"""

Implementing A* Algorithm (Maze Problem)
"""

The goal of the A* algorithm is to find the shortest path from the starting point to the goal point as fast as possible. The
#full path cost (f) for each node is calculated as the distance to the starting node (g) plus the distance to the goal node(h)
#Distances is calculated as the manhattan distance (taxicab geometry) between nodes.

```
class Node:

    # Initialize the class

    def_init_(self, position:(), parent:()):

        self.position = position

        self.parent = parent

        self.g = 0 # Distance to start node

        self.h = 0 # Distance to goal node

        self.f = 0 # Total cost

    # Compare nodes

    def_eq_(self, other):

        return self.position == other.position

    # Sort nodes

    def_lt_(self, other):

         return self.f < other.f

    # Print node

    def_repr_(self):

        return ('({0},{1})'.format(self.position, self.f))

# Draw a grid

def draw_grid(map, width, height, spacing=2, **kwargs):

    for y in range(height):

        for x in range(width):

            print('%%-%ds' % spacing % draw_tile(map, (x, y), kwargs), end='')

        print()

# Draw a tile
```

```
def draw_tile(map, position, kwargs):

    # Get the map value
    value = map.get(position)
    # Check if we should print the path
    if 'path' in kwargs and position in kwargs['path']: value = '+'
    # Check if we should print start point
    if 'start' in kwargs and position == kwargs['start']: value = '@'
    # Check if we should print the goal point
    if 'goal' in kwargs and position == kwargs['goal']: value = '$'
    # Return a tile value
    return value
  # A* search
  def astar_search(map, start, end):

    # Create lists for open nodes and closed nodes
    open = []
    closed = []
    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)
    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:
      # Sort the open list to get the node with the lowest cost first
      open.sort()
      # Get the node with the lowest cost
      current_node = open.pop(0)
      # Add the current node to the closed list
      closed.append(current_node)

      # Check if we have reached the goal, return the path
      if current_node == goal_node:
```

```
            path = []
            while current_node != start_node:
                path.append(current_node.position)
                current_node = current_node.parent
            #path.append(start)
            # Return reversed path
            return path[::-1]
        # Unzip the current node position
        (x, y) = current_node.position
        # Get neighbors
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        # Loop neighbors
        for next in neighbors:
            # Get value from map
            map_value = map.get(next)
            # Check if the node is a wall
            if(map_value == '#'):
                continue
            # Create a neighbor node
            neighbor = Node(next, current_node)
            # Check if the neighbor is in the closed list
            if(neighbor in closed):
                continue
            # Generate heuristics (Manhattan distance)
            neighbor.g = abs(neighbor.position[0] - start_node.position[0]) + abs(neighbor.position[1] - start_node.position[1])
            neighbor.h = abs(neighbor.position[0] - goal_node.position[0]) + abs(neighbor.position[1] - goal_node.position[1])
            neighbor.f = neighbor.g + neighbor.h
            # Check if neighbor is in open list and if it has a lower f value
            if(add_to_open(open, neighbor) == True):
                # Everything is green, add neighbor to open list
                open.append(neighbor)
    # Return None, no path is found
    return None
# Check if a neighbor should be added to open list
```

```
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f >= node.f):
            return False
    return True
# The main entry point for this module
def main():
    # Get a map (grid)
    map = {}
    chars = ['c']
    start = None
    end = None
    width = 0
    height = 0
    # Open a file
    fp = open('data\\maze.in', 'r')

    # Loop until there is no more lines
    while len(chars) > 0:
        # Get chars in a line
        chars = [str(i) for i in fp.readline().strip()]
        # Calculate the width
        width = len(chars) if width == 0 else width
        # Add chars to map
        for x in range(len(chars)):
            map[(x, height)] = chars[x]
            if(chars[x] == '@'):
                start = (x, height)
            elif(chars[x] == '$'):
                end = (x, height)

        # Increase the height of the map
        if(len(chars) > 0):
            height += 1
    # Close the file pointer
```

Guneet Kohli 1805172

```
    fp.close()

    # Find the closest path from start(@) to end($)

    path = astar_search(map, start, end)

    print()

    print(path)

    print()

    draw_grid(map, width, height, spacing=1, path=path, start=start, goal=end)

    print()

    print('Steps to goal: {0}'.format(len(path)))

    print()

# Tell python to run main method

if __name___== "_main_": main()
```