Unicamp - Instituto de Computação

MC886 B / MO416 A

Introdução à Inteligência Artificial

Prof. Dr. Jacques Wainer

Tarefa 1

Integrantes do Grupo:

David de Melo Almeida dos Reis - 203757
George Gigilas Junior - 216741
Ítalo Fernandes Gonçalves - 234990

Construção do Grafo

Para trabalhar com polígonos convexos e não convexos, o grupo desenvolveu um algoritmo para tratar os casos como os da figura 1a e 1b. Na figura 1a, no ponto P só devem ser visíveis os vértices: B, A, H, F, E e D enquanto que na figura 1b, a partir do vértice A só é possível visualizar o vértice B e o vértice D.

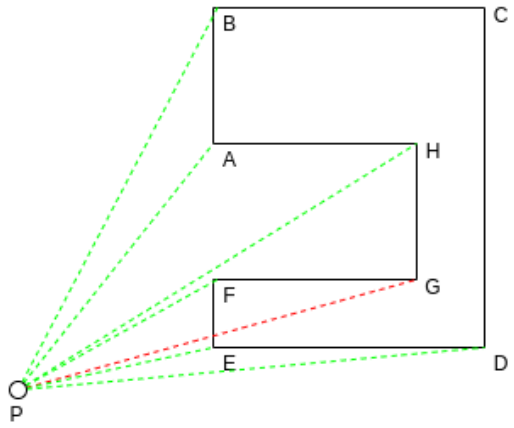

Figura 1a

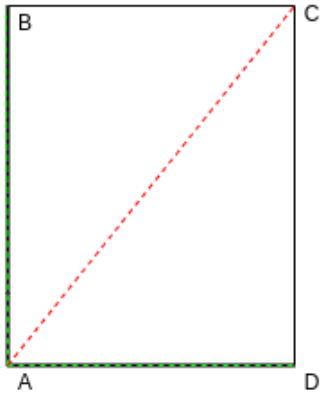

Figura 1b

O algoritmo segue 2 etapas:

- Garantir que dado um segmento de reta, de um ponto a outro, para que seja visível, não podem haver outros segmentos de retas (as arestas dos polígonos) - caso 1a.
- Garantir que esse segmento de reta, caso não haja intersecções com arestas, que ele não corta algum polígono internamente - caso 1b.

Dessa forma foi implementada a função de intersecção de segmento de retas para a primeira etapa do algoritmo e para segunda, caso seja válido para a primeira, foi escolhido um ponto pertencente ao segmento de reta sendo avaliado - escolhido nesse caso, o ponto médio, e verificado se este pertence ao interior do polígono. Como foi realizado e verificado, o algoritmo rodou com bastante eficiência e corretude.

Execução dos algoritmos

Os algoritmos foram implementados e a saída obtida pode ser observada logo abaixo:

Best First Search:
visitations: 23
cost: 17.664583015341176
path: ['initial', 'l', 'm', 'target']

ID:
visitations: 61
cost: 17.664583015341176
path: ['initial', 'l', 'm', 'target']

A*:
visitations: 17
cost: 11.580967354152477
path: ['initial', 'd', 'k', 'o', 'n', 'v', 'target']

IDA*:
visitations: 701
cost: 11.580967354152477
path: ['initial', 'd', 'k', 'o', 'n', 'v', 'target']

Sendo *initial* nosso ponto de partida e *target* nosso objetivo.

Discussões sobre os algoritmos

Best First Search:

Comparando com os outros algoritmos, o grupo percebeu que o Best First Search não encontrou o melhor caminho da origem até o destino e nem foi o que visitou o menor número de vértices.

Quanto ao caminho, esse algoritmo possui um problema bem claro: a partir do momento em que um nó é vizinho do destino, o algoritmo determina que esse é o caminho e para.

Para entender o porquê disso ser um problema, considere o seguinte exemplo: o algoritmo percorre 9 unidades de distância para chegar até o nó X e 10 unidades de distância para chegar ao nó Y. Supondo que X e Y são vizinhos do nó de destino, seja 5 a distância entre X e o destino e 1 a distância de Y até o destino. Ao executar o programa, o algoritmo visita X antes de visitar Y, pois a distância percorrida até ele é menor. Quando são analisados os nós vizinhos de X, o algoritmo encontra o destino e termina a execução, determinando que esse é o menor caminho encontrado entre o início e o destino (distância 14). Porém, se o algoritmo visitasse o nó Y, ele encontraria distância 11, que seria uma distância menor. Portanto, isso faz com que o algoritmo não tenha encontrado o caminho ótimo.

Já quanto ao número de vértices visitados, por se tratar de um algoritmo de busca em largura, ele visita todos os nós que possuem distância percorrida (em relação à origem) menor do que o nó que antecede o nó de destino no caminho encontrado. Isso faz com que ele visite boa parte dos nós disponíveis.

A*:

Logo que o algoritmo A* foi executado, o grupo percebeu que o caminho encontrado foi o caminho ótimo, e que o número de vértices visitados foi menor.

Apesar de ser muito parecido com o Best First Search, a mudança na heurística faz total diferença, e faz com que o programa seja “mais inteligente”, escolhendo os vértices mais interessantes e promissores. Essa estratégia é

responsável pelo menor número de vértices visitados, já que ele evita nós desnecessários.

Seguindo o exemplo dado para o Best First Search, a nova heurística para o nó X seria de 9 + 5 = 14 (distância percorrida da origem até o X, somada à distância percorrida de X até o destino) e para o nó Y seria de 11. Dessa forma, o algoritmo iria visitar o nó Y antes do nó X, obtendo um caminho mais curto.

ID:

Pela sua natureza de repassar por vértices várias vezes, (conforme o valor de corte vai ajustando), o algoritmo ID tem uma alta taxa de visitas, em contrapartida de um baixo uso de memória. O caminho encontrado por um problema em que o peso das arestas não é igual a 1 não é ótimo, pois o algoritmo apenas reduz a profundidade caminhada no grafo, porém não o custo.

IDA*:

Da mesma forma que o algoritmo ID, esse algoritmo é caracterizado por passar pelos mesmos vértices várias vezes, por não fazer o uso de programação dinâmica, o que reduz o uso de memória. Mas diferentemente do ID, por usar uma heurística apropriada, o algoritmo encontra um caminho ótimo.

Código desenvolvido

O código desenvolvido pelo grupo segue em anexo e pode ser encontrado no link: https://github.com/fernandesitalo/artificial-intelligence-unicamp

```python
from read_and_process_input import *
from geometric_functions import *
from IdaStar import IdaStar
from IterativeDeepening import ID
from BestFirstSearch import BestFirstSearch
from AStar import AStar

def main():
    archive = "input.txt"
    obj_input = read_and_process_input(archive)
    print(obj_input.graph)
    map_letter_point = obj_input.get_map_letter_point()

    bestFS = BestFirstSearch(obj_input.graph, map_letter_point)
    visitations, distance, path = bestFS.run()
    print("Best First Search")
    print("visitations:", visitations)
    print("cost:", distance)
    print("path:", path)
    print("")

    id = ID(obj_input.graph, map_letter_point)
    visitations, distance, path = id.run()
    print("IterativeDeepening")
    print("visitations:", visitations)
    print("cost:", distance)
    print("path:", path)
    print("")

    aStar = AStar(obj_input.graph, map_letter_point)
    visitations, distance, path = aStar.run()
    print("A*")
    print("visitations:", visitations)
    print("cost:", distance)
    print("path:", path)
    print("")

    idaStar = IdaStar(obj_input.graph, map_letter_point)
    visitations, distance, path = idaStar.run()
    print("IDA*")
    print("visitations:", visitations)
    print("cost:", distance)
    print("path:", path)
    print("")


if __name__ == '__main__':
    main()
```

```python
from geometric_functions import *

class input_problem:
    polygons = None
    map_letter_point = None
    graph = None

    def __init__(self, polygons, map_letter_point):
        self.map_letter_point = map_letter_point
        self.polygons = polygons
        self.graph = {}
        self.make_graph()

    def make_graph(self):
        letters = list(self.map_letter_point.keys())
        n = len(letters)

        for letter in letters:
            self.graph[letter] = []
        for i in range(n):
            for j in range(i+1, n):
                point_i = self.map_letter_point.get(letters[i])
                point_j = self.map_letter_point.get(letters[j])
                if self.check_connectivity(point_i, point_j):
                    self.graph[letters[i]].append(letters[j])
                    self.graph[letters[j]].append(letters[i])


    def check_connectivity(self, point_a, point_b):
        is_intersect = False
        for a_polygon in self.polygons:
            if segment_intersect_polygon(point_a, point_b, a_polygon, self.map_letter_point):
                is_intersect = True
                break
        if is_intersect == True:
            return False
        midpoint_ab = point((point_a.x+point_b.x)/2.0,(point_a.y+point_b.y)/2.0)
        for a_polygon in self.polygons:
            if point_inside_polygon(midpoint_ab, a_polygon, self.map_letter_point) > 0:
                return False
        return True

    def get_map_letter_point(self):
        return self.map_letter_point

def read_and_process_input(archive_name):
    f = open(archive_name,'r')

    amount_of_points = int(f.readline())
    map_letter_point = {}
    for _ in range(amount_of_points):
        letter, point_x, point_y = map(str,f.readline().split())
        map_letter_point[letter] = point(float(point_x), float(point_y))

    polygons = []

    amount_of_polygons = int(f.readline())
    for _ in range(amount_of_polygons):
        input_polygon = []
        input_polygon = f.readline().split()
        number = input_polygon.pop(0)
        polygons.append(polygon(int(number), input_polygon))

    x_initial, y_initial = map(float,f.readline().split())
    x_target, y_target = map(float,f.readline().split())

    initial = point(x_initial, y_initial) # this vertex I will call of "initial"
    target = point(x_target, y_target) # this vertex I will call of "target"

    map_letter_point["initial"] =  initial
    map_letter_point["target"] = target

    return input_problem(polygons, map_letter_point)
```

```
class point:
    x = None
    y = None

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

    def is_equal(self, other_point):
        return self.x == other_point.x and self.y == other_point.y

class polygon:
    points = None
    number = None
    def __init__(self, number, points):
        self.number = number
        self.points = points


################################################################################
# geometric functions


'''
    dado que os pontos (p), (q) e (r) estão alinhados, queremos descobrir se o ponto (r) esta entre (p) e (q)
'''
def isOnSegment(p,q,r):
    if q.x <= max(p.x, r.x) and q.x >= min(p.x, r.x) and q.y <= max(p.y, r.y) and q.y >= min(p.y, r.y):
        return True
    return False

'''
    qual a orientação do (ptc) em relação a (pta) e (ptb)
    0 estão alinhados
    1 esquerda
    -1 direita
'''
def orientation(pta, ptb, ptc):
    det = +(ptb.x*ptc.y) +(pta.y*ptc.x) +(pta.x*ptb.y) -(pta.y*ptb.x) -(pta.x*ptc.y) -(ptc.x*ptb.y)
    if det == 0:
        return 0
    if det > 0:
        return 1
    return -1

'''
    se o segmento (p1)-(q1) intersecta o segmento (p2)-(q2)
'''
def segments_intersect(p1, q1, p2, q2):
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)
    if o1 != o2 and o3 != o4:
        return True
    if o1 == 0 and isOnSegment(p1, p2, q1):
        return True
    if o2 == 0 and isOnSegment(p1, q2, q1):
        return True
    if o3 == 0 and isOnSegment(p2, p1, q2):
        return True
    if o1 == 0 and isOnSegment(p2, q1, q2):
        return True
    return False


'''
    sentido: horario ou anti-horario!
    retorno:
        0 se o ponto esta na borda
        1 se o ponto esta dentro
        -1 se o ponto esta fora
'''
def point_inside_polygon(p, polygon, map_letter_point):
    n = len(polygon.points)
    windingNumber = 0

    for i in range(n):
        point_i = map_letter_point.get(polygon.points[i])
        if p.is_equal(point_i):
            # print("DEBUG 1")
            return 0

        point_j = map_letter_point.get(polygon.points[(i+1)%n]) # next point!!!!!!

        if point_i.y == p.y and point_j.y == p.y:
            if min(point_i.x, point_j.x) <= p.x and p.x <= max(point_i.x, point_j.x):
                return 0
        else:
            below = point_i.y < p.y
```

```
            if below != (point_j.y < p.y):
                ori = orientation(point_i,point_j,p)
                if ori == 0:
                    return 0
                if below == (ori > 0):
                    windingNumber += (1 if below else -1)

    return 1 if windingNumber%2 != 0 else -1


'''
    Dado um polygono -> lista de vertices, queremos saber se alguma das arestas intersecta o segmento (p)-(q)
    importante: não podemos considera intersecção entre vertices!
'''
def segment_intersect_polygon(p, q, polygon, map_letter_point):
    n = len(polygon.points)
    for i in range(n-1):
        point_i = map_letter_point.get(polygon.points[i])
        point_j = map_letter_point.get(polygon.points[(i+1)%n])
        if segments_intersect(p, q, point_i, point_j):
            if not p.is_equal(point_i) and not p.is_equal(point_j) and not q.is_equal(point_i) and not q.is_equal(point_j):
                return True
    return False
```

```
class ID:
    def __init__(self, graph, map_letter_point):
        self.graph = graph
        self.map_letter_point = map_letter_point

    def __RevertPath__(self, path):
        list = []
        actualPosition = 'target'
        while actualPosition != 'initial':
            list.append(actualPosition)
            actualPosition = path[actualPosition]
        list.append("initial")
        list.reverse()

        return list

    def __EuclidianDistance__(self, src, dst):
        x1 = self.map_letter_point[src].x
        y1 = self.map_letter_point[src].y
        x2 = self.map_letter_point[dst].x
        y2 = self.map_letter_point[dst].y

        return ((x2-x1)**2 + (y2-y1)**2)**0.5

    def run(self):

        visitations = 0
        maxDepth = 0
        path = {}
        dst = {}
        findSolution = False
        dst['initial'] = 0

        while not findSolution:

            maxDepth += 1
            stack = [('initial', 0)]
            while len(stack) > 0:
                topOfStack = stack.pop()
                depth = topOfStack[1]
                actualPosition = topOfStack[0]

                if depth > maxDepth:
                    continue
                visitations += 1
                if actualPosition == 'target':
                    findSolution = True
                    break

                for child in self.graph[actualPosition]:
                    stack.append((child, depth + 1))

                    if child not in path:
                        path[child] = actualPosition
                        dst[child] = dst[actualPosition] + self.__EuclidianDistance__(actualPosition, child)

        path = self.__RevertPath__(path)
        return (visitations, dst['target'], path)
```

```
class IdaStar:
    def __init__(self, graph, map_letter_point):
        self.graph = graph
        self.map_letter_point = map_letter_point

    def __RevertPath__(self, path):
        list = []
        actualPosition = 'target'
        while actualPosition != 'initial':
            list.append(actualPosition)
            actualPosition = path[actualPosition]
        list.append("initial")
        list.reverse()

        return list

    def __EuclidianDistance__(self, src, dst):
        x1 = self.map_letter_point[src].x
        y1 = self.map_letter_point[src].y
        x2 = self.map_letter_point[dst].x
        y2 = self.map_letter_point[dst].y

        return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

    def run(self):

        visitations = 0
        path = {}
        dst = {}
        findSolution = False
        threshold = self.__EuclidianDistance__('initial', 'target')
        dst['initial'] = 0

        while not findSolution:

            stack = [('initial', 0)]
            thresholdList = []
            while len(stack) > 0:
                topOfStack = stack.pop()
                actualPosition = topOfStack[0]
                Gx = topOfStack[1]
                Hx = self.__EuclidianDistance__(actualPosition, 'target')
                Fx = Gx + Hx

                if Fx > threshold:
                    thresholdList.append(Fx)
                    continue

                visitations += 1

                if actualPosition == 'target':
                    findSolution = True
                    break

                for child in self.graph[actualPosition]:
                    stack.append((child, Fx))

                    if child not in path:
                        path[child] = actualPosition
                        dst[child] = dst[actualPosition] + self.__EuclidianDistance__(actualPosition, child)

            threshold = min(thresholdList)

        path = self.__RevertPath__(path)
        return (visitations, dst['target'], path)
```

```python
from queue import PriorityQueue
from math import sqrt

class BestFirstSearch:

    def __init__(self, graph, map_letter_point):
        self.graph = graph
        self.map_letter_point = map_letter_point
        self.g = {}

    def initializePoint(self, source, destination):
        x = self.map_letter_point[source].x
        y = self.map_letter_point[source].y

        xi = self.map_letter_point[destination].x
        yi = self.map_letter_point[destination].y

        self.g[destination] = self.g[source] + sqrt((x - xi)**2 + (y - yi)**2)

    def revertPath(self, path):
        list = []
        currentPosition = 'target'

        while currentPosition != 'initial':
            list.append(currentPosition)
            currentPosition = path[currentPosition]

        list.append("initial")
        list.reverse()

        return list

    def run(self):
        priority_queue = PriorityQueue()
        priority_queue.put((0, "initial"))
        visited = ["initial"]
        self.g["initial"] = 0
        visitations = 0
        path = {}
        flag = False

        while (not priority_queue.empty()):
            visitations += 1
            point = priority_queue.get()[1]
            if point == "target":
                flag = True
                break

            else:
                for key in self.graph[point]:
                    if key not in visited:
                        visited.append(key)
                        path[key] = point
                        self.initializePoint(point, key)
                        priority_queue.put((self.g[key], key))

        if (flag):

            path = self.revertPath(path)
            distance = self.g["target"]
            return (visitations, distance, path)
```

```python
from queue import PriorityQueue
from math import sqrt

class AStar:

    def __init__(self, graph, map_letter_point):
        self.graph = graph
        self.map_letter_point = map_letter_point
        self.f = {}
        self.g = {}

    def initializePoints(self, source, destination):

        x = self.map_letter_point[destination].x
        y = self.map_letter_point[destination].y

        xi = self.map_letter_point[source].x
        yi = self.map_letter_point[source].y

        xf = self.map_letter_point["target"].x
        yf = self.map_letter_point["target"].y

        if (destination != "initial"):
            self.g[destination] = self.g[source] + sqrt((x - xi)**2 + (y - yi)**2)
        else:
            self.g[destination] = 0

        self.f[destination] = self.g[destination] + sqrt((x - xf)**2 + (y - yf)**2)

    def revertPath(self, path):
        list = []
        distance = 0
        currentPosition = 'target'

        while currentPosition != 'initial':
            list.append(currentPosition)

            x1 = self.map_letter_point[currentPosition].x
            y1 = self.map_letter_point[currentPosition].y
            x2 = self.map_letter_point[path[currentPosition]].x
            y2 = self.map_letter_point[path[currentPosition]].y

            distance += sqrt((x1 - x2)**2 + (y1 - y2)**2)
            currentPosition = path[currentPosition]

        list.append("initial")
        list.reverse()

        return (list, distance)

    def run(self):
        priority_queue = PriorityQueue()
        priority_queue.put((0, "initial"))
        visited = ["initial"]
        self.initializePoints("initial", "initial")
        visitations = 0
        path = {}
        flag = False

        while (not priority_queue.empty()):
            visitations += 1
            point = priority_queue.get()[1]

            if point == "target":
                flag = True
                break

            else:
                for key in self.graph[point]:
                    if key not in visited:
                        visited.append(key)
                        path[key] = point
                        self.initializePoints(point, key)
                        priority_queue.put((self.f[key], key))

        if (flag):
            path, distance = self.revertPath(path)
            return (visitations, distance, path)
```

```
22
a 1 2
b 4.5 2
c 1 0.5
d 4.5 0.5
e 2 5
f 1 3
g 3 3
h 5 5
i  6 5
k 6 1
j 5 1
l 7 -5
m 8 -5
n 8 3
o 7 3
p 7 2
q 7.5 2
r 7.5 1
s 7 1
t 8 5
u 9 5
v 8.5 2
5
1 c d b a c
2 e f g e
3 h j k i h
4 l m n o p q r s l
5 t v u t
0 0
9 3
```