U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Buscas
Relatório Final

Unidade Curricular: Inteligência Artificial

Regentes: Inês Dutra

Francesco Renna

Elementos: Beatriz Marques de Sá - up202105831

Marisa Peniche Salvador Azevedo - up202108624

Marta Luísa Monteiro Pereira - up202105713

12 de março de 2023

Índice

Índice de Figuras

1. Introdução

1.1. O que é um problema de busca/procura?

Um problema de busca é um tipo de problema que envolve encontrar uma solução de entre várias possibilidades, seguindo algumas regras ou condições. O objetivo é encontrar a melhor solução possível, levando em conta critérios como tempo de execução, eficiência, custo, entre outros. Para resolver este tipo de problemas, é preciso usar técnicas e algoritmos específicos que permitam avaliar e comparar as diferentes opções e escolher a mais adequada para o caso em questão.

Quando não sabemos o que fazer para resolver um problema, podemos criar um plano. Esse plano inclui uma sequência de ações que nos levam ao resultado que queremos alcançar. Chamamos o agente que faz isso de "agente de resolução de problemas". Para encontrar a solução, o agente começa numa situação inicial e usa um processo de busca para chegar à solução final. Esse processo de busca é um método que o agente utiliza para encontrar a resposta.

Para resolver um problema de busca, o primeiro passo é definir o estado inicial (ponto de partida) e o estado final (objetivo a ser alcançado). Para isso, é necessário estabelecer um conjunto de possíveis estados do problema, e utilizar a função sucessor para gerar novos estados a partir do estado anterior.

A representação do espaço de estados é feita através de uma árvore de procura, onde cada nó da árvore representa um estado possível. A partir de cada nó, é possível gerar novos estados através da função sucessor.
Cada nó da árvore deve conter informações importantes, tais como: o estado atual, a profundidade a que se encontra, o nó pai (ou outra forma para representar o caminho percorrido) e o custo desde o estado inicial até ao nó atual.

Em resumo, um problema de busca é um tipo de problema que envolve encontrar um caminho através de um conjunto de estados até ao estado final. Para resolver um problema de busca, é necessário definir o estado inicial e o estado final, o conjunto de estados possíveis e a representação do espaço de estados através de uma árvore de procura.

1.2. Quais são os métodos utilizados para resolver problemas de procura?

Existem vários métodos utilizados para resolver problemas de procura. Alguns dos mais comuns são:

- Procura em profundidade (DFS): explora um caminho até o final antes de voltar e tentar outro caminho.

- Procura em largura (BFS): explora todos os caminhos no mesmo nível antes de passar para o próximo nível.

- Procura em profundidade limitada (DLS): é semelhante à busca em profundidade, mas estabelece um limite na profundidade máxima que pode ser explorada.

- Procura em profundidade iterativa (IDFS): é uma combinação de busca em profundidade limitada e busca em largura, aumentando gradualmente a profundidade máxima que pode ser explorada em cada iteração.

- Procura bidirecional: utiliza duas buscas, uma a partir do estado inicial e outra a partir do estado objetivo, encontrando um estado intermédio onde os dois caminhos se encontram.

- Procura heurística: utiliza uma função heurística para estimar o quão perto um nó está do objetivo, explorando primeiro os nós com a menor estimativa. A busca gulosa é uma forma de busca heurística que só leva em conta a estimativa heurística, enquanto o A* tem em conta tanto a estimativa heurística quanto a distância percorrida até o momento.

2. Descrição do problema de procura estudado (jogo dos 15)

O Jogo dos 15 é um jogo onde temos um tabuleiro 4x4 com 15 peças numeradas e um espaço vazio que podemos movimentar para permitir que as outras peças sejam movidas de lugar. O objetivo do jogo é conseguir colocar as peças na ordem correta, movimentando a peça vazia para permitir que as outras peças sejam colocadas nas suas posições corretas.

O problema de procura no Jogo dos 15 consiste em encontrar uma sequência de movimentos que nos leve da configuração inicial do tabuleiro para a configuração desejada. Ou seja, temos que descobrir uma sequência correta de movimentos que permitirá que todas as peças sejam movidas para as suas posições corretas.

O desafio é encontrar uma solução que utilize o menor número possível de movimentos para chegar à configuração desejada do tabuleiro. Para isso, podemos usar algoritmos de busca. No entanto, alguns desses algoritmos podem ser mais eficientes que outros em termos de encontrar a solução mais rapidamente.

Este jogo é um problema muito conhecido na área de inteligência artificial e é frequentemente utilizado como base para testar algoritmos de busca. Além disso, uma oportunidade para desenvolver técnicas de otimização e heurísticas de busca.

3. Estratégias de Procura

Durante as aulas, aprendemos sobre dois tipos de algoritmos de busca: os não informados e os informados. Nos algoritmos não informados, essa estimativa não está disponível e o agente precisa buscar todas as possíveis soluções para encontrar a melhor opção, enquanto que nos algoritmos informados, o agente possui uma estimativa da distância ou custo para chegar ao objetivo.

3.1. Procura não guiada

A estratégia de procura não guiada é um algoritmo de busca que não utiliza qualquer informação adicional sobre o espaço de busca, além do estado inicial e final. Isso significa que não segue um caminho específico para alcançar a solução, mas sim usa uma abordagem aleatória para explorar o espaço de busca em todas as direções possíveis.

Uma das vantagens da busca não guiada é que ela pode explorar áreas do espaço de soluções que outros métodos de busca não conseguem alcançar. Como o algoritmo não está limitado por qualquer estrutura de informação, ele pode percorrer livremente o espaço de busca, o que pode levar a soluções inesperadas e até mesmo ótimas.

No entanto, a desvantagem da busca não guiada é que pode ser muito lenta, pois é possível que o algoritmo explore uma grande quantidade de soluções antes de encontrar uma solução satisfatória. Além disso, como não há nenhuma informação sobre o espaço de busca, o algoritmo pode acabar percorrendo caminhos desnecessários e não produtivos, o que pode aumentar ainda mais o tempo de execução.

3.1.1. Profundidade (DFS - Depth-First Search):

Este algoritmo, é o método de pesquisa mais simples e geralmente é implementado como uma pesquisa numa árvore.

Começa-se no nó inicial (estado inicial) e, em seguida, verifica-se se é uma solução. Se for, o algoritmo termina, caso contrário, procura-se o próximo nó ainda não visitado. Se não houver mais nós a visitar, volta-se para o nó anterior e continua a busca. Este processo continua até que se encontre uma solução ou se esgotem todas as possibilidades.

Na figura seguinte pode ver-se um exemplo da ordem de pesquisa deste algoritmo.

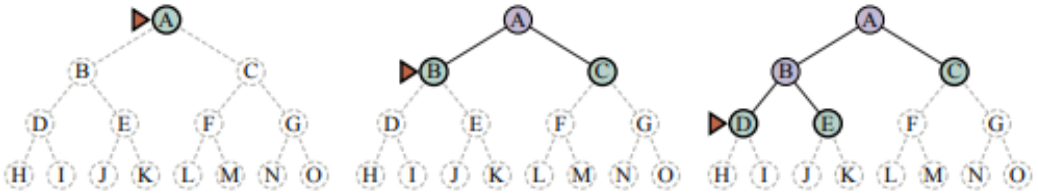

Figura 1 - Árvore de Profundidade

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach.

O algoritmo precisa de percorrer cada ramo até à sua profundidade máxima, resultando numa complexidade temporal de O(b^d), onde b é o fator de ramificação e d é a profundidade máxima. Contudo, apenas precisa de guardar o estado dos filhos dos nós que fazem parte da solução atual (do início até ao nó atual), para evitar a repetição de estados. Desta forma, a complexidade espacial é de O($b\ x\ d$). Se não for necessário evitar estados repetidos, ou seja, apenas é necessário guardar o caminho, resultando numa complexidade espacial de O(d). No entanto, este algoritmo não é completo, uma vez que não consegue encontrar soluções em caminhos infinitamente grandes, e não é ótimo, pois a primeira solução encontrada pode não ser a que requer menos passos.

3.1.2. Largura (BFS - Breadth-First Search):

Este algoritmo é implementado como uma árvore de pesquisa, mas ao contrário da pesquisa em profundidade, dá prioridade aos nós mais próximos da raiz.

Começa-se no nó inicial (estado inicial) e é criada uma fila vazia, em seguida, verifica-se se é uma solução. Se for, o algoritmo termina, caso contrário, se o nó tiver sucessores, são adicionados apenas aqueles que ainda não foram visitados ao final da fila. Depois disso, é retirado o nó que está no topo da fila e é repetido o processo para esse nó.

Na figura seguinte pode ver-se um exemplo da ordem de pesquisa deste algoritmo.

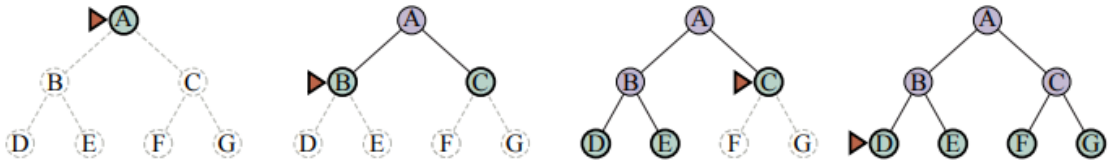

Figura 2 - Árvore de Largura

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach.

Esta estratégia é eficaz na busca da solução com o menor número possível de ações, pois quando o algoritmo está a expandir os nós na profundidade d, já terá expandido todos os nós na profundidade $d - 1$. Isso significa que, se a solução estiver presente, ela será encontrada. No entanto, essa abordagem funciona bem apenas para problemas em que todas as ações têm o mesmo custo, pois problemas com custos variáveis podem exigir uma abordagem diferente.

Uma desvantagem deste algoritmo é que todos os nós gerados são armazenados na memória, o que leva a uma complexidade temporal e espacial de O(b^d).

Uma vantagem é que o algoritmo é ótimo, pois a primeira solução encontrada é a melhor possível e, se o fator de ramificação b for finito, o algoritmo também é completo.

3.1.3. Busca Iterativa Limitada em Profundidade:

Este algoritmo funciona como uma árvore de pesquisa, onde são explorados os estados sucessores de cada estado inicial até uma profundidade máxima definida.

Para começar a busca, define-se a profundidade máxima que se pretende alcançar. Depois, começa-se a explorar a árvore partindo da raiz com uma profundidade de 0, ou seja, expandindo cada nó sucessor até essa profundidade. Caso a solução não seja encontrada, aumenta-se a profundidade da busca em 1 e repete-se o processo até atingir a profundidade máxima definida.

Assim, a cada iteração da busca, os nós sucessores são expandidos até à profundidade definida, e a busca continua a aumentar a profundidade até encontrar a solução ou até chegar à profundidade máxima definida sem encontrar uma solução.

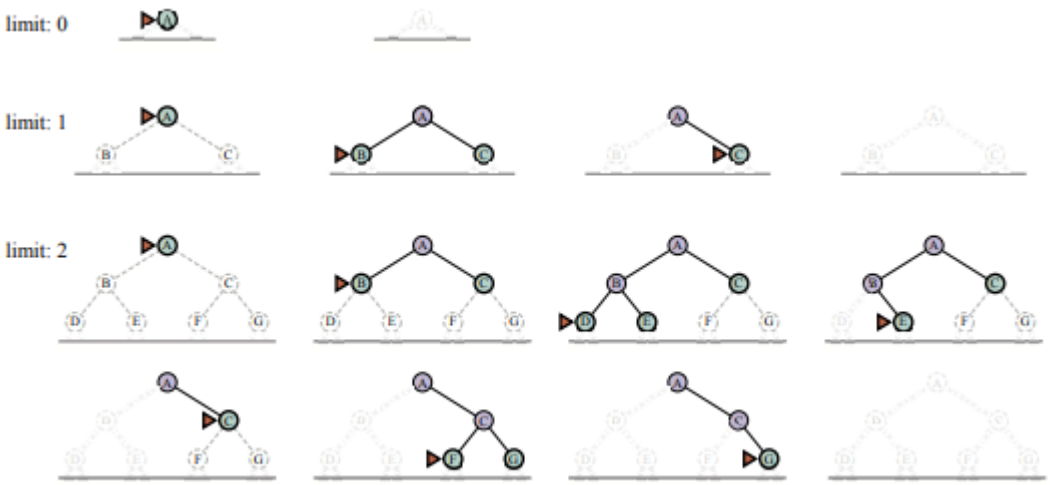

Figura 3 - Árvore de Profundidade Limitada

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach.

Portanto, esta estratégia expande os nós até uma determinada profundidade máxima permitida na busca ld e pelo fator de ramificação b.

A complexidade temporal da busca iterativa limitada em profundidade é O(b^{ld}), o que significa que o tempo necessário para realizar a busca aumenta exponencialmente com o limite de profundidade. Quanto maior o limite de profundidade, maior será o número de nós que precisam ser explorados e, portanto, mais tempo será necessário para encontrar a solução.

Já a complexidade espacial é O($b \times ld$), o que significa que a quantidade de espaço de memória necessária para realizar a busca aumenta linearmente com o limite de profundidade e o número de sucessores. Ou seja, quanto maior o fator de ramificação e quanto maior o limite de profundidade, mais espaço de memória será necessário para armazenar as informações dos nós explorados durante a busca.

3.2. Procura guiada

Ao contrário dos métodos de procura não guiada, os métodos de procura guiada possuem algumas informações do problema, nomeadamente algum tipo de função heurística que permita dar um valor a um determinado estado. Com este valor é possível escolher aquela que se acredita ser a melhor opção, ignorando as outras. É de notar, no entanto, que a função heurística não deve ser uma função qualquer.

3.2.1. O que é uma heurística?

Uma heurística é uma estratégia de resolução de problemas que utiliza regras práticas e aproximações para chegar a uma solução aceitável, em vez de seguir um algoritmo formal e exato.

Para ser eficaz, a heurística deve ser simples em comparação com o cálculo real do estado, que por vezes pode até corresponder à solução do problema. Devendo sempre estar o mais próximo possível do valor real, já que assim chegamos mais rapidamente à solução. Além disso, a heurística deve ser admissível, se o objetivo for minimizar o custo.

3.2.2. Busca Gulosa

A busca Gulosa é um algoritmo que utiliza a informação heurística para decidir qual o melhor caminho a seguir.

Esta estratégia escolhe sempre o nó com a menor heurística, ignorando o custo do caminho percorrido até aquele momento, e expande-o para criar uma nova geração de nós. Essa nova geração é avaliada com a heurística e o processo é repetido até que a solução seja encontrada. No entanto, pode não encontrar a melhor solução, pois pode escolher um caminho que parece ser o mais próximo do objetivo, sendo que, na verdade, este caminho pode ser mais longo do que os outros.

A complexidade temporal do algoritmo Gulosa é $O(b^d)$, onde b é o fator de ramificação da árvore de pesquisa e d é a profundidade máxima da árvore. Sendo que uma boa função heurística pode diminuir consideravelmente a complexidade temporal.

A complexidade espacial é $O(b^d)$ uma vez que o algoritmo precisa de armazenar todos os nós na memória.

Em suma, este algoritmo é frequentemente usado em problemas onde o espaço de busca é muito grande, não sendo prático avaliar todos os nós.

3.2.3. Busca A*

A busca A* é um algoritmo que utiliza a informação do custo do caminho e a informação heurística para decidir qual o melhor caminho a seguir.

O funcionamento do método A* dá-se através da construção de uma árvore de busca, onde cada nó representa um estado de problema e as arestas representam o custo do nó pai aos nós filhos que são gerados pela função sucessora. A pesquisa inicializa-se pela expansão do estado inicial, onde vai ser calculado o custo final que é dado pela função f(p) = g(p) + h(p), onde g(n) é o custo do estado atual ao estado n (estado gerado pelo estado atual) e h(n) é o custo do estado n (estado gerado pelo estado atual) ao estado final.

$$\overbrace{\underbrace{start \xrightarrow{actual} n}_{g(p)} \xrightarrow{estimate} goal}^{f(p)} \quad \underbrace{\qquad\qquad}_{h(p)}$$

A complexidade temporal deste algoritmo A* é O(b^d), mas se a heurística escolhida nunca superestimar o custo, a complexidade temporal pode reduzir para O(b^h), onde h é o custo estimado da solução mais curta. Como esta função guarda todos os nós visitados, terá complexidade espacial de O(b^d).

Este algoritmo é aplicado numa variedade de problemas, incluindo jogos, planeamento de rotas e robótica. Sendo mais eficiente em problemas em que o espaço de estado é grande e há muitas ações possíveis em cada estado. No entanto, a sua eficiência pode ser comprometida em problemas com espaços de estado muito grandes ou em que a heurística utilizada não é admissível.

3.2.4. Qual foi a heurística escolhida e porquê?

A escolha da heurística é uma decisão importante, uma vez que influencia o desempenho do algoritmo, sendo que uma boa heurística pode reduzir o tempo de procura. Para a implementação das procuras A* e Gulosa usamos as heurísticas Manhattan Distance e Misplaced Tiles.

A Manhattan Distance é uma função que mede a distância vertical e horizontal entre dois pontos do plano. No jogo dos 15, calcula a soma das distâncias entre cada peça de um estado ao estado final.

A Misplaced Tiles é calculada somando-se todas as peças que estão em posições diferentes daquelas do estado final.

Neste tipo de jogo as duas heurísticas são admissíveis, pelo facto que nunca superestimam o custo do caminho para o estado final. Apesar disso, a função Manhattan Distance é considerada uma heurística mais forte e eficiente para este tipo de jogo, uma vez que, tem em consideração a localização das peças em relação à posição final, enquanto que a função Misplaced Tiles apenas contabiliza o número de peças fora do lugar.

4. Descrição da Implementação

4.1. Escolha da linguagem de programação

A linguagem utilizada foi Python, pois é uma linguagem de programação de alto nível, fácil de ler e escrever, com sintaxe clara e concisa, e uma grande quantidade de bibliotecas disponíveis para resolver diferentes tipos de problemas. Além disso, Python é uma das linguagens mais populares e amplamente utilizadas em todo o mundo, o que significa que há uma grande comunidade de desenvolvedores e muitos recursos disponíveis para a sua aprendizagem e suporte.

Para o jogo dos 15, Python é uma escolha adequada devido à sua capacidade de lidar com matrizes e estruturas de dados, como as usadas para representar o tabuleiro do jogo. Outra vantagem em utilizar Python para o jogo dos 15 é a facilidade de implementação de algoritmos de busca e resolução de problemas, como a busca em largura ou profundidade, que podem ser usados para encontrar soluções para o jogo. Python também suporta programação orientada a objetos, o que facilita a organização e a manutenção do código.

Em resumo, Python é uma escolha adequada para implementar o jogo dos 15 devido à sua facilidade de leitura e escrita, ampla comunidade de desenvolvedores, grande quantidade de bibliotecas disponíveis, e capacidade de lidar com matrizes e estruturas de dados, além de suportar programação orientada a objetos e algoritmos de busca e resolução de problemas.

4.2. Estruturas de dados utilizadas

As estruturas de dados que foram utilizadas para a implementação do jogo dos 15 foram:

- Listas;
- Tuplos;
- Conjuntos;
- Filas.

Optámos por utilizar listas para armazenar as diferentes configurações do jogo.

Os tuplos também foram necessários, uma vez que criámos uma classe para armazenar cada configuração juntamente com a sua profundidade, e por vezes era necessário aceder a uma parte específica do tuplo.

Os conjuntos são semelhantes às listas, mas não permitem elementos duplicados, e foram úteis para guardar as configurações já visitadas, tornando o algoritmo mais eficiente.

Para algumas das estratégias, adaptamos listas para funcionarem como filas ou pilhas, enquanto para outras utilizamos filas prioritárias.

4.3. Estruturação do Código

Para a estruturação do código decidimos dividir os códigos em vários ficheiros:

- fifteen_puzzle.py: o método desejado será chamado

- board.py: representação visual do tabuleiro

- solvability.py: verificar se uma configuração é solucionável

- puzzle.py: possui uma classe Puzzle que é usada para representar um estado do jogo. Os estados do jogo são criados com base em uma lista que representa a disposição das peças no tabuleiro. Cada estado do jogo é criado com uma profundidade, um custo e um parent, array que contém a sequência de movimentos necessários para chegar a esse estado. Possui também quatro métodos que retornam uma nova classe Puzzle após um movimento do espaço em branco para a esquerda, direita, cima ou baixo.

- "heuristic.py": contém a heurística Misplaced e a heurística distância de Manhattan, que calcula o número de peças no tabuleiro que não estão na posição final e a a distância de Manhattan entre dois tabuleiros do jogo dos 15, respectivamente.

- "strategies.py": tem a implementação das estratégias de busca pedidas para a realização deste trabalho, neste caso, a busca em profundidade, a busca em largura, a busca em profundidade iterativa, a busca gulosa Misplaced e Manhattan e a busca A* Misplaced e Manhattan.

Para mais informações sobre a estruturação do código, deixamos aqui o link para o gitHub onde se encontra o repositório com o mesmo:

- https://github.com/marisaazevedo/trabalho1_IA

5. Resultados

Para comparar a eficiência das estratégias decidimos usar 4 configurações:

1. configuração inicial: 1 2 3 4 5 6 8 12 13 9 0 7 14 11 10 15
   configuração final: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

Estratégia	Tempo (seg)	Espaço	Encontrou a solução?	Profundidade	Custo
DFS	—	—	Não	—	—
BFS	0.386988	52604	SIm	12	—
IDFS	0.210271	6080	Sim	12	—
Gulosa Misplaced	0.001175	88	Sim	12	68
Gulosa Manhattan	0.000996	48	Sim	12	66
A* Misplaced	0.001339	100	Sim	12	142
A* Manhattan	0.0010237	53	Sim	12	144

2. configuração inicial: 2 1 4 0 7 10 3 8 6 5 11 9 12 14 13 15
   configuração final: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

Estratégia	Tempo (seg)	Espaço	Encontrou a solução?	Profundidade	Custo
DFS	—	—	Não	—	—
BFS	—	—	Não	—	—
IDFS	—	—	Não	—	—
Gulosa Misplaced	0.894159	24132	Sim	341	2292
Gulosa Manhattan	0.678995	15904	Sim	265	2170
A* Misplaced	10.896059	346008	Sim	251	4536
A* Manhattan	2.705399	84480	Sim	183	4419

3.

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

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

Estratégia	Tempo (seg)	Espaço	Encontrou a solução?	Profundidade	Custo
DFS	—	—	Não	—	—
BFS	—	—	Não	—	—
IDFS	—	—	Não	—	—
Gulosa Misplaced	1.257135	33952	Sim	296	2083
Gulosa Manhattan	1.048189	20504	Sim	362	3479
A* Misplaced	13.245138	338896	Sim	314	5935
A* Manhattan	6.711156	210172	Sim	192	50856

4.

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

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

Estratégia	Tempo (seg)	Espaço	Encontrou a solução?	Profundidade	Custo
DFS	—	—	Não	—	—
BFS	—	—	Não	—	—
IDFS	—	—	Não	—	—
Gulosa Misplaced	0.514089	17332	Sim	247	1715
Gulosa Manhattan	0.429825	11848	Sim	273	2264
A* Misplaced	14.500981	352424	Sim	329	6007
A* Manhattan	1.357310	40408	Sim	157	4781

6. Comentários Finais e Conclusões

O jogo dos 15 é um problema clássico que tem sido muito explorado como um problema de procura. Neste trabalho, foram implementados 5 métodos, sendo dois deles (A* e Gulosa) implementados com duas funções heurísticas diferentes (Misplaced Tiles e Manhattan Distance), onde comparamos o desempenho e a eficiência de cada um deles para o jogo dos 15.

Para a primeira configuração dada, conseguimos perceber que quase todos os métodos encontram a solução à mesma profundidade, exceto o DFS que não encontra uma solução, atingindo o máximo de memória física do computador. Sendo o método Gulosa Manhattan o que demora menos tempo a encontrar a solução.
Para a segunda, terceira e quarta configuração dada, só os métodos guiados foram capazes de encontrar uma solução, num tempo útil, tendo sido o método Gulosa Manhattan o que demora menos tempo. No entanto, o A* Manhattan encontra uma solução mais próxima ao estado inicial.

Com base nos resultados acima apresentados podemos verificar que as estratégias de procura guiadas, na resolução deste problema, têm uma maior eficiência do que as estratégias de procura não guiadas. Além disso, os resultados mostram que a melhor função heurística é a Manhattan Distance, pois tem um tempo de execução menor. Sendo assim, consideramos que os métodos A* e o Guloso com a função heurística Manhattan Distance são os mais eficientes pelo menor tempo de resposta e profundidade.

Em suma, a realização deste trabalho permitiu-nos obter um entendimento mais aprofundado e claro acerca do funcionamento dos algoritmos de busca.

7. Referências Bibliográficas

WebGrafia:

https://pt.wikipedia.org/wiki/Jogo_dos_15
https://pt.wikipedia.org/wiki/Heur%C3%ADstica
https://pt.stackoverflow.com/questions/103681/o-que-é-um-algoritmo-guloso
https://www.geeksforgeeks.org/a-search-algorithm/
https://mathworld.wolfram.com/15Puzzle.html
https://www.geeksforgeeks.org/search-algorithms-in-ai/
https://techvidvan.com/tutorials/ai-search-algorithms/

BiblioGrafia:

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach.

Alan Mackworth, David Lynton Poole (2010). Artificial intelligence: foundations of computational agents.

Nils Nilsson (1998). Artificial Intelligence: a new synthesis.

Elaine Rich and Kevin Knight (2017). Artificial Intelligence.