CENTRO UNIVERSITÁRIO FEI

VICTOR BIAZON
RA: 119.115-4

RELATÓRIO III – PROGRAMAÇÃO CIENTIFICA
QUEBRA CABEÇA DE OITO PEÇAS

SÃO BERNARDO DO CAMPO
2019

VICTOR BIAZON
RA: 119.115-4

RELATÓRIO III – PROGRAMAÇÃO CIENTIFICA
QUEBRA CABEÇA DE OITO PEÇAS

Relatório de desenvolvimento de algoritmos e busca para resolução do quebra cabeça, desenvolvido pelo aluno Victor Biazon, RA 119.115-4, para disciplina PEL216 – Programação Cientifica, ministrada pelo professor Reinaldo Bianchi.

São Bernardo do Campo
2019

Sumário:

Motivação

Utilização de orientação a objetos para desenvolvimento de algoritmos de busca cega e informada para solução problemas comuns na área de Inteligência Artificial.

Objetivo:

Desenvolver as estruturas dos algoritmos de busca cega e busca informada com programação orientada a objeto.

Utilização dos conceitos de **Busca em Largura** e **Busca em Profundidade**, que são algoritmos de busca cega, para retornar a solução do quebra cabeça de 8 peças com o menor número de movimentos possível.

Utilização dos conceitos dos algoritmos de busca **A*** e **Subida de Encosta**, que são algoritmos de busca informada, para encontrar a solução que para o mesmo quebra cabeça com o menor custo heurístico e menor número de movimentos, estados explorados e tempo possível.

Teoria

Busca Cega

A busca cega trata de algoritmos de busca que verificam e procuram a solução de problemas sem qualquer informação relacionada a “proximidade” que se encontra o objetivo da busca. Por tal motivo os algoritmos de busca cega também são conhecidos por executarem seu papel por “força bruta”, ou seja, geralmente testam todas as possibilidades exaustivamente até encontrar a solução ou até não haverem mais possibilidades (quando não é possível encontrar a solução).

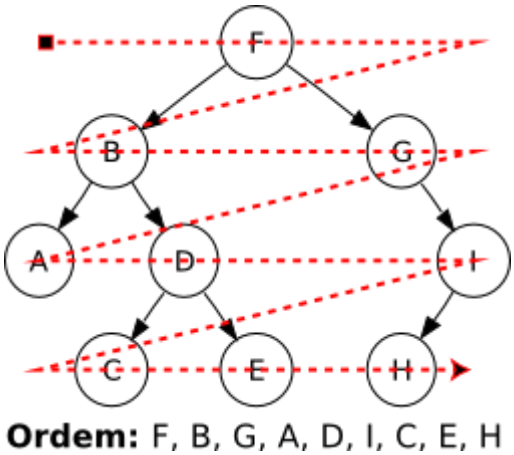
Busca em Largura

A busca em largura, em inglês BFS (Breath First Search), se trata de iniciar-se a busca a partir de um nó, considerado o estado inicial, e executando o processo de expansão de todas as possibilidades a partir das ações possíveis para cada nó que ainda não foram exploradas ou expandidas por nós explorados anteriormente. A exploração da busca em largura se dá através da inserção dos estados expandidos do nó sendo explorado no momento em uma fila e explorando sempre o próximo da fila e verificando se este é o objetivo.

Desta forma o algoritmo percorre camada a camada toda a arvore de busca gerada.

A busca em largura sempre acha soluções ótimas para problemas com custo uniforme, mas não quando custos variam de estado para estado.

Como exemplo a imagem abaixo ilustra o processo de busca em largura.



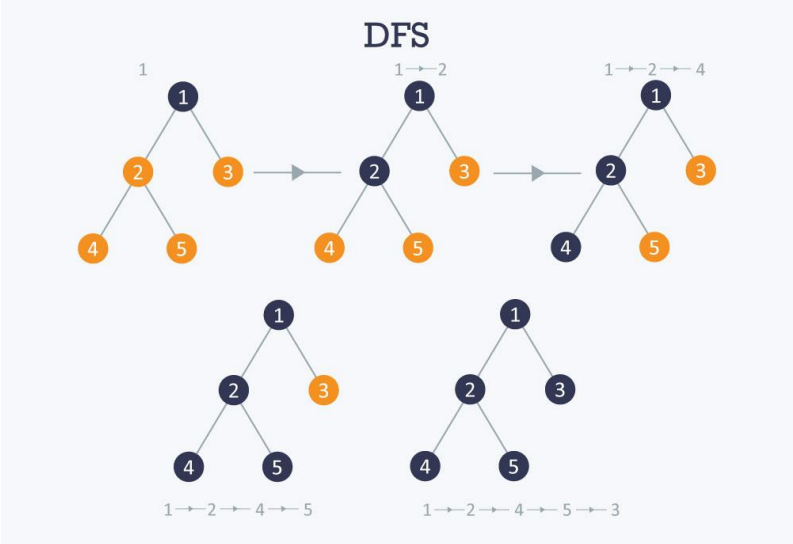
Ordem: F, B, G, A, D, I, C, E, H
[1]

Busca em Profundidade

A busca em profundidade, em inglês DFS (Depth First Search), se trata de iniciar-se a busca a partir de um nó, considerado o estado inicial, e executando o processo de expansão de todas as possibilidades a partir das ações possíveis para cada nó que ainda não foram exploradas ou expandidas por nós explorados anteriormente. A exploração da busca em profundidade se dá através da inserção dos estados expandidos do nó sendo explorado no momento em uma pilha e explorando sempre o topo da pilha e verificando se este é o objetivo.

Desta forma o algoritmo percorre buscando sempre o próximo elemento mais profundo da arvore até que não seja mais possível expandi-lo ou seja encontrada a solução. Caso não seja mais possível expandir em profundidade o nó em questão, o algoritmo marca este como explorado e pega o próximo da pilha, que na prática é na arvore o nó não explorado mais profundo atualmente na arvore. O algoritmo de busca em profundidade pode vir a demandar muito tempo caso o ramo em questão escolhido para exploração seja escolhido erroneamente, podendo inclusive a resposta estar na segunda camada da arvore de busca e o algoritmo ter de percorrer quase toda a arvore para encontrá-la.

Como exemplo a imagem abaixo ilustra o processo de busca em profundidade.



[2]

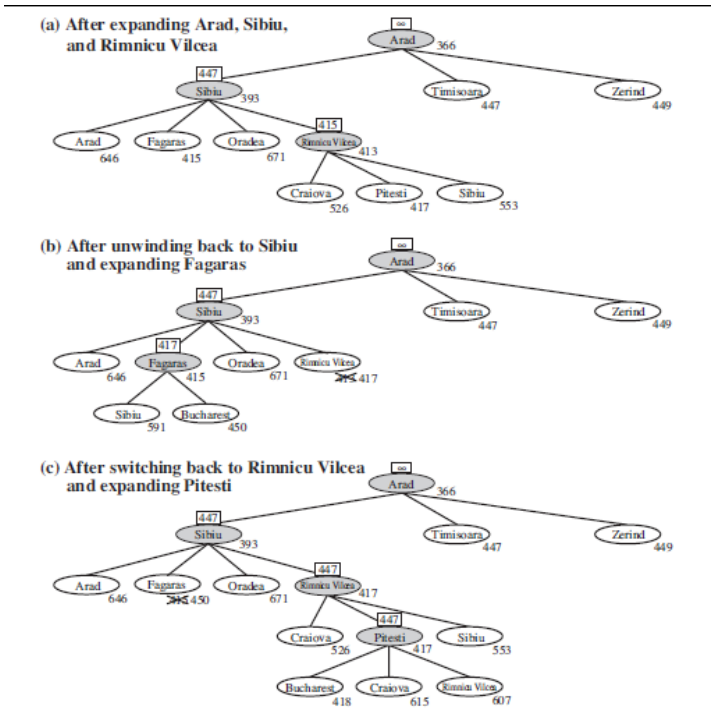
Busca Informada

Os algoritmos de busca informada executam a busca procurando o melhor “Caminho” a ser seguido, e para isso são munidos de ferramentas de avaliação do estado atual que se encontram para julgar qual o melhor nó a ser explorado. Para cada problema é necessário uma heurística diferente, está sendo o valor do custo estimado daquele estado até o estado desejado. É importante que está heurística não seja superestimada pois pode causar erros na busca.

Busca A*

O algoritmo de busca informada A* (A estrela), em inglês A Star Search, sempre segue o próximo nó que representará o menor custo total para a solução. Precisa de uma heurística eficiente para que seja possível encontrar rapidamente o objetivo e sempre retorna soluções ótimas e completas. Para o problema do quebra cabeça de 8 peças geralmente emprega ou a heurística de “número de peças fora do lugar” ou a “distancia de Manhattan”, que conta a distância em movimentos horizontais e/ou verticais para a posição correta da peça, sendo esta última mais eficiente do ponto de vista de coeficiente de geração de nós filhos, ou seja, gera menos nós filhos, pois chega na solução mais rapidamente e explora menos nós.

Na imagem abaixo é ilustrada a expansão de nós com o objetivo de ir de Arad a Bucharest com o menor custo. Nota-se que o algoritmo encontra Bucharest, mas não sendo este o caminho de menor custo, opta por expandir outro nó, encontrando uma solução melhor (418 contra 450 de custo da solução inicialmente encontrada).

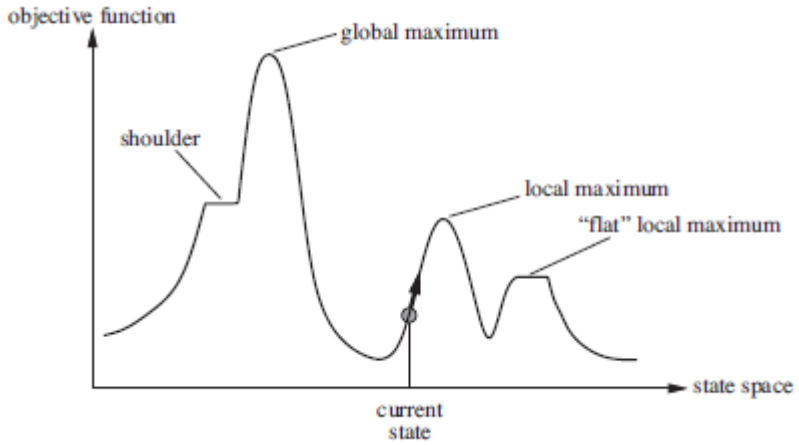


[3]

Busca Subida de Encosta

O algoritmo de busca subida de encosta, em inglês Hill Climbing Search, é um algoritmo guloso, ou greedy. Portanto este utiliza uma heurística para a partir do nó inicial expandir seus nós e em seguida escolher o nó com menor custo e explorá-lo, sempre escolhendo o nó de menor custo total. O Hill Climbing simples tem uma limitação que é o fato de que caso o nó escolhido retorne filhos os quais tem heurística maior que o nó pai o algoritmo deveria retornar que não é possível encontrar a solução, pois este encontrou um mínimo ou máximo local e não o mínimo ou máximo global que é a solução. Para contornar este problema existem diversas abordagens como a Subida de Encosta com Reinicio Randômico, o qual na eventualidade de um mínimo/máximo local reinicia randomicamente o algoritmo, ou Subida de Encosta com Reinicio de Menor custo, ou seja, na eventualidade de um mínimo local, este procura o nó de menor custo não explorado, e continua o algoritmo por este. Por ser um algoritmo guloso este encontra a solução mas não garante uma solução ótima.

Abaixo uma ilustração dos mínimos/máximos globais e locais de um problema genérico.

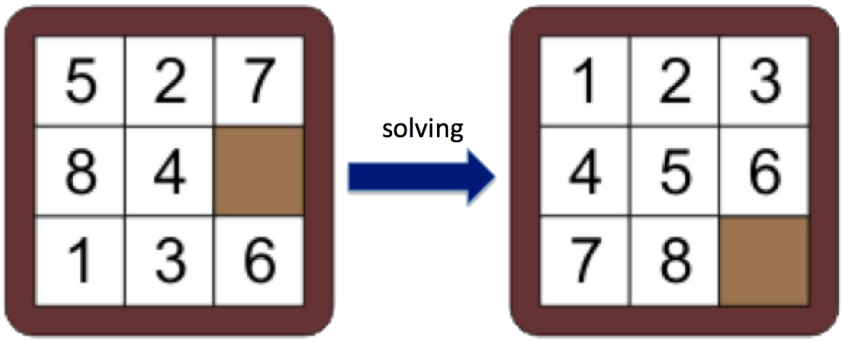


[4]

O Quebra cabeça de 8 peças deslizantes.

O quebra cabeça em questão segue as seguintes regras:

O objetivo é colocar os números em ordem com o espaço em branco no lugar onde ficaria o número 9. Para tal pode-se mover as peças em qualquer direção que seja possível, sempre uma a uma em direção ao espaço vago. A solução que demanda mais movimentos para este quebra cabeça tem 31 movimentos sendo estes os mais trabalhosos para os algoritmos de busca.



[5]

Implementação

Para implementação dos algoritmos de busca, tanto cega quanto informada, foram utilizadas três listas ligadas, sendo estas, a **Frontier** que guarda os estados a serem explorados, a lista **Explored** que guarda os estados já explorados pela busca e a lista **Solução** que após encontrado o objetivo da busca guarda o caminho da solução até o nó inicial para ser exibido.

Abaixo seguem os pseudocódigos dos algoritmos de busca:

Pseudo-código - Busca em Largura

```
BFS(Puzzle)
{
     // cria-se as listas frontier e explored, e define o nó inicial Puzzle.
     frontier = new Lista<Node>();
     explored = new Lista<Node>();
NPuzzle = new Node(Puzzle);

//Insere o nó na fronteira.
     frontier->InsereFinal(NPuzzle);

     // Inicializa a contagem do número de verificações, movimentos e o estado booleano que indica que a solução foi encontrada.
     int Qtde_Verificacoes = 0;
     int movimentos = 0;
     bool isGoal = false;

     //Define o alvo a ser explorado como o primeiro elemento da fronteira tratando-a como uma fila.
     Alvo = new Elemento<Node>;
     Alvo = frontier->GetIni();

     // Verifica se o alvo inicial é o objetivo antes de explorá-lo.
     Se Alvo == Objetivo {
          isGoal = true;
          Imprime("Solucao encontrada");
     }
     Incrementa Qtde_verificacoes;

     // Avalia o próximo elemento da fronteira enquanto a solução não for encontrada.
     Enquanto (isGoal == false)
```

```
            {
                  // Insere na lista explored o primeiro elemento da
fronteira.
                  explored->InsereFinal(frontier->RetiraInicio());
                  //Determina o alvo como o último elemento do explorado.
                  Alvo = explored->GetFim();
                  //Mostra na tela qual quebra cabeça está sendo explorado.
                  Imprime(Alvo);
                  // Gera os filhos do alvo já verificando se são o objetivo,
se algum deles for a função retorna True para IsGoal
                  isGoal   =   Alvo->dado->GerarFilhos(frontier,   explored,
Qtde_Verificacoes);
            }

      Imprime Solução.
```

Pseudo-código - Busca em Profundidade

```
      DFS(Puzzle)
      {
            // cria-se as listas frontier e explored, e define o nó inicial
Puzzle.
            frontier = new Lista<Node>();
            explored = new Lista<Node>();
      NPuzzle = new Node(Puzzle);

      //Insere o nó na fronteira.
            frontier->InsereFinal(NPuzzle);

            // Inicializa a contagem do número de verificações, movimentos e
o estado booleano que indica que a solução foi encontrada.
            int Qtde_Verificacoes = 0;
            int movimentos = 0;
            bool isGoal = false;

            //Define o alvo a ser explorado como o último elemento da
fronteira tratando-a como uma pilha.
            Alvo = new Elemento<Node>;
            Alvo = frontier->GetFim();

            // Verifica se o alvo inicial é o objetivo antes de explorá-lo.
            Se Alvo == Objetivo {
                  isGoal = true;
                  Imprime("Solucao encontrada");
```

```
            }
            Incrementa Qtde_verificacoes;

            // Avalia o próximo elemento da fronteira enquanto a solução não
for encontrada.
            Enquanto (isGoal == false)
            {
                  // Insere na lista explored o ultimo elemento da fronteira.
                  explored->InsereFinal(frontier->RetiraFinal());
                  //Determina o alvo como o último elemento do explorado.
                  Alvo = explored->GetFim();
                  //Mostra na tela qual quebra cabeça está sendo explorado.
                  Imprime(Alvo);
                  // Gera os filhos do alvo já verificando se são o objetivo,
se algum deles for a função retorna True para IsGoal
                  isGoal   =   Alvo->dado->GerarFilhos(frontier,   explored,
Qtde_Verificacoes); }
                  Imprime Solução.
```

Pseudo-código - Busca A*

```
        A_STAR(Puzzle)
        {
            // cria-se as listas frontier e explored, e define o nó inicial
Puzzle.
            frontier = new Lista<Node>();
            explored = new Lista<Node>();
        NPuzzle = new Node(Puzzle);

        //Insere o nó na fronteira.
            frontier->InsereFinal(NPuzzle);

            // Inicializa a contagem do número de verificações, movimentos e
o estado booleano que indica que a solução foi encontrada.
            int Qtde_Verificacoes = 0;
            int movimentos = 0;
            bool isGoal = false;

            //Define o alvo a ser explorado como o primeiro elemento da
fronteira tratando-a como uma fila.
            Alvo = new Elemento<Node>;
            Alvo = frontier->GetIni();

            // Verifica se o alvo inicial é o objetivo antes de explorá-lo.
            Se Alvo == Objetivo {
```

```
                isGoal = true;
                Imprime(“Solucao encontrada”);
            }
            Incrementa Qtde_verificacoes;
        // Insere na lista explored o primeiro elemento da fronteira.

            explored->InsereFinal(frontier->RetiraFinal());
            // Avalia o próximo elemento da fronteira enquanto a solução não
for encontrada.
            Enquanto (isGoal == false)
            {


                //Determina o alvo como o último elemento do explorado.
                Alvo = explored->GetFim();
                //Mostra na tela qual quebra cabeça está sendo explorado.
                Imprime(Alvo);
                // Gera os filhos do alvo já verificando se são o objetivo
e calculando a heuristica, se algum deles for a função retorna True para
IsGoal
                Alvo->dado->GerarFilhos_Informado(frontier,     explored,
Qtde_Verificacoes);
                //  Procura  o  elemento  de  menor  custo  atualmente  na
fronteira.
                index = ProcuraMenorCusto(frontier);
                //Insere na lista explored o próximo alvo para exploração
retirando-o da frontier.
        explored->InsereFinal(frontier->RetiraElemento(index))
        //Verifica se o alvo de menor custo também é objetivo
                Se Alvo == Objetivo {
                      isGoal == True;
        }

            }

        Imprime Solução.
```

Pseudo-código - Busca Subida de Encosta

```
        Hill_Climbing(Puzzle)
        {
            // cria-se as listas frontier e explored, e define o nó inicial
Puzzle.
            frontier = new Lista<Node>();
            explored = new Lista<Node>();
```

```
        NPuzzle = new Node(Puzzle);

        //Insere o nó na fronteira.
             frontier->InsereFinal(NPuzzle);

             // Inicializa a contagem do número de verificações, movimentos e
o estado booleano que indica que a solução foi encontrada.
             int Qtde_Verificacoes = 0;
             int movimentos = 0;
             bool isGoal = false;

             //Define o alvo a ser explorado como o primeiro elemento da
fronteira tratando-a como uma fila.
             Alvo = new Elemento<Node>;
             Alvo = frontier->GetIni();

             // Verifica se o alvo inicial é o objetivo antes de explorá-lo.
             Se Alvo == Objetivo {
                   isGoal = true;
                   Imprime("Solucao encontrada");
             }
             Incrementa Qtde_verificacoes;
        // Insere na lista explored o primeiro elemento da fronteira.

             explored->InsereFinal(frontier->RetiraFinal());
             // Avalia o próximo elemento da fronteira enquanto a solução não
for encontrada.
             Enquanto (isGoal == false)
             {


                   //Determina o alvo como o último elemento do explorado.
                   Alvo = explored->GetFim();
                   //Mostra na tela qual quebra cabeça está sendo explorado.
                   Imprime(Alvo);
                   // Gera os filhos do alvo já verificando se são o objetivo
e calculando a heuristica, se algum deles for a função retorna True para
IsGoal
                   Alvo->dado->GerarFilhos_Informado(frontier,     explored,
Qtde_Verificacoes);
                   //  Procura  o  elemento  de  menor  custo  atualmente  na
fronteira.
                   index = ProcuraMenorCusto(frontier);
                   //Insere na lista explored o próximo alvo para exploração
retirando-o da frontier.
        explored->InsereFinal(frontier->RetiraElemento(index))
        //Verifica se o alvo de menor custo também é objetivo
```

```
            Se Alvo == Objetivo {
                  isGoal == True;
}

     }

Imprime Solução.
```

Experimentos e resultados

Para testar os algoritmos implementados primeiramente foram testados com o seguinte problema proposto:

[4] [1] [6]
[3] [2] [8]
[7] [ ] [5]

Com objetivo de atingir o estado:

[1] [2] [3]
[4] [5] [6]
[7] [8] [ ]

Começando com a busca cega:

- Busca em largura

```
[1 3 0]
[4 2 6]
[7 5 8]

[1 0 3]
[4 2 6]
[7 5 8]

[1 2 3]
[4 0 6]
[7 5 8]

[1 2 3]
[4 5 6]
[7 0 8]

[1 2 3]
[4 5 6]
[7 8 0]
O problema foi resolvido com 2044 verificacoes e 13 movimentos.
Fim
Tempo de execucao BFS: 4801ms
```

- Busca em profundidade

```
[1 0 3]
[4 2 6]
[7 5 8]

[1 2 3]
[4 0 6]
[7 5 8]

[1 2 3]
[4 5 6]
[7 0 8]

[1 2 3]
[4 5 6]
[7 8 0]
O problema foi resolvido com 81095 verificacoes e 44665 movimentos.
Fim
Tempo de execucao DFS: 981475ms
```

Em seguida busca informada:

- Busca A*

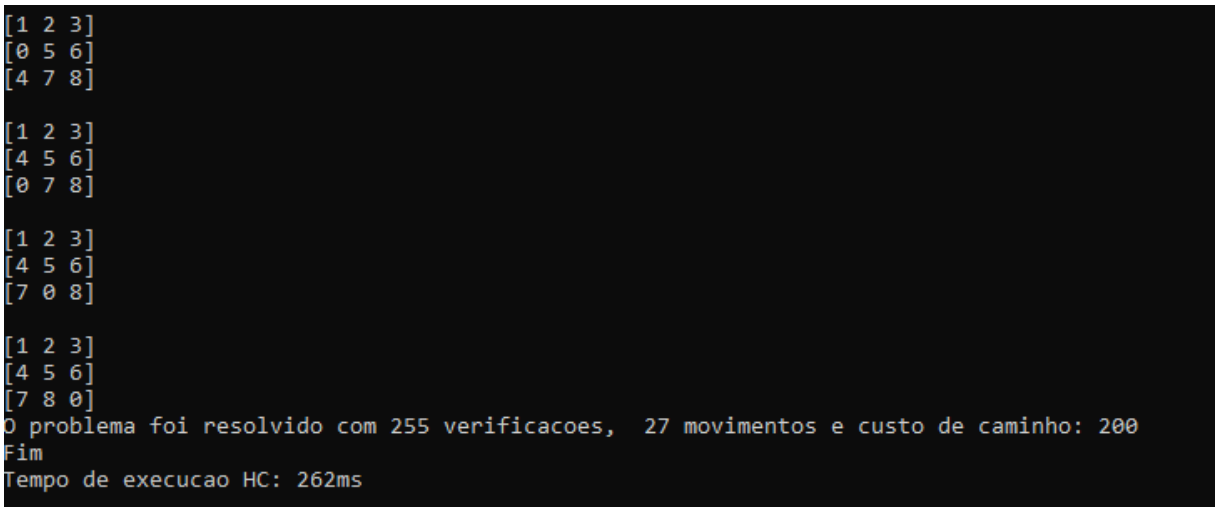
```
[1 0 3]
[4 2 6]
[7 5 8]

[1 2 3]
[4 0 6]
[7 5 8]

[1 2 3]
[4 5 6]
[7 0 8]

[1 2 3]
[4 5 6]
[7 8 0]
O problema foi resolvido com 242 verificacoes,  13 movimentos e custo de caminho: 98
Fim
Tempo de execucao A*: 232ms
```

- Busca Subida de encosta



Como se pode notar a busca cega toma muito mais tempo e explora muito mais estados do que a busca informada, especialmente a busca em profundidade que pode ter seu tempo de execução variando muito de acordo com a posição da solução na arvore e a escolha da ordem das ações a serem tomadas. No caso do A* e da Busca em largura ambos encontraram a solução ótima para o problema que neste quebra cabeça tem como característica a solução mais “rasa” sempre ser a ótima. Pode-se notar que a solução da subida de encosta por ser uma solução gulosa encontrou um caminho muito mais longo que o A* e a busca em largura, porem a sua rapidez na resolução do problema é muito eficiente. Como pode ser visto em problemas com soluções com mais passos, é onde este algoritmo realmente se destaca em velocidade.

Para demonstrar a eficiência da busca informada sobre a busca cega foram feitos mais dois testes sendo estes os seguinte problemas:

[4] [8] [3]

[7] [2] [5]

[6] [1] [ ]

Que pode ser resolvido em 20 movimentos;

E o problema:

[8] [6] [7]

[2] [5] [4]

[3] [ ] [1]

Que é o um dos piores casos deste quebra cabeça pois demanda 31 passos para se resolver.

Resultados do segundo teste:

[4] [8] [3]

[7] [2] [5]

[6] [1] [ ]

- Busca em largura

```
[1 0 3]
[4 2 5]
[7 8 6]

[1 2 3]
[4 0 5]
[7 8 6]

[1 2 3]
[4 5 0]
[7 8 6]

[1 2 3]
[4 5 6]
[7 8 0]
O problema foi resolvido com 39259 verificacoes e 20 movimentos.
Fim
Tempo de execucao BFS: 237531ms
```

- Busca em profundidade

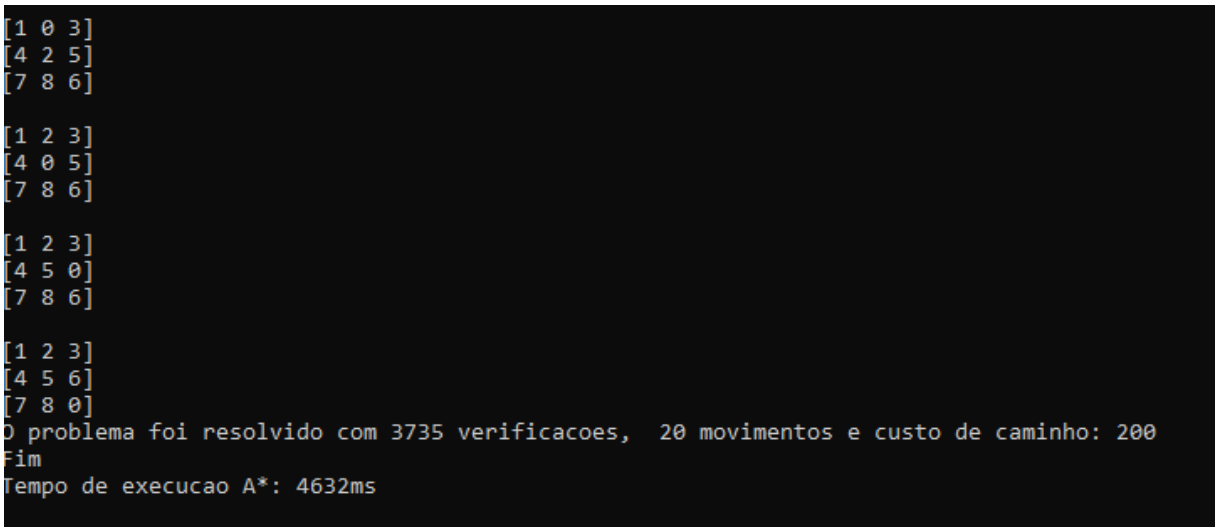
```
[1 2 3]
[0 4 6]
[7 5 8]

[1 2 3]
[4 0 6]
[7 5 8]

[1 2 3]
[4 5 6]
[7 0 8]

[1 2 3]
[4 5 6]
[7 8 0]
O problema foi resolvido com 41653 verificacoes e 23008 movimentos.
Fim
Tempo de execucao DFS: 265371ms
```

- Busca A*

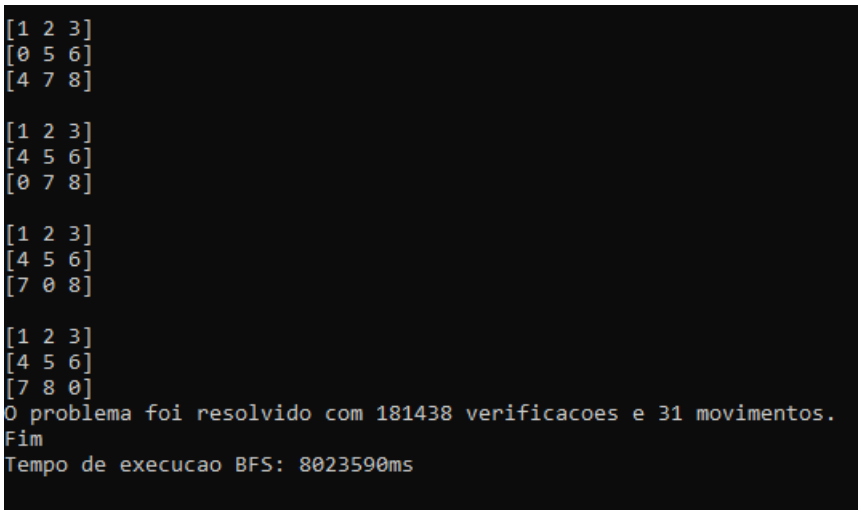


- Busca Subida de encosta

Neste caso já é possível notar que no primeiro exemplo a busca em profundidade que havia tomado 981 segundos tomou apenas 265s, e fez aproximadamente a metade das verificações, provando-se muito situacional. Já a busca em largura se provou com complexidade temporal exponencial onde o tempo foi muito maior para achar uma solução na 20ª camada do que na 13ª. O algoritmo A* novamente se mostrou muito eficiente, retornando a solução em apenas uma fração da busca em largura. E finalmente a subida em encosta que em eficiência temporal novamente superou a todos os outros algoritmos sem ter um custo muito mais alto que a solução ótima.

Resultados do terceiro teste:

- Busca em Largura

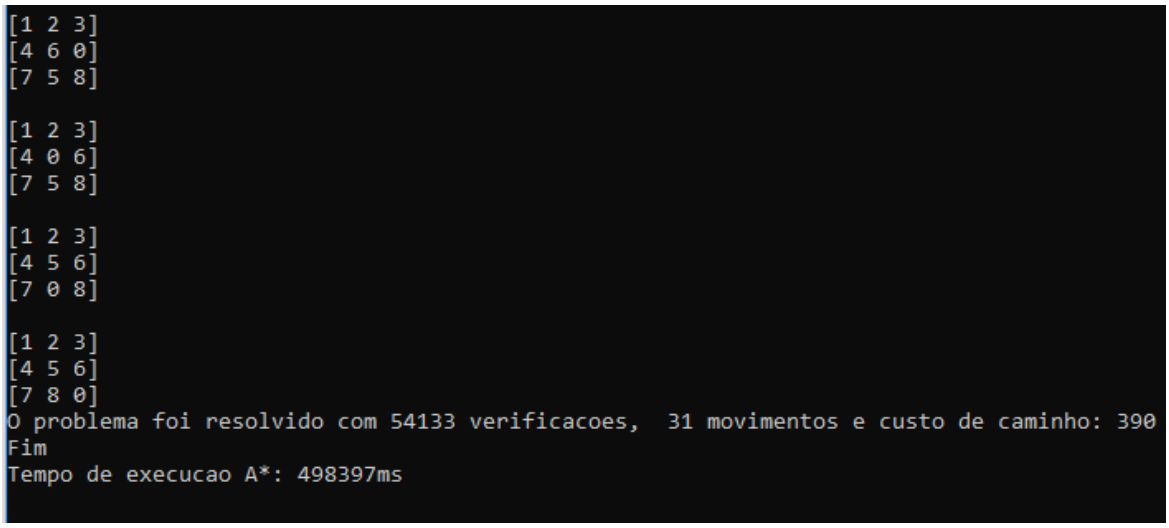


- Busca em profundidade

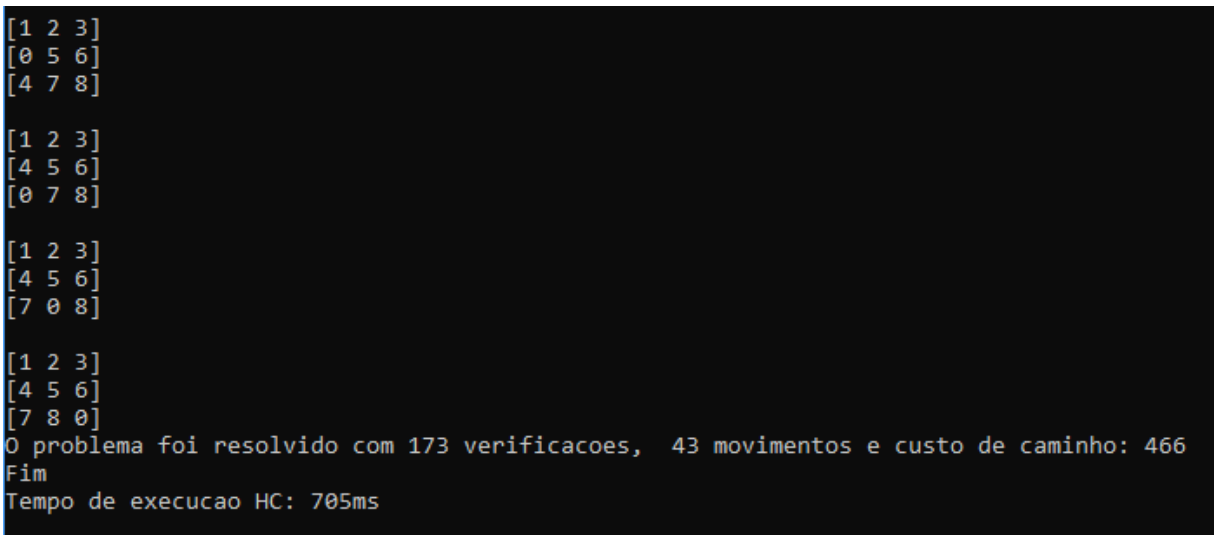
Especialmente para este problema, nota-se que a busca em largura foi especialmente ineficiente pois fez 181438 verificações sendo que o problema tem exatamente 9!/2 possibilidades que correspondem a 181440 possibilidades. Ou seja, a busca teve te percorrer quase todos os estados possíveis para achar a solução. Isso sem contar no grande dispêndio de tempo onde foram necessários 2 horas 13 minutos e 43 segundos para encontrar a solução.

No caso da busca em profundidade, foram necessários milhares de passos a mais, mas solução foi encontrada em menos de 15 minutos e nem perto da metade das verificações.

- Busca A*



- Busca Subida de Encosta



Neste caso do A* nota-se que o mesmo teve de fazer muito mais verificações do que nos exemplos anteriores, isso decorre do fato de que quanto mais longe do nó inicial os nós de menor caminho deste problema tendem a terem custos parecidos aos olhos do A* sendo necessários ele voltar na arvore e explorar outros estados inúmeras vezes para poder chegar ao resultado.

Como o algoritmo da subida de encosta é guloso, novamente se mostrou muito eficiente, gastando mais movimentos para chegar na solução, mas apenas uma ínfima fração do tempo dos outros algoritmos.

Trabalhos correlatos

iBFS: Concurrent Breadth-First Search on GPUs

Hang Liu H. Howie Huang Yang Hu Department of Electrical and Computer Engineering George Washington University {asherliu, howie, huyang}@gwu.edu

https://www.uml.edu/docs/ibfs_tcm18-284417.pdf

Block Architecture Problem with Depth First Search Solution and Its Application

Robbi Rahim[1,6], Dahlan Abdullah[2,6], Janner Simarmata[3,6], Andri Pranolo[4,6], Ansari Saleh Ahmar[5,6], Rahmat Hidayat[6,7], Darmawan Napitupulu[6,8], Heri Nurdiyanto[6,9], Bayu Febriadi[10] and Z Zamzami[10]

https://iopscience.iop.org/article/10.1088/1742-6596/954/1/012006/pdf

The Feature Selection Method based on Genetic Algorithm for Efficient of Text Clustering and Text Classification

Sung-Sam Hong, Wan Hee Lee, Myung-Mook Han

Published 2015

https://www.semanticscholar.org/paper/The-Feature-Selection-Method-based-on-Genetic-for-Hong-Lee/1a985a5a14ed977182c690895e23369e929d0c7a

Conclusão

Com estes experimentos e implementações dos algoritmos de busca cega e busca informada pudemos notar a grande superioridade em eficiência da busca informada, onde em certos casos a complexidade temporal e espacial diminui exponencialmente caso seja possível calcular e utilizar uma heurística admissível.

Como pôde-se notar, para o quebra cabeça em questão, a busca em largura sempre encontrou a solução ótima para o problema, pois neste quebra cabeça em especifico a quantidade de movimentos é o determinante da qualidade da solução. Portanto a busca em largura e o algoritmo de busca A* sempre chegam ao mesmo caminho para a solução.

No caso da busca em profundidade a eficiência do algoritmo é muito situacional, onde dependem fortemente da ordem de prioridade da escolha das ações e também da posição relativa da solução em relação ao nó inicial podendo ser extremamente rápido ou extremamente moroso para pouca variação nos quebra cabeças problema.

O algoritmo da subida de encosta se mostrou muito eficiente temporalmente, embora não apresente comumente a solução ótima, este é extremamente rápido e não apresenta soluções demasiadamente longas, onde geralmente no pior caso apresenta soluções com o dobro de passos do que a solução ótima, não sendo assim tão dispendioso em comparação com casos da busca em profundidade onde a quantidade de passos encontrados para a solução chega a milhares.

Sendo assim é conclusivo que a busca informada se possível é muito melhor que a busca cega, no entanto demanda mais planejamento e complexidade na construção e execução do programa a ser desenvolvido para avaliação da mesma.

Referências bibliográficas

[1] https://pythonhelp.wordpress.com/2015/01/19/arvore-binaria-de-busca-em-python/

[2] https://www.hackerearth.com/pt-br/practice/algorithms/graphs/depth-first-search/tutorial/

[3] Página 100, RUSSEL, Stuart; NOVIG, Peter; Artificial Inteligence: A Modern Approach, 3rd Edition.

[4] Página 121, RUSSEL, Stuart; NOVIG, Peter; Artificial Inteligence: A Modern Approach, 3rd Edition.

[5] https://inst.eecs.berkeley.edu/~cs61c/fa14/projs/02/