UNIVERSIDADE ESTADUAL DE MARINGÁ – UEM

DEPARTAMENTO DE INFORMÁTICA

Disciplina: Modelagem e Otimização Algorítmica

Professor: Ademir Aparecido Constantino

Implementação do Algoritmo A* para a Resolução do Jogo do Tabuleiro de Quinze Peças

ALUNO:

VINICIUS VIVAN RA: 81730

Maringá, janeiro 2016.

Sumário

Introdução

Os algoritmos de busca são utilizados para encontrar uma sequência de ações que, partindo de um estado inicial, levem a uma determinada configuração desejada. Assim, estes algoritmos geram novos estados a partir da aplicação de operadores no estado corrente, até que seja alcançada a solução. Portanto, dado um problema, deve ser definido seu estado inicial, final e os operadores que serão aplicados para gerar novos estados.

Este trabalho se propõe a exibir o funcionamento do método de busca A* aplicado no jogo do tabuleiro de quinze peças. O jogo dos quinze, consiste em um tabuleiro de 4 linhas por 4 colunas que, partindo de qualquer estado inicial que contenha algarismos de 1 a 15 e mais um quadrado vazio, representado pelo algarismo 0, chegue ao seguinte estado final:

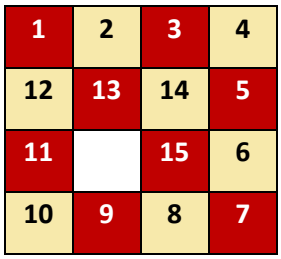


Figura 1: Estado final.

Os operadores podem ser representados como mover o algarismo 0 para cima, para a esquerda, para a direita ou para baixo.

Em algoritmos como a busca gulosa e a busca A* tem como características a utilização de funções heurísticas para auxiliar na solução do problema. Heurística é um método desenvolvido com o objetivo de encontrar soluções, sendo um procedimento de simplificação, ou aproximação que reduz ou limita a busca por soluções em domínios que são difíceis e pouco compreendidos.

Desenvolvimento Experimental

O método de busca A* foi implementado neste trabalho na linguagem Python. A busca A*, para gerar um novo estado, dá preferência ao que possui a melhor função de avaliação, considerando nesta função a soma do custo e a avaliação heurística.

Foram desenvolvidas cinco heurísticas, sendo elas:

- h'1(n) = número de peças foras de seu lugar na configuração final.

- h’2(n) = número de peças fora de ordem na sequência numérica das 15 peças, seguindo a ordem das posições no tabuleiro.

- h’3(n) = para cada peça fora de seu lugar somar a distância retangular (quantidade de deslocamentos) para colocar em seu devido lugar. Neste caso considera-se que o caminho esteja livre para fazer o menor número de movimentos.

- h’4(n) = p1*h'1(n) + p2*h’2(n) + p3*h’3(n), sendo p1 + p2 + p3 são pesos (número real) tais que p1 + p2 + p3 = 1. A escolha desses pesos deverá ser feita conforme os resultados dos experimentos. Para o código em anexo foram utilizados os seguintes valores: h’1 = 0,05; h’2 = 0,01; h’3 = 0,94.

- h’5(n) = max (h'1(n), h’2(n), h’3(n)).

Para a elaboração do algoritmo foi dada a preferência a linguagem Python, uma vez que esta proporciona uma grande praticidade para se trabalhar com listas e matrizes, o que foi muito utilizado para desenvolver leituras e movimentos no tabuleiro do jogo.

O código do A* foi baseado no pseudocódigo fornecido em slide, existem alguns pontos na implementação e escolhas de estruturas que devem ser destacados. Primeiramente foi criada uma classe “No” que é utilizada para cada nova posição do tabuleiro o qual será tratado. Essa classe possui os seguintes atributos:

tab: matriz com a posição dos números no tabuleiro.

hash: chave para verificar se tab pertence ao conjunto dos abertos, fechados ou é o estado final.

hlin: heurística h’(n) que está sendo usada.

pai: ponteiro para o Nó “tabuleiro” o qual a partir dele o nó atual foi gerado.

g: g(n)

f: função com a soma do custo g(n) e a heurística h’(n).

zl: linha do zero no tabuleiro.

zc: coluna do zero no tabuleiro.

Método construtor __init__(self, tabuleiro, pai):

```
def __init__(self, tabuleiro, pai):
        self.tab = tabuleiro #atribuição do tabuleiro passado por parâmetro
        self.hash = hash(tuple(tuple(i[:]) for i in self.tab)) #gera um hash a partir de tab
        self.hlin = h3(tabuleiro)  #heuristica usada
        if pai is not None: #verifica se não é o estado inicial
            self.pai = pai #Objeto passado por parâmetro
            self.g = pai.g+ 1 #calculo do g
        else: self.g = 0 #caso seja o estado inical
        self.f = self.g +  self.hlin #calculo do f
        self.zl, self.zc = posicao_zero(self.tab) #busca a posição do zero em tab
#metodos usados pelo SortedSet(), comparam o elemento atual com os contidos no conjunto SortedSet(), no caso, cj_abertos.
    def __lt__(self, other):
        if isinstance(other, (int, float)):
            return self.f < other
        return self.f < other.f

    def __le__(self, other):
        return self.f <= other.f

    def __gt__(self, other):
        return self.f > other.f

    def __ge__(self, other):
        return self.f >= other

    def __eq__(self, other):
        return self.hash == other.hash

    def __nq__(self, other):
        return self.hash != other.hash

    def __hash__(self):
        return self.hash
```

Funções auxiliares:

```
def posicao_zero(matriz): #busca a posição do zero em uma matriz
    for k in range(4):
        for j in range(4):
            if(matriz[k][j] == 0):
                return k,j # retorna uma tupla com a linha e coluna do zero na matriz


#pega os dados da entrada no formato “8 2 5 3 12 15 1 4 10 9 14 0 11 7 13 6” e os coloca em uma matriz
def entrada(matriz):
    string = input() #recebe a string “8 2 5 3 12 15 1 4 10 9 14 0 11 7 13 6”
    string = string.split(" ") #separa os elementos a cada espaço em branco encontrado
    contador = 0                                                    #e os coloca em um vetor
    for k in range(4):
        for j in range(4):
            matriz[k][j] = int(string[contador]) #colocar os elementos na matriz
            contador += 1
    return matriz
```

A entrada é digitada e todos os dados são organizados em uma matriz quatro por quatro que é passada por parâmetro e armazenada no atributo *“tab”* do primeiro nó instanciado, ainda na função *“main()”* S é o nó inicial e a função “*aestrela()*” é chamada para S.

Em meu projeto adotei uma biblioteca não padrão do Python, mas que por sua vez, é disponibilizada para download em seu site oficial (https://pypi.python.org/pypi/sortedcontainers), trata-se da biblioteca “sortedcontainers-1.4.4.” ela fornece uma estrutura de dados muito interessante para a proposta do trabalho, é uma lista de elementos que se mantem ordenada a cada inserção, ou seja, a cada iteração o elemento com o menor “f” sempre será o elemento de índice 0 da lista. O custo para manter esta lista ordenada é O(n) já que para inserir um elemento em sua posição são feitas no máximo n comparações, sendo n a quantidade de elementos já contidos na lista. Esta estrutura também verifica se o elemento que está sendo inserido já não está contido no conjunto *(“in”, “not_in”*), caso esteja o elemento não é inserido novamente, esta comparação é feita através de um *hash* único gerado no método construtor da classe a partir dos elementos da matriz e sua ordem.

Na função *"aestrela()"* é criado o conjunto dos estados abertos *"cj_abertos"*, este usa a estrutura *SortedList()*, trata-se de uma lista de nós que se mantem ordena crescentemente pelo valor de *f(m)* de cada nó. Isso torna rápida a busca do elemento com o menor valor de f, neste caso a busca possui tempo constante já que este elemento é o primeiro da lista *"cj_abertos.pop(0)"*, a estrutura *SortedList()* se mantem ordenada, mas, não mantém a unicidade de seus elementos, esta é mantida com o auxílio de dicionários que contém os conjuntos dos estados abertos e estados fechados. Em seguida o estado inicial *"S"* é inserido no conjunto dos estados abertos. O conjunto dos estados finais é criado, este é organizado em um dicionário já que precisamos apenas verificar se os tabuleiros avaliados estão contidos nele ou não. Por não manter este conjunto ordenado economizamos processamento. A partir deste ponto é iniciado o ciclo "*while(cj_abertos):*" que é verdadeiro desde que "*cj_abertos*" não esteja vazio. Código comentado abaixo:

```
def aestrela(S):
    cj_abertos = SortedList()   #criação do conjunto dos estados abertos
    cj_abertos.add(S)    #O estado inicial S é inserido no conjunto dos estados abertos
    abertos_dict = {S.key: S.f} #criação e inicialização do dicionário dos estados abertos
    fechados_dict = {}  #criação do dicionário dos estados fechados
    while(cj_abertos):  #enquanto cj_abertos != vazio
        menor = cj_abertos.pop(0)   #Nó com menor valor de f é removido dos abertos
        try:                    #tratamento de erros caso a chave já tenha sido removida
            del abertos_dict[menor.key]
        except KeyError:
            pass
        if (menor.key == final):  #verifica se o menor se trada do estado final
            return menor.g
        fechados_dict[menor.key] = menor.f  #coloca menor no dicionário
        cj_sucessores = gerar_jogadas(menor) #gera jogadas (sucessores) para o menor
        for sucessor in cj_sucessores: #enquanto existirem sucessores
            if sucessor.key in abertos_dict: #se o sucessor existir no conjuntos dos abertos
                if abertos_dict[sucessor.key] > sucessor.f:  #AND f > f'
                    abertos_dict[sucessor.key] = sucessor.f  #faz atualização do dicionário
                    cj_abertos.discard(sucessor)  #remove o valor antigo dos abertos
                    cj_abertos.add(sucessor) #insere o novo nó com
```

```
        elif sucessor.key in fechados_dict: #caso não esteja nos fechados
            if fechados_dict[sucessor.key] > sucessor.f: #AND f > f'
                cj_abertos.add(sucessor) #insere o novo nó nos abertos
                abertos_dict[sucessor.key] = sucessor.f #Insere no dicionário dos abertos
                del fechados_dict[sucessor.key] #e deleta dos fechados

        else: #caso o sucessor não esteja nem nos abertos e nem nos fechados
            cj_abertos.add(sucessor) #insere o novo nó nos abertos
            abertos_dict[sucessor.key] = sucessor.f #Insere no dicionário dos abertos

    return "fracasso"
```

Um nó é passado por parâmetro para a função "*gerar_jogadas(S)*" a partir da posição do zero em S (*"S.zl", "S.zc"*) são verificados quais os possíveis sucessores (Jogadas). Para cada jogada possível a função "*troca_posicao_zero(linha, coluna, S)*" é chamada esta função troca a posição do zero no tabuleiro com a posição linha/coluna passada por parâmetro, instancia um novo objeto "No" com a nova jogada e o retorna para a função "*gerar_jogadas(S)*" que por sua vez o coloca em uma lista de sucessores "*sucessores.append(troca_posicao_zero(S.zl, S.zc -1, S))*" esta lista é explorada na função "*aestrela(S)*".

Heuristicas:

```
def h1(matriz):
    nfordem = 0          #Inicia um contador
    for i in range(4):
        for j in range(4):
            if(matriz[i][j] != tab_final[i][j]): #compara a tabela atual com a desejada
                nfordem = nfordem + 1    # incrementa o contador para cada valor diferente
    return nfordem                       #retorna o contador




def h2(matriz):
    vetor = [] #criação de um vetor
```

```
    fora_ordem = 0
    vetor.append(matriz[0][0])   #Coloca os elementos da matriz passada por parâmetro
    vetor.append(matriz[0][1])   #para o vetor em ordem de caracol
    vetor.append(matriz[0][2])
    vetor.append(matriz[0][3])
    vetor.append(matriz[1][3])
    vetor.append(matriz[2][3])
    vetor.append(matriz[3][3])
    vetor.append(matriz[3][2])
    vetor.append(matriz[3][1])
    vetor.append(matriz[3][0])
    vetor.append(matriz[2][0])
    vetor.append(matriz[1][0])
    vetor.append(matriz[1][1])
    vetor.append(matriz[1][2])
    vetor.append(matriz[2][2])
    vetor.append(matriz[2][1])
    for k in range(14):
        if(vetor[k+1] != (vetor[k]+1)): #verifica se o elemento atual é diferente do anterior+1
            fora_ordem += 1 #para cada diferente incrementa o contador em 1
    return fora_ordem       #retorna o contador


#heurística 3
aux = [[0 for x in range(2)] for x in range(16)] #cria matriz 2x16
def h3(matriz):
    for k in range(4):
        for j in range(4):
            aux[tab_final[k][j]][0] = k #armazena a linha do elemento n da matriz final na linha n e coluna 0 do vetor
            aux[tab_final[k][j]][1] = j  #armazena a coluna do elemento n da matriz final na linha n e coluna 1 do vetor
    heuristica = 0 #inicia um contador
    for k in range(4):
        for j in range(4):
            posK = aux[matriz[k][j]][0] #passa a linha que elemento k,j da matriz passada por parâmetro deveria estar
```

```
            posJ = aux[matriz[k][j]][1]  #passa a coluna que elemento k,j da matriz passada por parâmetro deveria estar
            heuristica += math.fabs(k - posK) + math.fabs(j - posJ) #faz o cálculo da distância retangular


    return heuristic      #retorna a distância retangular

def h4(matriz):
    n = 0.05*h1(matriz) + 0.01*h2(matriz) + 0.94*h3(matriz)
    return(n)

def h5(matriz):
    n = max(h1(matriz),h2(matriz),h3(matriz))
    return n
```

Resultados

Para testarmos o algoritmo foram usados os diferentes casos de teste disponibilizados no moodle e no RunCodes. Tal programa foi compilado e executado pelo programa **PyPy 2.4.0. 32bits** em uma máquina com sistema operacional Windows 8.1, com 8GB de memória RAM e processador Intel Core i7.

Obs.: O algoritmo deste trabalho não passou para o caso 3 do RunCodes devido ao fato do interpretador utilizado pelo servidor ser o CPython e este faz interpretação pura. Já o PyPy possui um compilador Just in Time que o torna o Python tão eficiente quanto outras linguagens como Java ou C++.

O caso 1 do RunCodes é um caso básico do problema em que sabemos que o menor número de movimento para chegarmos à organização original (estado final) deverá ser 9.

Executamos o programa para as cinco heurísticas diferentes e obtivemos os seguintes resultados:

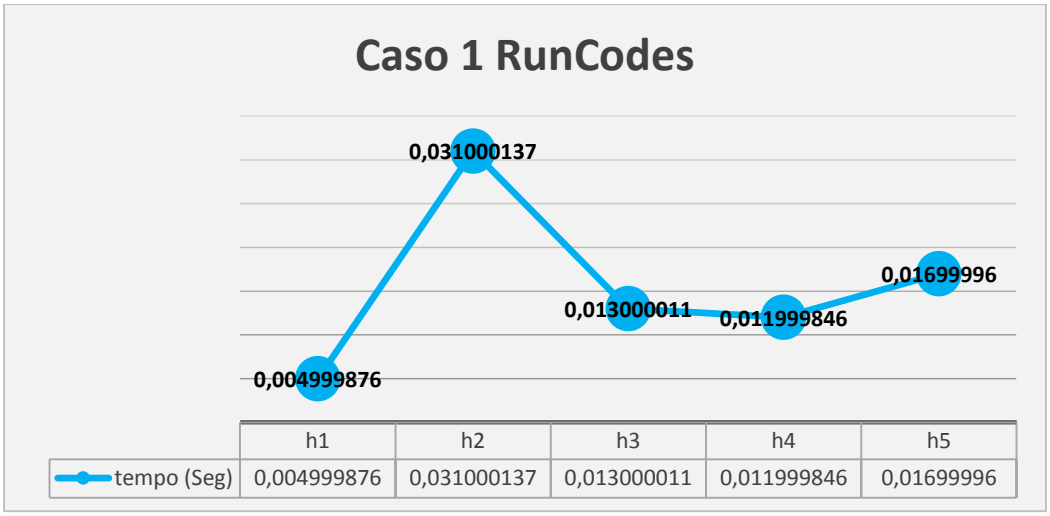
Obs.: Os casos em que as heurísticas não possuem valor atribuídos excederam o limite memória.

Caso 1 RunCodes:

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

Saída: 9

Gráfico 1: Gráfico que representa do tempo de execução para o caso 1 do RunCodes em cada heurística.

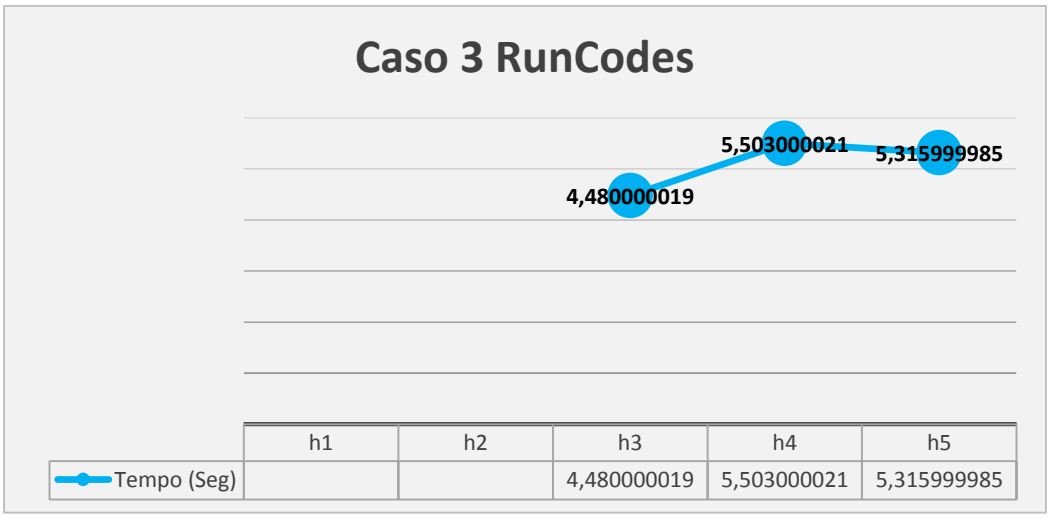


Caso 3 RunCodes:

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

Saída: 38

Gráfico 2: Gráfico que representa do tempo de execução para o caso 3 do RunCodes em cada heurística.

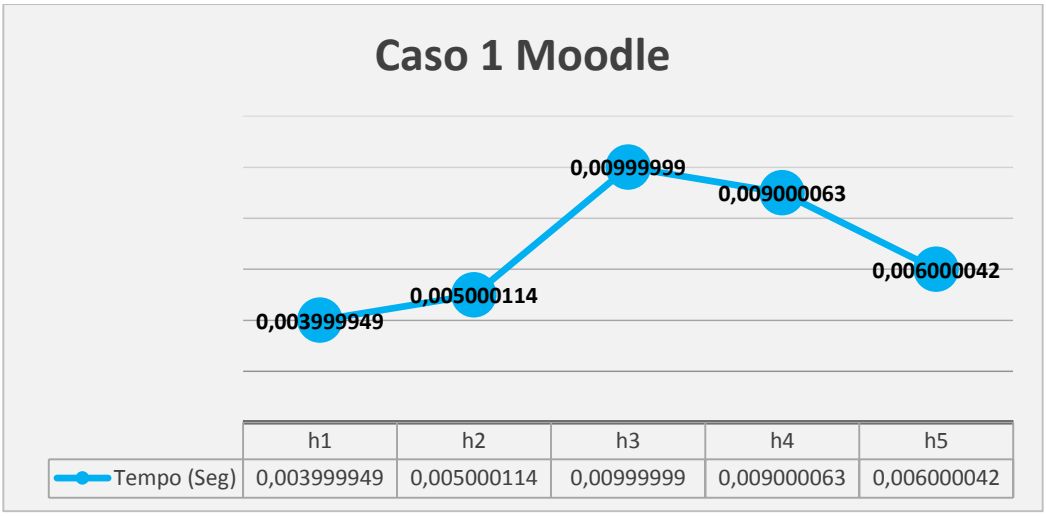


Caso 1 Moodle:

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

Saída: 8

Gráfico 3: Gráfico que representa do tempo de execução para o caso 1 do Moodle em cada heurística.

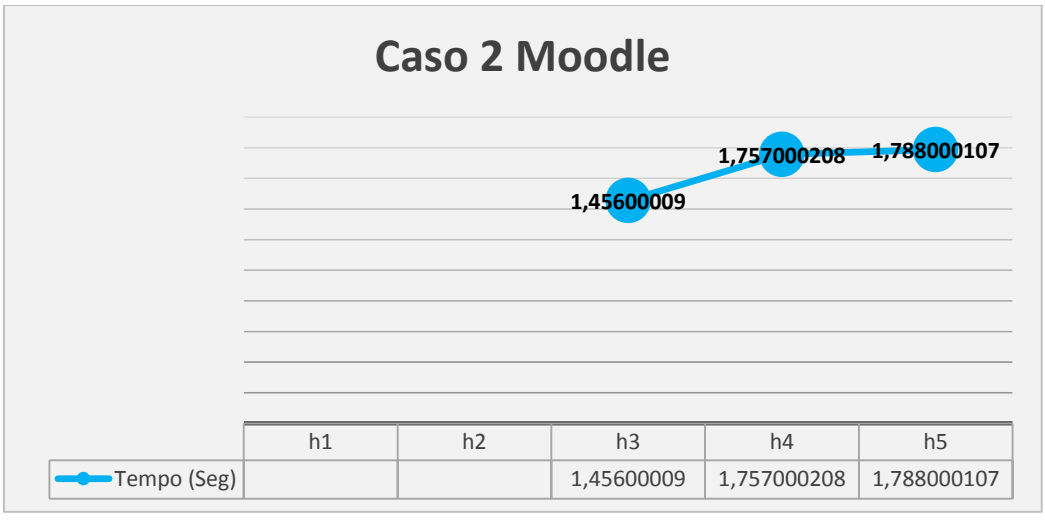


Caso 2 Moodle:

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

Saída: 36

Gráfico 4: Gráfico que representa do tempo de execução para o caso 2 do Moodle em cada heurística.

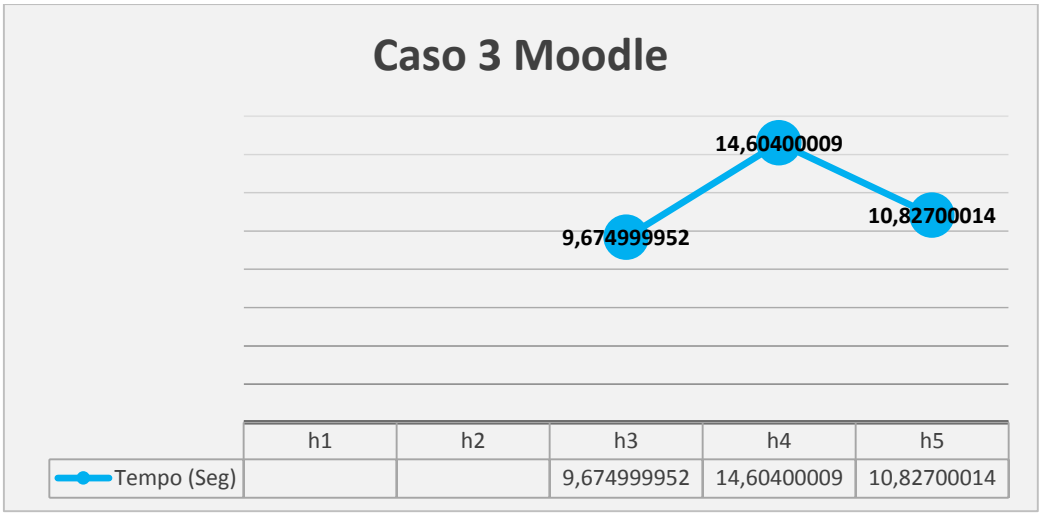


Caso 3 Moodle:

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

Saída: 41

Gráfico 5: Gráfico que representa do tempo de execução para o caso 3 do Moodle em cada heurística.

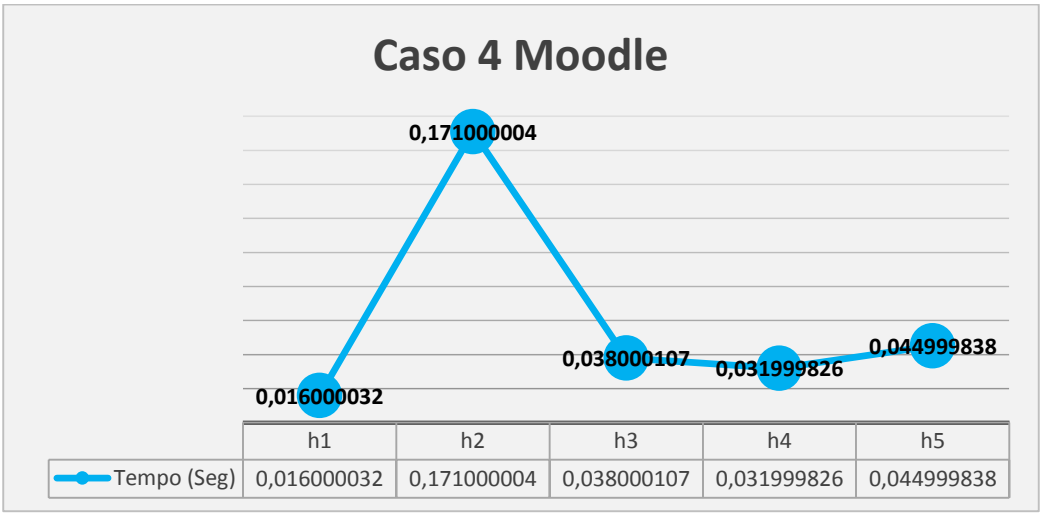


Caso 4 Moodle:

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

Saída: 13

Gráfico 6: Gráfico que representa do tempo de execução para o caso 4 do Moodle em cada heurística.

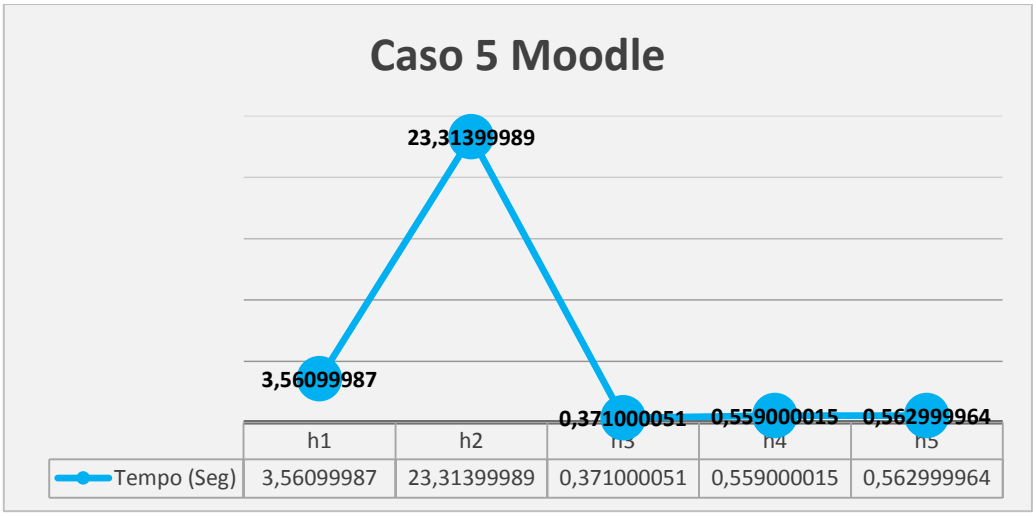


Caso 5 Moodle:

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

Saída: 28

Gráfico 7: Gráfico que representa do tempo de execução para o caso 5 do Moodle em cada heurística.

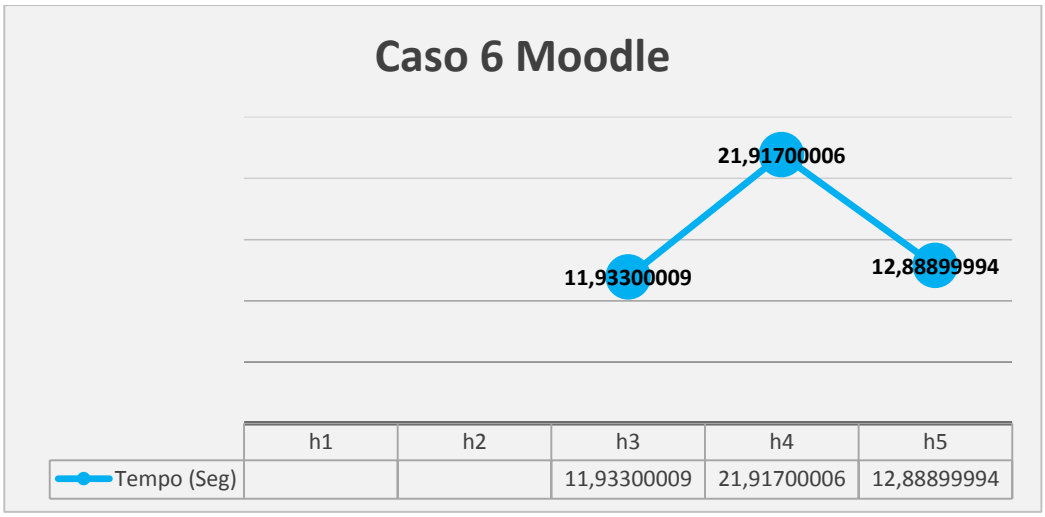


Caso 6 Moodle:

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

Saída: 46

Gráfico 8: Gráfico que representa do tempo de execução para o caso 6 do Moodle em cada heurística.

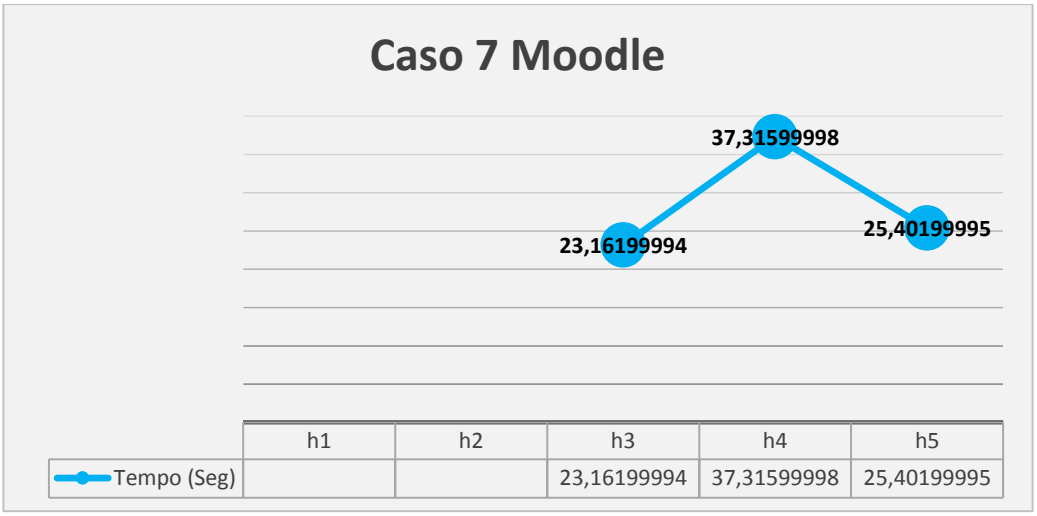


Caso 7 Moodle:

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

Saída: 48

Gráfico 9: Gráfico que representa do tempo de execução para o caso 7 do Moodle em cada heurística.

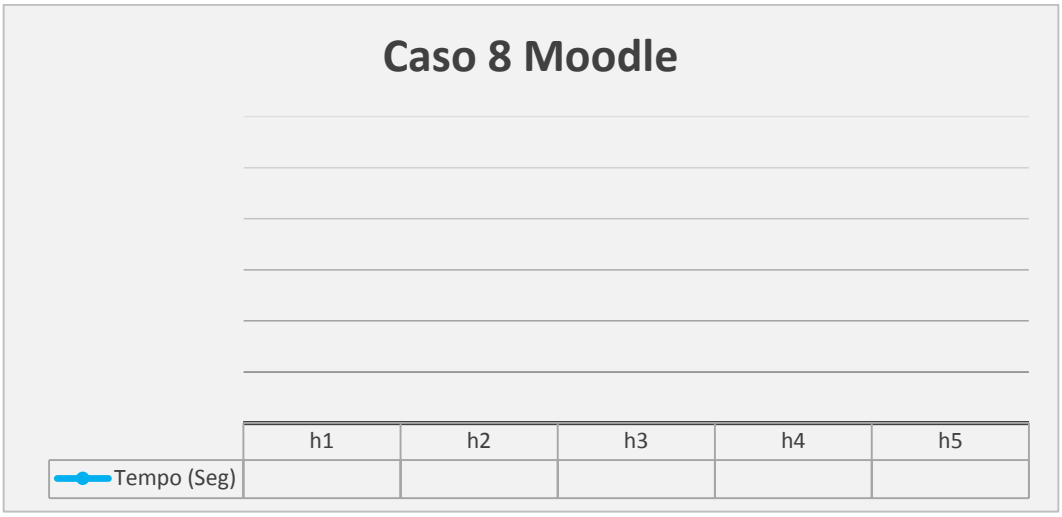


Caso 8 Moodle:

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

Saída:

Gráfico 10: Gráfico que representa do tempo de execução para o caso 8 do Moodle em cada heurística.

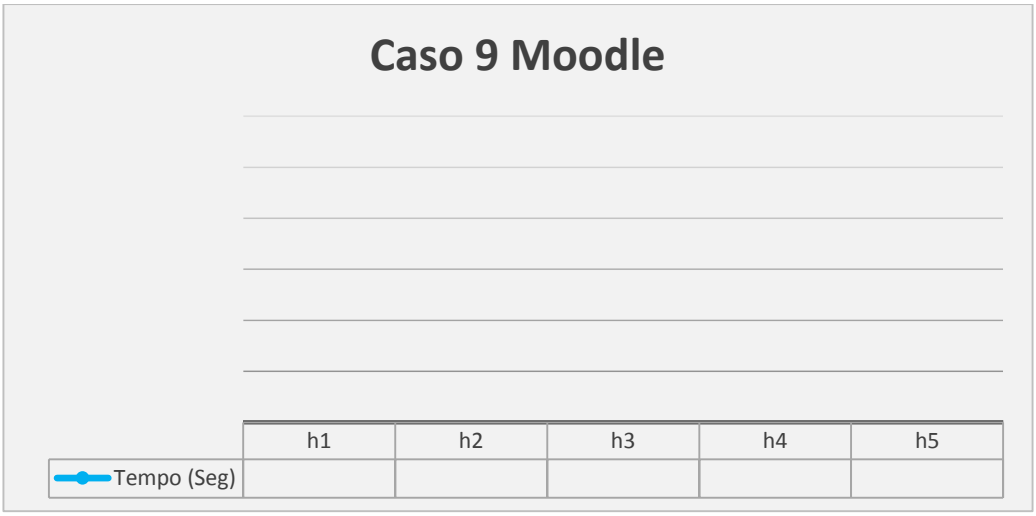


Caso 9 Moodle:

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

Saída:

Gráfico 11: Gráfico que representa do tempo de execução para o caso 9 do Moodle em cada heurística.

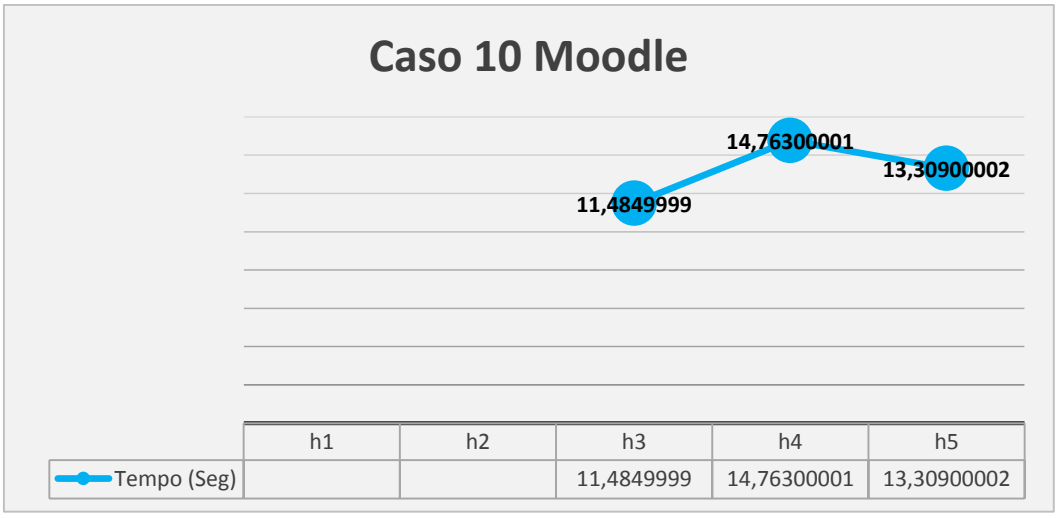


Caso 10 Moodle:

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

Saída: 42

Gráfico 10: Gráfico que representa do tempo de execução para o caso 10 do Moodle em cada heurística.



Conclusão

Com os resultados obtidos concluímos que o algoritmo da busca A* é uma boa estratégia para combinar o custo mínimo do caminho g(n) com o resultado da heurística

h’(n). Obtendo o resultado da menor sequência de movimentos para chegar ao estado final do jogo.

Em relação às heurísticas testadas, nota-se que há variações de desempenho entre as mesmas para diferentes entradas de dados. Isso ocorre porque uma função heurística é específica para cada tipo de problema, onde se estima um custo para o estado em que se encontra. Sendo assim, dependendo da organização do tabuleiro do jogo podemos ter, por exemplo, um maior número de peças fora da sua posição final ou fora de sequência, e com isso uma heurística pode ser melhor do que outra para uma dada instância.