy usando como heurística a Manhattan Distance, por esta ser considerada uma heurística admissível, ou seja, nunca vai sobrestimar o custo de alcançar o nosso objectivo.

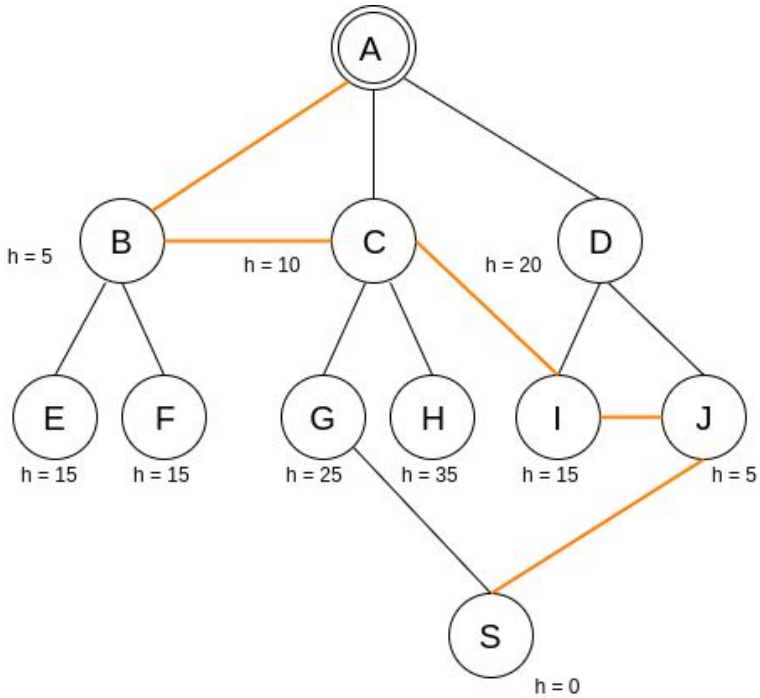


Fig. 4 - No greedy começamos por escolher os filhos do nó A e colocamos na priority queue, vamos sempre expandindo os filhos do nó que tiver melhor heurística (h). Neste caso ficamos com o caminho representado a amarelo.

2.2.2. Pesquisa A*

A escolha do nó a expandir é efetuada tal como no algoritmo greedy diferindo apenas na função que nos indica qual a escolha a efetuar. [14]

Vamos então evitar expandir caminhos com um custo muito elevado e expandir apenas os que são mais promissores. Se tomarmos g(n) como sendo o custo para chegar ao nó em questão, h(n) como sendo o custo estimado para chegar à solução (valor dado pela heurística) e f(n) o custo total estimado do caminho para chegarmos à nossa solução, então vamos ficar com:

f(n) = g(n) + h(n)

Poderemos então implementar o algoritmo A* usando uma priority queue e incrementando f(n). [12] [15]

Este algoritmo poderá ser usado em casos muito extensos ou em que pretendemos obter uma solução ótima de uma forma rápida, contanto que seja utilizada uma boa heurística. Neste trabalho utilizamos a heurística Manhattan Distance já referida anteriormente, devido às suas características vantajosas para este tipo de pesquisa.

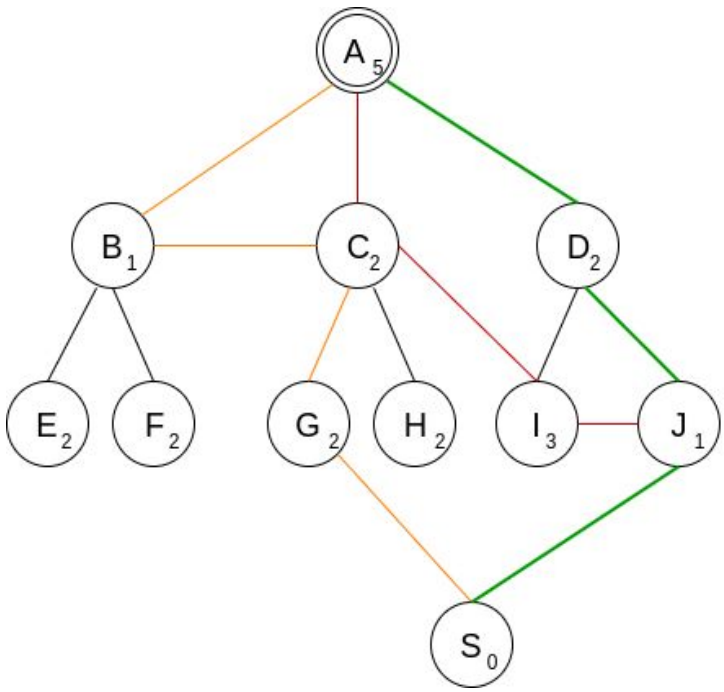


Fig. 5 - No algoritmo A* temos em conta não só a heurística mas também o custo do caminho desde o nó inicial, neste caso consideremos que em cada nível do grafo o custo é incrementado em 1 valor, assim garantimos que a solução encontrada será a solução ótima.

3. Descrição da Implementação

3.1. Linguagem utilizada

Na implementação deste trabalho nós decidimos utilizar a linguagem Java por ser uma linguagem a que ambas estamos familiarizadas e também porque esta é uma linguagem orientada a objectos. Como sabíamos que iria ser necessário implementar listas, filas e pilhas, entre outros, achamos desnecessário e excessivamente trabalhoso estar a fazer essas implementações manualmente e decidimos aproveitar todas as potencialidades da API do Java.

3.2. Estruturas de dados utilizadas

Na implementação deste programa utilizamos a LinkedList do Java para guardar os nós da nossa árvore de pesquisa. Utilizando esta LinkedList nós imitamos o comportamento das diferentes estruturas de dados normalmente utilizadas nas pesquisas efetuadas, nomeadamente uma stack na BFS e IDDFS, uma queue na BFS e uma priority queue no greedy e na A*.

Utilizamos ainda um hashMap do Java para controlar os nós já visitados da matriz nos algoritmos de pesquisa cega, nos algoritmos de pesquisa guiada o controle é efetuado pela heurística. Esta estrutura permite colocar os nós a ser explorados na LinkedList, sem repetições. Usar esta estrutura ao invés de outra LinkedList permitiu-nos salvar muito espaço.

4. Resultados

Estratégia	**Tempo (segundos)**	**Espaço (nós)**	**Encontrou a solução?**	**Profundidade/ Custo**
DFS	N/A	N/A	Não	N/A
BFS	0,983	36789	Sim	12
IDDFS	0,326	39669	Sim	12
Gulosa	0.005	87	Sim	12
A*	0,005	30	Sim	12

5. Comentários Finais e Conclusões

A nossa ideia inicial ao começar este trabalho era implementar um programa que resolvesse o problema do jogo dos 15, adaptável a várias dimensões de matriz. Contudo, à medida que fomos implementando o código para resolver o problema constatamos que este era um problema bastante complexo e que generalizar para uma matriz de qualquer dimensão iria trazer ainda mais dificuldades do que as que já estávamos a experienciar.

Desta forma apenas colocamos no início a possibilidade de o utilizador introduzir o tamanho pretendido do tabuleiro de jogo e implementamos uma função que indica se o mesmo terá solução no caso de um tabuleiro de tamanho par. A implementação das estratégias de pesquisa/busca foi feita apenas para matrizes de 4x4, tal como era o objetivo do trabalho.

O primeiro passo foi implementar uma função que, através de um cálculo simples, indicasse se era possível chegar da nossa configuração inicial à configuração final. Seguindo a documentação facultada pela professora nas aulas práticas chegamos à seguinte expressão lógica:

(#inversions even) == (blank on odd row from bottom) [16]

Passamos então a implementar uma função que de cada vez que fosse chamada devolvesse os descendentes possíveis de uma determinada configuração, ou seja, que mediante uma matriz inicial com o zero numa determinada posição, devolvesse todas as configurações que poderiam resultar de uma jogada válida a partir da configuração inicial (é considerada uma jogada válida mover o 0 para cima, para baixo, para a direita ou para a esquerda dentro dos limites do tabuleiro do jogo).

Estes descendentes são então guardados na nossa LinkedList de acordo com a ordenação exigida pelo método de pesquisa a utilizar. Assim sendo a matriz gerada é guardada no Node junto com outras informações pertinentes para a pesquisa, tais como, a altura a que o nó está na matriz e a sua heurística.

Todas as pesquisas implementadas seguem o algoritmo geral dado nas aulas, sendo que estas diferem maioritariamente na forma como são inseridos os descendentes na LinkedList. Isto vai influenciar a ordem pela qual são expandidos os nós, sendo esta ordem específica de cada tipo de pesquisa. Nas pesquisas Greedy e A* tivemos que calcular uma heurística, que vai ser usada para ordenar os nós por ordem decrescente de prioridade, isto é, o nó mais prioritário (com heurística menor) fica à cabeça da lista e os restantes vão sendo colocados conforme a sua prioridade até ao último, que é o nó menos prioritário.

Sempre que retiramos um nó da LinkedList para expandir os seus filhos/descendentes, fazemos primeiramente a comparação com a nossa

configuração final para verificar se esse nó é solução e só depois expandimos os seus filhos.

	Complexidade Temporal	Complexidade Espacial	Completude	Optimalidade
DFS	$O(b^d)$	O(bd)	Não	Não
BFS	$O(b^d)$	$O(b^d)$	Sim	Sim
IDDFS	$O(b^d)$	O(bd)	Sim	Sim
Greedy	$O(b^d)$	$O(b^d)$	Não	Não
A*	$O(b^d)$	$O(b^d)$	Sim	Sim

Quadro 2 - Comparação entre os diferentes algoritmos implementados no nosso programa, em que **b** representa o factor de ramificação de **d** a profundidade da solução, de notar que a profundidade poderá não ser a mesma nos diferentes algoritmos. [9] [17]

Os dados apresentados nesta tabela são consistentes com os dados obtidos nos testes que efetuamos com o nosso programa. Confirmamos que a DFS não é ótima nem completa quando comparada com as restantes, sendo que no nosso caso de teste nem sequer chegou a encontrar uma solução em tempo útil. Tal como esperado quando comparadas com as pesquisas cegas a A* e a Greedy levam em média menos tempo e gastam menos espaço, pois expandem menos nós.

Para casos relativamente pequenos como o nosso caso de teste a maioria das pesquisas consegue encontrar a solução em tempo útil, sendo que em certos casos até na DFS é possível encontrar a solução, que poderá não ser ótima. Tendo em conta as diferenças observadas podemos concluir que em casos maiores ainda que o nosso a pesquisa A* será a mais indicada, quer a nível de tempo quer espaço. Se usarmos uma boa heurística vamos conseguir encontrar a solução ótima mais rapidamente e gastar menos espaço.

6. Referências Bibliográficas

[1] - http://www.dictionary.com/browse/search-problem (em 25/02/2017)
[2] - http://jeiks.net/wp-content/uploads/2013/05/Metodos_de_Busca.pdf (em 25/02/2017)
[3] - http://www.cs.nott.ac.uk/~pszgxk/courses/g5aiai/003blindsearches/blind_searches.htm (em 26/02/2017)
[4] - https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm (em 26/02/2017)
[5] - http://intelligence.worldofcomputing.net/ai-search/depth-first-search.html#.WLQMIFWLSUk (em 27/02/2017)
[6] - http://intelligence.worldofcomputing.net/ai-search/breadth-first-search.html#.WLQWZVWLSUk (em 27/02/2017)
[7] - http://artint.info/html/ArtInt_62.html (em 28/02/2017)
[8] - http://www.cs.ubbcluj.ro/~csatol/log_funk/prolog/slides/7-search.pdf (em 28/02/2017)
[9] - http://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/ (em 28/02/2017)
[10] - https://heuristicswiki.wikispaces.com/Manhattan+Distance (em 28/02/2017)
[11] - https://heuristicswiki.wikispaces.com/Admissible+Heuristic (em 28/07/2017)
[12] - http://centurion2.com/AIHomework/AI260/ai260.php (em 01/03/2017)
[13] - https://en.wikipedia.org/wiki/Best-first_search (em 01/03/2017)
[14] - http://centurion2.com/AIHomework/AI260/ai260.php (em 01/03/2017)
[15] - https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_popular_search_algorithms.htm (em 01/03/2017)
[16] - https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html (em 05/03/2017)
[17] - https://www.ics.uci.edu/~welling/teaching/271fall09/UninformedSearch271f09.pdf (em 05/03/2017)

Na implementação do programa recorremos também às seguintes fontes:

https://www.youtube.com/watch?v=ySN5Wnu88nE&t=3s
http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
http://javaconceptoftheday.com/how-to-check-the-equality-of-two-arrays-in-java/