**Open-Ended Lab**

**2D-Maze-Solver**

**Objective:** This program solves a 2D maze with the help of several search algorithms.

**Background:**

The program firstly reads the input file you provided and creates the maze.

After that, the program start to search the maze based upon the algorithms it contains. After the algorithm finishes, it prints out the cost of the solution found, the solution path itself and the list of expanded nodes. Here are the algorithms that is contained within the program itself:

* Depth-First Search (DFS)
* Breath-First Search (BFS)
* Iterative-Deepening Search (IDS)
* Uniform-Cost Search (UCS)
* Greedy-Best-First Search (GBFS)
* A-Star Search (A\*)

There are four different files to run this program:

* Main.py
* Graph.py
* Maze.py
* Search.py

**Code:**

**Main.py:**

import search

from graph import Graph

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

graph = Graph()

search.graph = graph

search.depth\_first\_search()

search.breath\_first\_search()

search.iterative\_deepening\_search()

search.uniform\_cost\_search()

search.greedy\_best\_first\_search()

search.a\_star\_search()

**Graph.py**

import sys

from maze import Maze

class Node:

def \_\_init\_\_(self):

self.x = 0

self.y = 0

self.cost = 0

self.parent = None

self.east = None

self.south = None

self.west = None

self.north = None

self.heuristic = 0

def check\_equality(self, x, y):

return x == self.x and y == self.y

def \_\_str\_\_(self):

return "[" + str(self.x) + ", " + str(self.y) + "]"

class Graph:

nodes = []

maze = None

def \_\_init\_\_(self):

self.maze = Maze()

self.root = self.create\_node(self.maze.start[0], self.maze.start[1])

self.maximum\_depth = self.find\_maximum\_depth() - 1

self.create\_heuristic()

self.root.cost = 0

def create\_node(self, x, y):

node = Node()

node.x = x

node.y = y

self.nodes.append(node)

if self.maze.traps[node.x][node.y] == 1:

node.cost = 7

else:

node.cost = 1

if self.maze.can\_pass(node.x, node.y, "east"):

node.east = self.node\_exists(node.x, node.y + 1)

if node.east is None:

node.east = self.create\_node(node.x, node.y + 1)

node.east.parent = node

if self.maze.can\_pass(node.x, node.y, "south"):

node.south = self.node\_exists(node.x + 1, node.y)

if node.south is None:

node.south = self.create\_node(node.x + 1, node.y)

node.south.parent = node

if self.maze.can\_pass(node.x, node.y, "west"):

node.west = self.node\_exists(node.x, node.y - 1)

if node.west is None:

node.west = self.create\_node(node.x, node.y - 1)

node.west.parent = node

if self.maze.can\_pass(node.x, node.y, "north"):

node.north = self.node\_exists(node.x - 1, node.y)

if node.north is None:

node.north = self.create\_node(node.x - 1, node.y)

node.north.parent = node

return node

def node\_exists(self, x, y):

for node in self.nodes:

if node.check\_equality(x, y):

return node

return None

def find\_maximum\_depth(self):

maximum\_depth = 0

for node in self.nodes:

current\_node = node

local\_depth = 0

while current\_node is not None:

current\_node = current\_node.parent

local\_depth += 1

maximum\_depth = max(maximum\_depth, local\_depth)

return maximum\_depth

**Maze.py**

import re

class Maze:

size = []

wall\_vertical = [[]]

walls\_horizontal = [[]]

traps = [[]]

start = []

goals = []

def \_\_init\_\_(self):

self.read\_maze()

def read\_maze(self):

file = open("maze.txt", "r")

line = file.readline().rstrip("\n\r")

empty\_line = 0

while empty\_line < 2:

if not line:

empty\_line += 1

else:

empty\_line = 0

if line == "Size":

first\_size = file.readline().rstrip("\n\r")

second\_size = file.readline().rstrip("\n\r")

self.set\_size(first\_size, second\_size)

elif line == "Walls":

walls = []

line = file.readline().rstrip("\n\r")

# We are going to read every line until a blank line...

while line:

walls.append(line)

line = file.readline().rstrip("\n\r")

self.set\_walls(walls)

elif line == "Traps":

traps = []

column\_indexes = walls[i+1].split()

for index in column\_indexes:

self.wall\_vertical[row\_index - 1][int(index) - 1] = 1

elif "column" in walls[i]:

column\_index = int(re.sub("[^0-9]", "", walls[i]))

row\_indexes = walls[i + 1].split()

for index in row\_indexes:

self.walls\_horizontal[int(index) - 1][column\_index - 1] = 1

def set\_traps(self, traps):

for trap in traps:

indexes = list(map(int, trap.split()))

self.traps[indexes[0] - 1][indexes[1] - 1] = 1

def set\_start(self, start):

indexes = list(map(int, start.split()))

self.start.append(indexes[0] - 1)

self.start.append(indexes[1] - 1)

def set\_goals(self, goals):

for goal in goals:

indexes = list(map(int, goal.split()))

indexes = list(map(lambda x: x - 1, indexes))

self.goals.append(indexes)

def can\_pass(self, row, column, direction):

# Check if the player can pass

if direction == "east":

if column == (self.size[1] - 1):

return False

return self.wall\_vertical[row][column] == 0

elif direction == "south":

if row == (self.size[0] - 1):

return False

return self.walls\_horizontal[row][column] == 0

elif direction == "west":

if column == 0:

return False

return self.wall\_vertical[row][column - 1] == 0

elif direction == "north":

if row == 0:

return False

return self.walls\_horizontal[row - 1][column] == 0

**Search.py**

from collections import OrderedDict

graph = None

frontier = []

visited = OrderedDict() # To prevent duplicates, we use OrderedDict

def depth\_first\_search():

graph.clear\_parents()

dfs\_bfs\_ids\_ucs("Depth First Search(DFS):")

def breath\_first\_search():

graph.clear\_parents()

dfs\_bfs\_ids\_ucs("Breath First Search(BFS):")

def iterative\_deepening\_search():

graph.clear\_parents()

dfs\_bfs\_ids\_ucs("Iterative Deepening Search(IDS):")

def uniform\_cost\_search():

graph.clear\_parents()

dfs\_bfs\_ids\_ucs("Uniform Cost Search(UCS):")

def greedy\_best\_first\_search():

graph.clear\_parents()

heuristic\_search("Greedy Best First Search(GBFS):", return\_heuristic)

def a\_star\_search():

graph.clear\_parents()

heuristic\_search("A Star Search(A\*):", return\_cost\_and\_heuristic)

def heuristic\_search(algorithm, sort\_by):

goal\_state = None

solution\_cost = 0

solution = []

frontier.clear visited.clear()

frontier.append(graph.root)

if "UCS" in algorithm:

sort\_frontier(return\_cost)

current\_node = frontier.pop(pop\_index)

visited[current\_node] = None

if is\_goal(current\_node):

goal\_state = current\_node

break

if "IDS" in algorithm:

parent = current\_node

for i in range(iteration):

parent = parent if parent is None else parent.parent

if parent is None:

add\_to\_frontier(current\_node, "DFS")

else:

add\_to\_frontier(current\_node, algorithm)

for node in visited:

expanded\_nodes.append(node)

if "IDS" not in algorithm:

break

if goal\_state is None:

print("No goal state found.")

return

current = goal\_state

while current is not None:

solution\_cost += current.cost

solution.insert(0, current)

current = current.parent

print\_results(algorithm, solution\_cost, solution, expanded\_nodes)

def add\_to\_frontier(current\_node, algorithm):

nodes\_to\_add = []

if current\_node.east is not None and not is\_in\_visited(current\_node.east):

nodes\_to\_add.append(set\_parent(current\_node, current\_node.east, algorithm))

if current\_node.south is not None and not is\_in\_visited(current\_node.south):

nodes\_to\_add.append(set\_parent(current\_node, current\_node.south, algorithm))

if current\_node.west is not None and not is\_in\_visited(current\_node.west):

nodes\_to\_add.append(set\_parent(current\_node, current\_node.west, algorithm))

if current\_node.north is not None and not is\_in\_visited(current\_node.north):

nodes\_to\_add.append(set\_parent(current\_node, current\_node.north, algorithm))

if "DFS" in algorithm:

nodes\_to\_add.reverse()

for node in nodes\_to\_add:

frontier.append(node)

def set\_parent(parent\_node, child\_node, algorithm):

if "DFS" in algorithm or child\_node.parent is None:

child\_node.parent = parent\_node

return child\_node

def is\_in\_visited(node):

if node in visited:

return True

return False

def is\_goal(node):

for goal in graph.maze.goals:

if goal[0] == node.x and goal[1] == node.y:

return True

return False

def print\_results(algorithm, solution\_cost, solution, expanded\_nodes):

print(algorithm)

print("Cost of the solution:", solution\_cost)

print("The solution path (" + str(len(solution)) + " nodes):", end=" ")

for node in solution:

print(node, end=" ")

print("\nExpanded nodes (" + str(len(expanded\_nodes)) + " nodes):", end=" ")

if "IDS" in algorithm:

print()

for i in range(len(expanded\_nodes) - 1):

if type(expanded\_nodes[i+1]) == str:

print(expanded\_nodes[i])

else:

print(expanded\_nodes[i], end=" ")

else:

for node in expanded\_nodes:

print(node, end=" ")

print("\n")

def return\_cost(node):

return node.cost

def return\_heuristic(node):

return node.heuristic

def return\_cost\_and\_heuristic(node):

return node.heuristic + node.cost

def sort\_frontier(sort\_by):

frontier.sort(key=sort\_by)

**Output:**

**Depth First Search(DFS):**

Cost of the solution: 31

The solution path (20 nodes): [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 1] [5, 0] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [2, 5] [2, 6]

Expanded nodes (35 nodes): [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [2, 5] [2, 6]

**Breath First Search(BFS):**

Cost of the solution: 23

The solution path (12 nodes): [2, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [2, 5] [2, 6]

Expanded nodes (42 nodes): [2, 1] [3, 1] [2, 0] [3, 2] [4, 1] [3, 0] [1, 0] [4, 2] [2, 2] [4, 0] [0, 0] [5, 2] [5, 0] [0, 1] [5, 3] [5, 1] [6, 0] [1, 1] [4, 3] [6, 1] [7, 0] [1, 2] [3, 3] [6, 2] [7, 1] [1, 3] [0, 2] [3, 4] [6, 3] [7, 2] [2, 3] [0, 3] [4, 4] [7, 3] [2, 4] [0, 4] [5, 4] [2, 5] [1, 4] [0, 5] [6, 4] [2, 6]

**Iterative Deepening Search(IDS):**

Cost of the solution: 31

The solution path (20 nodes): [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 1] [5, 0] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [2, 5] [2, 6]

Expanded nodes (498 nodes):

Iteration 0: [2, 1]

Iteration 1: [2, 1] [3, 1] [2, 0]

Iteration 2: [2, 1] [3, 1] [3, 2] [4, 1] [2, 0] [3, 0] [1, 0]

Iteration 3: [2, 1] [3, 1] [3, 2] [4, 2] [2, 2] [4, 1] [2, 0] [3, 0] [4, 0] [1, 0] [0, 0]

Iteration 4: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [2, 2] [4, 1] [2, 0] [3, 0] [4, 0] [5, 0] [1, 0] [0, 0] [0, 1]

Iteration 5: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [5, 1] [2, 2] [4, 1] [2, 0] [3, 0] [4, 0] [5, 0] [6, 0] [1, 0] [0, 0] [0, 1] [1, 1]

Iteration 6: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [5, 1] [5, 0] [2, 2] [4, 1] [2, 0] [3, 0] [4, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2]

Iteration 7: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [5, 1] [5, 0] [6, 0] [4, 0] [2, 2] [4, 1] [2, 0] [3, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [0, 2]

Iteration 8: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [5, 1] [5, 0] [6, 0] [6, 1] [7, 0] [4, 0] [3, 0] [2, 2] [4, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [0, 2] [0, 3]

Iteration 9: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [2, 2] [4, 1]

Iteration 10: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [2, 2] [4, 1]

Iteration 11: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [2, 2] [4, 1]

Iteration 12: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [2, 2] [4, 1]

Iteration 13: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [2, 2] [4, 1]

Iteration 14: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [2, 2] [4, 1]

Iteration 15: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [0, 2] [2, 2] [4, 1]

Iteration 16: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [0, 2] [0, 3] [2, 2] [4, 1]

Iteration 17: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [0, 2] [0, 3] [0, 4] [2, 2] [4, 1]

Iteration 18: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [2, 5] [1, 4] [0, 2] [0, 3] [0, 4] [0, 5] [2, 2] [4, 1]

Iteration 19: [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [2, 5]

**Uniform Cost Search(UCS):**

Cost of the solution: 18

The solution path (19 nodes): [2, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [0, 7] [1, 7] [2, 7] [3, 7] [4, 7] [5, 7] [5, 6]

Expanded nodes (50 nodes): [2, 1] [3, 1] [2, 0] [3, 2] [4, 1] [3, 0] [1, 0] [4, 2] [2, 2] [4, 0] [0, 0] [5, 2] [5, 0] [0, 1] [5, 3] [5, 1] [6, 0] [1, 1] [4, 3] [6, 1] [7, 0] [1, 2] [3, 3] [6, 2] [7, 1] [0, 2] [3, 4] [6, 3] [7, 2] [0, 3] [4, 4] [7, 3] [0, 4] [5, 4] [0, 5] [1, 4] [6, 4] [0, 6] [2, 4] [0, 7] [2, 3] [1, 7] [2, 7] [1, 6] [3, 7] [1, 5] [4, 7] [3, 6] [5, 7] [5, 6]

**Greedy Best First Search(GBFS):**

Cost of the solution: 23

The solution path (18 nodes): [2, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [0, 7] [1, 7] [1, 6] [1, 5] [2, 5] [2, 6]

Expanded nodes (48 nodes): [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [4, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [2, 2] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [1, 4] [0, 7] [1, 7] [2, 7] [1, 6] [3, 7] [3, 6] [1, 5] [2, 5] [2, 6]

**A Star Search(A\*):**

Cost of the solution: 18

The solution path (19 nodes): [2, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [0, 7] [1, 7] [2, 7] [3, 7] [4, 7] [5, 7] [5, 6]

Expanded nodes (48 nodes): [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1] [4, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [2, 2] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [1, 4] [0, 7] [1, 7] [2, 7] [1, 6] [3, 7] [3, 6] [1, 5] [4, 7] [5, 7] [5, 6]