**ABSTRACT**

As the world is booming with the amazing technologies, introduced by **Artificial Intelligence** **(AI)**, different industries are getting benefits from it day by day. Some industries like **Smart Manufacturing**, **Robotics,** and **Game Development** are taking advantage of such technologies from the very beginning. Talking about AI in the Game Development industry, we use different types of AI-based **Search Algorithms** for finding the shortest path from the start point to the destination point.

Let’s say, there is a robot, who is trapped inside a Maze. To figure its way out, he’ll require a Search Algorithm for finding the shortest path to its destination. This scenario can be considered a **“Maze Traversal Problem”**.

The goal of the Search Algorithm is to provide the robot with the path which is the shortest and also takes less time to compute (**Time Complexity**).

The usage of an inappropriate algorithm can increase the processing time for calculating the shortest path from the current point for the robot. There might be some path that will end up at a different point, rather than the actual destination.

For avoiding this and getting our robot to the final destination with the shortest path and shortest time, we tested different types of Search Algorithms and tried to find out the best among them.

Here, we used:

* **DFS (Depth First Search) Algorithm**
* **BFS (Breadth First Search) Algorithm**
* **DFID (Depth First Iterative Deepening) Algorithm**

Moreover, the usage of the **File System** for providing the **Input**, containing the first line as the **Choice (0, 1, or 2)** for choosing, which algorithm to use by the robot, and a **Maze Map**, is adopted.

Maze Map contains a combination of **Plus (+)**, **Minus (-)**, **Asterisk (\*), and Pipe (|)** symbols to form the map structure. Then, depending on the selected algorithm the robot finds the best path for reaching its destination.

After testing out each of the 3 mentioned algorithms, we found out that the best algorithm is DFID for finding the shortest path to the destination in the shortest possible time.

**1. Introduction**

In this world, the term **Artificial Intelligence** is repetitively used everywhere. The major advantage it provides over **procedural programming** is that it tries to **imitate human behavior or actions**. These AI-Based Learning models enable us to develop intelligent machines which can **act, think, and make decisions like humans do**.

Let’s not forget to consider the subsets of AI, which is also very popular these days, **Machine Learning (ML)**, which also contains another popular subset called **Deep Learning (DL)**.

**Machine Learning (ML)** is divided into 4 main categories:

* **Supervised Machine Learning**
* **Unsupervised Machine Learning**
* **Semi-Supervised Machine Learning**
* **Reinforcement Machine Learning**

**Deep Learning (DL)** contains many categories:

* **RNN (Recurrent Neural Network)**
* **CNN (Convolutional Neural Network)**
* **NLP (Natural Language Processing), etc**

Further, we can develop an **Expert System** by using the power of AI and its subsets to solve a particular problem statement. Some of the major fields in AI that is excelling are **E-Commerce, Stock Market, Game Development, Automation, Smart Manufacturing Robotics, etc**.

Considering in mind the **Gaming Industry**, many AI Algorithms are used to optimize the steps taken by the **NPCs (Non-Playable Players),** present in the game. **AI-based Game Engines like Alpha-GO** is trained for millions of generations, to beat world champions and many more such extraordinary examples.

Coming back to our **“Maze Game”**, we need to find the best path from the start to the goal, considering the fact, that it needs to take less amount of time to compute the next step and the shortest possible path. Hence, we can say that it is a complex and perplexing network of passages. Some Search Algorithms are used for finding the best possible solution for the Maze Game.

**Elements of the Maze Map:**

* **Plus (+)** – It is used to separate the map in a row-wise manner.
* **Minus (-)** – it is used to create the horizontal walls in the map.
* **Asterisk (\*)** – It is used to specify the destination/goal point.
* **Pipe (|)** – It is used to create the vertical walls on the map.
* **Cell** – It is the most fundamental unit of the maze map, it can contain walls/obstacles or it can be free, giving the way for the robot to move.
  1. **Search Algorithms**

It is the heart of **Path Finding Algorithms**, these types of algorithms are widely used in **Maps (Navigation Systems), Video Games, Robotics, and Metabolic pathways**.

It finds the shortest route between the two points, keeping in mind it needs to avoid obstacles if any, some examples can be **Traffic Routing, Maze Navigation, and Robot Path Planning**.

Some of the major challenging aspects of pathfinding in Video Games is figuring out how to avoid obstacles while looking for the most efficient way to reach the goal.

There are 2 types of search algorithms **Uninformed/Blind search Algorithm and the Informed Search Algorithm**.

In this case, we have only tested the Uninformed Search Algorithms.

Let’s discuss the types of Search Algorithms…

* + 1. **Uninformed/Blind Search Algorithm**

This is also called **Blind Search** because it works with no information about the search space, other than to distinguish the goal state from all the others. It has no preference on which state should be expanded next.

**Some types of Uninformed Search Algorithm are:**

* **Breadth-First Search (BFS):** This type of search algorithm is used to search a tree or graph data structure for a note that meets the set of criteria. Here the algorithm starts from the start node and works its way to all the successor nodes at the current level before moving to the nodes in the next level. It uses **Queue Data Structure (FIFO) First-In-First-Out**.
* **Depth-First Search (DFS):** This type of search is used in **Topological Sorting, Scheduling Problems, Cycle Detection in graphs, and Solving Puzzles** with only one solution. It starts searching from the start node and explores each path to its greatest depth node before moving to the next path. It uses **Backtracking**, which addresses issues in a loop by seeking to systematically create a solution. It uses **Stack Data Structure (LIFO) Last-In-First-Out**.
* **Depth-First Iterative Deepening (DFID):** # DFID Here

**Some common characteristics:**

* **Problem Graph (Maze with a Start Node and a Goal Node)**
* **Strategy (The Approach)**
* **Data Structure (Holding All Potential States)**
* **Tree (Result of Travelling)**
* **Solution (Sequence)**
  + 1. **Informed Search Algorithms:**

In this type of Search Algorithm, the algorithm has prior information about the goal state with the aid of **Heuristics**, this aids to be more effective. It uses the idea of **Heuristic Search**.

As in this case, we are not using **Informed Search Algorithm**, so we’ll skip the types of Informed Search Algorithms…

**2. Research Methodology**

The main goal is to find the shortest path in the shortest possible time. Here, we compared 3 **Uninformed Search Algorithms (BFS, DFS, and DFID)** on a maze map and generated 3 separate outputs for each algorithm applied. After passing the input maze to 3 different algorithms, we can determine which algorithm is best to use in **robot pathfinding in a maze**.

**Here, we measure the algorithm’s performance on two parameters:**

1. **Time Complexity:** The Compute Time used for finding the solution.
2. **Path Length:** The length, that algorithm required to reach the destination
   1. **Depth First Search Algorithm**

***Pseudocode***

* Append the start cell in the Visited and Unvisited Array.
* Iterate until we reach the goal or unvisited empty cell.
  + “unvisited.pop()” – removing the last cell
  + Directions (East, West, North, and South)
  + Finding the next possible cell
  + Do nothing, if already explored
  + Otherwise, append the next cell to both Explored and Frontier
  1. **Breadth First Search Algorithm**

***Pseudocode***

* Append the start cell in the Visited and Unvisited Array
* Iterate until we reach the goal or unvisited empty cell.
  + “unvisited.pop(0)” – removing the first cell
  + Directions (East, West, North, and South)
  + Finding the next possible cell
  + Do nothing, if already visited
  + Otherwise, append the next cell to both Visited and Unvisited
  1. **Depth-First Iterative Deepening**

***Pseudocode***

**# DFID**

1. **Results & Discussion**

**APPENDIX**

**Code**

import sys

sys.setrecursionlimit(9000)

choice = -1

graph= []

no\_of\_states = 0

length\_of\_path = 1

closed = []

open\_list = []

obj\_list = []

bfs\_count = 0

class\_count = 0

m = 0                                      *#initial number or row for the robot*

output = open("output2.txt","w")           *#output file*

*#Taking the Input as the grid for the robot to walk along with the type of algorithm in the first line*

file\_path = "input.txt"

file = open(file\_path,"r")

choice = file.readline()

choice = int(choice[0])

for i in file.readlines():

    if(len(i) != 0 and i!=" " and i!="\n"):

        m = m+1

        if(i[len(i)-1] == "\n"):

            i = i[:len(i)-1]

        graph.append(list(i))

n = len(graph[0])                          *#initial number or columns for the robot*

*#class to store coordinate and prev, next in the path*

class path:

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

        self.coordinate = coordinate

        self.prev = None

        self.next = None

current\_position=[0,0]   *#variable to store the current position*

head = path([0,0])

*#close neighbors*

def movegen(pos):

    x,y =pos

    global closed, open\_list, graph, m, n

    list = []

    if(x+1<m and (graph[x+1][y] == ' ' or graph[x+1][y]=='\*') and [x+1,y] not in open\_list):

        list.append([x+1,y])

    if(x-1>0 and (graph[x-1][y] == ' ' or graph[x-1][y]=='\*') and [x-1,y] not in open\_list):

        list.append([x-1,y])

    if(y+1<n and (graph[x][y+1] == ' ' or graph[x][y+1]=='\*') and [x,y+1] not in open\_list):

        list.append([x,y+1])

    if(y-1>0 and (graph[x][y-1] == ' ' or graph[x][y-1]=='\*') and [x,y-1] not in open\_list):

        list.append([x,y-1])

    return list

*#Function to check if we have reached the goal state*

def goaltest(pos):

    global graph

    x,y =pos

    if(graph[x][y] == '\*'):

        return True

    else:

        return False

*#Function to print the output maze*

def print\_path(pointer):

    global graph, m, n, length\_of\_path, no\_of\_states

    graph[0][0]='0'

    while(pointer.prev != None):

        x = pointer.coordinate

        graph[x[0]][x[1]] = '0'

        length\_of\_path = length\_of\_path +1

        pointer = pointer.prev

    no\_of\_states = str(no\_of\_states) + "\n"

    length\_of\_path = str(length\_of\_path) + "\n"

    output.write(no\_of\_states)

    output.write(length\_of\_path)

    for i in range(m):

        string=""

        for j in range(n):

            string+=graph[i][j]

        string += "\n"

        output.write(string)

    sys.exit(0)

*#Depth First Search*

def dfs(pointer):

    global closed,open\_list, head, current\_position, no\_of\_states

    no\_of\_states = no\_of\_states +1

    current\_position = pointer.coordinate

    closed.append(current\_position)

    if(goaltest(current\_position)):

        print\_path(pointer)

    lst = movegen(current\_position)

    for x in lst:

        open\_list.append(x)

    for x in lst:

        if x not in closed:

            y = path(x)

            y.prev = pointer

            pointer.next = y

            dfs(y)

*#Breadth First Search*

def bfs(pointer):

    global closed, current\_position, bfs\_count, class\_count, obj\_list, no\_of\_states

    obj\_list.append(path([0,0]))

    while(bfs\_count != len(closed)):

        no\_of\_states = no\_of\_states + 1

        current\_position = obj\_list[bfs\_count].coordinate

        if(goaltest(current\_position)):

            print\_path(obj\_list[bfs\_count])

        lst = movegen(current\_position)

        for x in lst:

            if x not in closed:

                class\_count = class\_count + 1

                closed.append(x)

                obj\_list.append(path(x))

                obj\_list[class\_count].prev = obj\_list[bfs\_count]

                obj\_list[bfs\_count].next = obj\_list[class\_count]

        bfs\_count = bfs\_count + 1

*#Depth First Iterative Deepening*

def dfid(pointer, d):

    if(d==0):

        return

    global closed, current\_position,limit, no\_of\_states, count, condition

    no\_of\_states = no\_of\_states +1

    count = count+1

    if(count > limit):

        condition = False

    current\_position = pointer.coordinate

    closed.append(current\_position)

    if(goaltest(current\_position)):

        print\_path(pointer)

    lst = movegen(current\_position)

    for x in lst:

        open\_list.append(x)

    for x in lst:

        if x not in closed:

            y = path(x)

            y.prev = pointer

            pointer.next = y

            dfid(y,d-1)

*#MAIN*

if choice==0:    *#BFS*

    closed.append(current\_position)

    bfs(head)

elif choice==1:  *#DFS*

    dfs(head)

elif choice==2:  *#DFID*

    depth = 1

    limit = m\*n

    condition = True

    while(condition):

        count = 0

        dfid(head,depth)

        depth = depth + 1

        closed = []

        open\_list =[]

else:

    print("\nEnter a valid choice\n")