FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO
DEPARTAMENTO DE CIÊNCIAS
DOS COMPUTADORES

RELATÓRIO DE TRABALHO

{ (BUSCA_NÃO_INFORMADA) , (BUSCA_INFORMADA) }

Disciplina

Inteligência Artificial

CC2006

Docente

Inês Dutra

dutra@fc.up.pt

Grupo

André Meira

up201404877@fc.up.pt

Dinis Costa

up201406364@fc.up.pt

Conteúdo

INTRODUÇÃO

Neste relatório são apresentadas, descritas e analisadas diferentes estratégias de busca. Abordamos estratégias não informadas, conhecidas como **busca em profundidade**, **busca em largura** e **busca em profundidade iterativa**, bem como estratégias de busca informadas, **busca gulosa** e **busca A***.

Neste trabalho foram aplicadas as cinco buscas referidas em cima ao **Jogo dos Oito**. Este jogo é representado por uma matriz 3 x 3 que contém oito células numeradas de 1 a 8 e uma célula em branco.

8	5	2
6	7	1
3		4

Partindo de uma configuração inicial com as células baralhadas, o objetivo do jogo consiste em chegar a uma determinada configuração final. Para chegar de uma configuração a outra, pode ser feito um número ilimitado de jogadas que consistem em mover a peça em branco, para cima, para baixo, para a esquerda ou para a direita, mas nunca na diagonal.

Neste contexto em particular, uma busca contém um estado inicial e um estado final. Para além disso, uma função que indica quais os estados acessíveis a partir de cada estado. Cada busca consiste em, partindo do estado inicial, encontrar uma sequência de movimentos que nos permita atingir o nosso objetivo, ou seja, chegar ao estado final.

Assim sendo, o objetivo principal deste trabalho é analisar e comparar o comportamento dos algoritmos de busca enunciados primeiramente. Serão usados como critérios o número de nós guardados em memória, o número total de nós visitados, a profundidade máxima de um nó visitado, a profundidade da solução e o tempo gasto para encontrar a solução. Posto isto, será possível concluir em que tipo de problema se deve aplicar cada uma das diferentes estratégias.

1	2	3
7	8	4
	6	5

1	2	3
	8	4
7	6	5

1	2	3
8		4
7	6	5

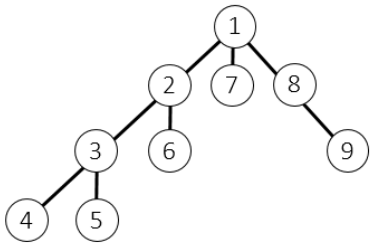
ESTRATÉGIAS DE PROCURA

Algoritmos Não Informados

Algoritmos não informados ou de busca cega, utilizam força bruta para fazerem a pesquisa pelos nós da árvore, pelo que não têm em conta qualquer informação sobre o custo de um nó ou qual é o melhor caminho para atingir a solução. A não ser em casos específicos estes algoritmos têm maiores tempos de execução e não garantem que a solução encontrada seja ótima.

Busca em Profundidade (DFS)

O algoritmo de busca em profundidade começa por expandir a raiz da árvore de pesquisa e aprofunda cada vez mais, até encontrar a solução ou então um nó sem filhos, nó que tem o nome de folha. Ao encontrar um nó folha que não seja solução, ou se o nó for igual a um dos seus antecessores, o algoritmo faz *backtracking* e continua a sua pesquisa, ou num nó irmão do nó atual, ou então num nó antecessor. A imagem seguinte mostra uma árvore onde os números dos nós representam a ordem em que foram visitados.

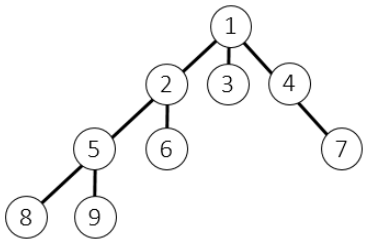


Quando a árvore é implícita (árvore em que os nós filhos podem ser gerados por um algoritmo, como no Jogo dos Oito, movimento da célula vazia para cima, para baixo, para a esquerda ou para a direita), a complexidade temporal da DFS é $O(b^D)$, onde 'b' é o fator de ramificação da árvore e 'D' é a maior profundidade dos nós visitados. A complexidade espacial de pesquisa em profundidade é $O(D \times b)$.

Este algoritmo não deve ser usado em árvores em que a profundidade possa ser ilimitada porque podemos estar a analisar um ramo de profundidade ilimitada que não contém solução. Assim, esta estratégia de busca não é completa nem ótima.

Busca em Largura (BFS)

O algoritmo de busca em largura começa por expandir a raiz da árvore de pesquisa e vai expandindo todos os nós filhos de cada nível de profundidade, até encontrar a solução ou até ter verificado todos os nós. Em seguida, uma imagem que mostra a ordem pelo qual os nós são visitados neste algoritmo.



Caso exista solução, a pesquisa em largura garante que esta será encontrada. Portanto, estamos a falar de uma estratégia completa. Se existir mais do que uma solução, retorna sempre a menos profunda, o que a torna ótima caso não haja diminuição do custo do caminho. Ambas as complexidades, temporal e espacial, são $O(b^d)$, onde 'b' é o fator de ramificação da árvore e 'd' é a profundidade da solução, pois todos os nós folha tem que estar armazenados simultaneamente e visitados.

Busca em Profundidade Iterativa (IDDFS)

O algoritmo de busca em profundidade interativa combina a eficiência de memória da busca em profundidade, com a completude da pesquisa em largura. Esta busca utiliza um limite de profundidade que aumenta iterativamente, sendo que é inicialmente 0, na segunda iteração é 1, e assim sucessivamente, até uma profundidade 'd', que é a menor profundidade a que se encontra uma solução. A IDDFS explora todos os nós de um nível antes de passar ao nível seguinte, de forma semelhante ao BFS.

Este algoritmo de busca é ótimo e completo. A complexidade temporal do algoritmo é $O(b^d)$ e a complexidade espacial é $O(b \times d)$, onde 'b' é o fator de ramificação e 'd' a profundidade a que se encontra a solução.

Utiliza-se esta estratégia de busca quando a árvore de busca é muito grande e a solução tem profundidade desconhecida, pois tem baixo custo de memória e garante que a solução, se existir, é sempre encontrada.

Algoritmos Informados

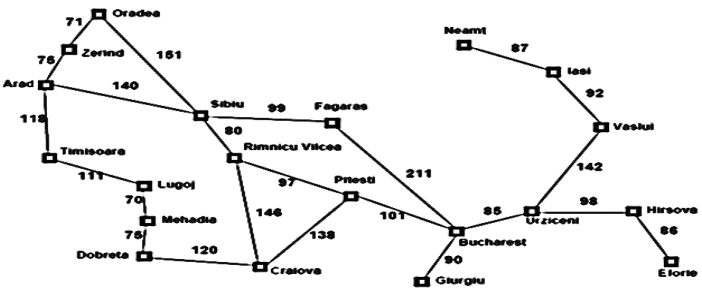
Os algoritmos informados utilizam conhecimento específico do problema em causa para encontrar a solução desejada. Neste tipo de algoritmos utiliza-se uma função de avaliação que descreve a prioridade com que um nó deve ser expandido. Essa função é denominada por função heurística que é uma estimativa de custo. Portanto, é um processo que pretende simplificar quando estamos na presença de questões difíceis e complexas. As suas respostas devem ser consideradas viáveis, ou, no melhor dos casos, altamente viáveis.

Busca Gulosa

A busca gulosa começa por expandir o primeiro nó da árvore de pesquisa, conhecido como nó raiz, e, a cada iteração, utiliza uma função heurística. Esta função estima a distância entre o estado atual e o objetivo, e, eventualmente o custo associado, expandido sempre o nó cuja função estimou como sendo o mais próximo do objetivo, independentemente da profundidade a que este se encontra.

Como vimos, contrariamente às outras buscas, a busca gulosa não visita os nós por uma ordem definida previamente. Começa por expandir o nó raiz e segue para o seu filho que apresenta o valor mais baixo devolvido pela função. Após expandir esse nó, o algoritmo não tem em conta apenas os seus filhos. Para a escolha do próximo nó a expandir também são tidos em conta todos os filhos de todos os nós anteriormente expandidos.

O algoritmo tem complexidade temporal e espacial iguais, $O(b^m)$, onde *'b'* é o fator de ramificação e '*m'* a profundidade máxima da árvore. Dependendo da qualidade da função heurística desenvolvida para o problema, ambas as complexidades podem variar. Esta não é uma estratégia de busca completa uma vez que, *by default*, não verifica a existência de nós repetidos no caminho. Em muitos problemas, uma estratégia gulosa não produz uma solução ótima, mas mesmo assim pode produzir soluções que se aproximam de uma solução ótima.



Busca A*

O algoritmo de busca A* foca-se na minimização da distância e do custo do caminho para atingir a solução. Sabendo que a Busca Gulosa diminui o custo para atingir a solução e que a Busca de Custo Uniforme diminui a distância, ambas foram combinadas. Para além disso, uma não é nem ótima nem completa e a outra, apesar de o ser, é muito ineficiente. Assim sendo, a busca A*, combina as funções de ambas as buscas, obtendo-se f(n) = h(n) + g(n). Onde h(n) é a função heurística da Busca Gulosa, g(n) a função do algoritmo de Busca de Custo Uniforme e f(n) é a soma da estimativa do custo para chegar à solução.

É uma estratégia completa, apenas para grafos com fator de ramificação finito, e ótima. No pior caso a complexidade temporal e espacial da busca A* é exponencial. No entanto, nenhum outro algoritmo ótimo garante expandir menos nós do que este.

Resumo

Estratégia de busca	Complexidade temporal	Complexidade espacial
DFS	$O(b^D)$	$O(D \times b)$
BFS	$O(b^d)$	$O(b^d)$
IDDFS	$O(b^d)$	$O(d \times b)$
Gulosa	$O(b^m)$	$O(b^m)$
A*	$O(b^m)$ ou $O(b \times m)$	$O(b^m)$

Onde,
b – fator de ramificação
d – profundidade da solução
D – maior profundidade dos nós visitados
m – profundidade máxima da árvore

VERIFICAÇÃO DA SOLUÇÃO

Antes de se iniciar uma busca a partir de qualquer configuração do Jogo dos Oito é importante verificar se a configuração em causa tem solução para não se perder tempo desnecessariamente.

Um método para avaliar se uma configuração tem solução, consiste em determinar se a paridade[1] da configuração inicial é igual à paridade da configuração que pretendemos atingir.

Explicando detalhadamente, para cada peça do tabuleiro de uma determinada configuração é somado o número de peças posteriores à peça em questão, cujo valor é superior ao seu. A soma de todos esses valores indica-nos a paridade da configuração. Se a paridade da configuração inicial for igual à paridade da configuração final, então existe solução, caso contrário a mesma não existe. Vejamos as seguintes configurações:

3	4	2
5	1	7
6		8

Inicial
{ 3, 4, 2, 5, 1, 7, 6, 0, 8 }

Peça 3 – 5 inversões
Peça 4 – 4 inversões
Peça 2 – 4 inversões
Peça 5 – 3 inversões
Peça 1 – 3 inversões
Peça 7 – 1 inversão
Peça 6 – 1 inversão
Peça 0 – 0 inversões
Peça 8 – 0 inversões

Paridade Impar (17),
há solução.

6	2	7
5		3
8	1	4

Inicial
{ 6, 2, 7, 5, 0, 3, 8, 1, 4 }

Peça 6 – 2 inversões
Peça 2 – 5 inversões
Peça 7 – 1 inversão
Peça 5 – 1 inversão
Peça 0 – 4 inversões
Peça 3 – 2 inversões
Peça 8 – 0 inversões
Peça 1 – 1 inversão
Peça 4 – 0 inversões

Paridade Par (16),
não há solução.

1	2	3
8		4
7	6	5

Objetivo
{ 1, 2, 3, 8, 0, 4, 7, 6, 5 }

Peça 1 – 7 inversões
Peça 2 – 6 inversões
Peça 3 – 5 inversões
Peça 8 – 0 inversões
Peça 0 – 4 inversões
Peça 4 – 3 inversões
Peça 7 – 0 inversões
Peça 6 – 0 inversões
Peça 5 – 0 inversões

Paridade Impar (25).

1 – A paridade de uma matriz será par ou ímpar consoante for par ou ímpar a soma das inversões efetuadas a cada um dos números de um vetor resultante da listagem das linhas da matriz.

DESCRIÇÃO DA IMPLEMENTAÇÃO

Os diferentes algoritmos foram implementados na linguagem C++. Utilizamos esta linguagem porque é aquela em que nós os dois, elementos do grupo, nos sentimos mais familiarizados. Para além disso é uma linguagem de programação de alto nível com facilidades para o uso em baixo nívelO código está organizado da seguinte forma:

1. A estrutura principal é o *Node* e é usado para representar cada nó da árvore, sendo que as matrizes estão codificadas em inteiro para as comparações serem mais rápidas e eficazes e melhor utilizar a memória por cada nó;

```
struct Node {
    int matrix;
    int zero_x;
    int zero_y;
    const Node* prev;
    int depth;
    int cost;
};
```

2. Para representar a matriz por um inteiro utilizamos as funções *getMatrixEntry* e *setMatrixEntry*, que correspondem a efetuar as operações *matrix[i][j]* e *matrix[i][j] = k* em *arrays* bidimensionais de C++, respetivamente. Isto é, uma matriz com 'n' quadrículas é guardada por um inteiro com 'n' algarismos, onde cada algarismo é guardado/acedido através de um *index* calculado pela sua posição na matriz, *index = i*SIZE + j*. Este *index* é depois utilizado para aceder à posição correspondente da *LOOKUP_TABLE*, tabela que guarda as potencias de 10 entre 0 e 'n', que utilizamos por ser mais eficiente que usar a função *pow(a,b).*

3. As estruturas de dados utilizadas para armazenar os nós ao longo das diferentes pesquisas são *std::queue, std::stack e std::priority_queue*. Utilizamos a *queue* para a pesquisa em largura, a *stack* para ambas as pesquisa em profundidade e a *priority_queue* para as duas pesquisas informadas. A diferença entre a *queue* e a *stack* é a ordem de entrada e de saída dos elementos. Já a *priority_queue* é uma estrutura que permite manter os elementos ordenados através de um critério por nós definido.

4. Na função *main* obtemos a configuração inicial e final através do *input* do utilizador. Depois de escolhida o tipo de pesquisa, representada por um inteiro, chamamos a respetiva função enviando o nó inicial. Posteriormente, a função *generate_children* gera os nós possíveis a partir da configuração atual, sendo que estes nós são guardados temporariamente numa estrutura *vector[]*, até se verificar se já foram utilizados, para serem depois adicionados à estrutura principal da pesquisa.

RESULTADOS

DFS

```
Movimentos: 31075
Segundos: 6.10997
Nos Visitados: 31885
Nos Gerados: 89449
Profundidade da Solucao: 31075
Profundidade Maxima: 31075
```

BFS

```
Movimentos: 23
Segundos: 0.319563
Nos Visitados: 1060540
Nos Gerados: 2858645
Profundidade da Solucao: 23
Profundidade Maxima: 24
```

IDDFS

```
Movimentos: 23
Segundos: 0.384737
Nos Visitados: 372473
Nos Gerados: 3226065
Profundidade da Solucao: 23
Profundidade Maxima: 23
```

GULOSA

```
Movimentos: 69
Segundos: 0.0072755
Nos Visitados: 9664
Nos Gerados: 28510
Profundidade da Solucao: 69
Profundidade Maxima: 89
```

A*

```
Movimentos: 23
Segundos: 0.0006179
Nos Visitados: 1072
Nos Gerados: 3051
Profundidade da Solucao: 23
Profundidade Maxima: 23
```

ANÁLISE

Os resultados foram obtidos utilizando uma máquina em modo 32-bits com sistema de operação Windows 10, processador Intel Core i7-6700HQ 2.6 GHz e 16GB memória de RAM.

Facilmente se confirma as estratégias que tem a otimalidade como característica. As estratégias BFS, IDDFS e A* encontraram a solução ótima, enquanto as estratégias DFS e Gulosa não.

Como vimos na argumentação teórica, a profundidade da solução é igual à profundidade máxima nas pesquisas em profundidade e na A*. Na pesquisa em largura o que acontece é que já foram adicionados nós filhos à estrutura da pesquisa de nós que se encontram no mesmo nível da solução, sendo que a profundidade máxima atingida é um nível acima à profundidade da solução.

É ainda possível observar que os algoritmos informados têm tempos de execução bastante inferiores relativamente aos não informados. Pode-se ainda observar que o tempo de execução do IDDFS é, aproximadamente, duas vezes superior ao da BFS, mas utiliza muito menos memória e chega a uma solução ótima.

Os resultados confirmam que, caso seja conhecida uma função heurística aplicável ao problema, é favorável aplicar um método de busca informado, de forma a obter tempos de execução mais baixos e utilizar menos memória.

Caso contrário, dependendo do problema, deve ser utilizada a busca BFS, se a solução não se encontrar a uma profundidade muito elevada, pois é uma busca que necessita de muita memória. Se o nível for demasiado alto, deve ser usada a IDDFS, pois requer pouca memória e, tal como a BFS, garante que a solução ótima é encontrada.

Se não for necessário uma solução ótima, em problemas onde a profundidade da árvore é conhecida e não seja demasiado elevada, a DFS pode ter um tempo de execução mais reduzido.