INTELIGÊNCIA ARTIFICIAL

Relatório - trabalho 1

24 DE MARÇO DE 2022

FACULDADE DE CIÊNCIAS
Ana Catarina Gomes – 201804545, Cláudia Maia – 201905492, Maria Sobral – 201906946

INTRODUÇÃO

Um problema de busca consiste em chegar a um determinado estado final a partir de um estado inicial, utilizando uma árvore de pesquisa para esse efeito, que iremos percorrer de diferentes maneiras (consoante o método de pesquisa utilizado), de forma que no final obtenhamos o caminho que nos levou do ponto de partida até ao nosso objetivo.

Existem vários métodos para resolver este tipo de problemas. No presente trabalho vamos focar-nos nos seguintes: busca em profundidade (DFS - Depth First Search), busca em largura (BFS – Breadth First Search), busca iterativa limitada em profundidade (IDFS), busca gulosa e A*.

O JOGO DOS 15

O jogo dos 15 consiste numa matriz 4x4, onde há 15 células numeradas de 1 a 15 e uma célula em branco. O objetivo do jogo é partir de uma configuração inicial aleatória e chegar a uma outra configuração que nos é dada. Para isto, podemos apenas mover a célula em branco para cima, para baixo, para a direita ou para a esquerda (desde que esta jogada seja válida).

ESTRATÉGIAS DE PROCURA NÃO INFORMADAS

Quando se aplica cada uma?

As estratégias de pesquisa não informadas constituem a forma mais básica de encontrar a solução para dado problema. Dependendo do tipo de estratégia, nem sempre é possível encontrar a solução, por exemplo, para o jogo dos 15 a pesquisa em profundidade pode ou não encontrar a solução, visto que a árvore tem profundidade infinita. Já a pesquisa em largura encontra sempre uma solução, mas para tal percorre todos os nós anteriores à mesma. Ainda assim, estas estratégias não informadas podem ser mais úteis noutros contextos, principalmente quando a estimativa do custo é demasiado custosa, ou não se conhece nenhuma função que faça essa aproximação, ou ainda se a dimensão do problema não justificar o recurso a estratégias informadas.

Vamos abordar três buscas não informadas distintas:

- Busca em profundidade (DFS)

  No algoritmo DFS a árvore é percorrida em profundidade, ou seja, começamos na raiz e vamos expandindo sempre o primeiro filho de cada nó até ele não ter mais sucessores. Depois voltamos para trás na árvore e vamos para o próximo nó que ainda tem sucessores que não foram explorados e assim sucessivamente.

  A sua complexidade espacial é O(bm) e a sua complexidade temporal é $O(b^m)$, onde b é o fator de ramificação da árvore e m é a profundidade máxima da árvore.

  O ponto fraco desta estratégia é que não é nem completa nem ótima.

- Busca em largura (BFS)

  A BFS é um algoritmo onde o nó da raiz é expandido primeiro, logo de seguida são expandidos os seus sucessores, a seguir os sucessores destes últimos e assim sucessivamente. Ou seja, todos os nós de uma certa profundidade são expandidos antes que algum nó do nível seguinte seja expandido.

  A sua complexidade temporal e espacial é $O(b^d)$, onde d é a profundidade da solução que se encontra mais perto da raiz. Esta estratégia já é completa e também ótima.

- Busca Iterativa Limitada em Profundidade (IDFS)

  A IDFS pretende combinar os benefícios da BFS, no que respeita à completude e otimalidade, com os da DFS, nomeadamente uma boa complexidade espacial. Para isso, parte de um limite imposto à priori e vai aumentando gradualmente até encontrar a solução, que será a ótima, tendo em conta que no caso o custo é uma função não decrescente da profundidade de nó. Apesar de a estratégia desperdiçar tempo a visitar alguns nós diversas vezes, esse tempo não é problemático porque visita mais vezes os nós de profundidade pequena e é sabido que na parte superior da árvore há muito poucos nós. Esta ligeira perda de tempo é compensada pelo considerável ganho na poupança de memória, visto que apenas armazena os nós da subárvore na qual se encontra. Assim, a complexidade temporal é $O(b^d)$ e a espacial é O(bd), onde d representa a profundidade da solução encontrada.

ESTRATÉGIAS DE PROCURA INFORMADAS

Quando se aplica cada uma?

As estratégias de procura informadas são particularmente úteis em pesquisas ao longo de árvores que não têm um limite de profundidade, como é o caso do jogo dos 15. Estas estratégias visitam nós prioritários, isto é, que se julga estar mais próximo da solução. O objetivo é, além de permitir encontrar uma solução em casos de árvores não finitas, reduzir o número de nós visitados que não conduzem a uma solução e, em certos casos, encontrar a solução ótima de forma eficiente. Assim, a função que estima o custo assume um papel importante na eficiência e otimalidade da pesquisa informada, permitindo à máquina ter uma espécie de intuição sobre a utilidade de seguir por dado caminho.

Nas buscas informadas, utiliza-se uma função de avaliação f, que descreve a prioridade com que um nó deve ser expandido. Além disso, é importante ter também em conta o custo para atingir o estado final a partir de um determinado nó. À função h que estima estes custos dá-se o nome de heurística.

Para o problema em questão, iremos utilizar duas heurísticas: (somatório do número de peças fora do lugar) e (somatório das distâncias de cada peça ao seu lugar na configuração final).

- Busca Gulosa

  Este algoritmo tenta expandir o nó que está mais perto do objetivo final, para conseguir encontrar a solução rapidamente, retornando a primeira que encontra. Porém, este método avalia os nós apenas através da função heurística, ou seja, f(n) = h(n). Na teoria, esta estratégia de busca não é ótima, a escolha do nó a expandir em cada momento, apesar de informada, pode não corresponder ao nó que levará à solução ótima. Além disso, não é completa pelo facto de não verificar se um nó já foi expandido, havendo a possibilidade entrar em ciclo. No entanto, para o propósito deste trabalho, fizemos uma alteração ao algoritmo original, impedindo que entrasse em ciclo.

- A*

  Já o método A*, quando avalia um nó, tem também em conta o custo do caminho desde a raiz até ao nó corrente (g(n)), uma vez que a função avaliação usada é f(n) = g(n) + h(n). Ou seja, f(n) dá-nos o custo estimado da solução mais barata através de n. Esta estimativa permite guardar, de forma ordenada, os nós gerados e assim visitar o nó com um menor custo, independentemente de já se ter gerado uma solução, já que esta pode não ser a solução ótima, pelo que convém continuar a explorar as opções de mais baixo custo. Assim, a solução retornada é a que surge em primeiro lugar na fila ordenada de nós a visitar, dando-nos a garantia que é de facto a solução ótima.

DESCRIÇÃO DA IMPLEMENTAÇÃO

A linguagem de programação escolhida foi o Java, uma vez que é a linguagem que os elementos do grupo melhor conhecem. Além disso, por ser orientada a objetos, facilita na implementação dos nós de uma árvore com as especificidades necessárias para o problema, ou seja, um nó caracterizado pelo tabuleiro que representa profundidade, nó pai e heurística (este último atributo apenas no caso das buscas não informadas). Além destes dois tipos de nós, usaram-se as seguintes estruturas de dados: LinkedListStack (para armazenar os nós visitados e futuros nós a visitar no caso das buscas em profundidade), LinkedListQueue (para guardar os futuros nós a visitar na pesquisa em largura) e MinHeap (para guardar de forma inteligente os nós a visitar em buscas informadas, tendo como critério de ordenação a heurística de cada nó).

De referir que a escolha da estrutura de dados para guardar os nós armazenados foi feita tendo por base a ideia de que nenhuma das estruturas de que dispúnhamos serviria o propósito de forma eficiente. Isto é, tanto na LinkedListStack, como na LinkedListQueue e entre outras formas de usar LinkedList, para verificar se um nó pertence à lista é preciso percorrê-la até encontrar ou, pelo contrário, percorrer a lista inteira e verificar que o elemento não está na lista, sendo necessário voltar a colocar os elementos entretanto retirados de volta na lista. Fazer isto para cada nó gerado é de facto custoso. No entanto,

a única opção diferente seria guardar os nós num array. Neste caso evitaríamos, pelo menos, a necessidade de voltar a colocar os elementos que, entretanto, foram analisados. Mas, dado que o Java requer que um array tenha uma capacidade limite máxima definida à priori, para usar esta estrutura para o efeito pretendido teríamos que definir um limite superior bastante elevado para não correr o risco de esgotar o espaço do array durante a pesquisa. Assim, para a busca em largura, gulosa e A* o limite máximo a definir seria o pior caso possível (b^d) para a busca em profundidade seria b*d e para a iterativa em profundidade b^m, onde b é fator de ramificação (que no caso sabemos que é em média 2), d é o nível de profundidade da árvore e m o primeiro limite de profundidade imposto para a pesquisa iterativa em profundidade que permite encontrar uma solução. Ora o limite máximo do espaço a usar, no pior caso, para guardar os nós visitados não é usado na totalidade, uma vez que as pesquisas encontram a solução bem antes de usar todo este espaço, pelo que apesar de ser um limite seguro para a execução da procura, ultrapassa largamente o que realmente é necessário, pelo que não se justifica usar um array. Além disso, tendo por objetivo preparar o programa para receber qualquer input, não é aconselhável deduzir a profundidade da solução de um problema qualquer.

Prosseguindo com a explicação da estrutura do código, convém referir que a ideia genérica do jogo, bem como as funções intimamente ligadas a ele (jogadas possíveis, efetuar uma jogada, solvabilidade, etc) encontram-se no ficheiro IA01. As diferentes buscas encontram-se em ficheiros separados, por uma questão de organização, sendo que são chamadas à vez pelo main da class IA01. Assim, IA01 testa primeiramente se o problema que recebe é solúvel, parando de imediato se não for. Se for possível partir da configuração inicial e chegar à final, então prossegue para as diferentes buscas, nas quais entra apenas com a configuração inicial e final. De notar que a busca iterativa em profundidade requer mais um argumento – o limite imposto à priori. No caso, começa com um limite de 5.

RESULTADOS

Resultados para o conjunto de configurações:

Ex1 (profundidade da solução ótima: 12):

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

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

Ex2 (profundidade da solução ótima: 7):

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

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

Ex3 (profundidade da solução ótima: 16):

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

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

Ex4 (profundidade da solução ótima: 12):

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

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

Estratégia	**Tempo (milissegundos)**				**Espaço (número de nós em memória)**				**Encontrou solução?**				**Profundidade**			
Exemplos	**Ex1**	**Ex2**	**Ex3**	**Ex4**	**Ex1**	**Ex2**	**Ex3**	**Ex4**	**Ex1**	**Ex2**	**Ex3**	**Ex4**	**Ex1**	**Ex2**	**Ex3**	**Ex4**
DFS	∞	∞	∞	∞	∞	∞	∞	∞	N	N	N	N	∞	∞	∞	∞
BFS	156	41	518	86	21410	406	244731	12960	S	S	S	S	12	7	16	12
IDFS	64	5	465	57	20	10	28	20	S	S	S	S	12	7	16	12
Gulosa com heurística A	13	4	64	7	145	8	587	18	S	S	S	S	12	7	18	12
Gulosa com heurística B	9	4	3	3	21	11	22	24	S	S	S	S	12	7	18	12
A* com heurística A	11	2	12	6	146	18	473	47	S	S	S	S	12	7	16	12
A* com heurística B	5	4	8	13	86	52	210	105	S	S	S	S	12	7	16	12

CONCLUSÕES

Com base nos resultados obtidos, podemos verificar que:

- DFS: Não é completa nem ótima, tal como se verifica em todos os exemplos testados. Estes resultados devem-se ao facto de a profundidade da árvore, teoricamente, ser infinita. No entanto, conseguimos detetar 'dead ends', impedindo o algoritmo de visitar nós repetidos. Como sabemos esta estratégia de deteção de 'dead ends' pode impedir-nos de encontrar a solução ótima, mas neste caso é necessária para terminar o programa, mesmo que isso implique não encontrar solução nenhuma.

- BFS: é completa e encontra sempre solução ótima para este tipo de problema, visto que o custo de expansão de um nó é sempre o mesmo., a não ser que tenha um nível de profundidade muito alto, nesse caso não consegue encontrar pois excede a memória disponível. As suas complexidades temporal e espacial são as duas $O(b^d)$ sendo b a ramificação média da arvore de pesquisa que é 2, e d a profundidade da melhor solução.

- IDFS: não é completa se não formos aumentando a profundidade máxima, mas aumentando-a e começando com uma profundidade menor que a da solução ótima, a solução é ótima, pois a primeira que encontra é a com menos profundidade. Se começarmos com um limite de profundidade maior podemos obter uma solução mais custosa, pois a pesquisa é em profundidade e não em largura.

- Gulosa: não é completa e embora a heurística B seja melhor e bastante rápida, a solução encontrada esta muito longe da ótima, pois é muito custosa. Esta estratégia é pouco eficiente em termos de custo pois apenas tem em consideração as heurísticas dos nós e não o caminho percorrido, o que faz com que as profundidades sejam muito elevadas.

- A*: é completa e ótima, como estudado em aula, e a heurística B é melhor em termos de tempo de execução e memória. Este algoritmo é melhor que o guloso pois para além de ter em consideração as heurísticas de cada nó, tem também o custo do caminho, ou seja, tem a capacidade de voltar atrás caso o caminho comece a ser demasiado custoso.

Assim sendo, concluímos que o algoritmo mais eficiente é o A*, pois para além de ser o mais rápido e ocupar menos memória, encontra sempre solução, ou seja, é completo tal como tínhamos estudado. Além disso, é sempre ótima, contrastando com a pesquisa gulosa que, à partida, seria também uma candidata a melhor estratégia, por ser informada. Concluímos ainda que a heurística B é a melhor para ambos os algoritmos informados e que as pesquisas informadas são sempre mais rápidas e eficientes que as não informadas, como era de esperar.

REFERÊNCIAS BIBLIOGRÁFICAS

- Slides da cadeira Inteligência Artificial (CC2006) do ano letivo 2021/2022
- Artificial Intelligence: a Modern Approach, by Stuart Russell and Peter Norvig, 3rd edition, Prentice Hall
- https://en.wikipedia.org/wiki/A*_search_algorithm
- https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
- http://wiki.icmc.usp.br/images/1/11/Aula2_Busca.pdf