4 Março 2019 – João Paulo Santos Monteiro | Nuno José Moreira Bernardes

RESOLVER PROBLEMAS DE PESQUISA

INTRODUÇÃO

Neste relatório iremos apresentar as conclusões e justificações relativas à resolução de problemas de busca, passando por caracterizar um problema deste gênero e mostrar alguns algoritmos usados e os seus fatores de fiabilidade.

O que é agente de resolução de problemas, um problema de busca e como se caracteriza?

i. Os agentes de resolução de problemas usam representações atómicas, isto é, estados que são considerados nulos, sem nenhuma estrutura interna visível para os algoritmos de resolução. Este tipo de problemas começa com a definição do mesmo e as suas soluções. Para tal, iremos usar dois tipos de algoritmos de procura, os de procura não informada – não lhes é dada nenhuma informação sobre como resolver o problema sem ser a sua definição – e os de procura informada, em que lhes é dada uma linha guia de onde procurar por soluções. O objetivo do problema é adotar uma meta e avançar para ela para que o problema seja resolvido. Inicialmente, é necessário formular um objetivo. Consideramos o objetivo como sendo um conjunto de estados do problema, estados esses em que o problema é satisfazível. A função do agente de resolução é procurar saber como atuar para chegar ao objetivo. Mas, antes de fazer isso, ele precisa de decidir que tipo de ações e estados ele deve de considerar. Esse processo é chamado de formulação de problema, processo esse em que, dado um objetivo, se decide que ações e estados devem ser considerados. Um agente com várias opções de valores desconhecidos pode decidir o que fazer examinado primeiramente as ações futuras que poderão originar estados de valores conhecidos. A solução para qualquer problema é uma sequência fixa de ações.

ii. O processo de procurar a sequência de ações que atingem o estado final é chamado de busca. Um algoritmo de busca recebe um problema e retorna a solução na forma de sequências de ações.

iii. Um problema pode ser definido em cinco componentes. O estado inicial em que o agente começa a sua avaliação, a descrição das possíveis ações disponíveis para o agente dado um estado, um modelo de transição que retorna o resultado de fazer uma ação sobre um estado, um espaço de estados que guarda todos os estados já atingidos desde o estado inicial por uma sequência de ações, um teste de solução que verifica se um dado estado corresponde ao estado que queremos atingir e um custo de caminho que guarda um valor da quantidade de ações realizadas até ao estado em que nos encontramos.

iv. A solução ótima é a solução que tem o menor valor de custo de caminho dentro de todas as soluções apresentadas.

v. Para medir a performance dos algoritmos de resolução dos problemas usamos quatro formas distintas. Completude, que verifica se o algoritmo garante encontrar uma solução quando ela existe. Otimalidade, que verifica se a estratégia usada encontrou a solução ótima, complexidade temporal, que retorna quanto tempo demora para encontrar uma solução e complexidade espacial, que informa quanta memória é necessária para realizar a procura. As complexidades temporais e espaciais são sempre consideradas com respeito por alguma medição da dificuldade do problema. A complexidade é expressa em três quantidades, a profundidade do nó mais próximo do objetivo (d), o fator de expansão ou o máximo número de sucessores de um qualquer nó (b) e o comprimento máximo de qualquer caminho no espaço de estados (m).

ESTRATÉGIAS DE PROCURA

Pesquisas não informada são as estratégias que não têm informações adicionais sobre os estados a não ser as informações dadas na definição do problema. Tudo o que podem fazer é gerar sucessores e distinguir o que é um estado final de um estado não final. Todas as estratégias distinguem-se pela ordem com que os tabuleiros são expandidos. Aqui iremos explorar três algoritmos, pesquisa em largura, em profundidade (profundidade limitada) e iterativo em profundidade.

Estratégia de procura não informada (não guiada)

vi. Pesquisa em largura (BFS) é uma estratégia na qual o tabuleiro raiz é expandido primeiro, depois todos os seus sucessores são também expandidos, e assim sucessivamente. Todos os tabuleiros são expandidos numa determinada profundidade na árvore de pesquisa antes de qualquer um dos outros tabuleiros na próxima profundidade sejam expandidos. Isto é alcançável usando uma fila no formato FIFO (os novos nós são adicionados no fim da fila e os nós mais antigos são expandidos primeiro). A verificação de se já atingimos a solução é feita em todos os nós gerados. De notar que o algoritmo descarta qualquer nó que já tenha sido explorado pois qualquer caminho deve ser pelo menos tão profundo quanto o já encontrado. O algoritmo é completo. O tabuleiro mais próximo do tabuleiro final pode nem sempre ser necessariamente ótimo. O BFS é ótimo se o custo do caminho for uma função crescente da profundidade do tabuleiro. O cenário mais comum é que todas as ações têm o mesmo custo. A complexidade temporal e espacial é de $O(b^d)$. Os requisitos de memória são um maior problema para o BFS do que o tempo de execução, mas, existem outras estratégias que requerem menos memória.

vii. Pesquisa em profundidade (DFS) expande sempre os nós mais profundos na fila atual da árvore de pesquisa. A pesquisa procede sempre para o nível mais profundo da árvore, e, à medida que os nós são explorados, são retirados da fila. O algoritmo usa uma fila do tipo LIFO ou seja, o tabuleiro adicionado mais recentemente é o escolhido para expansão. Este tem de ser o nó mais profundo que ainda não foi expandido pois é o único mais profundo que o seu pai, que, na iteração anterior, era

o nó que teria sido escolhido para ser expandido por ser o mais profundo. O problema deste tipo de procura é que falha num número infinito de profundidade, então, em vez de usarmos a DFS, usamos a busca em profundidade limitada (DLS) que funciona da mesma forma que a DFS mas limita a profundidade de forma a que, quando o nó a ser expandido estiver com um custo igual ao limite, ele passa a analisar o próximo nó mais profundo com custo menor que o limite imposto. O problema é que o DLS introduz um outro problema, em este algoritmo pode ser incompleto, nos casos em que o limite é menor que a profundidade da solução ótima do problema. DLS não irá ser ótimo pois irá explorar sempre a subárvore mais à esquerda, até mesmo se o nó objetivo não está nessa subárvore, mas, se outro nó igual à solução for encontrado nessa subárvore mais à esquerda e a profundidade não for ótima, ele vai retornar esse valor. A sua complexidade temporal é $O(b^l)$ e a sua complexidade espacial é O(bl).

viii. Pesquisa em profundidade iterativa (IDFS) é uma estratégia usada juntamente com o DFS que encontra o melhor valor limite de profundidade. O que ele faz é aumentar o limite de profundidade gradualmente (começando em 0, depois 1, 2, 3 ... até infinito) até encontrar solução. IDFS combina os benefícios de DFS e BFS. Como a DFS, os seus requisitos de memória são acessíveis (O(bd)). Como a BFS, é completo com o fator de expansão e finito e ótimo quando o custo do caminho não é decrescente. Mesmo o algoritmo gerando os mesmos estados múltiplas vezes, essa geração não é muito custosa. A sua complexidade temporal é $O(b^d)$, igualando-se assim ao BFS.

Estratégias informadas conseguem encontrar soluções mais eficientemente do que as não informadas. O algoritmo que se baseia na expansão dos tabuleiros nestas estratégias é o BFS em que um nó é selecionado para expansão baseando-se na sua função de avaliação $f(n)$. Esta função é construída como um custo estimado, para que o nó com o valor mais baixo de avaliação seja expandido primeiro. A escolha do valor de f determina a estratégia de procura. A função $f(n)$ denota de duas componentes, $h(n)$que é o custo estimado do caminho menos custoso do estado n até ao estado final e $g(n)$, que é o custo de movimentos até ao estado n. Nestas estratégias iremos falar das heurísticas e de dois algoritmos, um deles já conhecido de unidades curriculares anteriores, o greedy, e o outro é o A* ("A estrela").

Estratégia de procura informada (guiada)

ix. Pesquisa greedy/gulosa tenta expandir sempre o nó que está mais perto do objetivo, na ideia de que provavelmente será o nó que irá chegar à solução mais rapidamente. Por isso, este algoritmo avalia os nós usando apenas a função heurística $f(n) = h(n)$. O algoritmo encontra uma solução sem sequer expandir o nó que está no caminho para a solução, e por isso, o seu custo de procura é sempre mínimo. O algoritmo não é ótimo e isso prova o porquê do algoritmo ter o nome de greedy/guloso, a cada passo tenta sempre chegar o mais perto possível ao objetivo.

É também incompleto, até num estado espacial finito. A complexidade espacial e temporal deste algoritmo é $O(b^m)$, podendo esta ser minimizada se aplicarmos uma boa heurística.

x. A* avalia os nós combinando $g(n)$ e $h(n)$ originado assim a função $f(n) = g(n) + h(n)$, em que $f(n)$ é o custo estimado da solução menos custosa por n. O algoritmo tenta primeiro o tabuleiro com o menor valor de f, fazendo com que esta estratégia, por culminar dois fatores importantes e formando uma boa função, seja completa e ótima. A primeira condição para a otimalidade é que $h(n)$ tem de ser uma heurística admissível. Uma heurística admissível é qualquer uma que não sobrestima o custo para chegar ao objetivo. Porque $g(n)$ é o verdadeiro custo para chegar a n sobre o caminho corrente, e $f(n) = g(n) + h(n)$, temos então uma consequência imediata de que $f(n)$ nunca sobrestima o custo exato da solução sobre o caminho n. Heurísticas admissíveis não sempre otimista porque pensam que o custo de resolver é sempre menor do que realmente o é. A* normalmente fica sem memória antes de ficar sem tempo, concluindo-se que o A* não é prático para problemas de grande escala.

xi. Outras heurísticas usadas são a *memory-bounded heuristic search*, a BFS recursiva, MA* (*memory-bounde A**) e SMA* (*simplified MA**).

xii. As funções heurísticas propostas são duas, $h1$ (o número de peças fora do sítio) e $h2$ (soma das distâncias das peças até a sua posição objetivo, mas conhecida por *Manhattan distance*). Uma forma de caracterizar a qualidade de uma heurística é o fator de expansão efetivo (b*). Se um total número de tabuleiros é gerado por A* para um determinado problema é *N* e a profundidade da solução é d, então b* é o fator de expansão que uma árvore uniforme de profundidade d deve de ter para conter N+1 tabuleiros:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

Uma heurística bem delineada teria um valor de b* perto de 1, permitindo problemas de grande escala serem resolvidos a um custo computacional muito razoável.

xiii. Para o nosso problema usamos só a heurística $h2$ pois é mais eficiente do que $h1$. É fácil de analisar pela definição das duas heurísticas que, para qualquer tabuleiro n, $h2(n) \geq h1(n)$. O termo correto é que $h1$ domina $h2$. Dominar implica ser mais eficiente. O algoritmo A* a usar o $h2$ nunca vai expandir mais tabuleiros que A* a usar $h1$ (exceto para alguns casos particulares). Qualquer tabuleiro com $f(n) < C^*$ será expandido. Podemos também afirmar que qualquer tabuleiro em que $h(n) < C^* - g(n)$ será também expandido. Como $h2$ é tão grande como $h1$ para todos os tabuleiros, todos os tabuleiros que serão expandidos por A* com $h2$ serão também expandidos por $h1$ e $h1$ pode causar outros tabuleiros a expandirem-se. Portanto, genericamente, é melhor usar uma função heurística com valores mais elevados, tendo sempre cuidado de que o tempo computacional para a heurística não é elevado.

DESCRIÇÃO DO PROBLEMA ESTUDADO, ESTRUTURAS DE DADOS UTILIZADAS E LINGUAGEM UTILIZADA

O jogo das 15 peças consiste num tabuleiro 4x4 com 15 peças numeradas e uma peça branca. Uma peça que seja adjacente à peça branca pode movimentar-se no espaço da peça branca. O objetivo é chegar a uma disposição final das peças realizando as movimentações antes referidas. A formulação de um tabuleiro de jogo é descrita da seguinte forma: Um estado, que guarda a posição das peças nesse determinado estado, um estado inicial, que guarda o estado com que começamos o problema, as ações descritas pelo tabuleiro, que são os possíveis movimentos que um determinado estado pode fazer para atingir os estados sucessores a ele, uma condição que testa se o estado atual já representa o estado final e um valor que armazena o custo de movimento do estado.

Linguagem utilizada e vantagens

xiv. Para o nosso trabalho usamos a linguagem Java pois, para além de ser uma linguagem que ambos estamos bem familiarizados e por ser uma linguagem que segue o paradigma de programação orientada a objetos, facilita a manipulação de dados entre a estrutura que usamos no nosso programa.

Estruturas de dados utilizadas

xv. Estes tipos de algoritmos de pesquisa requerem uma estrutura de dados que facilite a manipulação dos dados de cada tabuleiro. Para cada tabuleiro (ou nó) n da árvore de procura, temos uma estrurtura que contém sete componentes:

1) *n.state*: o estado em que as peças se encontram;
2) *n.parent*: o tabuleiro que originou o tabuleiro em que estamos;
3) *n.action*: a ação feita para originar este tabuleiro;
4) *n.path_cost*: o custo de movimentos até chegar ao tabuleiro atual;
5) *n.outOfPlace*: valor da heurística das peças fora do sítio;
6) *n.manhattanValue*: valor da heurística de Manhattan e
7) *n.edSize*: número de tabuleiros em memória até ao tabuleiro atual.

xvi. Para procurarmos as soluções temos de seguir uma estrutura também já predefinida. Os algoritmos de procura funcionam a considerar várias possibilidades de sequências de ações. As sequências possíveis de ações começam dum estado inicial numa árvore de procura com o estado inicial na raiz da árvore. Depois precisamos de considerar avançar em várias ações. Quando fazemos isto, expandimos a árvore a partir do estado em que nos encontramos, originando assim, um novo nível de estados. Após originar o novo nível, temos de decidir em qual das possibilidades queremos avançar, dependendo esta decisão do algoritmo de procura que estamos a usar. O processo repete-se até encontrarmos uma solução.

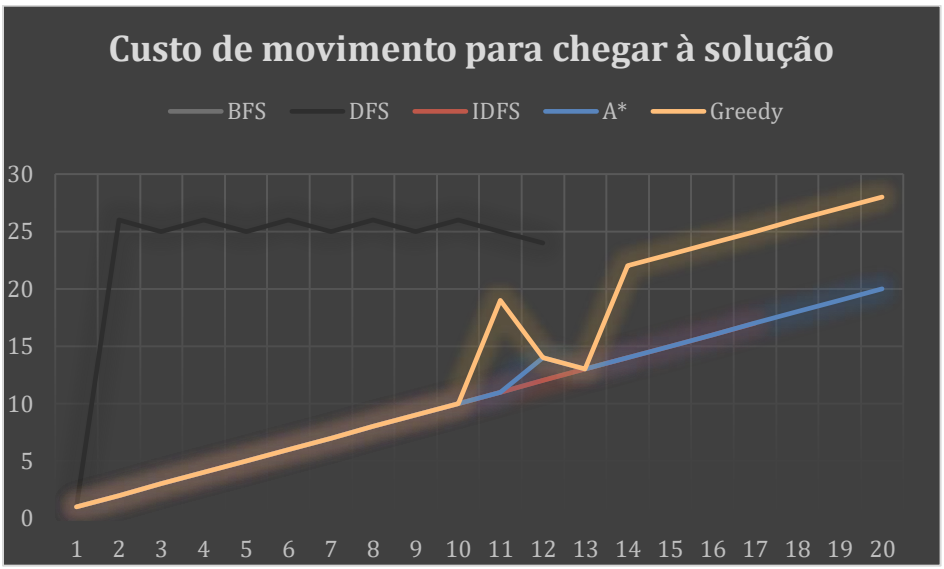
xvii. Os tabuleiros que estão a ser pesquisados terão de ser guardados de forma a que os algoritmos de pesquisa possam escolher qual o próximo tabuleiro a ser expandido seguindo a estratégia escolhida. A estrutura de dados apropriada para isto é uma fila (ou *Queue*). A fila vai-nos permitir decidir a maneira como queremos armazenar os tabuleiros. As variantes usadas no nosso programa são filas FIFO (*first-in first-out*), que retira o elemento mais antigo na fila, filas LIFO (*last-in first-*

out), que retira o elemento mais recente da fila e as filas de prioridade, que vão retirar o elemento da fila com maior prioridade seguindo uma função de ordenação. Temos também de implementar um Set de explorados que nos vai guardar os estados dos tabuleiros que já foram explorados para evitar ciclos e repetições. Para isso usamos um *Set* que pode ser implementado com uma *Hash Table* para permitir a máxima eficiência a procurar por estados repetidos. Com a nossa implementação, a inserção e verificação de estados conseguem ser feitos quase em tempo constante.

RESULTADOS E CONCLUSÕES

Para demonstrar os resultados obtidos pelo nosso programa, fizemos um estudo intensivo com puzzles desde profundidade um até profundidade vinte. Apontamos os resultados obtidos por cada algoritmos e tiramos as conclusões através de gráficos comparativos. (Nota: nos três gráficos que serão apresentados de seguida, quando um valor resultante de uma execução é muito elevado ou quando a execução dá erro de memória, os dados a partir dessa profundidade são descartados, pois o que nos interessa é saber quais os algoritmos que são mais fiáveis para este tipo de problema).

1) Gráfico de custo de movimento para chegar à solução pretendida:



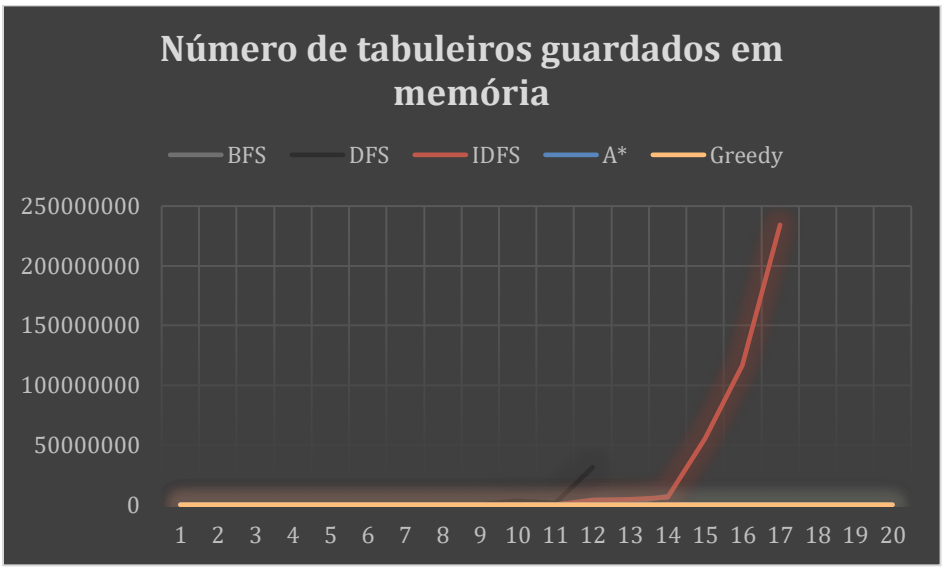
Conseguimos concluir que:

- O BFS, IDFS e A* irão sempre encontrar a solução ótima (a sua reta é linear);

- O DFS irá sempre encontrar solução perto do limite definido (neste caso o limite foi 25) e

- O algoritmo greedy, quanto maior a profundidade, menos probabilidade têm de retornar a solução ótima.

2) Gráfico de número de tabuleiros guardados em memória

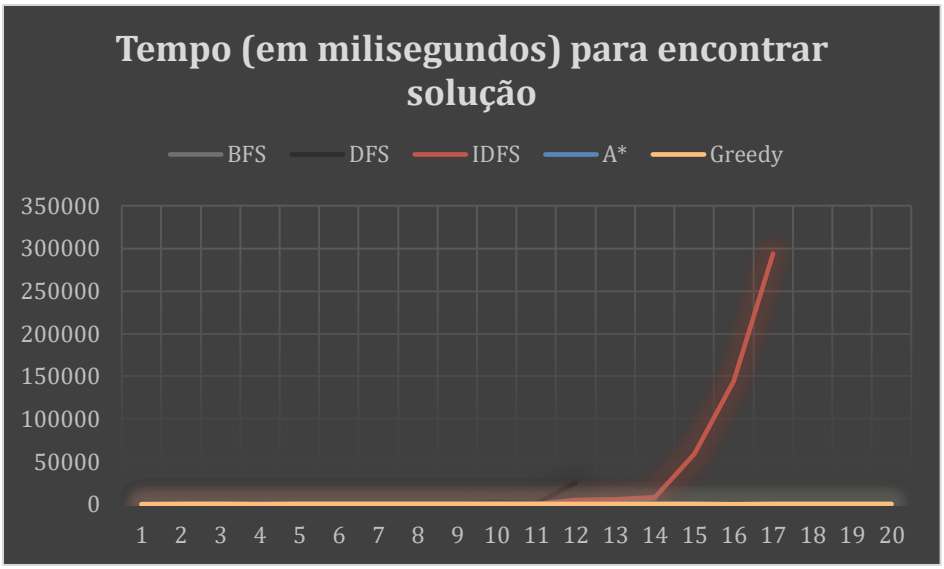


Concluímos deste gráfico e dos dados recolhidos que:

- o BFS, DFS e IDFS guardam muita memória porcausa da maneira como pesquisam sobre os tabuleiros e

- A* e Greedy mantêm sempre valores muito baixos de memória.

3) Gráfico de tempo de execução até chegar à solução (em milisegundos)



Concluímos a partir dos dados recolhidos e do gráfico que:

- o BFS, DFS e IDFS são rápidos para problemas de pouca profundidade;

- A* e Greedy, graças a pesquisar seguindo-se por funções heurísticas, mantêm tempos de execução baixos para a profundidade máxima testada.

Depois de termos analisado bem os valores alcançados por cada algoritmo, conseguimos fazer uma conclusão final de que os algoritmos que seguem uma busca informada conseguem ser muito mais eficazes do que os que não seguem. Mas, dos algoritmos informados que estudamos (A* e Greedy) ainda é preciso descobrirmos qual na realidade é o melhor. Para isso, e como a profundidade vinte não chegou para retirarmos a conclusão final, testamos com um tabuleiro de profundidade 35 e obtemos o seguinte resultado:

Algoritmo	*Movimentos*	*Memória*	*Tempo (ms)*
*A**	35	34605	294
Greedy	177	1478	42

Portanto, se quisermos obter uma solução com poucos movimentos iríamos certamente escolher o A*. No entanto, se quisermos obter a solução num menor período de tempo, a gastar o mínimo de memória possivel e não nos preocupando tanto com a quantidade de movimentos que precisou para chegar à solução, iríamos escolher o Greedy.

BIBLIOGRAFIA

O nosso relatório foi baseado no livro recomendado pela professora no início da unidade curricular:

- *Artificial Intelligence: A Modern Approach* (3ª Edição) *Stuart Russel & Peter Norvig;*

Todas as conclusões, estruturas de dados e algoritmos usados foram confirmados neste livro, pois têm justificações muito completas que tornam fácil o entendimento deste trabalho.