**Project 1: Robot Path Planning**

**Group Members: Anna Teng, Sunny Li**

**Program Running Instructions:**

Program should be run via the terminal with 3 argument being the python file containing the project code, input file name, and k in the format: **python3 <robot\_path\_finding\_file> <input\_file> <k\_value>**.

For example, with input 1 and k of 0:

python3 RobotPathPlanning.py "Input1.txt" 0

**Program Source Code:**

"""

Robot Path Finding Project

Authors: Anna Teng, Sunny Li

"""

import heapq

import math

import copy

import argparse

DIRECTIONS = [(1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1)]

# OutputModel class to simplify the code relating to the output

class OutputModel():

def \_\_init\_\_(self, output\_dict: dict):

self.depth = output\_dict["depth"]

self.generated\_nodes = len(output\_dict["generated\_nodes"])

self.moves = " ".join(map(str, output\_dict["moves"]))

self.f\_values = " ".join(map(str, output\_dict["f\_values"]))

rows = [" ".join(map(str, row)) for row in output\_dict["workspace"]]

self.output\_workspace = "\n".join(rows)

# Node class

class Node():

def \_\_init\_\_(self, pos, path\_cost, total\_cost, last\_angle=0, parent=None):

self.pos = pos

self.path\_cost = path\_cost

self.total\_cost = total\_cost

self.last\_angle = last\_angle

self.parent = parent

def \_\_lt\_\_(self, other):

# Used for heapq

return self.total\_cost < other.total\_cost

def \_\_repr\_\_(self):

# Used for debugging

return str(self.pos)

def process\_input(input\_file\_path):

"""

Used to process input file into workable data

:param input\_file\_path: path of input file

:return: tuple of starting and goal positions, and a 2D array of the robot workspace

"""

try:

input\_file = open(input\_file\_path, 'r')

except FileNotFoundError:

print("File not found!")

return

first\_line = input\_file.readline()

first\_line\_data = first\_line.strip().split()

start\_pos = (int(first\_line\_data[0]), int(first\_line\_data[1]))

goal\_pos = (int(first\_line\_data[2]), int(first\_line\_data[3]))

workspace = []

for line in input\_file:

if line != "\n":

curr\_line = []

nums = line.strip().split()

for num in nums:

curr\_line.append(int(num))

workspace.append(curr\_line)

input\_file.close()

workspace.reverse()

return start\_pos, goal\_pos, workspace

def calculate\_heuristic(curr\_pos, goal\_pos):

"""

Calculate the h(n) value based on current and goal position

:param curr\_pos: tuple of current location

:param goal\_pos: tuple of goal position

:return: The heuristic value calculated

"""

return math.sqrt((curr\_pos[0]-goal\_pos[0])\*\*2+(curr\_pos[1]-goal\_pos[1])\*\*2)

def is\_valid\_pos(pos, workspace):

"""

Returns whether a position is valid (not out of bound and not blocked)

:param pos: position to check

:param workspace: 2D list of workspace

:return: boolean of validity

"""

if pos[1] < 0 or pos[0] < 0 or pos[1] >= len(workspace) or pos[0] >= len(workspace[0]):

return False

return workspace[pos[1]][pos[0]] != 1

def calculate\_angle\_cost(curr\_angle, new\_angle, k):

"""

Calculates the angle cost for changing direction based on curr\_angle and new\_angle

:param curr\_angle: angle of current node in degrees

:param new\_angle: angle of next node in degrees

:param k: penalty coefficient for direction change

:return: angle cost as a float

"""

delta\_theta = abs(new\_angle - curr\_angle)

if delta\_theta > 180:

delta\_theta = 360 - delta\_theta

return k \* (delta\_theta / 180)

def calculate\_distance\_cost(curr\_node, new\_pos):

"""

Calculates the distance cost to get to new\_pos from curr\_pos

:param curr\_node: coordinates of curr position

:param new\_pos: coordinates of new position

:return: distance cost as a float

"""

# Check if horizontal or vertical move

if abs(new\_pos[0] - curr\_node.pos[0]) + abs(new\_pos[1] - curr\_node.pos[1]) == 1:

distance\_cost = 1

else:

distance\_cost = math.sqrt(2)

return distance\_cost

def calculate\_step\_cost(curr\_node, new\_pos, k):

"""

Calculates the step cost by adding distance and angle costs

:param curr\_node: coordinates of curr position

:param new\_pos: coordinates of new position

:param k: penalty coefficient for direction change

:return: total step cost as a float

:return: new angle in degrees

"""

dx, dy = new\_pos[0] - curr\_node.pos[0], new\_pos[1] - curr\_node.pos[1]

new\_angle = math.degrees(math.atan2(dy, dx)) % 360

if curr\_node.parent is None:

angle\_cost = 0

else:

angle\_cost = calculate\_angle\_cost(curr\_node.last\_angle, new\_angle, k)

distance\_cost = calculate\_distance\_cost(curr\_node, new\_pos)

return distance\_cost + angle\_cost, new\_angle

def a\_star\_search\_algo(start\_pos, goal\_pos, workspace, k):

"""

Implementation of the A\* search algorithm

:param start\_pos: starting coordinate

:param goal\_pos: goal coordinate

:param workspace: workspace 2D list

:return: None if no solution; curr\_node, generated if solution is found

"""

start\_node = Node(start\_pos, 0, calculate\_heuristic(start\_pos, goal\_pos))

reached = {}

frontier = []

heapq.heappush(frontier, start\_node)

generated = [start\_node]

while frontier:

# Get the smallest value

curr\_node = heapq.heappop(frontier)

# If solution is found

if curr\_node.pos == goal\_pos:

return curr\_node, generated

# Generate all child nodes

for direction in DIRECTIONS:

new\_pos = (curr\_node.pos[0] + direction[0], curr\_node.pos[1] + direction[1])

# Append node to generated and frontier if it's valid

if is\_valid\_pos(new\_pos, workspace):

step\_cost, new\_angle = calculate\_step\_cost(curr\_node, new\_pos, k)

child\_path\_cost = curr\_node.path\_cost + step\_cost

child\_heuristic = calculate\_heuristic(new\_pos, goal\_pos)

child\_total\_cost = child\_path\_cost + child\_heuristic

if new\_pos in reached and reached[new\_pos] <= child\_total\_cost:

continue

child\_node = Node(new\_pos, child\_path\_cost, child\_total\_cost, last\_angle=new\_angle, parent=curr\_node)

reached[child\_node.pos] = child\_node.total\_cost

generated.append(child\_node)

heapq.heappush(frontier, child\_node)

def calculate\_output\_values(final\_node, workspace):

"""

Function used to calculate several values needed for output

:param final\_node: The last node in the path found

:param workspace: 2D list of the workspace

:return: dictionary of all the values

"""

# Just didn't want to change the original workspace

new\_workspace = copy.deepcopy(workspace)

curr\_node = final\_node

depth = -1

moves = []

f\_values = []

while curr\_node:

pos = curr\_node.pos

if new\_workspace[pos[1]][pos[0]] != 2 and new\_workspace[pos[1]][pos[0]] != 5:

new\_workspace[pos[1]][pos[0]] = 4

depth += 1

f\_values.append(curr\_node.total\_cost)

if curr\_node.parent is not None:

direction = (curr\_node.pos[0] - curr\_node.parent.pos[0],

curr\_node.pos[1] - curr\_node.parent.pos[1])

move = DIRECTIONS.index(direction)

moves.append(move)

curr\_node = curr\_node.parent

moves.reverse()

f\_values.reverse()

new\_workspace.reverse()

return {

"depth": depth,

"moves": moves,

"f\_values": f\_values,

"workspace": new\_workspace

}

def output\_into\_file(output: OutputModel, file="sample testing.txt"):

"""

Used to write all output data into a output file

:param output: The OutputModel used to help with output generation

:param file: the filepath of the output file

:return: None

"""

output\_file = open(file, "w")

print(output.depth, file=output\_file)

print(output.generated\_nodes, file=output\_file)

print(output.moves, file=output\_file)

print(output.f\_values, file=output\_file)

print(output.output\_workspace, file=output\_file)

output\_file.close()

def main():

parser = argparse.ArgumentParser(description="Run A\* search on a robot workspace")

parser.add\_argument("input\_file", type=str, help="Path to the input file")

parser.add\_argument("k", type=int, nargs='?', default=0, help="Angle cost penalty parameter (default: 0)")

args = parser.parse\_args()

start\_pos, goal\_pos, workspace = process\_input(args.input\_file)

result = a\_star\_search\_algo(start\_pos, goal\_pos, workspace, args.k)

if result:

final\_node, generated\_nodes = result

output\_dict = calculate\_output\_values(final\_node, workspace)

output\_dict["generated\_nodes"] = generated\_nodes

output = OutputModel(output\_dict)

output\_into\_file(output)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Program Output Files:**