```
# Sample code from https://www.redblobgames.com/pathfinding/a-star/
# Copyright 2014 Red Blob Games <redblobgames@gmail.com>
#
# Feel free to use this code in your own projects, including commercial projects
# License: Apache v2.0 <http://www.apache.org/licenses/LICENSE-2.0.html>
from numpy.linalg import norm
import numpy as np
import collections
# class SimpleGraph:
#     def __init__(self):
#         self.edges = {}

#     def neighbors(self, id):
#         return self.edges[id]

# example_graph = SimpleGraph()
# example_graph.edges = {
#     'A': ['B'],
#     'B': ['A', 'C', 'D'],
#     'C': ['A'],
#     'D': ['E', 'A'],
#     'E': ['B']
# }


class Queue:
    def __init__(self):
        self.elements = collections.deque()

    def empty(self):
        return len(self.elements) == 0

    def put(self, x):
        self.elements.append(x)

    def get(self):
        return self.elements.popleft()

# utility functions for dealing with square grids
def from_id_width(id, width):
    return (id % width, id // width)
###
#Draw function
###
def draw_tile(graph, id, style, width):
    r = "."
```

```
    if 'path' in style and id in style['path']: r='●'
    if 'number' in style and id in style['number']: r = "%.2f" % style['number'][id]
    if 'point_to' in style and style['point_to'].get(id, None) is not None:
        (x1, y1) = id
        (x2, y2) = style['point_to'][id]
        if x2 == x1 + 1 and y2 == y1: r = "←"
        elif x2 == x1 + 1 and y2 == y1 + 1: r = "↖"
        elif x2 == x1 + 1 and y2 == y1 - 1: r = "↙"

        elif x2 == x1 - 1 and y2 == y1: r = "→"
        elif x2 == x1 - 1 and y2 == y1 + 1: r = "↗"
        elif x2 == x1 - 1 and y2 == y1 - 1: r = "↘"

        elif y2 == y1 + 1 and x2 == x1: r = " ↑ "
        elif y2 == y1 - 1 and x2 == x1: r = " ↓ "
        else: print(x1,y1,x2,y2)

    if 'start' in style and id == (style['start'][0],style['start'][1]): r = "A"
    if id in graph.walls: r = "#" * width
    if 'goal' in style and id == (style['goal'][0],style['goal'][1]): r = "Z"
    #if 'path' in style and id in style['path']: r = "@"

    return r

def draw_grid(graph, width=2, **style):
    print("%%-%ds" % width % " ", end="")
    for i in range(graph.width):
        print("%%-%ds" % width % i, end="")
    print()
    for y in range(graph.height):
        print("%%-%ds" % width % y, end="")
        for x in range(graph.width):
            print("%%-%ds" % width % draw_tile(graph, (y, x), style, width), end="")
        print()


###
### Classes for A*
###
class SquareGrid:
    def __init__(self, width, height, walls=set(), robotsize = 1):
        self.width = width
        self.height = height
        self.walls = walls
        self.r = robotsize
```

```
    def in_bounds(self, id):
        (x, y) = id
        return 0 <= x and x < self.height and 0 <= y and y < self.width

    def passable(self, id):
        # ret = True
        # (x, y) = id
        # results = [(x+1, y), (x, y-1), (x-1, y), (x, y+1),
        #            (x+1, y+1), (x-1, y-1), (x+1, y-1), (x-1, y+1)]
        # for i in results:
        #     ret = ret and i not in self.walls

        return id not in self.walls
        # return id not in self.walls

    def neighbors(self, id):
        (x, y) = id
        results = [(x+1, y), (x, y-1), (x-1, y), (x, y+1), (x+1, y+1), (x-1, y-1), (x+1, y-1), (x-1, y+1)]
        # if (x + y) % 2 == 0: results.reverse() # aesthetics
        results = filter(self.in_bounds, results)
        results = filter(self.passable, results)
        return results

class GridWithWeights(SquareGrid):
    def __init__(self, width, height, robotsize = 0):
        super().__init__(width, height, robotsize = robotsize)
        self.weights = {}
    def cost(self, from_node, to_node):
        return self.weights.get(to_node, 1)

class MaptoGrid(SquareGrid):
    def __init__(self, Map, robotsize = 0):

        walls = set()
        _hei = Map.shape[0]
        _wid = Map.shape[1]
        for i in range(_hei):
            for j in range(_wid):
                if Map[i][j] == 0 or Map[i][j]==1:
                    continue
                elif Map[i][j] == 100:
                    for k in range(-robotsize+1,robotsize):
                        for l in range(-robotsize+1,robotsize):
                            if i+k < 0 or j+l< 0 or i+k>_hei or j+l>_wid: continue
                            walls.add((i+k,j+l))
        print(walls)
```

```
        super().__init__(_wid, _hei, walls)

        self.weights = {}
    def cost(self, a, b):
        (x1, y1) = a
        (x2, y2) = b
        return (abs(x1 - x2)**2 + abs(y1 - y2)**2)**0.5

# diagram4 = GridWithWeights(10, 10)
# diagram4.walls = [(1, 7), (1, 8), (2, 7), (2, 8), (3, 7), (3, 8)]
# diagram4.weights = {loc: 5 for loc in [(3, 4), (3, 5), (4, 1), (4, 2),
#                                        (4, 3), (4, 4), (4, 5), (4, 6),
#                                        (4, 7), (4, 8), (5, 1), (5, 2),
#                                        (5, 3), (5, 4), (5, 5), (5, 6),
#                                        (5, 7), (5, 8), (6, 2), (6, 3),
#                                        (6, 4), (6, 5), (6, 6), (6, 7),
#                                        (7, 3), (7, 4), (7, 5)]}

import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]


def heuristic(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return (abs(x1 - x2)**2 + abs(y1 - y2)**2)**0.5


def path_compression(path,graph):
    c_p = [path[1]]
    last = path[1]
    i=2
    while i <len(path)-1:
        xdel_prev = last[0]-path[i][0]
```

```
        xdel_next = path[i][0]-path[i+1][0]
        ydel_prev = last[1]-path[i][1]
        ydel_next = path[i][1]-path[i+1][1]

        slop_prev, slop_next = 0,0

        if xdel_prev:
            slop_prev = ydel_prev/xdel_prev
        if xdel_next:
            slop_next = ydel_next/xdel_next

        if xdel_prev and xdel_prev:
            if slop_prev == slop_next:
                i+=1
            else:
                c_p.append(path[i])
                last = c_p[-1]
                i+=1
        else:
            if xdel_prev or xdel_next:
                c_p.append(path[i])
                last = c_p[-1]
                i+=1
            else:
                i+=1

    if c_p[-1] != (path[-1]):
        c_p.append(path[-1])
    c_p = f_compression(c_p,graph)
    return c_p


def f_compression(path, graph):
    print(path)
    base = np.asarray(path[0])
    cur = np.asarray(path[1])
    cur_idx = 1
    next = np.asarray(path[2])
    next_idx = 2
    ret = [path[0]]
    print('start base:', base)
    print('start cur:', cur)
    print('start next:', next)
    print('start ret:', ret)
    print('start curidx and nextidx:', cur_idx, next_idx)
    print()
```

```
    for _ in range(2,len(path)):
        print('cur cur=',cur)
        print('cur next=', next)
        b_signal = 0
        div = norm(next - base)
        print('cur range:',(base[0],next[0]),(base[1],next[1]))
        for j in range(base[0],next[0]):
            for k in range(base[1],next[1]):
                cur_point = (j,k)
                d = norm(np.cross(next - base, base - cur_point)) / div
                print('cur point=',cur_point,'d=',d)
                if d > 1: continue
                else:
                    # if we cannot compress this point then proceed one step
                    print('cur point is',cur_point in graph.walls,'in walls')
                    if cur_point in graph.walls:
                        b_signal = 1
                        break
            if b_signal:
                break
        if not b_signal:
            print('no obstacles between these two points, pop cur')
            cur, next = next, np.asarray(path[next_idx])
            cur_idx, next_idx = next_idx, next_idx+1
        else:
            print('there is some obstacles here, add cur in ret')
            ret.append(tuple(cur))
            print(ret)
            cur_idx = next_idx
            next_idx += 1
            base, cur, next = cur, next, np.asarray(path[next_idx])
        print()
    ret.append(path[-1])
    return ret



def fc_help():
    pass




def enlarge(point,size = 1):
    goals = set()
```

```
    x = point[0]
    y = point[1]
    for i in range(-size,size+1):
        for j in range(-size,size+1):
            goals.add((x+i,y+j))
    return goals

def a_star_search(graph, start, goal):
#reverse!!!
    # goal = goal[::-1]
    # goals = enlarge(goal)
    # start = start[::-1]
#reverse!!!
    print('start=',start,'goal=',goal)
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0
    path = []
    foundpath = False
    print('begin searching!')
    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            foundpath = True
            break
        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic(goal, next)
                frontier.put(next, priority)
                came_from[next] = current
    if not foundpath: return [-1]
    path.append(current)
    while came_from[current]:
        path.insert(0,came_from[current])
        current = came_from[current]
    print('finish searching!')
    print('begin compression')
    print(path)
    npath = path_compression(path,graph)
    return npath, path
```

```
# return came_from, cost_so_far, path
```