```
import random
import math
import itertools
import heapdict
import heapq
from collections import deque
import shapely.geometry as sg
import shapely.ops as so
import matplotlib.pyplot as plt

class Polygon:
    def __init__(self, vertices):
        self.vertices = vertices

    def check_inside(self, x, y):
        equation_values = []
        direction=0
        for i in range(len(self.vertices)):
            if(i == len(self.vertices)-1):  # line connecting last and first verte
                x1, y1 = self.vertices[len(
                    self.vertices)-1].x, self.vertices[len(self.vertices)-1].y
                x2, y2 = self.vertices[0].x, self.vertices[0].y
            else:
                x1, y1 = self.vertices[i].x, self.vertices[i].y
                x2, y2 = self.vertices[i+1].x, self.vertices[i+1].y
            equation_values.append((-(y2-y1)*x) + ((x2 - x1)*y) + (-(-(y2-y1)*x1 +

        left=True
        right=True
        for x in equation_values:   #right
            if x>0:
                continue
            else:
                right=False
                break
        for x in equation_values:   #left
            if x<0:
                continue
            else:
                left=False
                break
        return left or right

class Vertex:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.parent = None
        self.g = 0  # g(n) value

    def __eq__(self, v2):
        if (self.x == v2.x and self.y == v2.y):
            return True
        return False
```

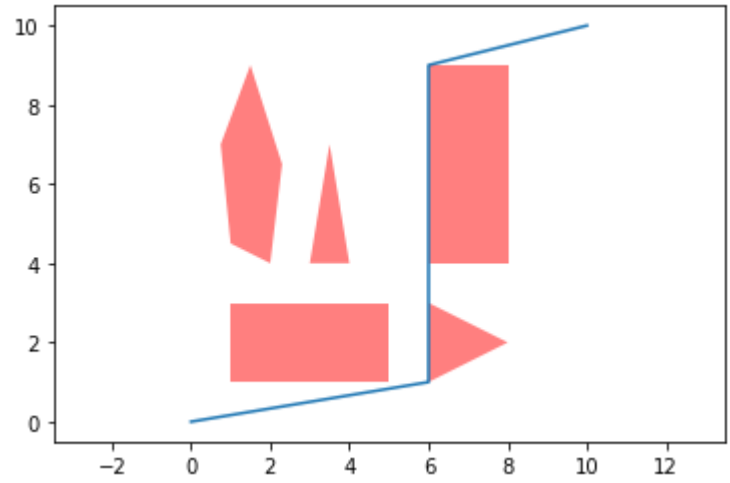
```
            return False

        def dist(self, v2):
            a = abs(self.x - v2.x)
            b = abs(self.y - v2.y)
            # root of a square plus b square
            return math.sqrt(math.pow(a, 2) + math.pow(b, 2))

        def __str__(self):
            return str(self.x) + str(self.y)

        def __hash__(self):
            return hash(str(self))


    class Edge:
        def __init__(self, v1, v2):  # stores 2 ends of lines
            self.x1 = v1.x
            self.y1 = v1.y
            self.x2 = v2.x
            self.y2 = v2.y

        def get_y(self, x):
            m = (self.y2 - self.y1) / (self.x2 - self.x1)
            y = m * (x-self.x1) + self.y1  # y=mx+c
            return y

        def check_inside(self, polygon):  # go point by point on line and check if ins
            if (self.x1 != self.x2):  # not vertical line
                for x in gen_spaces(min(self.x1, self.x2), max(self.x2, self.x1) , 0.0
                    y = self.get_y(x)
                    if (polygon.check_inside(x, y)):
                        return True
                return False
            else:  # vertical line
                for y in gen_spaces(min(self.y1, self.y2), max(self.y1, self.y2), 0.00
                    if (polygon.check_inside(self.x1, y)):
                        return True
                return False

    def gen_spaces(start, end, step):  # to generate steps as decimals
        if step !=0:
            no_spaces = int(abs(start-end) / step)
            return itertools.islice(itertools.count(start, step), no_spaces)

    class Env:
        def __init__(self, polygons, start_point, goal_point):
            self.polygons = polygons
            self.start = start_point
            self.goal = goal_point
            self.states = []
            for i in polygons:
                for j in i.vertices:
                    self.states.append(j)
            self.states.append(goal_point)
```

```
        def check_inside(self, line):
            for polygon in self.polygons:
                if (line.check_inside(polygon)):
                    return True
            return False

        def getNextStates(self, current):
            next_states = []
            for state in self.states:
                if state != current:
                    line = Edge(current, state)
                    if not self.check_inside(line):
                        next_states.append(state)
            return next_states

    def heuristic(state,goal):
        return goal.dist(state)

    def path(state,env):
        path_list=[]
        depth=state.g
        while depth>0:
            path_list.append([state.x,state.y])
            state=state.parent
            depth-=1
        path_list.append([env.start.x,env.start.y])
        path_list.reverse()
        return path_list

    def BFS(env):   #uninformed
        initial=env.start
        goal=env.goal
        frontier = deque([])
        failure = None
        explored = set()
        explored.add(initial)
        frontier.append([(initial.x, initial.y)])
        initial.g = 0
        if(initial==goal):
            return initial
        while len(frontier) != 0:
            path = frontier.popleft()
            node = path[-1]
            node = Vertex(node[0], node[1])
            for neighbor in env.getNextStates(node):
                newlist = list(path)
                neighbor.g = node.g + 1
                neighbor.parent=node
                if(neighbor==goal):
                    goallist = list(newlist)
                    goallist.append((neighbor.x, neighbor.y))
                    return goallist
                if neighbor not in explored:
                    explored.add(neighbor)
                    newlist.append((neighbor.x, neighbor.y))
```

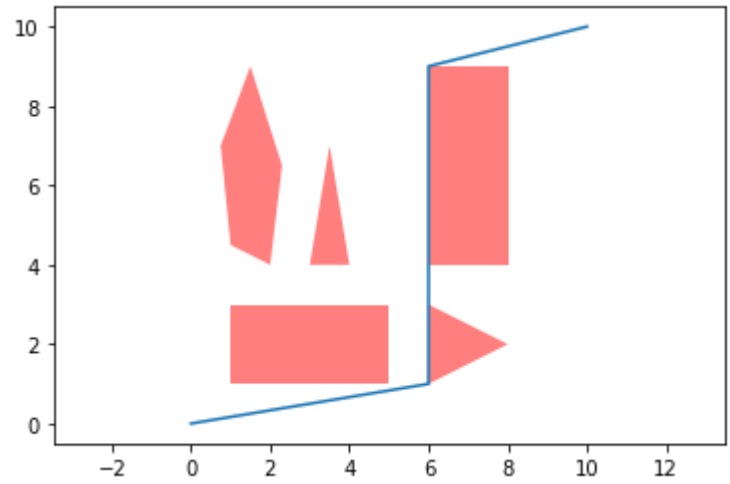
```
                frontier.append(newlist)

def GreedyBFS(env):
    initial=env.start
    goal=env.goal
    frontier = heapdict.heapdict()
    failure = None
    explored = set()
    frontier[initial] = heuristic(initial,goal)
    initial.g = 0
    while len(frontier) != 0:
        node = frontier.popitem()[0]
        if(node==goal):
            return node
        explored.add(node)
        for neighbor in env.getNextStates(node):
            neighbor.parent=node
            neighbor.g = node.g + 1
            found = 0
            if neighbor not in explored and neighbor not in frontier.keys():  #not
                frontier[neighbor]= heuristic(neighbor,goal)
    return failure


def AStar(env):
    initial=env.start
    goal=env.goal
    frontier = heapdict.heapdict()
    failure = None
    explored = set()
    frontier[initial] = heuristic(initial,goal)
    while len(frontier) != 0:
        node = frontier.popitem()[0]
        if(node==goal):
            return node
        explored.add(node)
        for neighbor in env.getNextStates(node):
            neighbor.parent=node
            neighbor.g = node.g+1
            new_cost = heuristic(neighbor,goal)+neighbor.g
            if neighbor not in explored and neighbor not in frontier.keys(): #not
                frontier[neighbor] = new_cost
            elif neighbor in frontier.keys() and frontier[neighbor] > new_cost: #a
                frontier[neighbor] = new_cost   #replacing cost with the lower cos
    return failure
p1=Polygon([Vertex(1,1),Vertex(1,3),Vertex(5,3),Vertex(5,1)])
p2=Polygon([Vertex(3,4),Vertex(4,4),Vertex(3.5,7)])
p3=Polygon([Vertex(2, 4),Vertex(1, 4.5),Vertex(0.75, 7),Vertex(1.5, 9),Vertex(2.3,
p4=Polygon([Vertex(6,4),Vertex(6,9),Vertex(8,9),Vertex(8,4)])
p5=Polygon([Vertex(6,1),Vertex(6,3),Vertex(8,2)])
env=Env([p1,p2,p3,p4,p5],Vertex(0,0),Vertex(10,10))

def print_path_line(final_path):
  x_path=[]
  y_path=[]
```

```
    for point in final_path:
      x_path.append(point[0])
      y_path.append(point[1])
    r1 = sg.Polygon([(1,1),(1,3),(5,3),(5,1)])
    r2 = sg.Polygon([(3,4),(4,4),(3.5,7)])
    r3 = sg.Polygon([(2, 4), (1, 4.5), (0.75, 7), (1.5, 9), (2.3, 6.5)])
    r4 = sg.Polygon([(6,4),(6,9),(8,9),(8,4)])
    r5 = sg.Polygon([(6,1),(6,3),(8,2)])

    new_shape = so.cascaded_union([r1,r2, r3,r4,r5])
    fig, axs = plt.subplots()
    axs.set_aspect('equal', 'datalim')

    for geom in new_shape.geoms:
        xs, ys = geom.exterior.xy
        axs.fill(xs, ys, alpha=0.5, fc='r', ec='none')
    plt.plot(x_path,y_path)
    plt.show()

print('A STAR')
point=AStar(env)
final_path=path(point,env)
print(final_path)
print_path_line(final_path)

print('GREEDY BFS')
point=GreedyBFS(env)
final_path=path(point,env)
print(final_path)
print_path_line(final_path)

print('BFS')
final_path=BFS(env)
print(final_path)
print_path_line(final_path)
```

A STAR
[[0, 0], [6, 1], [6, 9], [10, 10]]



GREEDY BFS
[[0, 0], [6, 1], [6, 9], [10, 10]]



BFS