U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Estratégias de Procura/Buscas

Relatório 1

Rui Santos, Tiago Teixeira e Carla Henriques

CC2006: Inteligência Artificial

Profª. Inês Dutra

10 de Março de 2023

Sumário

Estratégias de Procura/Buscas

1. Introdução

A inteligência artificial (IA) é um campo em constante evolução que tem como objetivo desenvolver sistemas computacionais capazes de realizar tarefas que normalmente exigem inteligência humana.

1.1 Problemas de Buscas/Procura

Um dos desafios fundamentais da IA é a busca/procura, que consiste no processo computacional que tenta encontrar uma solução para um problema num espaço de busca. Segundo Russel e Norvig (2022, pág. 82), este espaço pode ser composto por estados ou ações possíveis, no qual um agente (aquele que resolve o problema) tenta antecipar uma sequência de ações que eventualmente irão chegar a um dado objetivo. Ou seja, o agente planeia antecipadamente, tendo em consideração a sequência de ações que irão formar um caminho para o estado final.

1.2 Métodos Utilizados para Resolver Problemas de Procura

Diversos métodos, i.e., algoritmos de busca, são utilizados para resolver problemas de procura, tais como algoritmos de busca não informada, algoritmos de busca informada e algoritmos de busca local. Algoritmos de busca não informada percorrem o espaço de busca sem o auxílio de informações adicionais e o agente estima a distância do objetivo final, enquanto os algoritmos de busca informada o agente não faz estimações de distância e utilizam heurísticas para orientar a busca. Por sua vez, os algoritmos de busca local procuram aprimorar gradualmente uma solução num espaço de busca restrito.

2. Descrição do Problema

O jogo dos 15, chamado em inglês de “puzzle 15”, consiste em 15 peças, numeradas de 1 a 15, e um espaço vazio. As peças podem ser arranjadas inicialmente numa ordem aleatória num tabuleiro quadrado 4x4 e devem ser deslizadas a volta do tabuleiro até estarem numa certa ordem final pretendida. O único movimento valido é o deslize,

dentro dos limites do tabuleiro, do espaço em branco para cima, baixo, esquerda e direita.

O objetivo do jogo é chegar a um estado final a partir de um estado inicial usando apenas os movimentos validos. O mesmo nem sempre é possível.

O jogo dos 15, que era bastante popular no século XIX (Russel e Norvig, 2022, pág. 124), pertence a classe de problemas NP completos, e possui variações com tabuleiros de diferentes tamanhos, como por exemplo 3x3, o chamado de jogo dos 8.

3. Estratégias de Procura

3.1 Procura Não Guiada (Busca "Cega")

Os algoritmos de busca "cega" são aqueles que não levam em consideração conhecimentos heurísticos sobre o problema em a ser resolvido para determinar os ramos da árvore a seres explorados.

3.1.1. Profundidade (DFS – Depth-First Search)

Funcionamento

Imaginando uma árvore, iniciamos no nó na raiz e expandimos tanto quanto possível até chegarmos a um na folha ou até que encontramos a solução. No caso de estarmos num nó folha e ainda existirem subárvores para procurar, usamos *backtrack* e fazemos novamente a mesma "ideia" de pesquisa no próximo nó (figura 1).

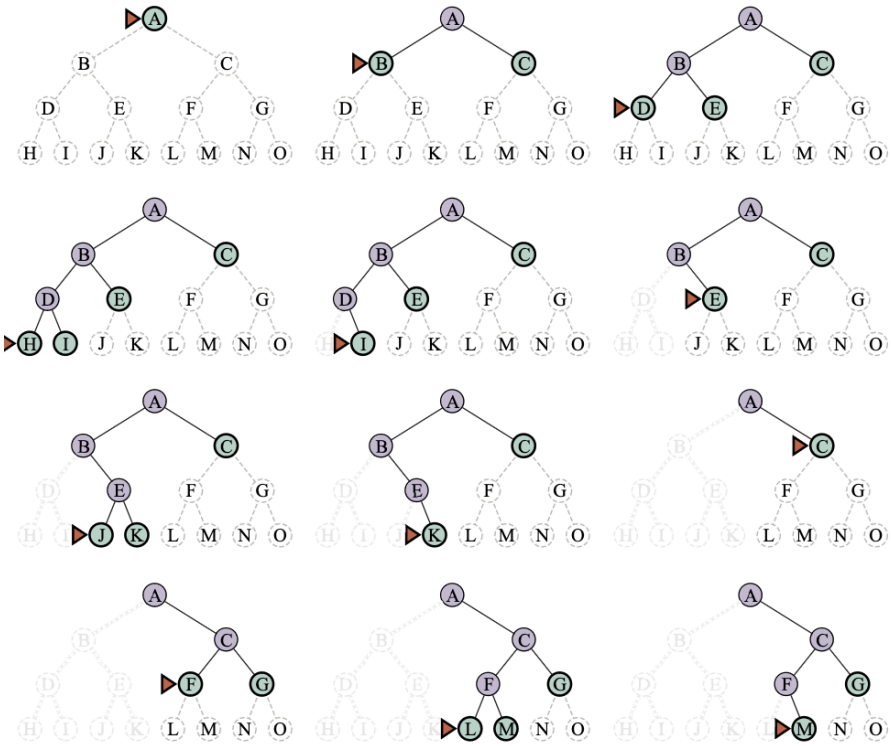

Figura 1 - Árvore binária com 12 passos em pesquisa em profundidade. Por Russel e Norvig (2022, pág. 97).

Aplicação

Aplicamos quando: não nos importamos com o tempo da procura, não queremos uma solução ótima e queremos ocupar menos espaço (menor complexidade espacial).

Complexidade

Considerando b "branching factor" e d profundidade da árvore.

Temporal:

- *Exponencial:* $O(b^d)$

Espacial:

- *Linear:* $O(b * d)$

3.1.2 Largura (BFS – Breath-First Search)

Funcionamento

Imaginando uma arvore, iniciamos no nó raiz e supondo que n é o nível atual, geramos todos os nós no nível n+1.Depois repetimos o processo para os restantes níveis até chegar a solução ou até não existir mais nós (figura 2, em que temos a ordem pelo qual seria "gerado" cada nó numa pesquisa em largura).

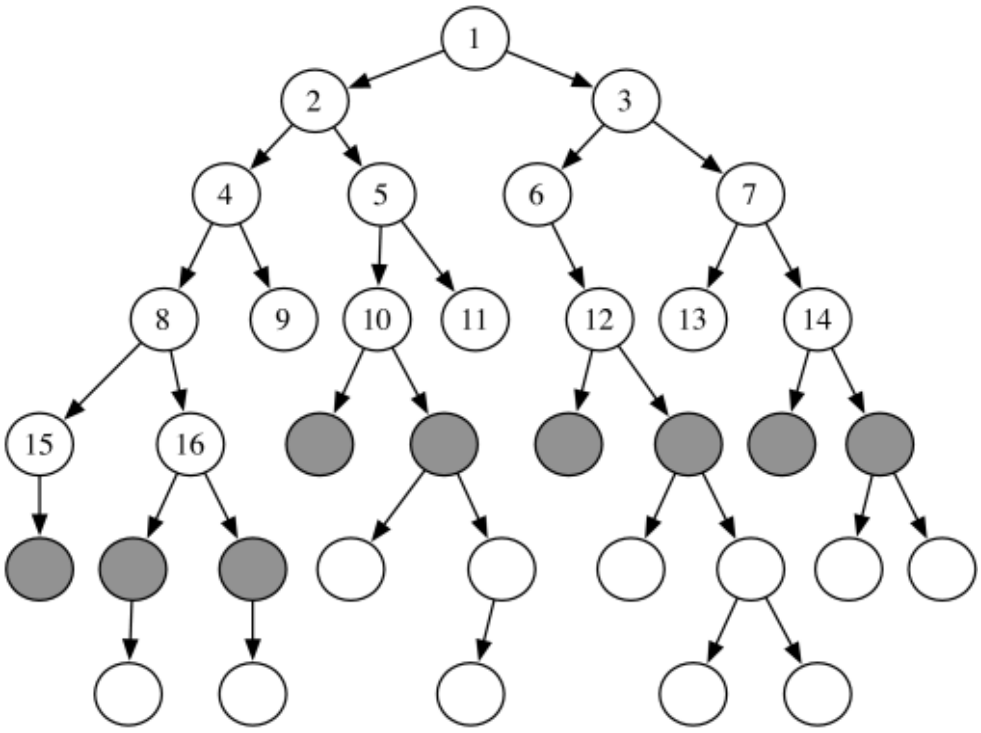

Figura 2 – Ordem na qual os nós são expandidos na pesquisa em largura. Por Poole e Mackworth (2017).

Aplicação

Aplicamos quando: não nos importamos com a complexidade espacial e quando queremos a solução ótima (mais curta).

Complexidade

Considerando b "branching factor" e d profundidade da árvore

Temporal e Espacial

- *Exponencial:* $O(b^d)$

3.1.3 Procura Iterativa Limitada em Profundidade

Funcionamento

Imaginando uma árvore e supondo que estamos a prosseguir da esquerda para a direita, iniciamos no nó raiz e definimos um limite de profundidade (inicialmente igual a 0). Se a solução não estiver no nó raiz aumentamos o limite de profundidade para 1 e geramos todos os descentes do nó raiz, se obtivermos a solução paramos se não aumentamos novamente o limite de profundida para 2 e, da esquerda para a direita, geramos os descendentes de cada nó repetindo o processo de verificar se tem ou não solução nesse nível. Para os restantes níveis o processo é o mesmo, e só termina quando chega ao limite máximo de profundidade ou quando chega a solução.

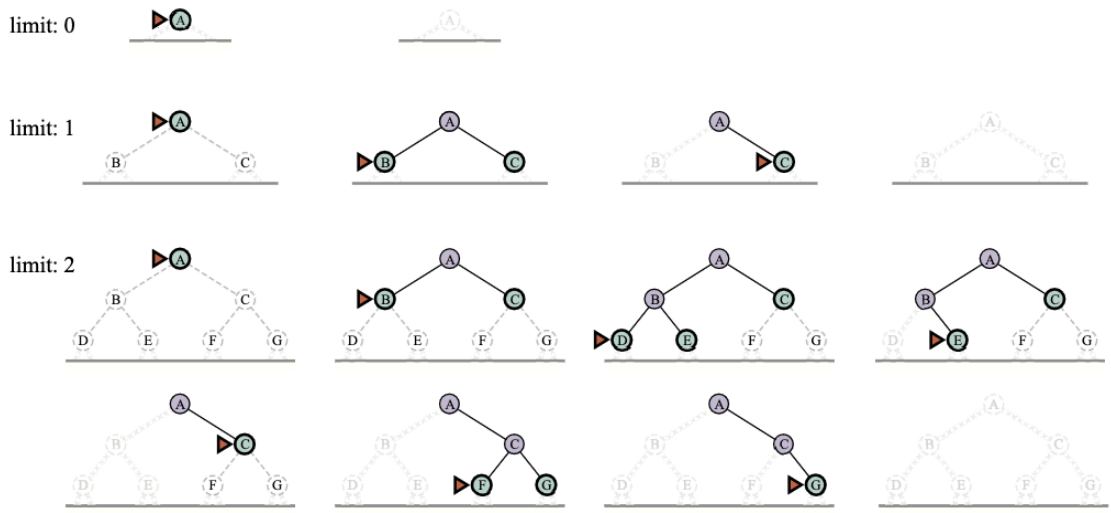

Figura 3 – Quatro iterações na busca iterativa limitada em profundidade. Por Russel e Norvig (2022, pág. 100).

Aplicação

Aplica-se quando: a árvore de procura é muito grande, profundidade da solução não é conhecida, queremos a solução ótima e queremos menor complexidade espacial, assim combinando as vantagens de um BL e DFS (Russel e Norvig, 2022, pág. 98).

Complexidade

Considerando b "branching factor" e d profundidade da árvore.

Temporal:

- *Exponencial:* $O(b^d)$

Espacial:

- *Linear:* O(b * d)

3.2 Procura Guiada

Os algoritmos de procura guiada geram novos estados a partir de informações específicas sobre o problema. É feita uma avaliação heurística para determinar a distância até a solução final.

3.2.1 Heurística

Uma heurística consiste numa função h(n) que atribui um valor a cada nó numa arvore de procura, esse valor ajuda a determinar qual dos ramos da arvore devemos seguir. Existem vários exemplos de funções heurísticas, como por exemplo: Manhattan distance, Euclidian distance ,etc… .É importante salientar que h(n) tem que ser admissível, ou seja, nunca pode sobrestimar o "custo real".

3.2.2 Gulosa

Funcionamento e Aplicação

Um algoritmo de procura gulosa ("greedy") resolve um problema selecionando a melhor opção disponível no momento através de estimativa da melhor avaliação heurística. Não se preocupa se o melhor atual leva para uma solução ótima.

Utiliza a função f(n) = h(n) , em que h(n) e o custo estimado do caminho mínimo entre o estado atual e o objetivo.

Aplicamos este método quando o problema que estamos a tentar resolver tem os seguintes requisitos: a solução ótima pode ser encontrada selecionando em cada passa a "melhor" opção disponível e se a solução ótima para o problema corresponde a na seleção da solução ótima para os seus sub-problemas.

Simmons, Hoeppke e Sutherland (2019), o algoritmo Greedy pode falar em maximizar o resultado de cima-abaixo na árvore, porque ele não antecipa as opções seguintes, assim escolhendo uma solução suboptima na sua iteração atual (figura 4).

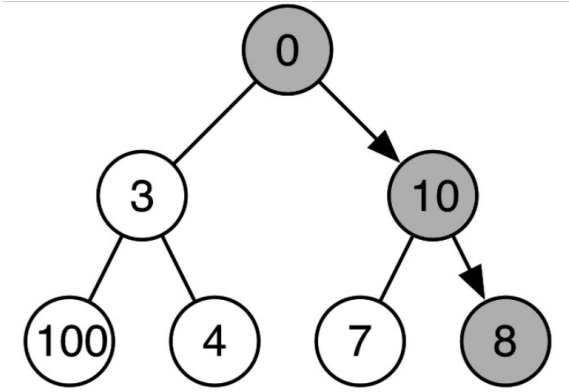

Figura 4 - Árvore que representa a utilização de um algoritmo “greedy”. Por Simmons, Hoeppke e Sutherland (2019).

Heurística Utilizada

Utilizamos duas heurísticas admissíveis, que são as mais comuns e “melhores” para resolver o problema dos 15. Essas heurísticas são:

- H1: número de pecas fora do lugar (não contando a peça em branco) que não estão no respetivo lugar representado no estado final.
- H2: somatório das distâncias de cada peça (não contando a peça em branco) ao seu “destino” no estado final (Manhattan distance).

Complexidade

Considerando b “branching factor” e d profundidade da árvore

Temporal e Espacial

- *Exponencial:* $O(b^d)$

*3.2.3 Busca A**

Funcionamento e Aplicação

Imaginando uma árvore, A* funciona gerando o caminho mínimo entre um no inicial e um no objetivo, utilizando uma função monótona f, que é definida da seguinte forma:

$f(n) = g(n) + h(n)$ onde:

- $f(n)$ é a função monótona que estima o "custo" de um nó n.
- $g(n)$ é o custo do caminho, até ao momento, para chegar ao nó n.
- $h(n)$ é a heurística admissível, ou seja, nunca sobrestima o custo real do caminho.

Aplicamos este algoritmo quando queremos encontrar o melhor caminho, envolvendo no processo o menor custo, em termos de distância e tempo.

Heurística Utilizada e Justificação

Utilizamos duas heurísticas admissíveis, que são as mais comuns e "melhores" para resolver o problema dos 15. Essas heurísticas são:

- H1: número de pecas fora do lugar (não contando a peça em branco) que não estão no respetivo lugar representado no estado final.
- H2: somatório das distâncias de cada peça (não contando a peça em branco) ao seu "destino" no estado final (Manhattan distance).

Complexidade

Considerando b "branching factor" e d profundidade da árvore

Temporal e Espacial

- *Exponencial:* $O(b^d)$

4. Descrição da Implementação

4.1 Linguagem Utilizada

A escolha da linguagem Java para implementar as buscas no problema do jogo dos 15 deve-se a várias razões. Em primeiro lugar, Java é uma linguagem de programação amplamente utilizada e com uma grande comunidade de desenvolvedores, o que torna mais fácil encontrar recursos e soluções para problemas específicos. Além disso, Java é uma linguagem orientada a objetos, o que significa que a implementação do problema pode ser modelada usando objetos e classes, tornando a implementação mais fácil de entender e manter.

Outra vantagem do uso de Java para resolver o problema do jogo dos 15 é a sua portabilidade. Java é uma linguagem multiplataforma, o que significa que o código pode ser executado em diferentes sistemas operacionais sem a necessidade de recompilação. Isso é útil para testar e executar as implementações das buscas em diferentes ambientes. Finalmente, Java também oferece uma grande variedade de bibliotecas e frameworks para trabalhar com estruturas de dados e algoritmos, o que pode tornar a implementação das buscas mais fácil e eficiente.

4.2 Estruturas de dados utilizadas

- Matriz 4*4, que representa um dado *puzzle*. Escolhemos esta estrutura de dados por ser a mais "simples" de entender e utilizar, contudo não é a opção mais eficiente para a manipulação dos dados (podíamos por exemplo ter usado um *array* de tamanho 16 para combater essa ineficiência).

- ArrayList<Node> para guardar os descendentes de um dado nó. Escolhemos esta estrutura por já ter uma implementação robusta no java e por ser análoga a um array. Em termos eficiência de manipulação dos dados achamos que para a objetivo, que é guardar e aceder a nós , é uma boa escolha.

- ArrayList <Puzzle> para guardar o caminho feito para chegar a um determinado no , a justificação de o porque que usamos e a sua eficiência e igual a do ponto anterior.

- PriorityQueue<Node> para usar em métodos de procura informado. Escolhemos esta estrutura de dados por já ter uma implementação robusta em java e porque se adequava melhor para a implementação dos métodos de procura informados. Em termos de eficiência de manipulação dados achamos que para o objetivo, que é guardar e aceder a nós, é uma boa escolha.
- Queue<Node>, estrutura auxiliar adicionada no código.
- Stack<Node>, estrutura auxiliar adicionada no código.

4.3 Estrutura do Código

Detalhamento das principais estruturas utilizadas no código que implementamos para o jogo dos 15:

4.3.1 Classe Main

boolean Check_Initial_to_Final(Puzzle p1, Puzzle p2)

Método que indica se um dado estado final(Puzzle p2) e atingível a partir de um estado inicial (Puzzle p1).

ArrayList<Node> MakeDescendants(Node n)

Método que gera todos os descendentes de um dado nó n e retorna um ArrayList<Node> com esses mesmo descendentes.

Start(Puzzle Inicial, Puzzle Final, String Queueingfunction)

Método que inicia o programa, caso seja possível chegar de Puzzle Inicial ao Puzzle Final , e executa o método de procura indicado por a String Queueingfunction.

Node DFS (Node startNode, Puzzle Final)

Implementação do método de pesquisa DFS .

Node BFS(Node startNode, Puzzle Final)

Implementação do método de pesquisa BFS , tendo a mesmo uma função auxiliar BFS_helper.

Node DFS_Iterative(Node startNode, Puzzle Final, int max_depth)

Implementação do método de pesquisa Iterative_DFS , tendo a mesmo uma função auxiliar DFS_Iterative_helper

int Hamming_Distance(Node n, Puzzle Final)

Método que retorna o valor da heurística "número de pecas fora do lugar (não contando a peça em branco) que não estão no respetivo lugar representado no estado final." de uma dado nó n.

int Manhattan_distance(Node n, Puzzle Final)

Método que retorna o valor da heurística "somatório das distâncias de cada peça (não contando a peça em branco) ao seu "destino" no estado final" de um dado nó n. Este método tens dois métodos auxiliares Manhantan_distance_helper(int i, int j, int k, int p, int[][] cur, int[][] Final) e find_position(int a, int[][] Final) .

Node A_Star_Humming(Node starNode, Puzzle Final)

Implementação do método de pesquisa A* , com a heurística "número de pecas fora do lugar (não contando a peça em branco) que não estão no respetivo lugar representado no estado final.".

Node A_Star_Manhattan(Node starNode, Puzzle Final)

Implementação do método de pesquisa A* , com a heurística "somatório das distâncias de cada peça (não contando a peça em branco) ao seu "destino" no estado final".

Node Greedy_With_Heuristic_Hamming(Node starNode, Puzzle Final)

Implementação do método de pesquisa Gulosa, com a heurística "número de pecas fora do lugar (não contando a peça em branco) que não estão no respetivo lugar representado no estado final.".

Node Greedy_With_Heuristic_Manhattan(Node starNode, Puzzle Final)

Implementação do método de pesquisa Gulosa, com a heurística "somatório das distâncias de cada peça (não contando a peça em branco) ao seu "destino" no estado final".

4.3.2 Classe Node

Classe que representa um nó, e contem os seguintes atributos:

- Puzzle current, puzzle que corresponde a o no “atual”;
- ArrayList <Puzzle> path, ArrayList que guarda o caminho de Puzzle’s que foi feito até nó atual;
- int depth, indica a profundidade do no atual;
- int cost, indica número de nós, sem contar o inicial, que foram necessários para chegar ao nó atual;
- Puzzle parente, Puzzle pai que gerou o nó filho (elemento anterior no path).

4.3.2 Classe Puzzle

Classe que representa uma configuração do Puzzle dos 15, e contem os seguintes atributos:

- int x, y ,que representam a posição do espaço em branco na matriz tab;
- int[][] tab , que representa o puzzle em si;
- int f_n , guarda o valor da função f para as pesquisas informadas.

int nr_inversions(int[] v)

Método que retorna o número de inversões de um Puzzle.

String get_permutation_parity()

Método que retorna à paridade da permutação do Puzzle.

boolean is_solvable()

Método que retorna se uma dada instancia do Puzzle tem solução.

boolean move_up(), move_down() ,move_left() e move_right

Método que executa os movimentos do Puzzle se forem validos e retorna verdadeiro ou falso consoante a validez do movimento.

void set_state(int[] vec)

Método que recebe um array de inteiros e coloca-o na matriz tab (carrega a configuração de um puzzle).

5. Resultados

Foram encontrados os seguintes os resultados apresentados na Tabela 1 para as execuções dos algoritmos em cada uma das configurações e estratégias implementadas.

Estratégia (Heurística)	***Tempo(s)***	***Espaço*** [(1)]	***Encontrou a solução?***	***Profundidade /Custo***
DFS	Não termina	-	Não	-
IDFS	3,447s	1131231	Sim	12
BFS	4,665s	5146014	Sim	12
Gulosa (Humming)	10,965s	5146014	Sim	12
Gulosa (Manhattan)	0,1s	63	Sim	12
A* (Humming)	0,126s	265	Sim	12
A* (Manhattan)	0,097s	107	Sim	12

(1) Máximo de nós armazenados simultaneamente em memória durante a execução.

Tabela 1 – Sumário dos resultados da execução de diferentes algoritmos implementados.

6. Conclusões Finais

Pode-se observar que os métodos de procura informados têm um desempenho melhor do que outras estratégias de busca, pois encontram a solução final com menos tempo, e/ou percorrendo menor caminho, e/ou ocupando menos espaço na memória.

A busca em profundidade (sem um limite imposto) não encontra solução, pois a Stack do Java atinge o seu limite. Já a busca em profundidade iterativa, tem uma expansão de nós menor do que a busca em largura, considerando que ao momento que o objetivo final é alcançado ela não percorre outros caminhos.

O algoritmo guloso tem o melhor desempenho com a heurística Manhattan, porque além de encontrar a solução ótima, ele ocupa menos espaço na memória. O A* na Manhattan também ocupa menos espaço na memória e tem um desempenho de tempo ao Guloso em Manhattan.

A busca em largura e a busca em custo uniforme ocupam mais espaço na memória, ainda assim encontrando uma solução ótima, com menor custo.

7. Bibliografia

Abiy, T., Pang, H. e Tiliksew, B. A* Search. *Brilliant.* Disponível em https://brilliant.org/wiki/a-star-search/

Poole, D. e Mackworth, A. (2017). *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press.

Nilsson, N. (1998). *Artificial Inteligence: a new synthesis.* Morgan Kaufmann Publishers.

Ravikiran, S. (2023). *A* Algorithm Concepts and Implementation.* Disponível em https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm

Ribeiro, P. (2022). *Estruturas de Dados.* Disponível em https://www.dcc.fc.up.pt/~pribeiro/aulas/edados2223/

Russell, S., Norvig, P. (2022). *Artificial Intelligence: A Modern Approach*. (4a edição). Pearson Education Limited. Global Edition.

Simmons, B., Hoeppke, C. e Sutherland, W. (2019). Beware Greedy Algorithms. *J Anim Ecol*. 88: 804– 807. Disponível em https://doi.org/10.1111/1365-2656.12963

Szabo, F. (2015). Manhattan Distance. *The Linear Algebra Survival Guide.* Disponível em https://www.sciencedirect.com/topics/mathematics/manhattan-distance

Busca em Profundidade. *Wikipedia.* Disponível em https://pt.wikipedia.org/wiki/Busca_em_profundidade

Depth First Search (DFS). *Programiz.* Disponível em https://www.programiz.com/dsa/graph-dfs

Greedy Algorithm. *Programiz*. Disponível em https://www.programiz.com/dsa/greedy-algorithm