**Birla Institute of Science and Technology**

**Artificial and Computational Intelligence**

**Problem solving by informed and uninformed search**

**Abstract:**

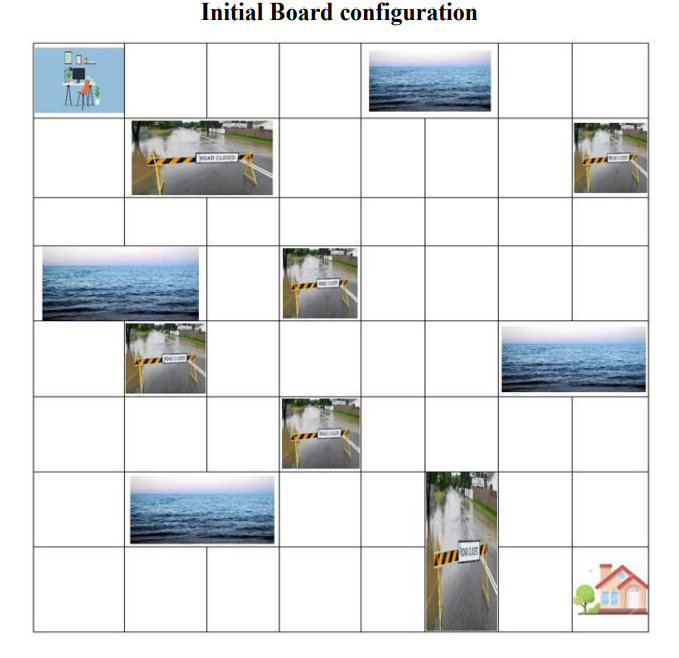
In this particular case, we will be dealing with the design of a Problem Solving Agent in the given environment, to reach from home to office.

**Problem Statement:**

The city Nellore is facing heavy rains due to Cyclone Michaung. Most of the areas are inundated with water.

Design of a navigation agent which you can use to find the safe routes without water. The agent is fed with the map of the city marked with areas of flood. The agent has to find the route that is the safest to take you home by choosing the next grid considering different factors. 5 points to be added each time the agent passes adjacent (Up, Down, Left, Right) safe places and 5 points to be deducted while the agent passes near water bodies and 3 points to be deducted if roads in the area are flooded with water.

**Grid Environment:**



|  |  |
| --- | --- |
| **Task Environment** | Grid Navigation (Static / Dynamic) |
| **Algorithm** | GBFS, Genetic |
| **Fully vs Partially Observable** | Fully |
| **Single vs Multi-Agent** | Single |
| **Deterministic vs Stochastic** | Deterministic |
| **Episodic vs Sequential** | Episodic |
| **Static vs Dynamic** | Static |
| **Discrete vs Continuous** | Discrete |

**PEAS Environment:**

|  |  |
| --- | --- |
| Performance\* | Find safest route to home |
| Environments | Navigation Agent’s current position in grid  City map  Road  Location of flood areas  Goal/home location  Info of neighbouring cells (safe/unsafe) |
| Sensors | Detect flood area and safe cell |
| Actuators | Agent allowed movements.  Move Left, Move Right, Move Up, Move Down |

**Task Environment:**

• Sensor based fully observable environment.

• Episodic Action

• Discrete State

• Single agent navigation

• Static and

• Deterministic environment

**Task Environment Details:**

With the confirmation of a fully observable environment, here's a breakdown of the task environment characteristics:

**Deterministic:**

* **States:** All grid positions, obstacles, start, and goal are known upfront, with no hidden elements or uncertainty impacting information accessibility.
* **Actions:** Movement actions (up, down, left, right) are assumed to be deterministic, always leading to the intended direction without unexpected events.
* **Outcomes:** Given the same state and action, the next state is always predictable (e.g., moving down from a specific position always leads to the position below).

**Static:**

* **Grid:** The 8x8 grid layout and obstacle positions remain constant throughout the task.
* **Start and Goal:** Both the start and goal nodes are fixed, and their locations never change.

**Episodic:**

* **Independent trials:** Each attempt to find the safe path from start to goal is considered an independent episode with a clear beginning (start node) and end (goal node).
* **Reset between episodes:** After reaching the goal or failing (e.g., hitting an obstacle), the agent is assumed to be reset to the starting position, starting a new episode.

**Discrete:**

* **States:** Individual grid positions represent distinct and countable states.
* **Actions:** The four movement options are distinct and finite.
* **Transitions:** Movement between states (e.g., from start to next position) happens in discrete steps, not continuously.

**Part a: Designing a PSA to reach goal safely using informed search Method: Greedy best first search**

GBFS is an informed search algorithm that always expands the node that is estimated to be closest to the goal based on the heuristic function h(n). It is not optimal because it may get stuck in local optima or loops, but it is often used when optimality is not a strict requirement and computational efficiency is important.

**Greedy Best First Algorithm:**

1. Initialize a tree with the root node being the start node in the open list.

2. If the open list is empty, return a failure, otherwise, add the current node to the closed list.

3. Remove the node with the lowest h(n) value from the open list for exploration.

4. If a child node is the target, return a success. Otherwise, if the node has not been in either the open or closed list, add it to the open list for exploration.

Heuristic function h(n):

A heuristic function, denoted as h(n), assesses the proximity of a successive node to the target node, favoring the immediate low-cost option. Consequently, while this approach may prioritize nodes that appear to be closer to the goal, it does not guarantee the discovery of the shortest path to reach the goal node.

Given, 5 points to be added each time the agent passes adjacent (Up, Down, Left, Right) safe places and 5 points to be deducted while the agent passes near water bodies and 3 points to be deducted if roads in the area are flooded with water.

Calculation of h(n):

s = 0

h(n) =

if (row, col) == flooded\_cell

s -= 3

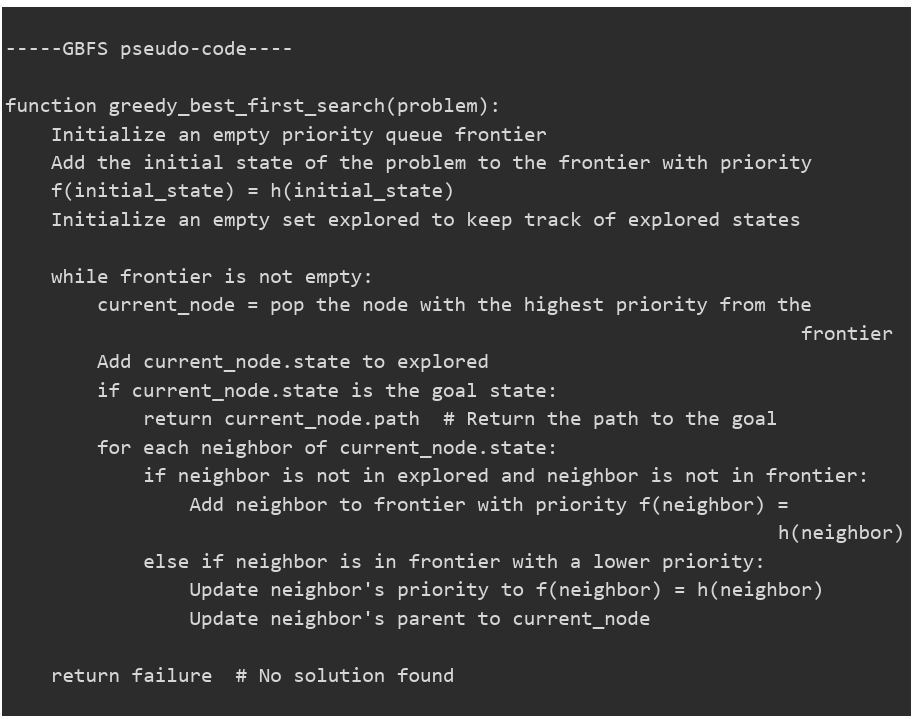
else if (row, col) == blocked\_cell

s -= 5

else if (row, col) == safe\_cell

s += 5

GBFS Pseudo Code:



**Part b: Designing a PSA to reach goal safely using uninformed search Method: Genetic Algorithm**

Uninformed search algorithms explore the search space without any additional information about the state space or the problem's constraints. They make decisions based solely on the available state transitions and do not use problem-specific knowledge beyond the definition of the problem itself. Examples of some other uninformed search algorithms include depth-first search, breadth-first search, and uniform-cost search.

**Genetic Algorithm:**

A genetic algorithm (GA) is a type of metaheuristic optimization algorithm inspired by the process of natural selection and evolution. It is used to find approximate solutions to optimization and search problems by mimicking the process of natural selection in biological organisms.

Here's a simplified explanation of how a genetic algorithm works:

1. **Initialization**: A population of potential solutions (individuals) to the optimization problem is randomly generated.

2. **Evaluation**: Each individual in the population is evaluated and assigned a fitness score, which indicates how well it solves the optimization problem.

3. **Selection**: Individuals are selected from the population to become parents for the next generation based on their fitness scores. Individuals with higher fitness scores are more likely to be selected.

4. **Crossover**: Pairs of selected parents are combined to produce offspring (new individuals) through crossover or recombination. This process involves exchanging genetic information (e.g., parts of a solution representation) between parents to create new candidate solutions.

5. **Mutation**: Occasionally, random changes (mutations) are introduced to the offspring's genetic information to add diversity to the population.

6. **Replacement**: The offspring replaces some individuals in the current population, creating a new generation of candidate solutions.

7. **Termination**: The process iterates through the selection, crossover, mutation, and replacement steps for a predetermined number of generations or until a satisfactory solution is found.

By repeating these steps over multiple generations, the population evolves towards better solutions to the optimization problem.

#Pseudo code for genetic Algorithm

A screenshot of a computer program

Description automatically generated

**Part c: Python Implementation**

**Part c(i): Folder structure**

#Folder Structure

A screenshot of a computer

Description automatically generated

**Part c (ii): main() Implementation / Initialisation of program**

#main.py

import argparse

from pprint import pprint

from utils.GridGenerator import GridGenerator

from utils.GridEnvironment import GridEnvironment

from Algorithms.GBFSearchAlgorithm import GBFSearchAlgorithm

from Algorithms.GeneticSearchAlgorithm import GeneticSearchAlgorithm

from Algorithms.SearchAlgorithmFactory import SearchAlgorithmFactory

def main():

    """

    This script runs a search algorithm on a grid environment.

    Usage:

        python main.py [-a | -b] [-d]

    Options:

        -a, --genetic   Run Genetic Algorithm

        -b, --gbfs      Run Greedy Best First Search

        -d, --display   Display grid

    Example:

        python main.py          ## Run both Genetic Algorithm and Greedy Best First Search

        python main.py -b -d    ## Run Greedy Best First Search

        python main.py -a       ## Run Genetic Algorithm

    """

    parser = argparse.ArgumentParser()

    group = parser.add\_mutually\_exclusive\_group()

    group.add\_argument("-a", "--genetic", action="store\_true", help="Run Genetic Algorithm")

    group.add\_argument("-b", "--gbfs", action="store\_true", help="Run Greedy Best First Search")

    parser.add\_argument("-d", "--display", action="store\_true", help="Display grid")

    args = parser.parse\_args()

    if(args.genetic == False) and (args.gbfs == False):

        grid\_gen = GridGenerator()

        grid\_gen.generate\_grid()

        env = GridEnvironment(grid\_gen.grid, args.display)

        print("Running both Genetic Algorithm and Greedy Best First Search")

        gbfs = GBFSearchAlgorithm(env)

        gbfs\_path, gbfs\_cost = gbfs.search()

        print("Path taken by the agent using Greedy Best First Search:", gbfs\_path)

        print("Total path cost using Greedy Best First Search:", gbfs\_cost)

        genetic\_search = GeneticSearchAlgorithm(env)

        gs\_path, gs\_cost = genetic\_search.search()

        print("Path taken by the agent using Genetic Algorithm:", gs\_path)

        print("Total path cost using Genetic Algorithm:", gs\_cost)

    else:

        search\_algorithm = SearchAlgorithmFactory.create\_search\_algorithm(args)

        path, total\_cost = search\_algorithm.search()

        print("Path taken by the agent:", path)

        print("Total path cost:", total\_cost)

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

    main()

**Part c (iii): Dynamic Generation of Grid**

#utils.GridGenerator.py

class GridGenerator:

    def \_\_init\_\_(self):

        self.grid = [

            ['.', '.', '.', '.', '#', '#', '.', '.'],

            ['.', 'F', 'F', '.', '.', '.', '.', 'F'],

            ['.', '.', '.', '.', '.', '.', '.', '.'],

            ['#', '#', '.', 'F', '.', '.', '.', '.'],

            ['.', 'F', '.', '.', '.', '.', '#', '#'],

            ['.', '.', '.', 'F', '.', '.', '.', '.'],

            ['.', '#', '#', '.', '.', 'F', '.', '.'],

            ['.', '.', '.', '.', '.', 'F', '.', '.']

        ]

    def generate\_grid(self):

        # Start position input and validation

        while True:

            start\_input = input("\nEnter the start position (row,col): ")

            start\_row, start\_col = map(int, start\_input.split(','))

            grid\_with\_start, error\_message = self.set\_position((start\_row, start\_col), 'S')

            if error\_message:

                print(error\_message)

            else:

                break

        # Goal position input and validation

        while True:

            goal\_input = input("\nEnter the goal position (row,col): ")

            goal\_row, goal\_col = map(int, goal\_input.split(','))

            grid\_with\_goal, error\_message = self.set\_position((goal\_row, goal\_col), 'G')

            if error\_message:

                print(error\_message)

            else:

                break

        # Print final grid with start and goal positions

        print("\nFinal grid with start and goal positions:")

        self.print\_grid()

    def set\_position(self, position, symbol):

        row, col = position

        if row < 0 or row > 7 or col < 0 or col > 7:

            return None, f"Invalid input! Row and column indices must be between 0 and 7."

        if self.grid[row][col] == '#':

            self.print\_grid()

            return None, f"Invalid {symbol} position! This cell is occupied by a water body ('#')."

        if self.grid[row][col] == 'F':

            self.print\_grid()

            return None, f"Invalid {symbol} position! This cell is flooded ('F')."

        if self.grid[row][col] == 'S':

            self.print\_grid()

            return None, f"Invalid {symbol} position! The start and the Goal position cannot be same."

        self.grid[row][col] = symbol

        return self.grid, None

    def print\_grid(self):

        for row in self.grid:

            print(' '.join(row))

**Part c (iv): Function for program statistics**

#utils.grid.py

import time

import platform

if platform.system() == "Windows":

    import psutil

elif platform.system() == "Mac" or platform.system() == "Linux":

    import resource

def track\_time\_and\_space(func):

    def wrapper(\*args, \*\*kwargs):

        start\_time = time.time()

        start\_memory = 0

        end\_memory = 0

        if platform.system() == "Mac" or platform.system() == "Linux":

            start\_memory = resource.getrusage(resource.RUSAGE\_SELF).ru\_maxrss

        elif platform.system() == "Windows":

            start\_memory = psutil.Process().memory\_info().rss

        result = func(\*args, \*\*kwargs)

        end\_time = time.time()

        if platform.system == "Mac" or platform.system() == "Linux":

            end\_memory = resource.getrusage(resource.RUSAGE\_SELF).ru\_maxrss

        elif platform.system() == "Windows":

            end\_memory = psutil.Process().memory\_info().rss

        execution\_time = end\_time - start\_time

        memory\_usage = (end\_memory - start\_memory) / 1024  # Convert to kilobytes

        print(f"Execution time: {execution\_time} seconds")

        print(f"Memory usage: {memory\_usage} KB")

        return result

    return wrapper

**Part c (v): Visualize the Grid using matplotlib from pyplot**

Use pip install

#utils.GridEnvironment.py

import matplotlib.pyplot as plt

class GridEnvironment:

    def \_\_init\_\_(self, grid: list[list[str]], display: bool = False):

        def find\_start\_and\_goal():

            for i in range(self.rows):

                for j in range(self.cols):

                    if self.grid[i][j] == 'S':

                        self.start = tuple((i, j))

                    elif self.grid[i][j] == 'G':

                        self.goal = tuple((i, j))

            print("Start {start}, Goal {goal}:".format(start=self.start, goal=self.goal))

        self.grid = grid

        self.rows = len(grid)

        self.cols = len(grid[0])

        self.display = display

        find\_start\_and\_goal()

    def visualize(self, path):

        # Define colors for different elements

        colors = {'S': 'green', '.': 'white', '#': 'black', 'F': 'red', 'G': 'blue'}

        # Create a plot

        fig, ax = plt.subplots()

        # Plot each element with specified colors

        for i in range(self.rows):

            for j in range(self.cols):

                ax.text(i, j, self.grid[i][j], ha='center', va='center', color=colors[self.grid[i][j]])

        # Customize ticks

        ax.set\_xticks(range(len(self.grid[0])))

        ax.set\_yticks(range(len(self.grid)))

        ax.set\_xticklabels([])

        ax.set\_yticklabels([])

        # Add description

        plt.title('S - Start, G - Goal')

        plt.xlabel('# - Water Bodies')

        plt.ylabel('F - Flooded Roads')

        # Add grid lines

        ax.grid(True, which='both', color='black', linewidth=1.5, linestyle='-', alpha=0.7)

        plt.plot(\*zip(\*path), marker='o', color='red', label='Path')

        plt.scatter(\*path[0], marker='o', color='green', label='Start')

        plt.scatter(\*path[-1], marker='o', color='blue', label='Goal')

    def is\_valid\_move(self, row, col):

        return (

            0 <= row < self.rows

            and 0 <= col < self.cols

            and self.grid[row][col] != '#'

            and self.grid[row][col] != 'F'

        )

    def get\_adjacent\_cells(self, row, col, algorithm=None):

        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)] #degrees of freedom - need to revise by prasnejit

        adjacent\_cells = [] # Return empty list of adjacent cells if all moves are invalid

        for dr, dc in directions:

            new\_row, new\_col = row + dr, col + dc

            if self.is\_valid\_move(new\_row, new\_col):

                if algorithm == "greedy" and self.display:

                    print(" The move is valid for this %d and %d" % (new\_row, new\_col))

                adjacent\_cells.append(tuple((new\_row, new\_col)))

        return adjacent\_cells

    def goal\_reached(self, row, col):

        return (row, col) == self.goal

**Part d: GBFS (Informed search) Implementation**

#GBFS Implementation:

Filepath: ./Algorithms/GBfSearchAlgorithm.py

from heapq import heappush, heappop

from collections import deque

from typing import List, Tuple

## Import the project files

from utils.grid import track\_time\_and\_space

from utils.GridEnvironment import GridEnvironment

from Algorithms.ISearchAlgorithm import ISearchAlgorithm

class GBFSearchAlgorithm(ISearchAlgorithm):

    """

    Greedy Best-First Search Algorithm implementation.

    This algorithm uses a priority queue to explore the search space based on a heuristic function.

    It expands the node with the lowest heuristic value, prioritizing the most promising paths towards the goal.

    Attributes:

        None

    Methods:

        search(grid\_env: GridEnvironment) -> Tuple[List[Tuple[int, int]], int]:

            Performs the greedy best-first search on the given grid environment.

        heuristic(row: int, col: int) -> int:

            Calculates the heuristic value for a given cell in the grid.

        time\_complexity() -> str:

            Returns the time complexity of the algorithm.

        space\_complexity() -> str:

            Returns the space complexity of the algorithm.

    """

    def \_\_init\_\_(self, grid\_env: GridEnvironment) -> None:

        super().\_\_init\_\_()

        self.grid\_env = grid\_env

        self.tree = {}

    @track\_time\_and\_space

    def search(self) -> Tuple[List[Tuple[int, int]], int, int]:

        """

        Performs the greedy best-first search on the given grid environment.

        Args:

            grid\_env (GridEnvironment): The grid environment to search in.

        Returns:

            Tuple[List[Tuple[int, int]], int]: A tuple containing the path taken by the agent and the total path cost.

        """

        start = self.grid\_env.start

        goal = self.grid\_env.goal

        visited = set()

        pq = [(self.heuristic(\*start), start)]

        came\_from = {}

        cost\_so\_far = {start: 0}  # Store the cost of reaching each cell

        total\_nodes\_expanded = 0

        total\_branching\_factor = 0

        depth\_of\_solution = 0

        while pq:

                print("Open List (Priority Queue):", pq)

                print("Closed List (Visited Set):", visited)

                priority, current = heappop(pq)  # Pop the item with lowest priority

                if current == goal:

                    # Reconstruct the path

                    path = deque()

                    total\_cost = cost\_so\_far[current]

                    while current != start:

                        path.appendleft(current)

                        current = came\_from[current]

                        depth\_of\_solution += 1  # Increment depth for each step towards the start node

                    path.appendleft(start)

                    break  # Exit the loop when the goal node is found

                visited.add(current)

                total\_nodes\_expanded += 1

                successors = self.grid\_env.get\_adjacent\_cells(\*current, algorithm="greedy")

                total\_branching\_factor += len(successors)

                for next\_cell in successors:

                    if next\_cell in visited:  # Check if the cell has already been visited

                        continue  # Skip to the next iteration if the cell has been visited

                    new\_cost = cost\_so\_far[current] + 1  # Assuming each step has a cost of 1

                    if next\_cell not in cost\_so\_far or new\_cost < cost\_so\_far[next\_cell]:

                        cost\_so\_far[next\_cell] = new\_cost

                        # print("Cost of next cell", cost\_so\_far[next\_cell])

                        priority = new\_cost + self.heuristic(\*next\_cell)

                        print("Priority of next cell", priority)

                        heappush(pq, (priority, next\_cell))  # Add the next cell to the priority queue

                        came\_from[next\_cell] = current

            # Store the depth of the optimal solution

        self.depth\_of\_solution = depth\_of\_solution

        print("Total Branching Factor:", round(total\_branching\_factor / total\_nodes\_expanded))

        print("Depth of the graph search tree is:", self.depth\_of\_solution)

            # Optionally, you can return the path and total cost if needed

        return list(path), total\_cost

    def heuristic(self, row: int, col: int) -> int:

        """

        Calculates heuristic value for a given cell in the grid. Considering

        if the adjacent cells are safe, water bodies, or flooded roads

        Args:

            row (int): The row index of the cell.

            col (int): The column index of the cell.

        Returns:

            int: The heuristic value for the cell.

        """

        score = 0

        for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:

            new\_row, new\_col = row + dr, col + dc

            if 0 <= new\_row < self.grid\_env.rows and 0 <= new\_col < self.grid\_env.cols:

                if self.grid\_env.grid[new\_row][new\_col] == '.':

                    score += 5  # Add 5 points for adjacent safe places

                elif self.grid\_env.grid[new\_row][new\_col] == '#':

                    score -= 5  # Deduct 5 points for adjacent water bodies

                elif self.grid\_env.grid[new\_row][new\_col] == 'F':

                    score -= 3  # Deduct 3 points for flooded roads

        return score

**Part e: Genetic Algorithm (uninformed search) Implementation**

#Genetic Algorithm

Filepath: ./Algorithms/GeneticSearchAlgorithm.py

import random

from abc import ABC

from utils.grid import track\_time\_and\_space

from utils.GridEnvironment import GridEnvironment

from Algorithms.ISearchAlgorithm import ISearchAlgorithm

class Individual(ABC):

    def \_\_init\_\_(self, grid\_env: GridEnvironment):

        self.individual\_type = "Child"  # Every individual is a child, until it is a parent

        self.grid\_env = grid\_env

        self.path = [tuple(grid\_env.start)]

        self.fitness = 0

        current\_pos = grid\_env.start

        while current\_pos != grid\_env.goal:

            adjacent\_cells = grid\_env.get\_adjacent\_cells(\*current\_pos, algorithm="genetic")

            next\_pos = random.choice(adjacent\_cells)

            self.path.append(next\_pos)

            current\_pos = next\_pos

    def set\_path(self, path: list):

        self.path = path

    def evaluate\_fitness(self):

        for pos in self.path:

            row, col = pos

            if self.grid\_env.grid[row][col] == '.':

                self.fitness += 5

            elif self.grid\_env.grid[row][col] == '#':

                self.fitness -= 5

            elif self.grid\_env.grid[row][col] == 'F':

                self.fitness -= 3

    def mutate(self, mutation\_rate: float):

        for i in range(1, len(self.path) - 1):

            if random.random() < mutation\_rate:

                adjacent\_cells = self.grid\_env.get\_adjacent\_cells(\*self.path[i], algorithm="genetic")

                self.path[i] = random.choice(adjacent\_cells)

class GeneticSearchAlgorithm(ISearchAlgorithm):

    def \_\_init\_\_(self, grid\_env):

        self.grid\_env: GridEnvironment = grid\_env

        self.population\_size = 10

        self.generations = 100

        self.mutation\_rate = 0.01

    def crossover(self, parent1, parent2):

        crossover\_point = random.randint(1, min(len(parent1.path), len(parent2.path)) - 1)

        # Single point crossover

        child\_path = parent1.path[:crossover\_point] + parent2.path[crossover\_point:]

        child = Individual(self.grid\_env)

        child.set\_path(child\_path)

        return child

    @track\_time\_and\_space

    def search(self):

        population = [Individual(self.grid\_env) for \_ in range(self.population\_size)]

        for generation in range(self.generations):

            for individual in population:

                individual.evaluate\_fitness()

            population.sort(key=lambda x: x.fitness, reverse=True)

            if population[0].fitness == 40:  # Max possible fitness - to be revised later

                break

            next\_generation = population[:2]  # Elitism

            while len(next\_generation) < self.population\_size:

                parent1 = random.choice(population[:self.population\_size // 2])

                parent2 = random.choice(population[:self.population\_size // 2])

                child = self.crossover(parent1, parent2)

                child.mutate(self.mutation\_rate)

                next\_generation.append(child)

            population = next\_generation

        best\_individual = population[0]

path = best\_individual.path

        cost = 0

        for pos in path:

            row, col = pos

            if self.grid\_env.grid[row][col] == '.':

                cost += 5

            elif self.grid\_env.grid[row][col] == '#':

                cost -= 5

            elif self.grid\_env.grid[row][col] == 'F':

                cost -= 3

        return path, cost

**Part f: Execution of the program / stack run.**

Execution of code:

C:\Users\HP\OneDrive\Desktop\MTech\assignments\mtech\semester\_1\03\_assignments\aci\ACI\_ASSIGNMENT\_1> python .\main.py

Enter the start position (row,col): 0,1

Enter the goal position (row,col): 7,7

Final grid with start and goal positions:

. S . . # # . .

. F F . . . . F

. . . . . . . .

# # . F . . . .

. F . . . . # #

. . . F . . . .

. # # . . F . .

. . . . . F . G

Start (0, 1), Goal (7, 7):

**Part g (i): Dynamic Generation of Grid - OUTPUT**

**Part g (ii): GBFS Search Algorithm – OUTPUT**

Open List (Priority Queue): [(7, (0, 1))]

Closed List (Visited Set): set()

Priority of next cell 3

Priority of next cell 6

Open List (Priority Queue): [(3, (0, 2)), (6, (0, 0))]

Closed List (Visited Set): {(0, 1)}

Priority of next cell 7

Open List (Priority Queue): [(6, (0, 0)), (7, (0, 3))]

Closed List (Visited Set): {(0, 1), (0, 2)}

Priority of next cell 9

Open List (Priority Queue): [(7, (0, 3)), (9, (1, 0))]

Closed List (Visited Set): {(0, 1), (0, 2), (0, 0)}

Priority of next cell 15

Open List (Priority Queue): [(9, (1, 0)), (15, (1, 3))]

Closed List (Visited Set): {(0, 1), (0, 2), (0, 3), (0, 0)}

Priority of next cell 8

Open List (Priority Queue): [(8, (2, 0)), (15, (1, 3))]

Closed List (Visited Set): {(0, 1), (0, 0), (0, 3), (0, 2), (1, 0)}

Priority of next cell 6

Open List (Priority Queue): [(6, (2, 1)), (15, (1, 3))]

Closed List (Visited Set): {(0, 1), (0, 0), (0, 3), (2, 0), (0, 2), (1, 0)}

Priority of next cell 17

Open List (Priority Queue): [(15, (1, 3)), (17, (2, 2))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (0, 3), (2, 0), (0, 2), (1, 0)}

Priority of next cell 16

Priority of next cell 14

Open List (Priority Queue): [(14, (1, 4)), (17, (2, 2)), (16, (2, 3))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (0, 3), (2, 0), (0, 2), (1, 0), (1, 3)}

Priority of next cell 25

Priority of next cell 15

Open List (Priority Queue): [(15, (1, 5)), (16, (2, 3)), (25, (2, 4)), (17, (2, 2))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (0, 3), (2, 0), (1, 4), (0, 2), (1, 0), (1, 3)}

Priority of next cell 26

Priority of next cell 18

Open List (Priority Queue): [(16, (2, 3)), (17, (2, 2)), (25, (2, 4)), (26, (2, 5)), (18, (1, 6))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (0, 2), (1, 0), (1, 3)}

Open List (Priority Queue): [(17, (2, 2)), (18, (1, 6)), (25, (2, 4)), (26, (2, 5))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (2, 3), (0, 2), (1, 0), (1, 3)}

Priority of next cell 8

Open List (Priority Queue): [(8, (3, 2)), (18, (1, 6)), (25, (2, 4)), (26, (2, 5))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (2, 3), (0, 2), (2, 2), (1, 0), (1, 3)}

Priority of next cell 19

Open List (Priority Queue): [(18, (1, 6)), (19, (4, 2)), (25, (2, 4)), (26, (2, 5))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (2, 3), (0, 2), (2, 2), (1, 0), (3, 2), (1, 3)}

Priority of next cell 27

Priority of next cell 12

Open List (Priority Queue): [(12, (0, 6)), (19, (4, 2)), (25, (2, 4)), (27, (2, 6)), (26, (2, 5))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (2, 3), (0, 2), (2, 2), (1, 0), (1, 6), (3, 2), (1, 3)}

Priority of next cell 10

Open List (Priority Queue): [(10, (0, 7)), (19, (4, 2)), (25, (2, 4)), (27, (2, 6)), (26, (2, 5))]

Closed List (Visited Set): {(0, 1), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (0, 2), (2, 2), (1, 0), (1, 6), (3, 2), (1, 3)}

Open List (Priority Queue): [(19, (4, 2)), (26, (2, 5)), (25, (2, 4)), (27, (2, 6))]

Closed List (Visited Set): {(0, 1), (0, 7), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (0, 2), (2, 2), (1, 0), (1, 6), (3, 2), (1, 3)}

Priority of next cell 10

Priority of next cell 12

Open List (Priority Queue): [(10, (5, 2)), (12, (4, 3)), (27, (2, 6)), (26, (2, 5)), (25, (2, 4))]

Closed List (Visited Set): {(0, 1), (0, 7), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (4, 2), (0, 2), (2, 2), (1, 0), (1, 6), (3, 2), (1, 3)}

Priority of next cell 11

Open List (Priority Queue): [(11, (5, 1)), (12, (4, 3)), (27, (2, 6)), (26, (2, 5)), (25, (2, 4))]

Closed List (Visited Set): {(0, 1), (0, 7), (2, 1), (0, 0), (1, 5), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (4, 2), (0, 2), (2, 2), (1, 0), (1, 6), (3, 2), (1, 3), (5, 2)}

Priority of next cell 25

Open List (Priority Queue): [(12, (4, 3)), (25, (2, 4)), (27, (2, 6)), (26, (2, 5)), (25, (5, 0))]

Closed List (Visited Set): {(5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (0, 1), (0, 7), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3)}

Priority of next cell 29

Open List (Priority Queue): [(25, (2, 4)), (25, (5, 0)), (27, (2, 6)), (26, (2, 5)), (29, (4, 4))]

Closed List (Visited Set): {(4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (0, 1), (0, 7), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3)}

Priority of next cell 18

Open List (Priority Queue): [(18, (3, 4)), (25, (5, 0)), (27, (2, 6)), (29, (4, 4)), (26, (2, 5))]

Closed List (Visited Set): {(4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3)}

Priority of next cell 27

Priority of next cell 27

Open List (Priority Queue): [(25, (5, 0)), (26, (2, 5)), (27, (2, 6)), (29, (4, 4)), (27, (4, 4)), (27, (3, 5))]

Closed List (Visited Set): {(3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3)}

Priority of next cell 16

Priority of next cell 8

Open List (Priority Queue): [(8, (4, 0)), (27, (3, 5)), (16, (6, 0)), (29, (4, 4)), (27, (4, 4)), (27, (2, 6)), (26, (2, 5))]

Closed List (Visited Set): {(3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3)}

Open List (Priority Queue): [(16, (6, 0)), (27, (3, 5)), (26, (2, 5)), (29, (4, 4)), (27, (4, 4)), (27, (2, 6))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3)}

Priority of next cell 22

Open List (Priority Queue): [(22, (7, 0)), (27, (3, 5)), (26, (2, 5)), (29, (4, 4)), (27, (4, 4)), (27, (2, 6))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (6, 0)}

Priority of next cell 18

Open List (Priority Queue): [(18, (7, 1)), (27, (3, 5)), (26, (2, 5)), (29, (4, 4)), (27, (4, 4)), (27, (2, 6))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (6, 0)}

Priority of next cell 19

Open List (Priority Queue): [(19, (7, 2)), (27, (3, 5)), (26, (2, 5)), (29, (4, 4)), (27, (4, 4)), (27, (2, 6))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (7, 1), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (6, 0)}

Priority of next cell 30

Open List (Priority Queue): [(26, (2, 5)), (27, (3, 5)), (27, (2, 6)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (1, 3), (7, 1), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (7, 2), (6, 0)}

Open List (Priority Queue): [(27, (2, 6)), (27, (3, 5)), (30, (7, 3)), (29, (4, 4)), (27, (4, 4))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (7, 2), (6, 0)}

Priority of next cell 18

Priority of next cell 15

Open List (Priority Queue): [(15, (2, 7)), (27, (3, 5)), (18, (3, 6)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 14

Open List (Priority Queue): [(14, (3, 7)), (27, (3, 5)), (18, (3, 6)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Open List (Priority Queue): [(18, (3, 6)), (27, (3, 5)), (30, (7, 3)), (29, (4, 4)), (27, (4, 4))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (5, 0), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Open List (Priority Queue): [(27, (3, 5)), (27, (4, 4)), (30, (7, 3)), (29, (4, 4))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (5, 0), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 18

Open List (Priority Queue): [(18, (4, 5)), (27, (4, 4)), (30, (7, 3)), (29, (4, 4))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (5, 0), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (3, 5), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 21

Open List (Priority Queue): [(21, (5, 5)), (27, (4, 4)), (30, (7, 3)), (29, (4, 4))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (3, 5), (5, 2), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 20

Priority of next cell 22

Open List (Priority Queue): [(20, (5, 6)), (22, (5, 4)), (30, (7, 3)), (29, (4, 4)), (27, (4, 4))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 23

Priority of next cell 16

Open List (Priority Queue): [(16, (5, 7)), (23, (6, 6)), (22, (5, 4)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 22

Open List (Priority Queue): [(22, (5, 4)), (23, (6, 6)), (22, (6, 7)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 23

Open List (Priority Queue): [(22, (6, 7)), (23, (6, 6)), (23, (6, 4)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 23

Open List (Priority Queue): [(23, (6, 4)), (23, (6, 6)), (23, (7, 7)), (29, (4, 4)), (27, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (6, 7), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 19

Priority of next cell 14

Open List (Priority Queue): [(14, (6, 3)), (27, (4, 4)), (19, (7, 4)), (29, (4, 4)), (30, (7, 3)), (23, (7, 7)), (23, (6, 6))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (6, 4), (6, 7), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0)}

Priority of next cell 28

Open List (Priority Queue): [(19, (7, 4)), (27, (4, 4)), (23, (6, 6)), (29, (4, 4)), (30, (7, 3)), (23, (7, 7)), (28, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (6, 4), (6, 7), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0), (6, 3)}

Open List (Priority Queue): [(23, (6, 6)), (27, (4, 4)), (23, (7, 7)), (29, (4, 4)), (30, (7, 3)), (28, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 4), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (6, 4), (6, 7), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0), (6, 3)}

Priority of next cell 14

Open List (Priority Queue): [(14, (7, 6)), (27, (4, 4)), (23, (7, 7)), (29, (4, 4)), (30, (7, 3)), (28, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 4), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (6, 4), (6, 7), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0), (6, 6), (6, 3)}

Open List (Priority Queue): [(23, (7, 7)), (27, (4, 4)), (28, (7, 3)), (29, (4, 4)), (30, (7, 3))]

Closed List (Visited Set): {(4, 0), (3, 4), (4, 3), (3, 7), (5, 4), (5, 1), (5, 7), (0, 2), (2, 2), (1, 0), (1, 6), (2, 5), (1, 3), (7, 4), (7, 1), (4, 2), (4, 5), (5, 0), (5, 6), (3, 6), (0, 1), (0, 7), (2, 4), (2, 1), (2, 7), (1, 5), (7, 0), (6, 4), (6, 7), (7, 6), (3, 2), (3, 5), (5, 2), (5, 5), (0, 0), (0, 3), (2, 0), (1, 4), (0, 6), (2, 3), (2, 6), (7, 2), (6, 0), (6, 6), (6, 3)}

Total Branching Factor: 3

Depth of the graph search tree is: 13

Execution time: 1.169076681137085 seconds

Memory usage: 16.0 KB

Path taken by the agent using Greedy Best First Search: [(0, 1), (0, 2), (0, 3), (1, 3), (1, 4), (2, 4), (3, 4), (3, 5), (4, 5), (5, 5), (5, 6), (5, 7), (6, 7), (7, 7)]

Total path cost using Greedy Best First Search: 13

Execution time: 4.774971961975098 seconds

Memory usage: 1152.0 KB

**Part g (iii): Genetic Algorithm – OUTPUT**

Path taken by the agent using Genetic Algorithm: [(0, 1), (0, 2), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 2), (0, 3), (0, 2), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (2, 2), (2, 3), (2, 2), (3, 2), (4, 2), (5, 2), (4, 2), (3, 2), (4, 2), (4, 3), (4, 2), (3, 2), (4, 2), (3, 2), (2, 2), (2, 1), (2, 0), (1, 0), (2, 0), (2, 1), (2, 0), (1, 0), (2, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (3, 2), (4, 2), (5, 2), (5, 1), (5, 0), (5, 1), (5, 2), (5, 1), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (5, 1), (5, 0), (5, 1), (5, 2), (4, 2), (3, 2), (2, 2), (3, 2), (4, 2), (4, 3), (4, 2), (3, 2), (2, 2), (2, 1), (2, 0), (1, 0), (0, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 1), (0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 4), (1, 5), (1, 6), (2, 6), (1, 6), (0, 6), (0, 7), (0, 6), (1, 6), (2, 6), (2, 7), (3, 7), (2, 7), (3, 7), (3, 6), (3, 5), (2, 5), (1, 5), (2, 5), (2, 6), (2, 7), (3, 7), (3, 6), (3, 7), (2, 7), (2, 6), (2, 7), (2, 6), (3, 6), (3, 7), (3, 6), (2, 6), (3, 6), (3, 5), (3, 4), (2, 4), (2, 5), (2, 4), (2, 5), (1, 5), (2, 5), (3, 5), (4, 5), (4, 4), (4, 5), (4, 4), (3, 4), (4, 4), (5, 4), (6, 4), (7, 4), (6, 4), (6, 3), (7, 3), (7, 4), (7, 3), (7, 4), (6, 4), (7, 4), (7, 3), (7, 4), (6, 4), (6, 3), (7, 3), (6, 3), (7, 3), (6, 3), (7, 3), (6, 3), (6, 4), (5, 4), (4, 4), (5, 4), (5, 5), (5, 4), (4, 4), (4, 5), (3, 5), (3, 6), (3, 7), (3, 6), (2, 6), (1, 6), (0, 6), (1, 6), (1, 5), (2, 5), (2, 4), (2, 5), (2, 6), (3, 6), (3, 5), (3, 6), (2, 6), (3, 6), (3, 7), (2, 7), (3, 7), (3, 6), (3, 5), (2, 5), (2, 4), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 6), (3, 6), (2, 6), (2, 7), (3, 7), (2, 7), (2, 6), (3, 6), (3, 5), (4, 5), (5, 5), (5, 4), (4, 4), (4, 3), (4, 4), (4, 5), (4, 4), (4, 5), (3, 5), (2, 5), (2, 6), (3, 6), (3, 5), (3, 4), (4, 4), (5, 4), (4, 4), (4, 5), (3, 5), (3, 4), (2, 4), (1, 4), (2, 4), (1, 4), (1, 3), (1, 4), (1, 5), (1, 6), (2, 6), (1, 6), (1, 5), (1, 6), (0, 6), (1, 6), (0, 6), (1, 6), (2, 6), (2, 7), (3, 7), (2, 7), (2, 6), (1, 6), (2, 6), (2, 5), (2, 6), (2, 7), (2, 6), (2, 7), (3, 7), (2, 7), (2, 6), (1, 6), (1, 5), (1, 4), (2, 4), (2, 5), (2, 4), (2, 5), (2, 6), (3, 6), (3, 5), (4, 5), (3, 5), (4, 5), (5, 5), (4, 5), (4, 4), (4, 3), (4, 2), (3, 2), (4, 2), (4, 3), (4, 4), (5, 4), (4, 4), (5, 4), (6, 4), (7, 4), (7, 3), (6, 3), (7, 3), (6, 3), (6, 4), (6, 3), (6, 4), (5, 4), (5, 5), (5, 4), (6, 4), (7, 4), (7, 3), (7, 2), (7, 3), (7, 4), (7, 3), (6, 3), (7, 3), (7, 4), (7, 3), (7, 4), (6, 4), (7, 4), (6, 4), (6, 3), (7, 3), (7, 4), (7, 3), (7, 2), (7, 1), (7, 0), (6, 0), (5, 0), (5, 1), (5, 0), (5, 1), (5, 0), (6, 0), (5, 0), (5, 1), (5, 2), (5, 1), (5, 0), (5, 1), (5, 0), (5, 1), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (5, 1), (5, 0), (4, 0), (5, 0), (6, 0), (5, 0), (5, 1), (5, 0), (6, 0), (5, 0), (6, 0), (7, 0), (6, 0), (7, 0), (7, 1), (7, 0), (6, 0), (5, 0), (4, 0), (5, 0), (6, 0), (5, 0), (6, 0), (5, 0), (5, 1), (5, 2), (4, 2), (5, 2), (5, 1), (5, 0), (5, 1), (5, 0), (5, 1), (5, 0), (6, 0), (7, 0), (6, 0), (7, 0), (6, 0), (5, 0), (6, 0), (5, 0), (4, 0), (5, 0), (5, 1), (5, 2), (4, 2), (3, 2), (4, 2), (4, 3), (4, 4), (5, 4), (6, 4), (7, 4), (7, 3), (6, 3), (6, 4), (5, 4), (6, 4), (5, 4), (4, 4), (4, 5), (4, 4), (3, 4), (2, 4), (1, 4), (2, 4), (3, 4), (2, 4), (2, 5), (2, 4), (2, 3), (2, 2), (2, 3), (1, 3), (2, 3), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4), (4, 3), (4, 2), (5, 2), (4, 2), (5, 2), (4, 2), (4, 3), (4, 2), (3, 2), (2, 2), (3, 2), (4, 2), (3, 2), (2, 2), (2, 3), (2, 2), (2, 1), (2, 0), (1, 0), (2, 0), (2, 1), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (0, 2), (0, 1), (0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 1), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (2, 0), (2, 1), (2, 0), (1, 0), (2, 0), (1, 0), (0, 0), (1, 0), (2, 0), (1, 0), (0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 1), (0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (2, 2), (2, 3), (1, 3), (1, 4), (1, 3), (2, 3), (2, 4), (2, 3), (2, 2), (2, 1), (2, 0), (2, 1), (2, 2), (2, 3), (2, 2), (2, 1), (2, 2), (2, 3), (1, 3), (0, 3), (0, 2), (0, 1), (0, 2), (0, 1), (0, 2), (0, 3), (0, 2), (0, 3), (1, 3), (2, 3), (1, 3), (0, 3), (0, 2), (0, 1), (0, 0), (1, 0), (2, 0), (1, 0), (0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (1, 0), (2, 0), (1, 0), (0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (2, 2), (2, 1), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (1, 4), (1, 3), (1, 4), (2, 4), (2, 3), (2, 2), (2, 1), (2, 0), (1, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 1), (0, 2), (0, 3), (1, 3), (0, 3), (1, 3), (2, 3), (2, 4), (2, 3), (1, 3), (2, 3), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (2, 5), (2, 6), (3, 6), (3, 7), (3, 6), (3, 5), (3, 6), (3, 7), (2, 7), (2, 6), (2, 7), (3, 7), (3, 6), (2, 6), (1, 6), (2, 6), (2, 7), (3, 7), (3, 6), (2, 6), (1, 6), (2, 6), (1, 6), (0, 6), (1, 6), (2, 6), (2, 5), (1, 5), (1, 4), (1, 3), (1, 4), (1, 5), (1, 6), (0, 6), (0, 7), (0, 6), (1, 6), (2, 6), (3, 6), (3, 7), (3, 6), (3, 7), (2, 7), (3, 7), (3, 6), (3, 5), (3, 6), (3, 7), (2, 7), (3, 7), (2, 7), (2, 6), (3, 6), (2, 6), (2, 7), (3, 7), (3, 6), (3, 7), (3, 6), (3, 7), (2, 7), (2, 6), (1, 6), (0, 6), (1, 6), (0, 6), (1, 6), (2, 6), (2, 5), (2, 6), (2, 7), (2, 6), (1, 6), (1, 5), (1, 6), (1, 5), (2, 5), (2, 6), (2, 5), (3, 5), (3, 4), (2, 4), (2, 5), (3, 5), (3, 4), (3, 5), (4, 5), (5, 5), (4, 5), (5, 5), (5, 4), (5, 5), (5, 4), (4, 4), (3, 4), (3, 5), (3, 6), (3, 5), (3, 4), (3, 5), (4, 5), (4, 4), (4, 3), (4, 2), (5, 2), (4, 2), (4, 3), (4, 2), (4, 3), (4, 2), (3, 2), (2, 2), (2, 3), (2, 2), (2, 1), (2, 2), (2, 3), (1, 3), (0, 3), (0, 2), (0, 3), (1, 3), (0, 3), (1, 3), (0, 3), (1, 3), (1, 4), (1, 3), (2, 3), (1, 3), (2, 3), (2, 2), (2, 3), (2, 2), (3, 2), (2, 2), (3, 2), (2, 2), (2, 1), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4), (3, 4), (3, 5), (4, 5), (3, 5), (2, 5), (1, 5), (1, 6), (1, 5), (2, 5), (1, 5), (1, 4), (2, 4), (2, 5), (2, 6), (1, 6), (1, 5), (1, 4), (2, 4), (1, 4), (1, 5), (2, 5), (3, 5), (2, 5), (2, 4), (2, 3), (2, 4), (2, 3), (2, 2), (2, 1), (2, 2), (2, 3), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (4, 5), (3, 5), (3, 6), (3, 5), (2, 5), (2, 4), (2, 5), (2, 4), (2, 5), (3, 5), (4, 5), (4, 4), (4, 3), (4, 2), (4, 3), (4, 2), (5, 2), (4, 2), (5, 2), (4, 2), (5, 2), (5, 1), (5, 2), (5, 1), (5, 0), (5, 1), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (5, 1), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (5, 1), (5, 0), (5, 1), (5, 2), (4, 2), (5, 2), (5, 1), (5, 2), (5, 1), (5, 0), (6, 0), (5, 0), (5, 1), (5, 2), (5, 1), (5, 2), (4, 2), (4, 3), (4, 2), (5, 2), (5, 1), (5, 2), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4), (2, 3), (2, 2), (3, 2), (2, 2), (2, 3), (2, 4), (1, 4), (1, 3), (0, 3), (0, 2), (0, 1), (0, 2), (0, 3), (0, 2), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (2, 0), (2, 1), (2, 0), (1, 0), (0, 0), (0, 1), (0, 0), (0, 1), (0, 2), (0, 1), (0, 2), (0, 1), (0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4), (4, 3), (4, 2), (5, 2), (4, 2), (3, 2), (4, 2), (3, 2), (4, 2), (4, 3), (4, 4), (5, 4), (4, 4), (4, 5), (3, 5), (2, 5), (2, 4), (3, 4), (3, 5), (4, 5), (3, 5), (2, 5), (1, 5), (1, 6), (2, 6), (3, 6), (2, 6), (2, 7), (3, 7), (3, 6), (3, 5), (2, 5), (2, 6), (3, 6), (3, 7), (3, 6), (3, 7), (2, 7), (2, 6), (1, 6), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (1, 6), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (1, 6), (0, 6), (0, 7), (0, 6), (0, 7), (0, 6), (1, 6), (1, 5), (1, 6), (0, 6), (1, 6), (1, 5), (1, 4), (2, 4), (1, 4), (1, 3), (1, 4), (2, 4), (2, 3), (1, 3), (2, 3), (2, 4), (2, 3), (2, 4), (1, 4), (1, 5), (1, 6), (1, 5), (1, 4), (1, 5), (1, 6), (1, 5), (1, 6), (1, 5), (1, 4), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (2, 5), (1, 5), (1, 4), (1, 5), (1, 6), (2, 6), (2, 7), (2, 6), (2, 7), (3, 7), (3, 6), (3, 7), (3, 6), (3, 7), (3, 6), (2, 6), (3, 6), (3, 5), (2, 5), (3, 5), (2, 5), (1, 5), (2, 5), (1, 5), (2, 5), (1, 5), (1, 6), (2, 6), (3, 6), (3, 7), (2, 7), (3, 7), (2, 7), (3, 7), (2, 7), (3, 7), (3, 6), (2, 6), (2, 5), (1, 5), (1, 6), (0, 6), (1, 6), (2, 6), (2, 7), (2, 6), (2, 7), (2, 6), (3, 6), (3, 7), (3, 6), (3, 5), (2, 5), (1, 5), (2, 5), (2, 4), (2, 3), (2, 2), (3, 2), (2, 2), (3, 2), (4, 2), (5, 2), (5, 1), (5, 0), (6, 0), (7, 0), (6, 0), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (5, 1), (5, 0), (5, 1), (5, 0), (5, 1), (5, 2), (4, 2), (4, 3), (4, 2), (4, 3), (4, 2), (4, 3), (4, 2), (3, 2), (2, 2), (2, 3), (2, 4), (3, 4), (3, 5), (3, 6), (2, 6), (2, 5), (2, 4), (2, 3), (1, 3), (0, 3), (0, 2), (0, 3), (0, 2), (0, 1), (0, 2), (0, 3), (0, 2), (0, 1), (0, 2), (0, 1), (0, 2), (0, 1), (0, 0), (0, 1), (0, 2), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (2, 0), (1, 0), (0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4), (4, 3), (4, 4), (4, 3), (4, 4), (4, 5), (4, 4), (4, 3), (4, 4), (5, 4), (5, 5), (5, 4), (4, 4), (5, 4), (6, 4), (6, 3), (7, 3), (6, 3), (6, 4), (6, 3), (7, 3), (6, 3), (6, 4), (7, 4), (6, 4), (5, 4), (4, 4), (3, 4), (2, 4), (1, 4), (2, 4), (1, 4), (1, 5), (2, 5), (2, 6), (3, 6), (2, 6), (3, 6), (2, 6), (1, 6), (1, 5), (2, 5), (1, 5), (2, 5), (1, 5), (1, 6), (1, 5), (2, 5), (2, 6), (2, 7), (3, 7), (2, 7), (2, 6), (1, 6), (2, 6), (2, 7), (2, 6), (2, 7), (3, 7), (2, 7), (2, 6), (1, 6), (2, 6), (3, 6), (3, 7), (3, 6), (3, 5), (4, 5), (4, 4), (5, 4), (4, 4), (5, 4), (6, 4), (5, 4), (4, 4), (4, 3), (4, 2), (5, 2), (5, 1), (5, 2), (5, 1), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (4, 0), (5, 0), (6, 0), (7, 0), (7, 1), (7, 0), (7, 1), (7, 0), (6, 0), (7, 0), (7, 1), (7, 0), (7, 1), (7, 0), (6, 0), (7, 0), (7, 1), (7, 0), (7, 1), (7, 0), (7, 1), (7, 0), (6, 0), (5, 0), (5, 1), (5, 0), (5, 1), (5, 0), (6, 0), (7, 0), (7, 1), (7, 0), (6, 0), (7, 0), (7, 1), (7, 2), (7, 1), (7, 2), (7, 3), (6, 3), (6, 4), (6, 3), (6, 4), (5, 4), (5, 5), (5, 6), (6, 6), (7, 6), (6, 6), (5, 6), (5, 7), (6, 7), (5, 7), (5, 6), (6, 6), (5, 6), (6, 6), (6, 7), (6, 6), (7, 6), (6, 6), (6, 7), (7, 7)]

Total path cost using Genetic Algorithm: 6080

**Part f(i): Time and Space complexity**

GBFS Algorithm Time and Space Complexity:

Total nodes expanded: 45

Time Complexity of GBFS: 4782969

Worst case Space Complexity of GBFS: 6

Genetic Algorithm Time and Space Complexity:

Memory Computation for Genetic Search is : 1000

Time Complexity of Genetic Search Algorithm: 1095000.0

Space Complexity of Genetic Search Algorithm: 11790

**Part f (ii): Comparison analysis using graphs**

A diagram of a graph

Description automatically generatedA grid with colored lines

Description automatically generated

**Part f (iii): Reference images – OUTPUT**

City map grid, start position and goal node

A screenshot of a computer program

Description automatically generated

GBFS algorithm statistics

A screenshot of a computer

Description automatically generated

Genetic algorithm statistic

A close up of numbers and symbols

Description automatically generated

Part f (iv): GLOSSARY

GIT Reference:

<https://github.com/vatsaaa/mtech/blob/main/semester_1/03_assignments/aci/ACI_ASSIGNMENT_1/ipynb/ACI_Assignment_1_SolutionTemplate_PS_1.ipynb>

Part g: Resources List