INSTITUTO FEDERAL DO ESPÍRITO SANTO

IFES – CAMPUS SERRA

André Felipe Santos Martins

RELATÓRIO: ALGORITMO DE BUSCA A*

SERRA - ES

2021

1. Explicação teórica

O A* (lê-se a-estrela) é um algoritmo muito utilizado na área de computação para encontrar caminhos entre dois pontos A e B, com base na introdução de uma heurística no algoritmo de busca, no intuito de planejar com antecedência cada passo a ser dado, de modo que uma decisão mais otimizada possa ser feita. O A* vem sendo utilizado amplamente para resolver vários problemas de diferentes áreas, como o alinhamento de múltiplas sequências de DNA em biologia, planejamento de caminhos em robótica e jogos digitais (Rios L.H.O., Chaimowicz L., 2010, pg. 1).

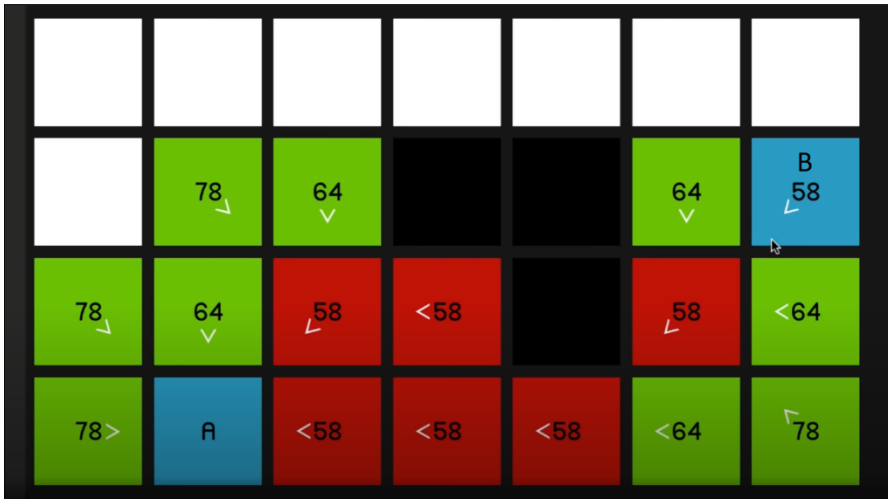

Figura 1- Exemplo de execução do A (fonte: A* Pathfinding vídeo).*

O A* é como uma extensão do algoritmo de Dijkstra, com adição de uma característica chamada Busca em largura (Breadth-first search), que funciona como uma árvore de caminho de custo ordenada por nós que contém o menor custo do nó inicial até o nó de destino. O que torna o A* diferente e melhor em alguns casos, é que para cada nó, o A* utiliza uma função $f(n)$ para calcular uma estimativa do custo total de um caminho a partir do nó avaliado. Essa função é o que contém o que chamamos de heurística, pois trata-se mais de uma estimativa e não necessariamente da distância correta.

Durante a execução do algoritmo, o custo é calculado da seguinte maneira:

$$f(n) = g(n) + h(n)$$

no qual,

n = nó avaliado;

$f(n)$ = custo total estimado do caminho através do nó n;

$g(n)$ = custo para chegar do nó inicial até n;

$h(n)$ = custo estimado de para percorrer de n até o nó final. Esse cálculo se trata é feito por estimativa, portanto, trata-se da heurística.

De acordo com a , o algoritmo A* inicia no ponto ***A***, em azul, e percorre todos os nós adjacentes em busca do nó que contém o menor custo $f(n)$. Uma lista é criada com todos os

nós adjacentes e obstáculos são desconsiderados. Então o nó de menor custo é selecionado e todas essas operações são feitas recursivamente até que o nó n encontrado seja o ***B***, que é o objetivo final. A função $h(n)$ pode ser calculada de diversas formas, mas o método mais comum é o método de Manhattan, é um método padrão para calcular a heurística em uma tabela ou matriz. O cálculo de $h(n)$ é feito da seguinte forma:

$$h = |x - x'| + |y - y'|$$

no qual,

x e y = coordenadas na matriz o nó atual;

x' e y' = coordenadas na matriz do nó final (objetivo).

Para calcular a heurística de movimentos não diagonais, o método de Manhattan consegue atender de forma satisfatória, porém quando se trata de movimentos diagonais, pois no método de Manhattan não é considerado que movimentos 45 graus são permitidos. Para isso utilizamos o método Octil. Essa heurística leva em conta o fato que se pode viajar em arestas diagonais, então, se a distância entre um nó (x, y) é x e y, então a heurística pode ser calculada como:

$$h = max(|\ x - x'|, |y - y'|)\ +\ (sqrt(2) - 1) * min(|\ x - x'|, |y - y'|)$$

no qual,

x e y = coordenadas na matriz o nó atual;

x' e y' = coordenadas na matriz do nó final (objetivo);

$min()$ e $max()$ servem para representar que $sqrt(2) - 1$ deve ser multiplicado pelo menor valor.

2. Problema proposto

O objetivo deste trabalho é implementar ou selecionar um bom algoritmo A* e fazer uma análise da sua implementação, destacando pontos importantes do código e dando o devido destaque a melhor forma de implementar o algoritmo. Além disso vamos realizar testes com matrizes de diversos e tamanhos e demonstrar os resultados.

Os requisitos que devem ser atendidos neste trabalho são:

1. Utilizar/Testar o mesmo modelo de mapa (grid) disponível no enunciado do trabalho;
2. Fazer entrada de parâmetros por linha de comando;
3. Exibir ao final do programa um mapa com o desenho da trajetória, a lista de coordenadas a serem percorridas e o tempo gasto na execução.

3. Implementação

Já que havia a possibilidade de selecionar um algoritmo já implementado, por questões de tempo de priorização de algumas atividades, optamos por selecionar uma implementação feita

na biblioteca em Python chamada python-pathfinding, que contém vários algoritmos de busca e encontrar caminhos, mas claramente com ênfase no A*. Este repositório foi escolhido pois a implementação é feita utilizando diversas classes e estruturas de dados e trabalha de forma paramétrica, ou seja, existe uma classe base genérica chamada Finder, que contém os métodos genéricos de calcular custo, aplicar heurística, encontrar vizinhos etc., e é injetada dentro da classe devida de busca, por exemplo AStarFinder.

Para utilizar a biblioteca eu removi outros arquivos referentes a outras buscas, deixando o repositório mais enxuto. Para os testes, implementei uma função principal que recebe como parâmetro via linha de comando o nome do arquivo, o ponto inicial e o ponto final. O mapa é lido de dentro do arquivo e armazenado dentro de uma classe chamada Grid, que contém toda a estrutura dos nós, ponto inicial e final. Como neste trabalho trata-se de uma análise do algoritmo A* baseado na biblioteca selecionada, irei demonstrar mais partes do código e fazer os devidos comentários sobre.

```
def main():
    # get params from sys
    filename = sys.argv[1]
    start = sys.argv[2]
    end = sys.argv[3]

    # get axis params
    start = str_to_axis(start)
    end = str_to_axis(end)

    # load maze into a matrix
    maze = load_maze(filename)

    # instantiate lib pathfinding components #
    # grid (maze matrix)
    grid = Grid(matrix=maze)

    # start and end nodes existing on the grid
    start = grid.node(start[0], start[1])
    end = grid.node(end[0], end[1])

    # instantiate finder and run
    finder = AStarFinder(diagonal_movement=DiagonalMovement.always)

    start_time = time.time()
    path, runs = finder.find_path(start, end, grid)
    end_time = time.time() - start_time

    print('operations:', runs, 'path length:', len(path), 'execu-
tion time:', end_time)
    print(grid.grid_str(path=path, start=start, end=end))
    print('path:', path)
# end-main()
```

Figura 2 - função main(). Fonte: Implementada pelo autor.

A Grid é uma classe que tem o objetivo de representar a matriz do "labirinto. Ela recebe como parâmetro uma matriz (lista) e constrói internamente uma estrutura de nós, que onde cada nó possui suas propriedades de posição (x, y), se é um caminho viável (não se trata de uma barreira ou canto), seus valores de $f(n)$, g(n) e h(n) que serão calculados posteriormente, e uma

propriedade auxiliar de aberto ou fechado, que é utilizado para verificar se o nó foi adicionado na lista de caminhos abertos ou fechados. A Grid contém os seguintes métodos:

- $build\_nodes()$ - Inicializa a estrutura de nós;
- $node()$ - Retorna um nó de uma posição (x, y);
- $inside()$ - Verifica se uma posição (x, y) está dentro do mapa;
- $walkable()$ - Verifica se um nó (x, y) não é um objeto ou parede;
- $neighbors()$ - Retorna uma lista de vizinhos de um nó (x, y);
- $cleanup()$ - Limpa grid;
- $grid\_str()$ - Imprime a grid definida, aceitando como parâmetros os pontos inicial e final e uma lista contendo o caminho descoberto.

Para instanciar a classe AStarFinder passamos apenas o parâmetro $diagonal\_movement$, que é um enumerador e permite que o programador decida se é permitido realizar caminhos usando diagonais. Enfim, para rodar o buscador são passados pelo menos três parâmetros, o primeiro é a Grid e os dois últimos são instância de um nó para o início e o fim, que pode ser retirado diretamente da Grid com o comando $node = grid.node(x, y)$.

A partir do momento que a função $find\_path()$ é chamada, todo algoritmo é realizado internamente pela biblioteca. A partir daqui todo código é referente ao repositório utilizado. Como citado anteriormente, tudo foi desenvolvido de uma forma genérica para que funcionasse com outros algoritmos de busca, então a $find\_path()$ apenas chama a sua equivalente dentro da classe Finder, que foi injetada no momento que a AStarFinder foi instanciada. Em seguida, temos a função de busca em si.

```
"""
AStarFinder Class - a_star.py
"""
def find_path(self, start, end, grid):
        start.g = 0
        start.f = 0
        return super(AStarFinder, self).find_path(start, end, grid)
```

Figura 3 - função find_path() da classe AStarFinder.

```
"""
Finder Class - finder.py
"""
def find_path(self, start, end, grid):
        self.start_time = time.time()  # execution time limitation
        self.runs = 0  # count number of iterations
        start.opened = True

        open_list = [start]

        while len(open_list) > 0:
            self.runs += 1
            self.keep_running()

            path = self.check_neighbors(start, end, grid, open_list)
            if path:
                return path, self.runs

        # failed to find path
        return [], self.runs
```

Figura 4 - função find_path() da classe Finder.

A função find_path() nada mais faz do que o que foi citado na 1. Explicação teórica: é criada uma lista aberta com o primeiro nó, e enquanto houver nós dentro dessa lista, o loop continua o looping. Um dos parâmetros opcionais da função find_path() é o tempo máximo de execução do programa, para isso serve a linha que contém $self.keep\_running()$, que verifica que o tempo limite foi excedido, porém não foi considerada essa funcionalidade neste trabalho.

```
def check_neighbors(self, start, end, grid, open_list,
                        open_value=True, backtrace_by=None):
        # pop node with minimum 'f' value
        node = heapq.nsmallest(1, open_list)[0]
        open_list.remove(node)
        node.closed = True

        # if reached the end position, construct the path and return it
        # (ignored for bi-
directional a*, there we look for a neighbor that is
        #  part of the oncoming path)
        if not backtrace_by and node == end:
            return backtrace(end)

        # get neighbors of the current node
        neighbors = self.find_neighbors(grid, node)
        for neighbor in neighbors:
            if neighbor.closed:
                # already visited last minimum f value
                continue
            if backtrace_by and neighbor.opened == backtrace_by:
                # found the oncoming path
                if backtrace_by == BY_END:
                    return bi_backtrace(node, neighbor)
                else:
                    return bi_backtrace(neighbor, node)

            # check if the neighbor has not been inspected yet, or
            # can be reached with smaller cost from the current node
            self.process_node(neighbor, node, end, open_list, open_value)

        # the end has not been reached (yet) keep the find_path loop runn
ing
        return None
```

Figura 5 - função check_neighbors() na classe AStarFinder.

A função $check\_neighbors()$ deve ser implementada individualmente, por isso ela se encontra dentro da classe AStarFinder com sua própria implementação. É nela que é selecionado o nó de menor custo, retirada da lista como numa estrutura de pilha utilizando o comando $node = heapq.nsmallest(1, open\_list)[0]$, e todos seus vizinhos são verificados e processados dentro do looping, removendo o nó atual da lista aberta, fechando-o e desconsiderando vizinhos que são obstáculos.

Dentro da função $self.find\_neighbors(grid, node)$ é chamada a função $grid.neighbors()$, que retorna a lista de vizinhos válidos. Cada vizinho verificado se ele já foi para lista fechada. A parte mais importante é $self.process\_node()$, no qual o vizinho é processado, calculando seu custo checando se se ele é o objetivo final. Ignore a função $bi\_backtrace()$, pois ela é utilizada apenas no algoritmo A* bidirecional.

```
def process_node(self, node, parent, end, open_list, open_value=True):
        '''
        we check if the given node is path of the path by calculating its
        cost and add or remove it from our path
        :param node: the node we like to test
            (the neighbor in A* or jump-node in JumpPointSearch)
        :param parent: the parent node (the current node we like to test)
        :param end: the end point to calculate the cost of the path
        :param open_list: the list that keeps track of our current path
        :param open_value: needed if we like to set the open list to some
thing
            else than True (used for bi-directional algorithms)

        '''
        # calculate cost from current node (parent) to the next node (nei
ghbor)
        ng = self.calc_cost(parent, node)

        if not node.opened or ng < node.g:
            node.g = ng
            node.h = node.h or \
                self.apply_heuristic(node, end) * self.weight
            # f is the estimated total cost from start to goal
            node.f = node.g + node.h
            node.parent = parent

            if not node.opened:
                heapq.heappush(open_list, node)
                node.opened = open_value
            else:
                # the node can be reached with smaller cost.
                # Since its f value has been updated, we have to
                # update its position in the open list
                open_list.remove(node)
                heapq.heappush(open_list, node)
```

Figura 6 - função process_node() na classe Finder.

Na função $process\_node()$ é verificado se o nó faz parte do caminho através do cálculo do seu custo e baseado nisso ele é adicionado ou removido do caminho final. O valor de $g()$ do nó enviado como parâmetro é calculado através da função $self.calc\_cost(parent, node)$, que basicamente verifica a distância entre o nó atual e seu vizinho, também verificando se é um vizinho direto ou um vizinho indireto (diagonal).

Se o nó ainda não foi processado em nenhum momento ou seu custo de movimento medido no momento for menor ($if\ not\ node.opened\ or\ ng\ <\ node.g$), o valor de $g()$ é mantido, e então calcula-se a heurística, caso ainda não calculada, utilizando $self.apply\_heuristic(node, end)\ *\ self.weight$. Nesta função é chamada a o cálculo da heurística, que nesse caso é o método Octil, citado na1. Explicação teórica. Calculado o $g()$ e o $h()$, é somado em $f()$ e armazenado. Se o nó não foi adicionado a lista aberta, ele é adicionado. Caso esteja aberto, a posição dele é atualizada dentro da lista aberta.

Seguindo a partir da Figura 5, quando o nó final é encontrado, é chamada a função $backtrace(end)$, que percorre o último nó e concatena todos os nós pais até chegar ao nó inicial.

4. Resultados

Para execução e teste foram criados dois arquivos de $main()$ diferentes, são eles:

- main.py – Arquivo que recebe parâmetros através do terminal, sendo eles o nome do arquivo e duas coordenadas no labirinto. Um exemplo de uso: $python$.\ $main.py$.\$Mapa_B.txt$ 2,3 14,3
- mainV2.py – Arquivo utilizado para testar vários pontos aleatórios em massa. Todos os parâmetros estão atribuídos dentro do código, não são aceitos parâmetros via linha de comando.

Os dois arquivos são tem a mesma funcionalidade e possuem funções para tratamento e preparação da entrada de dados para serem passados como parâmetros. Um exemplo de execução o arquivo main.py sobre o labirinto no arquivo Mapa_B.txt, é o seguinte:

Comando executado: $python$.\$main.py$.\$Mapa_B.txt$ 2,2 20,53

Figura 7- labirinto do arquivo Mapa_B.txt

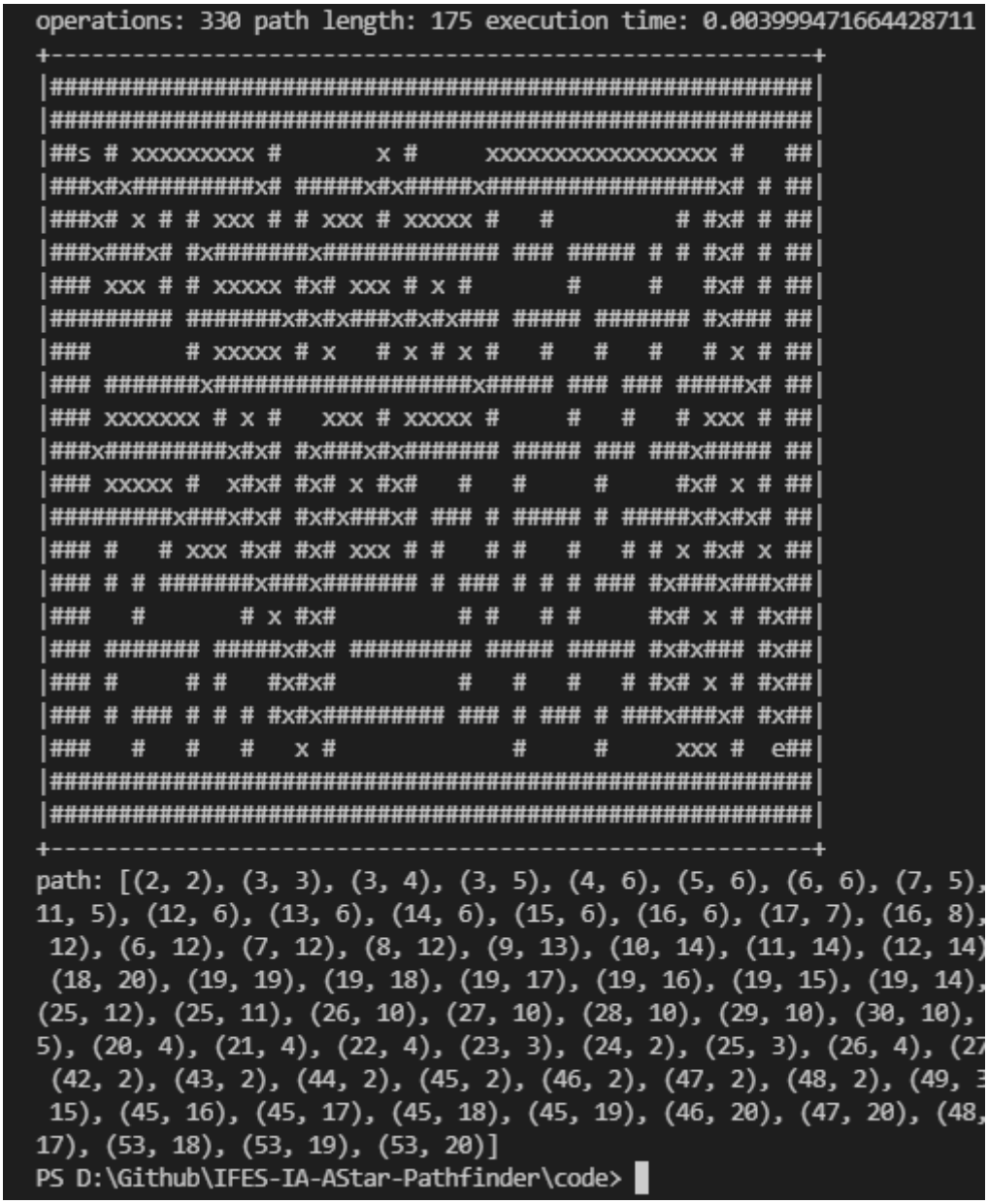

Figura 8 - Saída do programa.

Como podemos ver, no resultado temos o número de operações sobre a lista aberta, a quantidade de passos para chegar ao objetivo, o tempo de execução, o labirinto representando por X onde é o caminho e a lista inteira de coordenada dos caminhos. Como pode ver esse algoritmo pode performar em muito menos de um segundo. Mas para realmente testá-lo que foi criado o arquivo mainV2.py. Nele é feito testes em massa sobre os labirintos de diferentes tamanhos em pontos aleatórios do mapa.

Para efetuar os testes e armazenas os resultados, a saída do arquivo mainV2.py não consta com o mapa do labirinto nem o caminho percorrido, mas sim com dados referentes para análise a seguir. Comando executado: python .\mainV2.py > output.csv;

```
filename,maze_lines,maze_columns,start,end,path_length,time
.\Mapa_B.txt,22,55,(51; 18),(45; 10),160,0.001997709274291992
.\Mapa_B.txt,22,55,(31; 2),(17; 8),359,0.0050013065338134766
.\Mapa_B.txt,22,55,(12; 4),(15; 20),102,0.0010004043579101562
```

Figura 9 - exemplo de saída do arquivo mainV2.py.

A saída do teste da figura 9 foi feito com 100 coordenadas diferentes de início e fim. Com base no arquivo de CSV gerado, podemos gerar o seguinte gráfico:

Figura 10 - Gráfico de comparação de tempo de execução e o tamanho do caminho percorrido.

Na Figura 10 é possível observar que o algoritmo performa muito bem até quase 500 passos, levando alguns milésimos para encontrar o objetivo final. Considerando que este labirinto, como é exibido na saída CSV, possui um tamanho de 22 linhas e 55 colunas, vamos fazer um teste em um labirinto maior.

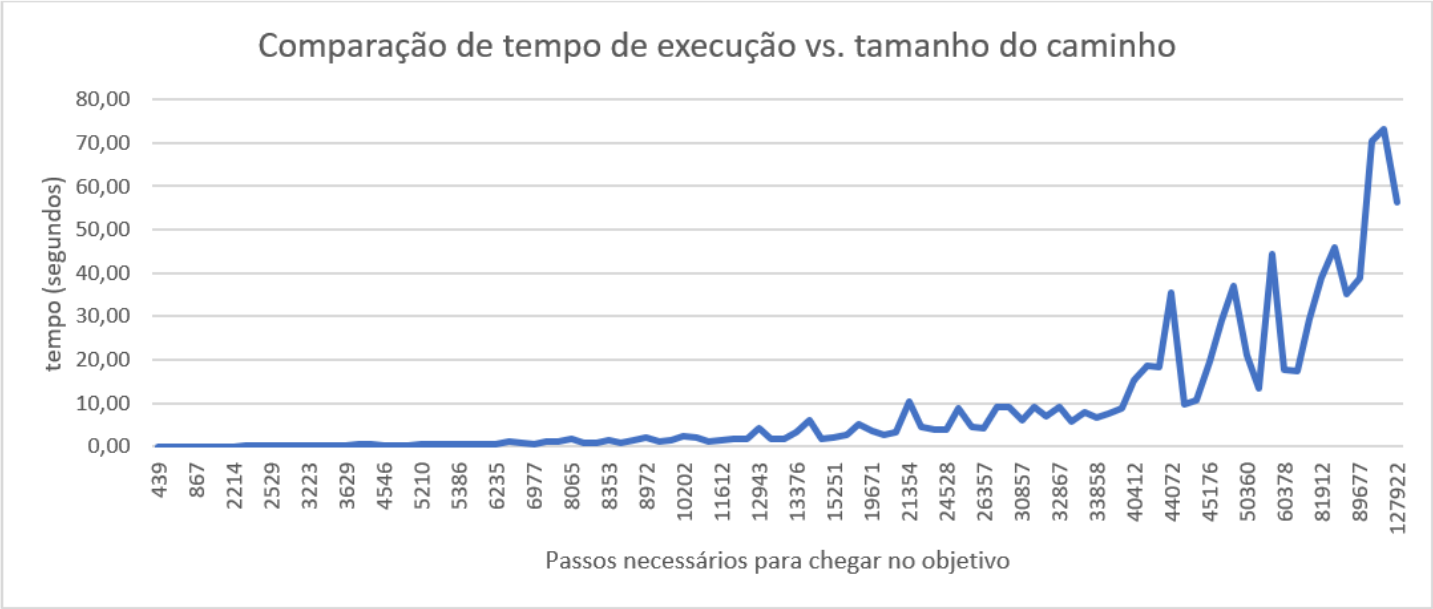

Figura 11 - comparação de tempo de execução e o tamanho do caminho no arquivo Mapa_A__V06.txt.

O teste da Figura 11 foi executado em 100 diferentes pontos de início e fim sobre o arquivo Mapa_A__V06.txt, de 1080 linhas e 540 colunas. Como podemos observar o gráfico, o algoritmo é bem performático até cerca de 33 mil passos distante do objetivo final, levando cerca de 10 segundos para encontrar o objetivo. Além disso, podemos observar que em cerca

de 83 mil passos o algoritmo já levou cerca de 70 segundos para encontrar o objetivo. Claramente isso possui variáveis, como a disposição dos obstáculos sobre o labirinto, mas no geral, atende muito bem. Para trabalhos futuros seria interessante um gerador de labirintos, que possibilite gerar labirintos superpopulosos de obstáculos ou repleto de corredores que dificultem o trabalho do algoritmo.

5. Referencias

https://homepages.dcc.ufmg.br/~chaimo/public/SBIA10.pdf

https://github.com/brean/python-pathfinding