INSTITUTO POLITÉCNICO DE BEJA

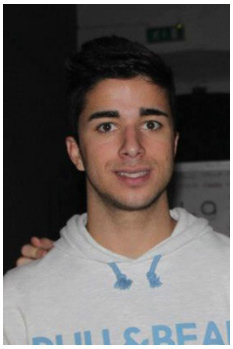
Escola Superior de Tecnologia e Gestão

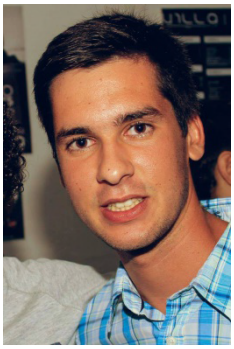
Licenciatura em Engenharia Informática

Estrutura de Dados e Algoritmos

Relatório

Estudo do funcionamento e da complexidade computacional do Algoritmo A*





12551, Pedro Miguel Montes Santos

12552, Emanuel Alexandre Cavaco Teixeira

Docente: José Jasnau Caeiro

Beja

2014

Índice

1- Introdução

Nesta unidade curricular foi-nos proposta a realização de um trabalho prático visando o estudo da complexidade computacional de um algoritmo de pesquisa heurística em grafos, mais conhecido por A*. Este algoritmo é uma extensão do algoritmo de *Dijkstra*.

O A* pode ser vulgarmente tratado como *Path Finding,* pois este algoritmo trata de encontrar o caminho com menor custo, de um vértice inicial até a um vértice final, utilizando a melhor heurística para o efeito pretendido.

Implementações deste algoritmo podem ser frequentemente encontradas em jogos ou em busca de rotas entre localidades.[1]

1.1- Objetivos e motivação

O presente trabalho tem como objectivo numa fase inicial a programação do próprio algoritmos A* com base em pseudocódigo[2] resultante de pesquisa. Decidimos para isso utilizar a linguagem *Python* por ser uma linguagem de alto nível, que nos proporciona um resultado final sucinto e elegante. Outro grande fator que nos levou a optar por esta linguagem foi o facto de possuirmos algumas lacunas na mesmas, aumentando o desafio pessoal e procurando aprender mais sobre a mesma.

Optamos por seguir as indicações do docente, utilizando a biblioteca *OpenCV*[3], muito útil e com um enorme leque de funções, as quais muitas delas não teremos oportunidade de explorar devidamente no decorrer do projeto. A *OpenCV* irá nos ajudar no momento em operemos com o ficheiro *.pgm* em analise.

Numa fase posterior, e depois de já alcançado sucesso no ponto anterior iremos elaborar um estudo computacional dos resultados experimentais obtidos. Estes resultados experimentais são obtidos com base no tempo que cada busca demora a encontrar um novo ponto.

1.2 - Contributos

O presente trabalho foi desenvolvido por um grupo de duas pessoas descritas na capa do mesmo. Ambos tiveram um contributo ativo no seu desenvolvimento, sendo benéfica a troca de ideias em algumas situações menos claras do processo. Usufruímos de alguns contributos externos mencionados em bibliografia. Recorremos a alguns pseudocódigos disponíveis pela *web,* tudo documentado bibliograficamente.

1.3- Estrutura do Documento

Este relatório é constituído por uma introdução que resume em poucos parágrafos os objetivos propostos para o trabalho. Segue-se a parte teórica, onde são abordados os aspetos teóricos do documento. A parte experimental consiste na apresentação dos resultados da realização da estrutura de dados e dos algoritmos desenvolvidos, bem como o seu estudo. São medidos tempos de execução e apresentados em gráficos ilustrativos. A conclusão engloba as possíveis dificuldades que tivemos na realização do projeto e as ilações resultantes da análise de todo o processo. A bibliografia representa todos os locais onde foi pesquisada informação referente ao tema proposto. Por fim, os anexos contém todo o código do algoritmo desenvolvido.

2- Parte Teórica

2.1- Introdução

Neste tópico será visto um pouco mais de teoria acerca dos assuntos que estamos a tratar. Podemos estudar três fundamentos teóricos relevantes para a percepção e realização deste trabalho, sendo eles os algoritmos e o *pathfinding*.

Um algoritmo é uma sequência finita de instruções, bem definidas e não ambíguas. O algoritmo serve para resolver problemas, desde que para isso esteja corretamente

implementado e seja adequado ao problema que se pretende tratar. A definição de algoritmo pode ser expressa como um conjunto de passos para realizar uma tarefa.

O conceito de um algoritmo foi formalizado em 1936 pela Máquina de Turing de Alan Turing e pelo calculo lambda de Alonzo Church.

Pathfinding é a maneira de buscar uma trajectória desde um ponto inicial, até um ponto final, evitando os pontos bloqueados, as chamadas barreiras, com o menor custo possível. Assenta numa formula base, ($F = G + H$), em que o H é a distancia estimada do ponto a tratar até ao ponto final, segundo uma determinada heurística.

2.2 - Pseudo-Código

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from, current_node)
    if current_node in came_from
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

FIG. 1 - PSEUDO-CÓDIGO

3- Parte Experimental

3.1- Realização experimental

Neste trabalho era-nos proposto a realização de uma aplicação em Python, C# ou Java, mas optámos pelo Python.

Foi-nos fornecido pelo docente uma máquina virtual com o Sistema Operativo - Debian já com ambientes de desenvolvimento e módulos incluídos. Decidimos utilizar a máquina virtual disponibilizada, pois tivemos dificuldades em instalar os módulos nos nossos sistemas operativos (Mac OSX Mavericks).

Para o desenvolvimento do código utilizámos o editor de texto *Sublime Text*. Este IDE é multiplataforma que suporta as mais variadas linguagens de programação e permite a instalação de pacotes que ajudam ao desenvolvimento do código de uma maneira mais eficiente e rápida, facilitando assim o trabalho do programador. Permite a instalação de temas para alterar o aspeto da interface, tentando agradar a todos os utilizadores.

Computadores utilizados:
MacBook Pro (ano 2011)

- Processador 2.8 GHz Intel Core i7
- 4GB de memória RAM.

MacBook Pro (ano 2012)

- Processador 2.5 GHz Intel core i5 (TurboBoost até 3.1 GHz)
- 4GB de memória RAM.

Quando utilizada a maquina virtual era utilizado o Virtualbox com 1*GB* de memória *RAM*.

3.2- Sistema experimental

Na realização do nosso trabalho optamos por dividir o código-fonte em dois módulos distintos, o *astar.py* e o *main.py*. Acreditamos que esta divisão seja benéfica para separar o que é o algoritmo de busca do caminho com menor custo do restante código que

por exemplo imprime a imagem no ecrã ou que escreve um ficheiro de dados com os pontos resultantes. Para o estudo da complexidade computacional do resultado obtido optamos por fazer a medição de tempos com recurso ao módulo *time,* mais precisamente à função com o mesmo nome. (*time.time())*

No trabalho desenvolvido foi implementada a heurística da distancia de *Manhattan.*

Na obtenção dos gráficos que demonstram as conclusões do resultado final foi utilizado o Gnuplot.

O módulo *astar.py* é composto por uma classe e várias funções:

- class AStar():
    - def __init__(self):
    - def g(self, neighbor):
    - def h(self, neighbor):
    - def a_star(self, img):

O módulo *main.py* é constituido por duas classes:

- class OpenImage(object):
    - def open_image(self, START_POINT, END_POINT, flag = True):
    - def show_image(self):
    - def save_image(self, name):
    - def draw_point(self, point):
- class WriteFile(object):
    - def write_file(self, name, closed_set):

3.3- Analise dos resultados experimentais obtidos

Para a obtenção de dados experimentais suficientes para conseguirmos fazer uma analise rigorosa sobre o comportamento do algoritmo em estudo foram realizados alguns testes repetidamente. Os testes em causa foram todos realizados nas mesmas circunstancias, para evitar discrepâncias a nível de resultados.

O primeiro teste (*fig. 1*) é uma execução do algoritmo A*. Os tempos aqui apresentados referem-se ao tempo que é necessário para calcular um novo ponto, o ponto

seguinte com um custo mais reduzido. Obtemos cerca de quinhentos e oitenta pontos, todos eles com tempos muito semelhantes, provocado assim um gráfico linear, com a complexidade O(n).

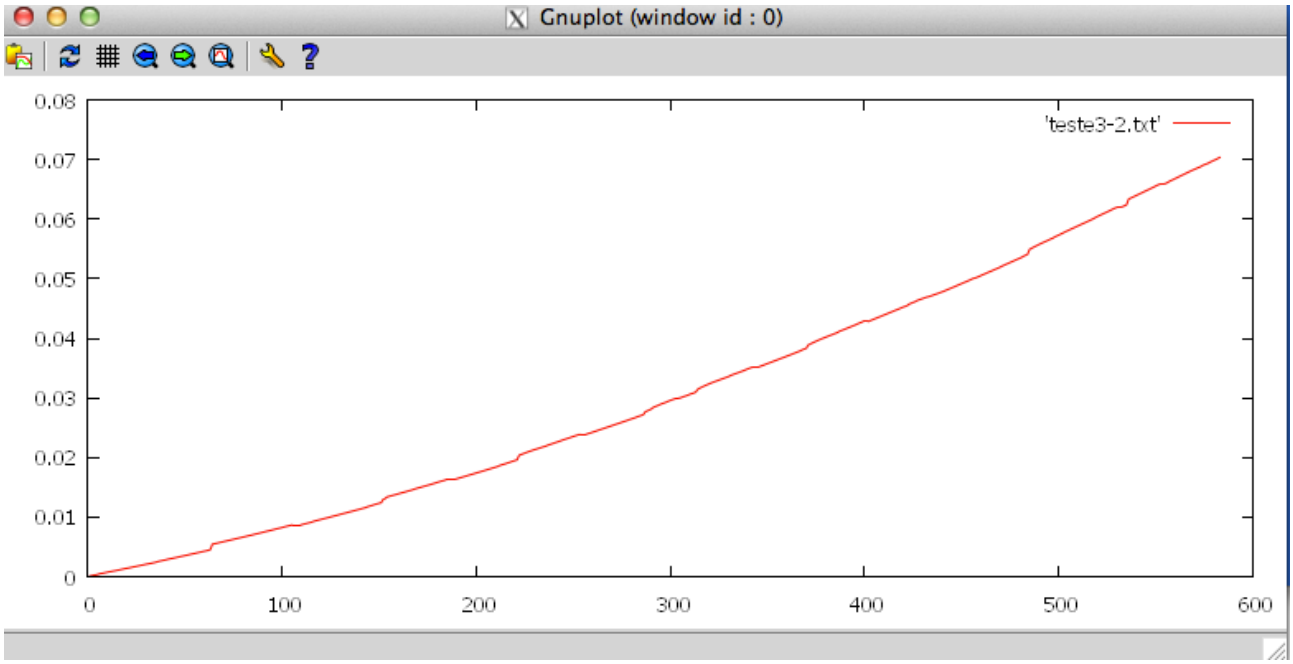


FIG. 2 - TESTE 1

Para um segundo teste (*fig. 2*) decidimos repetir a execução do algoritmo e calcular primeiramente a média dos tempos da procura dos pontos por cada execução, e de seguida calcular a média entre todas as execuções. Iniciamos nas dez repetições, atingindo as cem repetições. Continuamos a obter um gráfico linear, com tempos praticamente idênticos, com uma complexidade computacional de O(n).

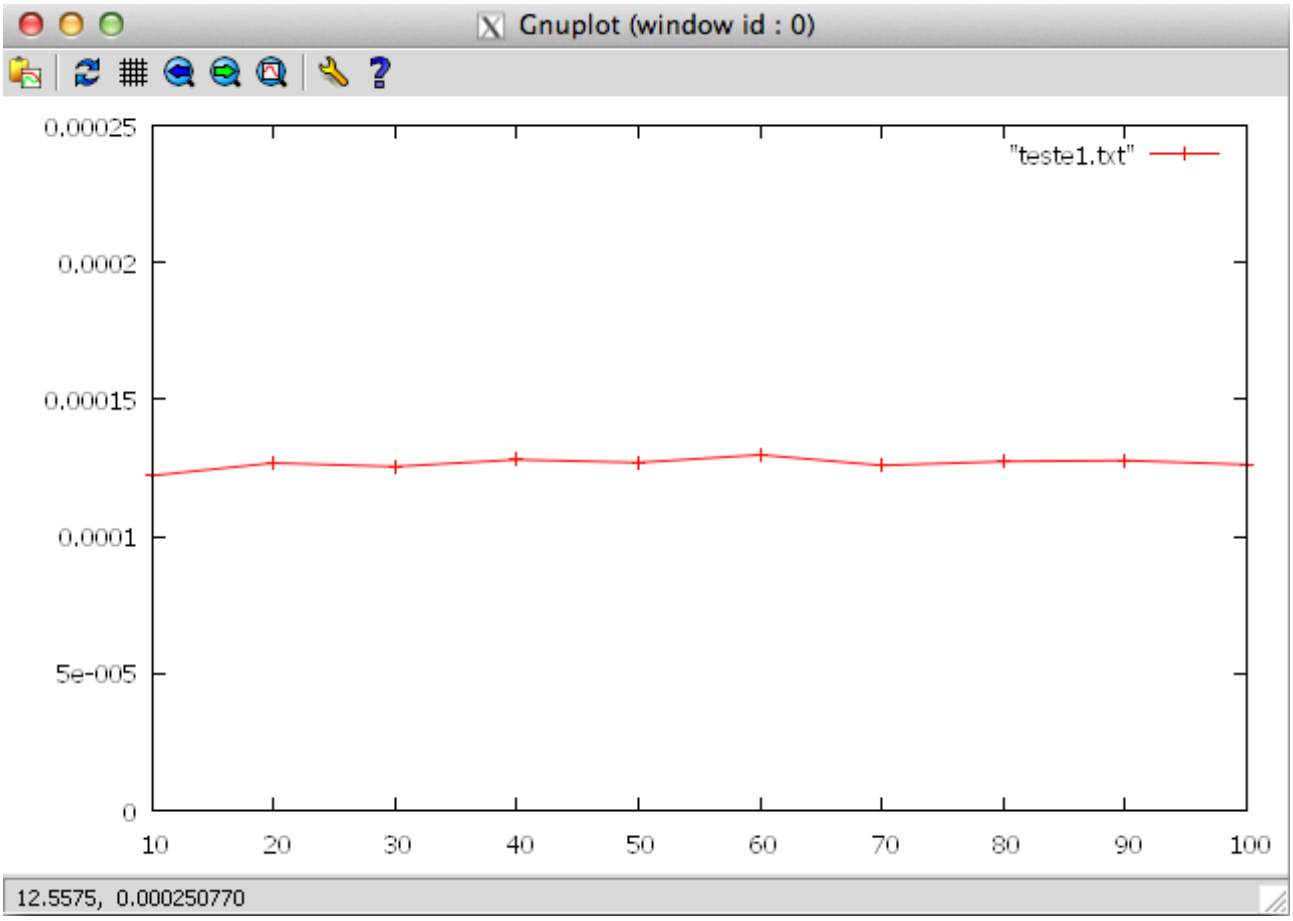


FIG. 3 - TESTE 2

No que toca à qualidade dos resultados obtidos estamos seguramente diante de uma boa pesquisa em termos custo. Para a obtenção deste resultado bastante aproximado ao pretendido recorremos a uma variável que reduz a importância da distancia em linha recta, dando um peso maior à intensidade do pixel em estudo. Quanto maior for esse valor mais peso é dado à intensidade do pixel. Recorremos ao método de tentativa e erro para procurar o valor que mais se ajustava ao caso em especifico. De notar que para diferentes níveis de aproximação basta alterar esse mesmo valor.

FIG. 4 - IMAGEM ORIGINAL

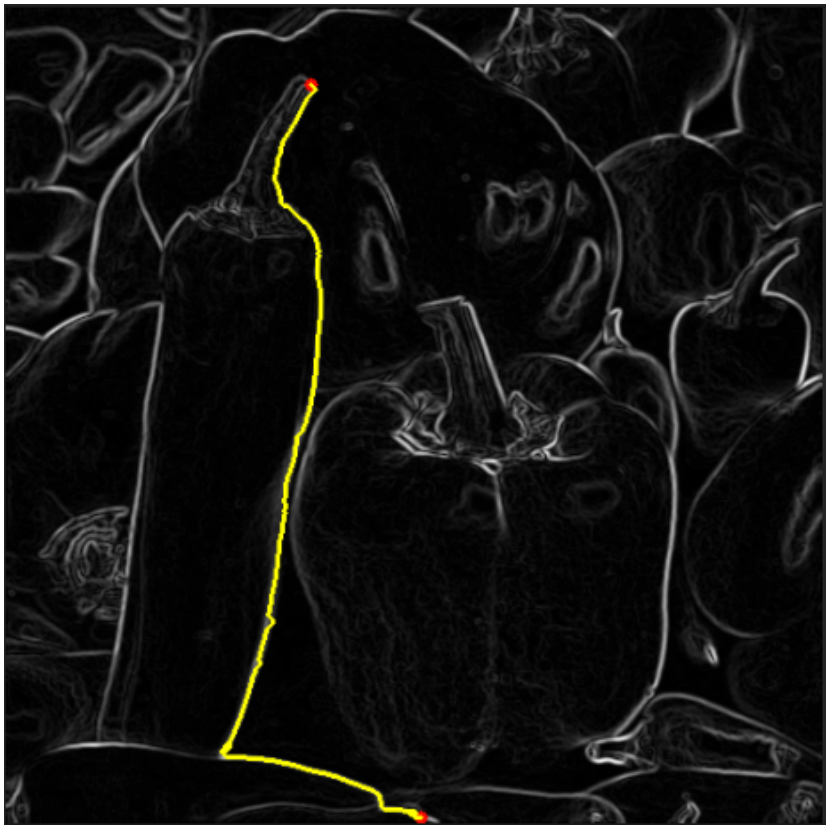


FIG. 5 - RESULTADO EXPERIMENTAL OBTIDO

4- Conclusão

Com a realização deste trabalho experimental podemos tirar algumas conclusões acerca da complexidade computacional do algoritmo em estudo. Tínhamos em nossa posse valores teóricos previstos para uma base comparativa com os resultados alcançados experimentalmente.

Pode concluir-se então que o algoritmo *A** apresenta uma complexidade computacional de O(n) para cada ponto da sua busca. Os valores experimentais aproximam-se dos valores teoricamente esperados.

A sua complexidade O(n), linear, deve-se a que cada um dos pontos em análise depende de outros pontos, para além dos seus vizinhos. Como um ponto tem um máximo de pontos vizinhos, neste caso em particular cada ponto apenas podia ter oito vizinhos, então não existem grandes variações nos tempos de execução para cada iteração do algoritmo.

Acreditamos que este algoritmo atinge uma boa performance no que toca a algoritmos de busca heurística pois apresenta um leque de pesquisa por cada ponto bastante reduzido, o que torna a busca mais rápida, talvez pecando na eficiência da busca. Poderíamos atingir ainda uma melhor performance ao implementar uma outra estrutura de dados no caso da lista fechada, para conseguir uma pesquisa dos seus elementos mais rápida.

No que toca ao código desenvolvido acreditamos ter encontrado uma boa solução, utilizando código pequeno e compacto, tentando utilizar o mínimo de operações, para atingir uma melhor eficiência. O resultado obtido em termos de pesquisa também foi o mais próximo do esperado, indo pelo caminho com menos custo.

Em suma os nossos objetivos foram realizados com sucesso, quer em termos de análise do comportamento do algoritmo quer do seu desenvolvimento propriamente dito.

5- Bibliografia

[1] OpenCV dev team. (2014, Abril 20). [Online]. Disponível em: http://opencv.org

[2] Autor desconhecido. (2014, Abril 14). [Online]. Disponível em: http://en.wikipedia.org/wiki/A*_search_algorithm

[3] Patrick Lester. (2014, Abril 14). [Online]. Disponível em: http://www.policyalmanac.org/games/aStarTutorial.htm

[4] Amit Patel. (2014, Abril 18). [Online]. Disponível em: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[5] Rajiv Eranki. (2014, Abril 22). [Online]. Disponível em: http://web.mit.edu/eranki/www/tutorials/search/

[6] Autor desconhecido. (2014, Abril 22). [Online]. Disponível em: http://stackoverflow.com/questions/11070248/a-time-complexity

6- Anexos

6.1 - astar.py

```
import math

'''
@authors 12551 Pedro Santos & 12552 Emanuel Teixeira
@date 11 de Maio de 2014
@obs: Algoritmo A* Path Finding para a
procura de caminhos com menos custo
em funcao da intensidade.
'''

class AStar():
     """Classe AStar e responsavel pelo algoritmo da busca
     do melhor caminho com menos custo"""
     def __init__(self):
          '''
          Inicializacao dos parametros da classe Astar
          '''
          self.START_POINT = (191, 48) #x, y
          self.END_POINT = (260, 508)
          self.ALPHA = 1
          self.MAX_COR = 255.0
          self.closed_set = []

     def g(self, neighbor):
          '''
          Funcao com o custo g, em funcao da intesndidade da cor
          Quanto maior for a intensidade menor sera o custo

          @param neighbor e o ponto vizinho que esta a ser testado
          @return a normalizacao da cor entre 0 e 1 do ponto neighbor
          '''
          return (1 - (self.img[neighbor[1], neighbor[0]][0] / self.MAX_COR))

     def h(self, neighbor):
          '''
          Funcao com o custo h, em funcao da distancia em linha recta ao ponto
final
          Quanto maior for a distancia maior sera o custo

          @param neighbor e o ponto vizinho que esta a ser testado
          @return a distancia em linha reta do ponto neighbor ao ponto END_POINT
          '''
          return math.sqrt((neighbor[0] - self.END_POINT[0]) ** 2 +
((self.END_POINT[1] - neighbor[1]) ** 2))

     def a_star(self, img):
          '''
          Funcao A* que busca o melhor caminho
          utilizando a soma das funcoes g e h para um conjuto
          de pontos e escolhendo o ponto que apresenta um custo mais baixo

          @param img e a matriz da imagem a testar com o valor dos pixeis
          '''
          self.img = img
          current = self.START_POINT
```

```
        self.closed_set = [self.START_POINT]
        while not (current[0] == self.END_POINT[0] and current[1] ==
self.END_POINT[1]):
            open_set = [(current[0] + x, current[1] + y) for x in xrange(-1,2)
for y in xrange(-1,2)
                        if (not (x == 0 and y == 0)) and ((current[0] + x,
current[1] + y) not in self.closed_set)] # procura dos vizinho, se estes ainda
nao estiverem na lista fechada
            if len(open_set) > 0:
                try:
                    temp = [self.g(neighbor) + self.h(neighbor) /
(self.ALPHA * 10) for neighbor in open_set]
                    current = open_set[temp.index(min(temp))]
                    self.closed_set.append(current)
                except Exception, e:
                    print 'Point out of image borders'
```

6.2 - main.py

```
import math
import cv2
import numpy as np
from astar import AStar

'''
@authors 12551 Pedro Santos & 12552 Emanuel Teixeira
@date 11 de Maio de 2014
@obs: Algoritmo A* Path Finding para a
procura de caminhos com menos custo
em funcao da intensidade.
'''

class OpenImage(object):
    """Classe onde a imagem vai ser
    aberta para estudo e tratamento"""
    def __init__(self, name):
        '''
        Inicializacao de parametros da classe

        @name nome do ficheiro da imagem a abrir
        '''
        super(OpenImage, self).__init__()
        self.name = name
        self.img = None
        self.width = 0
        self.height = 0

    def open_image(self, START_POINT, END_POINT, flag = True):
        '''
        Abertura e desenho na imagem dos pontos inicial e final
        para percepcao clara do caso de estudo

        @param START_POINT ponto incial
        @param END_POINT ponto final
        '''
        start = time.clock()
        self.img = cv2.imread(self.name)
        self.width, self.height = self.img.shape[:2]
        self.time = time.clock() - start
        if flag:
```

```
                cv2.line(self.img,(START_POINT[0], START_POINT[1]),
(START_POINT[0], START_POINT[1]),(0,0,255),8)
                cv2.line(self.img,(END_POINT[0], END_POINT[1]),(END_POINT[0],
END_POINT[1]),(0,0,255),8)

    def show_image(self):
        '''
        Abertura de uma caixa de dialogo com
        o resultado do algoritmo visivel
        '''
        cv2.imshow('A* - EDA 13/14', self.img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    def save_image(self, name):
        '''
        Guarda o conteudo da janela num ficheiro
        de imagem

        @param name nome do ficheiro de imagem a gravar
        '''
        cv2.imwrite(name, self.img)

    def draw_point(self, point):
        '''
        Funcao que desenha um ponto na imagem

        @param point ponto a desenhar na imagem
        '''
        cv2.line(self.img, point, point,(0,255,255), 2)

class WriteFile(object):
    '''
    Classe para a escrita de ficheiros
    '''
    def write_file(self, name, closed_set):
        '''
        Escrita de um ficheiro .txt para o armazenamento dos
        pontos resultantes do algoritmo

        @param name nome do ficheiro .txt onde serao armazenados os dados
        @closed_set lista de pontos resultantes do algoritmo
        '''
        data = open(name, 'w')
        for x in closed_set:
            data.writelines((str(x[0]), ' , ', str(x[1]), '\n'))
        data.close()


if __name__ == '__main__':
    open_image = OpenImage('peppersgrad.pgm')
    a_star = AStar()
    open_image.open_image(a_star.START_POINT, a_star.END_POINT)
    a_star.a_star(open_image.img)
    for point in a_star.closed_set:
        open_image.draw_point(point)
    open_image.show_image()
    open_image.save_image('result.png')
    write_file = WriteFile()
    write_file.write_file('data.txt', a_star.closed_set)
```