

# MC886 B/ MO416 A - Exercício 1

Ikaro J. U. Basso (174964), Luiz F. M. Pereira (203532), Matheus C. Lindino (203581)

31/03/2022

## Decisões de projeto

Quanto ao problema de expansão de vértices, um loop verifica todos os segmentos de reta possíveis, se algum deles cruza alguma aresta dos polígonos do problema. Isso é feito verificando as orientações dos pontos do segmento de reta atual com cada ponto de uma dada aresta. Caso as orientações sejam opostas, a reta cruza a aresta do polígono, o que a torna um passo inválido. Os casos de colinearidade são tratados a parte, já que o passo testado pode estar cruzando um polígono por algum de seus vértices. Quando o vértice colinear ao caminho testado for o único vértice desse tipo presente em um polígono, ele não invalida o possível passo. Quando houver dois vértices colineares ao possível passo num mesmo polígono, o caminho não é invalidado se eles forem adjacentes.

## A\* e IDA\*

O A\* é um algoritmo para busca de caminho, no qual dado um grafo, um ponto inicial e final, ele garante o melhor caminho entre eles, uma vez que este é uma combinação de aproximações heurísticas com o algoritmo Breadth First Search (BFS). Para o desenvolvimento do algoritmo, foi utilizado como heurística a distância euclidiana entre o ponto atual até o ponto final. Dessa forma, o algoritmo coloca os vértices gerados em uma fila de prioridade, baseado no valor de  $f(x)$ . Assim, o próximo nodo a ser visitado será aquele que possui o menor custo atual somado com o valor da heurística (aproximação da distância necessária para chegar no ponto final). Também criado uma lista de vértices visitados, para não entrar em loop o programa. O algoritmo é encerrado quando encontrado o vértice final, retornando o melhor caminho.

O IDA\* é uma mistura entre a busca por profundidade e o A. O IDA realiza uma busca por profundidade limitada no grafo. Essa limitação é baseada em um valor máximo de  $f(x)$ . Se não for encontrado uma solução com esse custo, ele atualiza o valor máximo com o custo do filho que é maior que o ele. Para o desenvolvimento do algoritmo, foi utilizado como heurística a distância euclidiana e iniciado o  $f(x)$  máximo (threshold) como a distância euclidiana entre o ponto inicial e o final. Após isso, é realizado a chamada da função recursiva search no qual realiza uma busca por profundidade. Caso não tenha encontrado o ponto final e tenha um nó com um custo maior que o threshold definido, o  $f(x)$  máximo passa a ser o custo desse nó, reiniciando a busca desde a raiz.

## Resultados

Ao que se refere a obtenção das menores distâncias, para a saída padrão obtivemos a seguinte ordem: BFS < A\* < IDA\* < IDS, Porém, vale destacar que o algoritmo BFS foi também o algoritmo responsável por gerar a maior quantidade de caminhos, seguido por IDA, A e por fim o IDS. Vale ressaltar que o IDS, além de gerar o caminho mais custoso, gera também o caminho que utiliza o maior número de vértices se comparado aos outros algoritmos, passando por 8 vértices (excluídos início e fim), contra 4 do IDA\* e 5 do IDA\*.

## Código fonte

classes.py

```
# Este módulo python contém as classes usadas no exercício.

import visibility

class LineSeg:

    def set_weight(self):
        self.weight = visibility.line_length(self)

    def __init__(self, v1, v2):
        self.v1 = v1
        self.v2 = v2

    def __eq__(self, other):
        return (self.v1 == other.v1) and (self.v2 == other.v2)

    def __lt__(self, other):
        return self.weight < other.weight

class Vertex:

    def __init__(self, name, x, y):
        self.name = name
        self.coord = Point(x, y)
        self.visible = []
        self.belongs_poly = None
        self.adjacent = []
        self.distance = 0
        self.visited = False
        self.parent = None

    def __eq__(self, other):
        if other.__class__ == Point:
            return self.coord.x == other.x and self.coord.y == other.y
        elif other.__class__ == Vertex:
            return self.name == other.name
        else:
            return False

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

    def __lt__(self, other):
        return self.distance < other.distance

    def print(self):
        out = f"{self.name}: x={self.coord.x}, y={self.coord.y};"
        if self.visible:
```

```

        out += f"\tVisible:{self.visible}"
    if self.belongs_poly:
        out += f"\tBelongs to:{self.belongs_poly.name}\n"
    return out

def __str__(self):
    out = f"{self.name}: x={self.coord.x}, y={self.coord.y};"
    if self.visible:
        out += f"\tVisible: "
        for v in self.visible:
            out += f"{v.name} "
    if self.belongs_poly:
        out += f"\tBelongs to:{self.belongs_poly.name}\n"
    return out

def belongs_to(self, polygon):
    self.belongs_poly = polygon

def set_visited(self, value):
    self.visited = value

def get_visited(self):
    return self.visited

def set_distance_and_visited(self, dest):
    self.visited = True
    self.distance = visibility.line_length(LineSeg(self, dest))

class Polygon:

    def __init__(self, name, vertices):
        self.name = name
        self.vertices = vertices
        self.concavity = False
        self.bay = []

    def __str__(self):
        out = f"{self.name}:"
        if self.vertices:
            for vert in self.vertices:
                out += "\t"
                out += str(vert)
            else:
                out += "Sem vertices definidos"
        out += "\n"
        return out

class Point:

    def __init__(self, x, y):
        self.x = float(x)

```

```

self.y = float(y)

def __eq__(self, other):
    return self.x == other.x and self.y == other.y

def print(self):
    print(f"x={self.x}, y={self.y};")

```

#### make\_dict.py

```

# Este programa faz um dicionário com vértices, polígonos, e
# pontos de chegada e de partida a partir de um arquivo texto.

import classes

def make_dict(filepath):

    dict_data = {
        "vertices": [],
        "polygons": [],
        "start_end_vertices": [],
        "paths": []
    }

    with open(filepath, "r") as raw_data:

        start = False
        line = raw_data.readline()
        while (line):
            line = line.split()
            # Vertices
            if (len(line) == 3):
                new_vert = classes.Vertex(name=line[0], x=line[1], y=line[2])
                dict_data["vertices"].append(new_vert)

            # Polygons
            elif (len(line) > 3):
                poly_name = line[0]
                vert_names_list = line[1:]
                poly_vert_list = []
                for vert_name in vert_names_list:
                    for vert in dict_data["vertices"]:
                        if (vert_name == vert.name):
                            poly_vert_list.append(vert)
                            continue
                new_poly = classes.Polygon(poly_name, poly_vert_list)
                for vert in poly_vert_list:
                    vert.belongs_poly = new_poly
                dict_data["polygons"].append(new_poly)

            # Start and end points
            elif (len(line) == 2):

```

```

        if not start:
            new_vert = classes.Vertex("S", line[0], line[1])
            start = True
        else:
            new_vert = classes.Vertex("G", line[0], line[1])
            dict_data["start_end_vertices"].append(new_vert)

    line = raw_data.readline()

return dict_data

```

### search\_algorithms.py

```

# Este módulo python contém implementações dos algoritmos de
# busca Best-First Search, Iterative Deepening Search, A*,
# e Iterative Deepening A*, aplicados ao problema de busca
# por um caminho de um ponto a outro do plano passando somente
# por vértices de polígonos sem que o caminho cruze qualquer
# aresta dos polígonos.

```

```
import copy
```

```
import classes
import visibility
```

```
# Algoritmo Best-First Search
```

```
def bfs(problem):
```

```
    i = 0
    path = []
```

```
    open_list = []
    closed_list = []
```

```
    root = problem["start_end_vertices"][0]
    final_dest = problem["start_end_vertices"][1]
```

```
    root = visibility.expand_vert(problem, root)
    open_list.append(root)
```

```
    while len(open_list) > 0:
        current = open_list.pop(0)
        closed_list.append(current)
```

```
        if current == final_dest:
            path = []
            while current != root:
                path.append(current)
                current = current.parent
            path.append(root)
            return path[::-1]

```

```

a = copy.copy(current)
problem["paths"].append([])
while a != None:
    problem["paths"][i].append(a)
    a = a.parent
    i += 1

for v in current.visible:
    children = copy.copy(v)

    children.parent = current
    children = visibility.expand_vert(problem, children)

    if children in closed_list:
        continue

    children.distance = current.distance + \
        visibility.line_length(classes.LineSeg(current, children))
    if children not in closed_list:
        open_list.append(children)

    open_list.sort(key=lambda v: v.distance)
return path

```

*# Algoritmo Iterative Deepening Search*

```

def ids(problem):

    def ids_search(node, target, current_depth, max_depth, path):
        if node == target:
            return node, True

        if current_depth == max_depth:
            if len(node.visible) > 0:
                return None, False
            else:
                return None, True

        bottom_reached = True
        for v in node.visible:
            children = copy.copy(v)
            children.parent = node
            children = visibility.expand_vert(problem, children)
            if children not in path:
                path.append(children)
                result, bottom_reached_search = ids_search(children, target,
                                                            current_depth + 1, max_depth,
                                                            path)

                if result is not None:
                    return result, True
                bottom_reached = bottom_reached and bottom_reached_search

        return None, bottom_reached

```

```

depth = 1
i = 0
bottom_reached = False
root = problem["start_end_vertices"][0]
root = visibility.expand_vert(problem, root)
final_dest = problem["start_end_vertices"][1]
while not bottom_reached:
    path = []
    path.append(root)
    result, bottom_reached = ids_search(root, final_dest, 0, depth, path)

    problem["paths"].append([])
    for p in path:
        problem["paths"][i].append(p)

    best_path = []
    a = copy.copy(path[-1])

    while a != None:
        best_path.append(a)
        a = a.parent

    if result is not None:
        # best_path[0], best_path[-1] = best_path[-1], best_path[0]
        return best_path[::-1]
    depth += 1
    i += 1
return None

# Algoritmo A* Search
def a_star(problem):

    def heuristic_1(children, final_dest):
        return visibility.line_length(classes.LineSeg(children, final_dest))

    i = 0
    path = []

    # Inicializa vetores de controle
    open_list = []
    closed_list = []

    # Inicializa vértice inicial e final
    root = problem["start_end_vertices"][0]
    final_dest = problem["start_end_vertices"][1]

    # Gera os filhos do vértice raiz
    root = visibility.expand_vert(problem, root)
    open_list.append(root)

    while len(open_list) > 0:

```

```

# Retira o vértice com o menor  $f(x) = g(x) + h(x)$  e adiciona nos visitados
current = open_list.pop(0)
closed_list.append(current)

# Caso o vértice atual é o final, encerra o programa retornando o caminho encontrado
if current == final_dest:
    path = []
    while current != root:
        path.append(current)
        current = current.parent
    path.append(root)
    return path[::-1]

a = copy.copy(current)
problem["paths"].append([])
while a != None:
    problem["paths"][i].append(a)
    a = a.parent
    i += 1

# Para cada filho gerado, calcula o  $f(x)$  e adiciona na fila de prioridade
for v in current.visible:
    children = copy.copy(v)
    children.parent = current
    children = visibility.expand_vert(problem, children)

    if children in closed_list:
        continue

    dist_to_children = visibility.line_length(
        classes.LineSeg(current, children))
    children.distance = current.distance + dist_to_children + heuristic_1(
        children, final_dest)

    if children not in closed_list:
        open_list.append(children)

# Ordena a fila baseado no  $f(x)$ 
open_list.sort(key=lambda v: v.distance)
return path

# Algoritmo Iterative Deepening A*
def ida_star(problem):

    def search(current, final_dest, previous_cost, threshold, path):

        # Caso o vértice atual for o final, para o programa
        if current == final_dest:
            return True

        # Calcula o  $f(x)$  do nodo atual
        cost = previous_cost + visibility.line_length(

```



```

        classes.LineSeg(current, final_dest))

# Caso o f(x) atual for maior que o f(x) máximo, retorna o f(x) atual
    if cost > threshold:
        return cost

    minimum = float('inf')

    # Para cada filho gerado, expande os nós e realiza a busca por profundidade baseado no f(x)
    for v in current.visible:
        children = copy.copy(v)
        children.parent = current

        children = visibility.expand_vert(problem, children)
        if children not in path:
            path.append(children)
            temp_cost = search(children, final_dest, cost, threshold, path)

            if temp_cost == True:
                return True

            if temp_cost < minimum:
                minimum = temp_cost

    return minimum

# Inicializa os vértices inicial e final
root = problem["start_end_vertices"][0]
final_dest = problem["start_end_vertices"][1]

# Gera os filhos da raiz
root = visibility.expand_vert(problem, root)

# Define o threshold inicial (distancia euclidiana entre o ponto inicial e final)
threshold = visibility.line_length(classes.LineSeg(root, final_dest))
counter = 0

while True:
    path = []
    path.append(root)

    # Busca em profundidade, limitada pelo threshold
    temp_cost = search(root, final_dest, 0, threshold, path)

    problem["paths"].append([])
    for p in path:
        problem["paths"][counter].append(p)

    best_path = []
    a = copy.copy(path[-1])
    while a != None:
        best_path.append(a)
        a = a.parent

```

```

if temp_cost == True:
    return best_path[::-1]
elif temp_cost == float('inf'):
    return None
else:
    threshold = temp_cost
    counter += 1

```

## visibility.py

```

# Este módulo python contém os algoritmos necessários para
# a expansão dos vértices, que consiste em descobrir quais
# outros vértices são visíveis para o vértice atual.

import numpy as np

import classes

# Registra no vértice atual (curr_vert) todos os outros
# vértices visíveis a partir dele.
def expand_vert(problem, curr_vert):

    visible = []
    curr_vert.visited = True

    for polygon in problem["polygons"]:

        # Checa o caso em que o polígono contém curr_vert.
        if (curr_vert.belongs_poly == polygon):
            visible_in_poly = get_visible_in_poly(curr_vert)
            for vis_vert in visible_in_poly:
                if (not vis_vert.visited):
                    visible.append(vis_vert)

        # Outros polígonos.
        else:
            for i in range(len(polygon.vertices) - 1):
                poly_vert = polygon.vertices[i]
                possible_line = classes.LineSeg(curr_vert, poly_vert)
                if (is_visible(possible_line, problem)):
                    if (not poly_vert.visited):
                        visible.append(poly_vert)

        # Checa o caso em que o destino é visível
        goal = problem["start_end_vertices"][1]
        possible_line = classes.LineSeg(curr_vert, goal)
        if (is_visible(possible_line, problem)):
            if (not goal.visited):
                visible.append(goal)

    curr_vert.visible = visible
    return curr_vert

```

```

# Retorna True se o segmento de reta não cruza nenhum polígono
# e False se há cruzamento.
def is_visible(possible_line, problem):

    for polygon in problem["polygons"]:
        if (do_cross_poly(possible_line, polygon)):
            return False

    return True

# Checa a concavidade dos polígonos e altera os atributos "bay"
# e "concavity" nos polígonos que possuem concavidade.
def make_concavity(problem):

    for polygon in problem["polygons"]:
        polygon = poly_concavity(polygon)

    return problem

# Como, na definição do problema, os vértices foram declarados
# no sentido anti-horário, quando a orientação de 3 pontos
# consecutivos for no sentido horário, eles fazem parte da baía.
# A função constrói o vetor "bay" e define o valor do parâmetro
# "concavity" como True no polígono de entrada, se ele possuir
# alguma concavidade.
def poly_concavity(polygon):

    no_vert = len(polygon.vertices) - 1
    if (no_vert > 3):
        for i in range(no_vert):
            v1 = polygon.vertices[i]
            v2 = polygon.vertices[(i + 1) % no_vert]
            v3 = polygon.vertices[(i + 2) % no_vert]
            p1 = v1.coord
            p2 = v2.coord
            p3 = v3.coord

            # Constrói a baía e modifica o parâmetro de
            # concavidade.
            if (orientation(p1, p2, p3) == -1):
                polygon.concavity = True
                for bay_vert in [v1, v2, v3]:
                    if (bay_vert not in polygon.bay):
                        polygon.bay.append(bay_vert)

    return polygon

# Calcula se os três pontos ordenados são orientados no sentido
# horário (negativo), anti-horário (positivo) ou são colineares

```

```

# (zero).
def orientation(p1, p2, p3):

    m = [[float(p1.x), float(p2.x), float(p3.x)],
          [float(p1.y), float(p2.y), float(p3.y)], [1.0, 1.0, 1.0]]
    det = np.linalg.det(m)

    # O determinante do numpy nem sempre retorna zero quando
    # há colinearidade.
    if (abs(det) < 0.000001):
        orientation = 0
    elif (det < 0):
        orientation = -1
    elif (det > 0):
        orientation = 1

    return orientation

# Retorna todos os vértices visíveis para "vert" que estão
# no mesmo polígono que ele.
def get_visible_in_poly(vert):

    visible = []
    polygon = vert.belongs_poly

    # Checa se o polígono que contém "vert" tem concavidade.
    # Se "vert" estiver na baía, todos pertencentes a ela são
    # visíveis, bem como seus adjacentes, e ainda, os
    # vértices adjacentes a seus visíveis, caso sejam
    # colineares à reta que os une.
    # Caso contrário, somente os adjacentes a "vert".
    if (polygon.concavity and vert in polygon.bay):
        for v in polygon.bay:
            visible.append(v)
        visible.remove(vert)

    for adj in get_adjacent(vert):
        if adj not in visible:
            visible.append(adj)

    for vis_vert in visible:
        for other_vis in get_adjacent(vis_vert):
            p1 = vert.coord
            p2 = vis_vert.coord
            p3 = other_vis.coord
            if (other_vis not in visible and other_vis != vert):
                if (orientation(p1, p2, p3) == 0):
                    visible.append(other_vis)

    else:
        visible = get_adjacent(vert)

```

```

return visible

def get_adjacent(vert):

    adjacent = vert.adjacent

    if not adjacent:
        polygon = vert.belongs_poly
        if vert.belongs_poly:
            no_vert = len(polygon.vertices) - 1
            for i in range(no_vert):
                if (vert == polygon.vertices[i]):
                    index_1 = (i - 1) % no_vert
                    index_2 = (i + 1) % no_vert
                    adjacent.append(polygon.vertices[index_1])
                    adjacent.append(polygon.vertices[index_2])
                    break

    return adjacent

def is_adjacent(v1, v2):
    if v1 in get_adjacent(v2):
        return True
    else:
        return False

# Entrada: dois segmentos de reta, que são formados por dois
# vértices cada.
# A função retorna True se os segmentos se cruzam, e False
# caso contrário.
# Se os segmentos se cruzam em um ponto apenas, considera-se
# que não se cruzam, já que os vértices se enxergam.
# Nos casos de colinearidade, também considera-se que não
# se cruzam, pelo mesmo motivo.
def do_cross_seg(line_seg_1, line_seg_2):

    p1 = line_seg_1.v1.coord
    p2 = line_seg_1.v2.coord

    p3 = line_seg_2.v1.coord
    orientation_1 = orientation(p1, p2, p3)

    p3 = line_seg_2.v2.coord
    orientation_2 = orientation(p1, p2, p3)

    if (orientation_1 == 0 and orientation_2 == 0):
        return [line_seg_2.v1, line_seg_2.v2]

    if (orientation_1 == 0):
        return line_seg_2.v1

```

```

if (orientation_2 == 0):
    return line_seg_2.v2

if (orientation_1 == orientation_2):
    return "No"

else:
    return "Yes"

# Checa se um segmento de reta cruza alguma das arestas
# de um polígono.
def do_cross_poly(line_seg, polygon):

    collinear = []

    for i in range(len(polygon.vertices) - 1):
        v1 = polygon.vertices[i]
        v2 = polygon.vertices[i + 1]

        if (line_seg.v1 in (v1, v2)):
            continue
        elif (line_seg.v2 in (v1, v2)):
            continue

        edge = classes.LineSeg(v1, v2)

        cross_seg_1 = do_cross_seg(line_seg, edge)
        cross_seg_2 = do_cross_seg(edge, line_seg)

        if (cross_seg_1 == cross_seg_2 and (cross_seg_1 == "Yes")):
            return True
        elif (cross_seg_1 == "No" or cross_seg_2 == "No"):
            continue
        elif (type(cross_seg_1) is list):
            for coll_vert in cross_seg_1:
                collinear.append(coll_vert)

    # Pelo menos um dos vértices da aresta em questão
    # está contido no segmento de reta.
    else:
        # Se for adjacente ao vértice de destino (que está
        # em um polígono diferente do vértice de partida),
        # então o vértice em cross_seg_1 é visível.
        # Se for adjacente ao vértice de partida, também.
        not_added = cross_seg_1 not in collinear
        start_adjacent = get_adjacent(line_seg.v1)
        dest_adjacent = get_adjacent(line_seg.v2)
        if (cross_seg_1 in dest_adjacent or cross_seg_1 in start_adjacent):
            continue
        elif (not_added):
            collinear.append(cross_seg_1)

```

```

# Um dicionário contendo como chaves cada um dos
# polígonos que contêm os vértices colineares ao
# segmento de reta atual, cujos valores são arrays
# contendo seus respectivos vértices que pertencem
# ao array "collinear".
aux_dic = {}

for vert in collinear:

    polygon = vert.belongs_poly
    if not aux_dic.keys():
        aux_dic[polygon.name] = [vert]

    elif polygon.name not in aux_dic.keys():
        aux_dic[polygon.name] = [vert]

    else:
        aux_dic[polygon.name].append(vert)

for poly_name in aux_dic.keys():
    for vert in aux_dic[poly_name]:

        # Este loop deve contemplar os casos nos quais
        # há vários vértices colineares ao segmento
        # atual em um certo polígono. Caso algum
        # desses vértices não for adjacente a nenhum
        # dos outros, e ele não estiver em uma baía,
        # então há cruzamento.
        do_continue = False
        for aux_v in get_adjacent(vert):
            if aux_v in aux_dic[poly_name]:
                do_continue = True

        if do_continue:
            continue

        # Se o comprimento de line_seg for maior que a
        # distância de seus dois vértices a algum dos
        # vértices colineares selecionados, então o
        # segmento cruza algum polígono.
        aux_seg_1 = classes.LineSeg(line_seg.v1, vert)
        aux_seg_2 = classes.LineSeg(line_seg.v2, vert)
        seg_length = line_length(line_seg)
        comp_1 = seg_length > line_length(aux_seg_1)
        comp_2 = seg_length > line_length(aux_seg_2)
        if (comp_1 and comp_2):
            return True

return False

# Retorna o tamanho de um segmento de reta.
def line_length(line_seg):

```

```

x_diff = line_seg.v1.coord.x - line_seg.v2.coord.x
y_diff = line_seg.v1.coord.y - line_seg.v2.coord.y

length = (x_diff**2 + y_diff**2)**0.5

return length

# Printa todos os vértices visíveis a partir de "vert"
def vert_visibility(vert):

    if vert.visible:
        print(f"Vertex {vert.name} sees: ", end="")
        for vert in vert.visible:
            print(f"{vert.name}", end=" ")
    else:
        print(f"Vertex {vert.name} not expanded.", end="")
    print("\n")

# Revela todas as concavidades do problema.
def print_bays(problem):

    for polygon in problem["polygons"]:
        if polygon.concavity:
            print(f"\nPolygon {polygon.name} has a bay.")
            print("Bay:", end="")
            for vert in polygon.bay:
                print(vert.name, end=" ")
            print("\n")

    return

def debug_vert_visibility(problem, vert):
    expand_vert(problem, vert)
    vert_visibility(vert)

```

## main.py

```

import sys

import classes
import make_dict
import search_algorithms as search
import visibility as vis

def calc_total_distance(solution):
    total_distance = 0
    for i in range(len(solution) - 1):
        total_distance += vis.line_length(
            classes.LineSeg(solution[i], solution[i + 1]))

```



```

return total_distance

def print_paths(paths):
    for path in paths:
        print("\t", end="")
        for vertex in path:
            if vertex != path[-1]:
                print(vertex.name, end=" ")
            else:
                print(f"{vertex.name}.")

def report(solution, paths, print_all):
    print(f"Path found: ", end="")
    for vertex in solution:
        if vertex.__class__ == classes.Vertex:
            if vertex != solution[-1]:
                print(vertex.name, end=" ")
            else:
                print(f"{vertex.name}.")
    if solution[-1].__class__ == classes.Vertex:
        print(f"Distance: {calc_total_distance(solution)}")
    if print_all:
        print("All paths:")
        print_paths(paths)
    print()

def pos_solution(problem):
    for i in range(2):
        problem["start_end_vertices"][i].visible = []
        problem["start_end_vertices"][i].distance = 0
        problem["start_end_vertices"][i].visited = False
        problem["start_end_vertices"][i].parent = None
        problem["paths"] = []
    for v in problem["vertices"]:
        v.visible = []
        v.distance = 0
        v.visited = False
        v.parent = None
    return problem

def main():
    # Constrói o dicionário do problema.
    if len(sys.argv) < 2:
        print("""
Usage: python main.py <filepath> [-a [all,bfs,ids,a_star,ida]] [-r]
-a: search algorithm to use. Default is all.
-r: print all paths.
""")

```

```

    exit(1)
else:
    filepath = sys.argv[1]
    algorithm = "all"
    print_all = False
    if len(sys.argv) > 2:
        if sys.argv[2] == "-a":
            algorithm = sys.argv[3]
        if sys.argv[2] == "-r" or sys.argv[4] == "-r":
            print_all = True
    problem = make_dict.make_dict(filepath)

    # Configura os polígonos não-convexos
    problem = vis.make_concavity(problem)

    # Algoritmos de busca

    if algorithm == "bfs":
        print("Best-First Search")
        bfs_path = search.bfs(problem)
        bfs_all_paths = problem["paths"]
        report(bfs_path, bfs_all_paths, print_all)
    elif algorithm == "ids":
        print("Iterative Deepening Search")
        ids_path = search.ids(problem)
        ids_all_paths = problem["paths"]
        report(ids_path, ids_all_paths, print_all)
    elif algorithm == "a_star":
        print("A*")
        a_star_path = search.a_star(problem)
        a_star_all_paths = problem["paths"]
        report(a_star_path, a_star_all_paths, print_all)
    elif algorithm == "ida":
        print("IDA*")
        ida_star_path = search.ida_star(problem)
        ida_star_all_paths = problem["paths"]
        report(ida_star_path, ida_star_all_paths, print_all)
    elif algorithm == "all":
        # BFS
        print("Best-First Search")
        bfs_path = search.bfs(problem)
        bfs_all_paths = problem["paths"]
        report(bfs_path, bfs_all_paths, print_all)
        problem = pos_solution(problem)
        # IDS
        print("Iterative Deepening Search")
        ids_path = search.ids(problem)
        ids_all_paths = problem["paths"]
        report(ids_path, ids_all_paths, print_all)
        problem = pos_solution(problem)
        # A*
        print("A*")
        a_star_path = search.a_star(problem)

```

```

a_star_all_paths = problem["paths"]
report(a_star_path, a_star_all_paths, print_all)
# IDA*
print("IDA*")
ida_star_path = search.ida_star(problem)
ida_star_all_paths = problem["paths"]
report(ida_star_path, ida_star_all_paths, print_all)

main()

```

## Saída padrão

Best-First Search

Path found: S d k o n v G.

Distance: 11.580967354152477

All paths:

```

S.
c S.
a S.
a c S.
f S.
f a S.
f c S.
f a c S.
g a S.
d S.
d c S.
g a c S.
g f S.
j d S.
g f a S.
j d c S.
e f S.
e f a S.
g f c S.
g f a c S.
b a S.
e f c S.
e f a c S.
b d S.
k d S.
b a c S.
b d c S.
k d c S.
k j d S.
b g a S.
k j d c S.
b j d S.
b j d c S.
b g a c S.
e g a S.
b f S.

```

b f a S.  
 b g f S.  
 b g f a S.  
 s d S.  
 e g a c S.  
 s k d S.  
 s d c S.  
 s k d c S.  
 s j d S.  
 s k j d S.  
 b f c S.  
 b f a c S.  
 h g a S.  
 s j d c S.  
 s k j d c S.  
 e g f S.  
 b g f c S.  
 b g f a c S.  
 e g f a S.  
 p k d S.  
 r s d S.  
 r k d S.  
 r s k d S.  
 p k d c S.  
 p k j d S.  
 r s d c S.  
 h g a c S.  
 r k d c S.  
 r s k d c S.  
 r j d S.  
 r k j d S.  
 r s j d S.  
 r s k j d S.  
 p k j d c S.  
 r j d c S.  
 r k j d c S.  
 r s j d c S.  
 r s k j d c S.  
 q k d S.  
 h g f S.  
 q k d c S.  
 q p k d S.  
 q k j d S.  
 h g f a S.  
 p s d S.  
 p s k d S.  
 q p k d c S.  
 q k j d c S.  
 q p k j d S.  
 p s d c S.  
 q s d S.  
 p s k d c S.  
 q s k d S.  
 p s j d S.

p s k j d S.  
 q p k j d c S.  
 q s d c S.  
 i h g a S.  
 q s k d c S.  
 p s j d c S.  
 p s k j d c S.  
 o k d S.  
 q s j d S.  
 q s k j d S.  
 h e f S.  
 o k d c S.  
 q s j d c S.  
 q s k j d c S.  
 h g f c S.  
 h g f a c S.  
 o k j d S.  
 h e f a S.  
 o p k d S.  
 o k j d c S.  
 q r s d S.  
 l S.  
 q r k d S.  
 q r s k d S.  
 o p k d c S.  
 r p k d S.  
 o p k j d S.  
 q r s d c S.  
 i h g a c S.  
 q r k d c S.  
 q r s k d c S.  
 q r j d S.  
 q r k j d S.  
 q r s j d S.  
 q r s k j d S.  
 o p k j d c S.  
 h b a S.  
 q r j d c S.  
 q r k j d c S.  
 q r s j d c S.  
 q r s k j d c S.  
 h e f c S.  
 h e f a c S.  
 i h g f S.  
 h d S.  
 i h g f a S.  
 h b d S.  
 o s d S.  
 o p s d S.  
 o s k d S.  
 o p s k d S.  
 h d c S.  
 h b a c S.  
 h b d c S.

o s d c S.  
 o p s d c S.  
 o s k d c S.  
 o p s k d c S.  
 h j d S.  
 o s j d S.  
 o s k j d S.  
 o p s j d S.  
 o p s k j d S.  
 l c S.  
 b e f S.  
 h b g a S.  
 h j d c S.  
 o s j d c S.  
 o s k j d c S.  
 o p s j d c S.  
 o p s k j d c S.  
 n o k d S.  
 b e f a S.  
 h b j d S.  
 i e f S.  
 n o k d c S.  
 n o k j d S.  
 i e f a S.  
 h b j d c S.  
 n o p k d S.  
 n o k j d c S.  
 m l S.  
 n o p k d c S.  
 n o p k j d S.  
 h b g a c S.  
 h e g a S.  
 n o p k j d c S.  
 h b f S.  
 i e f c S.  
 i e f a c S.  
 h b f a S.  
 h b g f S.  
 n o s d S.  
 n o p s d S.  
 h b g f a S.  
 h e g a c S.  
 i k d S.  
 n o s k d S.  
 n o p s k d S.  
 n o s d c S.  
 n o p s d c S.  
 i k d c S.  
 n o s k d c S.  
 n o p s k d c S.  
 i k j d S.  
 n o s j d S.  
 n o s k j d S.  
 n o p s j d S.

n o p s k j d S.  
 m l c S.  
 h b f c S.  
 h b f a c S.  
 t h g a S.  
 t i h g a S.  
 i k j d c S.  
 n o s j d c S.  
 n o s k j d c S.  
 n o p s j d c S.  
 n o p s k j d c S.  
 v n o k d S.  
 o i h g a S.  
 v n o k d c S.  
 l d S.  
 t o k d S.  
 v n o k j d S.  
 v n o p k d S.  
 l d c S.  
 t o k d c S.  
 v n o k j d c S.  
 t h g a c S.  
 t i h g a c S.  
 i p k d S.  
 t o k j d S.  
 i e g a S.  
 v n o p k d c S.  
 t o p k d S.  
 v n o p k j d S.  
 i p k d c S.  
 t o k j d c S.  
 i p k j d S.  
 t o p k d c S.  
 v n o p k j d c S.  
 t o p k j d S.  
 i p k j d c S.  
 t o p k j d c S.  
 t h g f S.  
 t i h g f S.  
 t h g f a S.  
 t i h g f a S.  
 i e g a c S.  
 n i h g a S.  
 v n o s d S.  
 v n o p s d S.  
 i s d S.  
 v n o s k d S.  
 v n o p s k d S.  
 i s k d S.  
 i p s d S.  
 i p s k d S.  
 v n o s d c S.  
 v n o p s d c S.  
 i s d c S.

u h g a S.  
 u i h g a S.  
 u t h g a S.  
 u t i h g a S.  
 t o s d S.  
 t o p s d S.  
 v n o s k d c S.  
 v n o p s k d c S.  
 i s k d c S.  
 i p s d c S.  
 t o s k d S.  
 t o p s k d S.  
 t n o k d S.  
 v n o s j d S.  
 v n o s k j d S.  
 v n o p s j d S.  
 v n o p s k j d S.  
 i s j d S.  
 i s k j d S.  
 i p s k d c S.  
 i p s j d S.  
 i p s k j d S.  
 t e f S.  
 i e g f S.  
 t h e f S.  
 t i e f S.  
 t o s d c S.  
 t o p s d c S.  
 t o s k d c S.  
 t o p s k d c S.  
 t n o k d c S.  
 v n o s j d c S.  
 v n o s k j d c S.  
 v n o p s j d c S.  
 v n o p s k j d c S.  
 t h g f c S.  
 t h g f a c S.  
 i s j d c S.  
 i s k j d c S.  
 t o s j d S.  
 t o s k j d S.  
 t o p s j d S.  
 t o p s k j d S.  
 t n o k j d S.  
 t e f a S.  
 i e g f a S.  
 t h e f a S.  
 t i e f a S.  
 n i h g a c S.  
 t n o p k d S.  
 l j d S.  
 t o s j d c S.  
 t o s k j d c S.  
 t o p s j d c S.



t o p s k j d c S.  
t n o k j d c S.

#### Iterative Deepening Search

Path found: S c a b d h e i u G.

Distance: 24.145726557818605

All paths:

S c d a f l.  
S c a d f l.  
S c a b f g d h j k l s.  
S c a b d e f g h j l.  
S c a b d h j k l s e g f i t u.  
S c a b d h e g i j t u k l o p q r s f.  
S c a b d h e g f i t u j k l r s.  
S c a b d h e g f i k l n o p s t u j r q.  
S c a b d h e g f i k j l o p q r s n m t v u G.

#### A\*

Path found: S d s o n v G.

Distance: 12.313270303364892

All paths:

S.  
c S.  
d S.  
a S.  
f S.  
j d S.  
k d S.  
s d S.  
b d S.  
l S.  
q s d S.  
r s d S.  
d c S.  
p s d S.  
b a S.  
q k d S.  
p k d S.  
g a S.  
h d S.  
s k d S.  
r k d S.  
o s d S.  
a c S.  
o k d S.  
g f S.  
f a S.  
b f S.  
k j d S.  
s j d S.  
r j d S.  
f c S.  
b j d S.  
e f S.

p q s d S.  
 n o s d S.  
 n o k d S.  
 o p s d S.  
 p r s d S.  
 r q s d S.  
 o p k d S.  
 i k d S.  
 o s k d S.  
 i s d S.  
 i h d S.  
 v n o s d S.  
 h b d S.  
 h j d S.  
 v n o k d S.  
 t o s d S.  
 t o k d S.  
 g b d S.  
 t h d S.  
 l d S.

IDA\*

Path found: S c d h u G.

Distance: 15.145726557818604

All paths:

S S c d a f l.  
 c S S c d b h j k l s a f.  
 d S S c d b h j k l s i p q r o a f.  
 a S S c d b h j k l s i p q r o a f e g.  
 f S S c d b h j k l s i p q r o a f g.  
 j d S S c a d f l.  
 k d S S c a d b h j k l s f.  
 s d S S c a d b h j k l s i p q r o f.  
 b d S S c a d b h j k l s i p q r o f e g.  
 l S S c a b f g d h j k l s i p q r o.  
 q s.  
 d S S c a b f g d h j k i l o p q r s.  
 r s.  
 d S S c a b f g d h e i j t u k l o p q r s.  
 d c S S c a b f g d h e i j t u k l o p q r s.  
 p s.  
 d S S c a b f g d h e i j t u k l o n p q r s.  
 b a S S c a b f g d h e i j t u m v G.  
 q k d S.  
 p k d S.  
 g a S.  
 h d S.  
 s k d S.  
 r k d S.  
 o s d S.  
 a c S.  
 o k d S.  
 g f S.  
 f a S.

b f S.  
k j d S.  
s j d S.  
r j d S.  
f c S.  
b j d S.  
e f S.  
p q s d S.  
n o s d S.  
n o k d S.  
o p s d S.  
p r s d S.  
r q s d S.  
o p k d S.  
i k d S.  
o s k d S.  
i s d S.  
i h d S.  
v n o s d S.  
h b d S.  
h j d S.  
v n o k d S.  
t o s d S.  
t o k d S.  
g b d S.  
t h d S.  
l d S.