Projeto e Análise de Algoritmos
Trabalho Prático 2 - Grafos

Jordan Silva[1]

[1] Instituto de Ciências Exatas
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

jordan@dcc.ufmg.br

1. Objetivo

Este trabalho tem como objetivo a implementação e análise da solução para o problema do *Quebra-Cabeça das N pastilhas*, proposto no módulo de grafos da disciplina de Projeto e Análise de Algoritmos. Este trabalho soluciona esse problema através do algoritmo **A*** e implementação de três heurísticas de programação diferentes: **Distância de Manhattan, Distância de Hamming e de Conflito Linear**.

A implementação deste trabalho foi realizada utilizando a linguagem *C++11*.

2. Problema

O trabalho proposto tem como o tema *O Quebra-Cabeça das N pastilhas*, no qual o problema à ser resolvido é um jogo de quebra-cabeça, onde você tem um tabuleiro de $N \times N$ espaços e $N - 1$ peças. As peças desse tabuleiro são definidas como valores inteiros, são enumeradas de 1 à $N - 1$ e estão distribuídas aleatoriamente no quebra-cabeça.

Para solucionar esse problema é necessário organizar as peças de forma a alcançar uma disposição ordenada, conforme apresentado na Figura 1. A única ação permitida para o jogador é movimentar qualquer peça da vizinha do espaço vazio para ele, sendo que esse movimento só pode ser realizado verticalmente ou horizontalmente.

Uma observação à ser feita para o problema do Quebra-Cabeça de N pastilhas é que 50% dos estados iniciais são irresolvíveis.

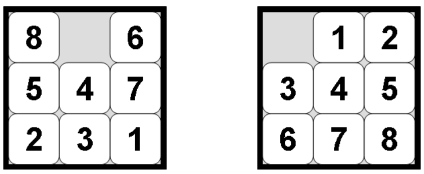

Figura 1. Quebra-cabeça 3x3

Neste trabalho será necessário modelar esse problema como um grafo, considerando cada estado do quebra-cabeça como um vértice, e expandir esse grafo através de arestas entre os vértices que possuem estados válidos, onde é possível alcançar através de movimentos do espaço vazio. Também será necessário que cada vértice armazene o seu estado atual, como também o seu vértice antecessor, e a sua movimentação (Figura 2). Essas informações serão necessárias para mapear as movimentações realizadas e encontrar a solução ótima do problema (i.e., O menor número de movimentos é considerado a solução ótima).

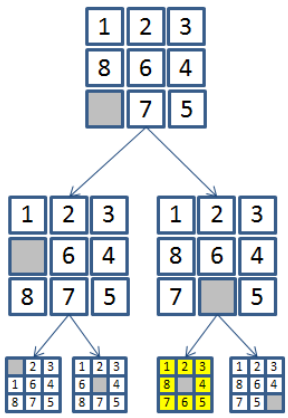

Figura 2. Expandindo a Árvore de estados

Dados de Entrada A entrada do programa consiste em um arquivo texto, ***input.txt***. O arquivo possui na sua primeira linha um valor inteiro N que informa a dimensão do quebra-cabeça. A seguir, o resto do arquivo é o estado inicial do quebra-cabeça, onde terá N linhas com N número inteiros separados por espaço. Excepcionalmente, uma das linhas possuirá um "_"(*underline*) informando a posição do espaço vazio (Tabela 1).

Dados de Saída Para a saída, haverá dois casos: *a*) O quebra-cabeça tem solução, então deve conter: na primeira linha do arquivo um valor inteiro que informa o tamanho do caminho achado, e em cada linha subsequente o movimento tomado pelo espaço vazio na sequência para resolver o tabuleiro, formatado em { acima, abaixo, esquerda, direita }, como pode ser visto na coluna ***output.txt*** da Tabela 1. *b*) O tabuleiro é irresolvível, então deve conter: 'Sem solução'.

input.txt	***output.txt***
	5
3	abaixo
1 _ 5	direita
3 2 4	acima
6 7 8	esquerda
	esquerda

Tabela 1. Exemplo dos dados

3. Modelagem e Solução

O problema foi modelado como o *8-puzzle problem*[1]. Este é um jogo de quebra-cabeça inventado e popularizado por Noyes Palmer Chapman nos anos de 1870. Esse quebra-cabeça é jogado em um tabuleiro 3 por 3, com 8 peças quadradas rotuladas de 1 à 8, e um espaço em branco. O objetivo é reorganizar essas peças de modo que fiquem em ordem. Apenas é permitido movimentar as peças verticalmente e horizontalmente para o espaço vazio.

[1] `http://www.cs.princeton.edu/courses/archive/spr10/cos226/assignments/8puzzle.html`

Para generalizar o nosso problema, vamos modelar como *N-puzzle problem*, onde o tabuleiro do quebra-cabeça pode assumir qualquer dimensão, assim dado um tabuleiro de dimensão $\boldsymbol{N}$, onde possuímos $\boldsymbol{N-1}$ peças enumeradas de $\boldsymbol{N-1}$ e 1 espaço em vazio. O nosso objetivo é reorganizar esse tabuleiro de forma que o espaço vazio esteja na posição 1 (i.e., superior esquerdo) do tabuleiro, e as outras peças organizadas em ordem crescente em suas respectivas posições.

Fez se necessário a realização de uma modelagem em forma de grafo $G(V, E)$ para esse problema, onde cada vértice V é dado como um estado do tabuleiro, e as arestas E demonstram os estados que são alcancáveis através desse grafo. Assim, realizamos uma busca através do estado inicial desse grafo, e expandimos esse grafo buscando as soluções viáveis até encontrar a solução ótima. Para realizar essa tarefa, será apresentado na próxima seção o **Algoritmo A*** que foi utilizado para solucionar o problema, tal como as heurísticas desenvolvidas.

4. Algoritmo A*

O **Algoritmo A*** é um algoritmo amplamente usado para **Busca de Caminhos** em um grafo, onde possui um ótimo desempenho e precisão para solucionar esse tarefa. Este algoritmo usa o *best-first search* e encontra o caminho com menor custo dado um estado vértice inicial até o seu vértice objetivo. Como o *breadth-first search*, o **A*** realiza uma busca completa, e sempre irá a solução se essa existir.

O **A***, diferentemente de um algoritmo BCU, não se baseia a busca somente pelo custo real do vértice, mas considera um valor heurístico para o custo. O objetivo da heurística é quantificar a distância do vértice atual para o vértice objetivo, no nosso problema as heurísticas serão baseadas nas posições das peças do quebra-cabeça. Assim, o cálculo de custo desse algoritmo é dado por

$$f(n) = g(n) + h(n) \tag{1}$$

onde a função $g(h)$ retorna o custo real total de se alcançar o vértice n, ou a profundidade desse vértice na expansão da árvore de soluções. A função $h(n)$ adicionada pelo **A***, é uma função heurística. Nesse trabalho utilizaremos somente de heurísticas admissíveis, onde o seu custo estimado gerado é no máximo o custo real, dado como

$$h(n) \leq g(n) \tag{2}$$

A implementação do **Algoritmo A*** nesse trabalho pode ser vista no Algoritmo 1, onde expandimos a nossa árvore de soluções através de todas as possibilidades viáveis de movimentos do nosso espaço vazio, e calculamos o custo para cada novo estado. O nosso algoritmo tem como condição de parada quando encontrar o vértice objetivo, i.e., quando a nossa função de heurística $h(n) = 0$, sendo a distância do estado atual para o estado objetivo zero.

4.1. Análise de Complexidade

A complexidade de tempo desse algoritmo depende da heurística implementada, mas no pior caso, em uma busca pelo espaço de soluções sem nenhuma restrição, o número de vértices expandidos será exponencial ao tamanho da solução, sendo assim

$$O(|V|) = O(b^d) \tag{3}$$

onde b é o *fator de ramificação* (i.e., a quantidade de filhos de cada vértice) [Russell and Norvig 1995]. Essa complexidade é dada assumindo que o a solução exista, e esta é alcançável a partir do estado inicial; senão, o espaço de estados será *infinito* e o algoritmo não terminará. A complexidade de tempo é polinômial quando o espaço de busca dos estados é uma árvore, existindo um único estado objetivo à ser alcançado, e a função heurística h satisfaz as seguintes condições:

$$|h(x) - h^*(x)| = O(\log h^*(x)) \tag{4}$$

onde h^* é a heurística ótima, o custo exato do estado corrente à solução. Dessa forma, o erro de h não irá crescer mais rápido que o logaritmo da "heurística perfeita" h^*. [Pearl 1984]

Algorithm 1 Pseudo-código A*

```
function ASTAR(startNode, goal_node, dimension, heuristic)
    result ← []
    visited ← []
    queue ← startNode
    last ← 0
    while !empty(queue) do
        curr_node ← top(queue)
        if curr_node ∉ visited then
            curr_node insert visited
        else if cost(curr_node) ≤ cost(visited[curr_node]) then
            visited[curr_node] ← curr_node
        end if
        if curr_node = goal_node then
            last ← curr_node
            break
        end if
        cildren ← GENERATECHILDREN(curr_node)
        for all children do
            if children ∉ visited or cost(children) ≤ cost(visited[children]) then
                children insert queue
            end if
        end for
    end while
    while last ≠ 0 do
        visited[last] insert result
        last ← visited[last].parent
    end while
    return result
end function
```

4.2. Distância de Manhattan

A **Distância de Manhattan** é a distância entre dois pontos medida ao longo dos eixos em ângulos retos. Esse nome faz uma alusão ao layout das ruas da cidade de Manhattan, o que proporciona o menor caminho que um carro teria que percorrer entre dois pontos da cidade.

No nosso problema do quebra-cabeça, se x_i e y_i são as coordenadas x e y da peça i no estado s, e se $\bar{x_i}$ e $\bar{y_i}$ são as coordenadas x e y de peça i no estado objetivo, a heurística é:

$$h(s) = \sum_{i=1}^{N}(|x_i(s) - \bar{x_i}| + |y_i(s) - \bar{y_i}|) \tag{5}$$

onde pode ser vista a implementação através do Algoritmo 2.

4.2.1. Análise de Complexidade

Complexidade de tempo A solução através dessa heurística é na verdade a mesma complexidade de solucionar dois problemas simples. Esse algoritmo basicamente segue a mesma abordagem do **qsort**, e irá percorrer todos os N elementos da matriz e realizará uma operação aritmética simples para cada elemento. Dessa forma, temos

$$O(n) \text{ no melhor e pior caso} \tag{6}$$

onde $\boldsymbol{n}$ é a quantidade de elementos no vetor.

Complexidade de espaço A complexidade de espaço na implementação realizada é irrelevante, pois na realização do cálculo do algoritmo está sendo armazenado somente o somatório das distâncias das peças até seus respectivos objetivos. Logo

$$O(1) \text{ no melhor e pior caso} \tag{7}$$

onde esse valor será armazenado em cada vértice V, como valor da função $h(n)$.

Algorithm 2 Pseudo-código Manhattan Distance

function MANHATTANDISTANCE($vector,\ dimension$)
  $distance \leftarrow = 0$
  for $i \leftarrow 1$ to $foes.size()$ **do**
    $row \leftarrow i\ /\ dimension$
    $column \leftarrow i\ \%\ dimension$
    if $vector[i] \neq blank\_position$ **then**
      $target\_row = vector[i]\ /\ dimension$
      $target\_row = vector[i]\ \%\ dimension$
      $distance \leftarrow distance + (abs(row - target\_row) + abs(column - target\_column))$
    end if
  end for
  return $distance$
end function

4.3. Distância de Hamming

A definição da **Distância de Hamming** ou *Misplaced Tiles* é dado pela quantidade de peças que não estão na sua posição final (desconsiderando o espaço vazio do tabuleiro). Assim o Algoritmo 3 é dado pela contagem das peças em posições erradas, dessa forma temos

$$h(s) = \sum_{i=1}^{N} I(s_i) \tag{8}$$

$$I(s) = \begin{cases} 1 & \text{if } x_i(s) \neq \bar{x_i} \text{ or } y_i(s) \neq \bar{y_i} \\ 0 & \text{else} \end{cases} \tag{9}$$

onde a função $h(s)$ é definida pelo somatório das peças que não estão na posição correta, qual é dada pelo indicador $I(s)$.

Algorithm 3 Pseudo-código Hamming Distance

```
function HAMMINGDISTANCE(vector, dimension)
    distance ← = 0
    for i ← 1 to foes.size() do
        if vector[i] ≠ blank_position and vector[i]! = i then
            distance ← distance + 1
        end if
    end for
    return distance
end function
```

4.3.1. Análise de Complexidade

Complexidade de tempo A solução através dessa heurística igualmente a heurística de Manhattan, percorre todos os N elementos da matriz e verifica somente se o valor está na posição correta. Assim, na nossa implementação (Algoritmo 3) teremos a complexidade

$$O(n) \text{ no melhor e pior caso} \tag{10}$$

onde $\boldsymbol{n}$ é a quantidade de elementos no vetor.

Complexidade de espaço O espaço utilizado nessa heurística irá utilizar somente de uma variável, onde será armazenado o somatório com a quantidade de peças do quebra-cabeça fora da posição correta. Logo

$$O(1) \text{ no melhor e pior caso} \tag{11}$$

onde esse valor será armazenado em cada vértice V, como valor da função $h(n)$.

4.4. Conflito Linear

Essa heurística tem como premissa verificar se existem peças trocas na mesma linha. Dessa forma temos um conflito linear caso duas peças t_j e t_k estejam na mesma linha; as posições finais dessas duas peças sejam nessa linha que elas estão; t_j está a direita da peça t_k, mas a posição final de t_j deveria ser à esquerda da posição final de t_k. Assim, teremos a nossa heurística da seguinte forma

$$h(s) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} I(s_i, s_j) \tag{12}$$

$$I(t_j, t_k) = \begin{cases} 1 & \text{if } t_j \leftrightarrow t_k \\ 0 & \text{else} \end{cases} \tag{13}$$

onde $\leftrightarrow$ significa se os elementos estão em posições invertidas, como descrito na heurística.

4.4.1. Análise de Complexidade

Complexidade de tempo A implementação dessa solução é um pouco mais trabalhosa que as anteriores. Essa heurística é necessário percorrer $N^(1.5)$ elementos, devido a necessidade de comparar cada elemento com os outros da mesma linha. Assim, na nossa implementação (Algoritmo 4) teremos a complexidade

$$O(n) \text{ no melhor e pior caso} \tag{14}$$

onde $\boldsymbol{n}$ é a quantidade de elementos no vetor.

Complexidade de espaço O espaço utilizado nessa heurística irá utilizar somente de uma variável, onde será armazenado o somatório com a quantidade de peças do quebra-cabeça fora da posição correta. Logo

$$O(1) \text{ no melhor e pior caso} \tag{15}$$

onde esse valor será armazenado em cada vértice V, como valor da função $h(n)$.

5. Experimentos

Neste trabalho foram realizados experimentos variando a entrada dos dados, a fim de analisar comparativamente as heurísticas implementadas, tal como o seu desempenho, a quantidade de nós explorados, análise de tempo de execução e memória utilizada. As diferentes combinações de quebra-cabeça foram geradas aleatoriamente. Esses experimentos, como também a especificação do ambiente onde os testes foram executados serão abordados nessa seção.

5.1. Ambiente de teste

A implementação dos algoritmos e os experimentos foram realizados utilizado a linguagem *c++11*, em um **MacBook Pro (13-inch, Mid 2012); 2,5 GHz Intel Core i5; 16 GB 1600 MHz DDR3; OS X El Captain 10.11 (14D136)**

Algorithm 4 Pseudo-código Linear Conflict

function LINEARCONFLICT($vector$, $dimension$)
  $distance \leftarrow$ = 0
  for $i \leftarrow 1$ to $foes.size()$ **do**
    for $j \leftarrow i + 1$ to $foes.size()$ **do**
      if $vector[i] \neq blank\_position$ **then**
        if CHECKINVERTED($vector$, i, j) **then** ▷ Verifica se a peça i e j estão invertidas na mesma linha, e se ambas estão na linha correta.
          $distance \leftarrow distance + 1$
        end if
      end if
    end for
  end for
  return $distance$
end function

Tabela 2. Tabuleiros

Custo	5	10	20	48	66
Tabelas	1 _ 5 3 2 4 6 7 8	6 1 2 4 _ 5 7 3 8	2 8 _ 4 1 6 7 5 3	14 7 6 1 2 8 4 13 9 3 15 10 12 5 11 _	12 9 13 15 11 10 14 3 7 2 5 4 8 6 1 _

5.2. Análise dos Experimentos

Os experimentos para análise do tempo de execução foram realizados através da variação da sequência de números no tabuleiro, variação das dimensões, e execução das três heurísticas individualmente e a combinação da heurística da Distância de Manhattan + Distância de Hamming. Primeiramente escolhemos três entradas de dados distintas (Tabela 2), com as dimensões 3x3 e realizamos uma análise da quantidade de movimentos necessários para solucionar cada tabuleiro pelo tempo gasto (Figura 3), e também pela quantidade de nós explorados até encontrar a solução (Figura 4).

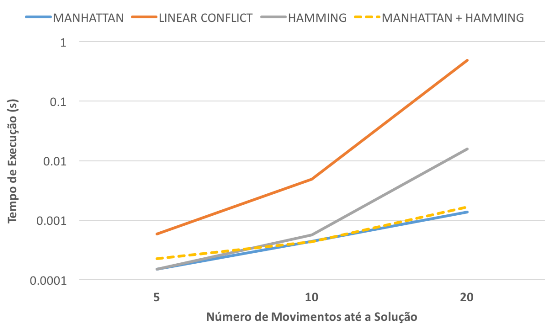

Figura 3. Tempo execução

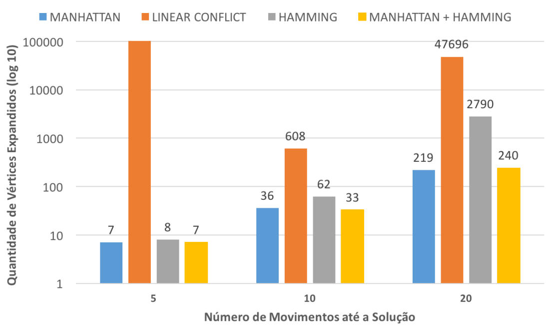

Figura 4. Qtd. vértices explorados

Após a análise de tempo de execução e quantidade de vértices explorados em cada heurística, realizamos uma análise da quantidade memória utilizada e tempo gasto para encontrar a solução do Tabuleiro de Custo 48 (Tabela 2). Como podemos visualizar na Figura 5, apenas as duas heurísticas **Manhattan (MD)** e **Manhattan + Hamming (MD+HD)** conseguiram solucionar o problema, e a combinação da **Distância de Manhattan** com a **Distância de Hamming (HD)** se mostrou muito superior à utilizar somente a Heurística de **Manhattan**.

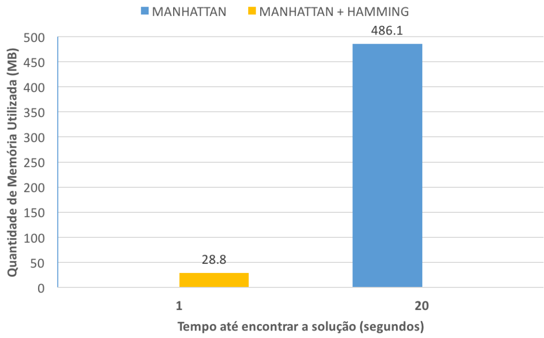

Figura 5. Tabuleiro 48 - Memória Utilizada

Finalizando os nossos testes, com a **Heurística MD+HD** conseguimos resolver um problema de dimensão 5x5, com o custo de 66 movimentos (Tabela 2). Essa heurística foi a única experimentada que conseguiu resolver esse quebra-cabeça, sendo necessário uma utilização de 1.22GB de Memória RAM; 6.314.429 de vértices explorados; e foi executado em 59 segundos.

6. Conclusão

De acordo com a proposta do trabalho, este documento apresentou quatro heurísticas admissíveis para a solucionar o problema apresentado na Seção 2, sendo que uma das heurísticas é realizada a partir da combinação de duas outras. Realizamos uma análise das heurísticas implementadas e percebemos como uma diferença sútil na implementação tanto das heurísticas, como do algoritmo A* impacta na exploração do espaço de soluções e para encontrar a solução ótima mais rapidamente. Acredito que foi cumprido a proposta desse trabalho, tal como um melhor entendimento sobre o conteúdo de grafos ministrado na disciplina de Projeto e Análise de Algoritmo.

Referências

Pearl, J. (1984). Heuristics: intelligent search strategies for computer problem solving.

Russell, S. and Norvig, P. (1995). Artificial intelligence: a modern approach.