AI Project
MAZE Application

Team Members:

Name: Mohamed Ismail Elsayed
ID: 32020246

Name: Mohamed Samy Abdul Azim
ID: 32020265

Name: Yasmine Ahmed Mohamed
ID: 32020165

Name:Salwa Ahmed Mohamed
ID:32021012

Name: Shimaa abdelbadea
ID:32020098

Under Supervision:

Doctor: Karim Badawi
ENG: Alaa Ayman

Table of Content:

Table of Figure:

Introduction

In this Jupyter Notebook, we will be discussing several search algorithms applied on a maze using the pyamaze module. The main idea of this module is to assist in creating customizable random mazes and to apply search algorithms with ease. By using this module, you don't need to program the GUI, and you don't need the Object-Oriented Programming since the module will provide you the support. This module uses the Tkinter GUI framework which is built-in in Python and you don't need to install any framework to use this module.

Python-Maze-World-pyamaze

The **pyamaze** module is designed to generate customizable random mazes and apply different search algorithms efficiently. This module uses the Tkinter GUI framework, which is built-in in Python, so you don't need to install any framework to use this module.

Graphs

Here's an example of a maze generated using `pyamaze`:

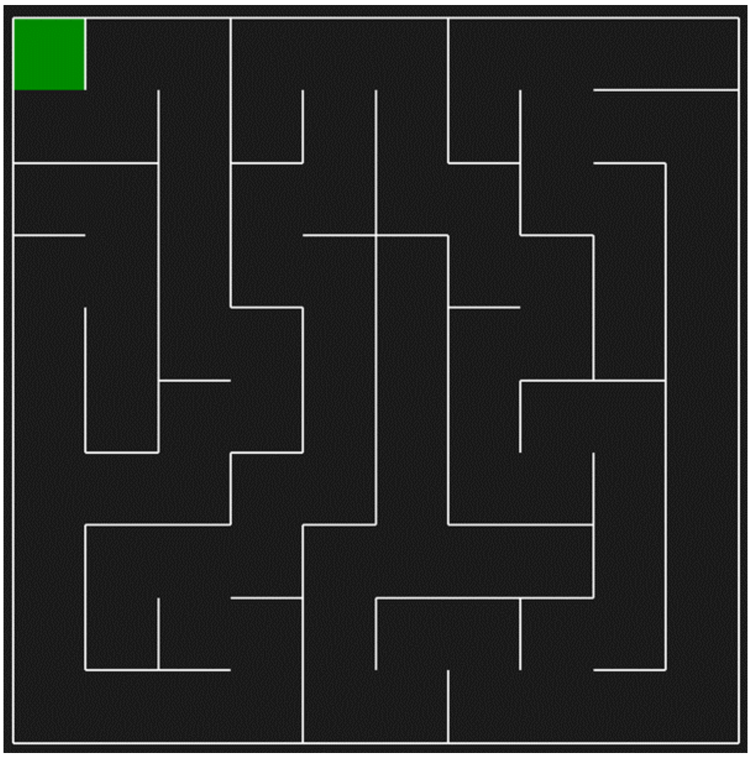


Figure 1 Maze

Depth-First Search Algorithm

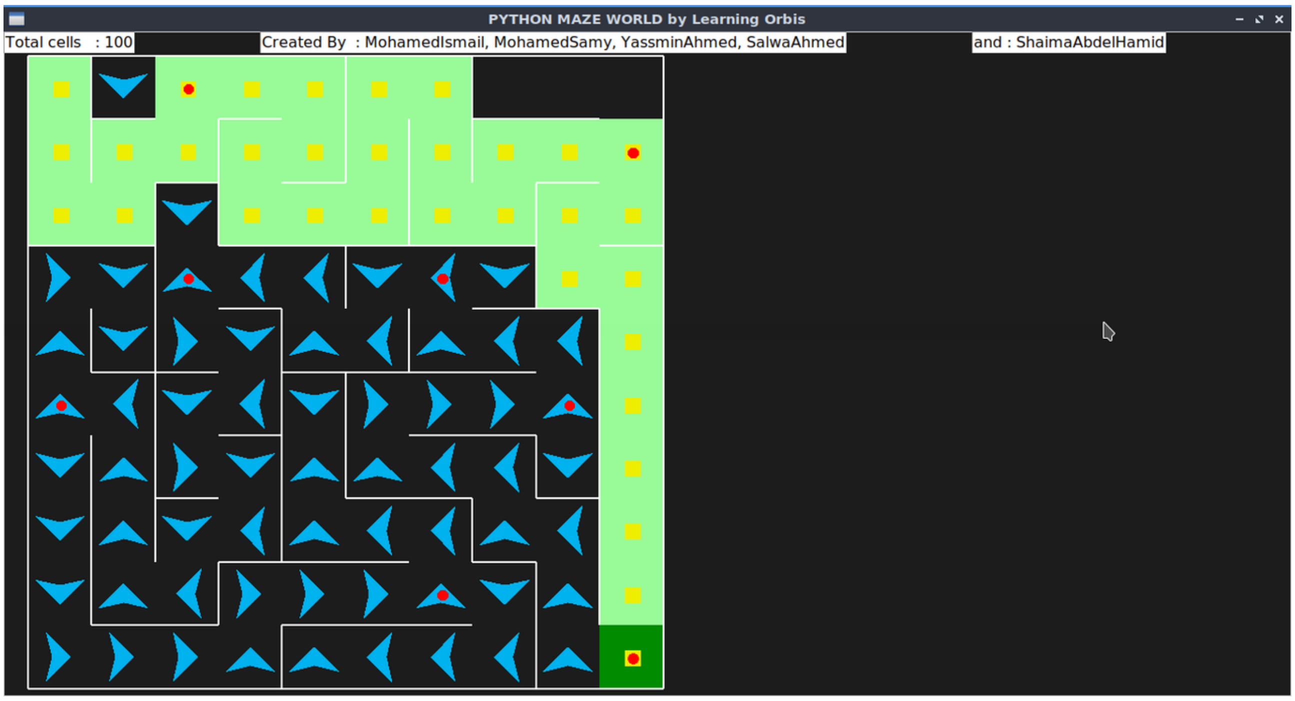
This is a Python program that implements the Depth-First Search (DFS) algorithm on a maze. The program uses the pyamaze module to create and display the maze, the agent, and the text labels.

How it works

The program defines a function called DFS that takes two arguments: a maze object and an optional starting cell coordinates. If the starting cell is not given, it defaults to the bottom-right corner of the maze.

The function uses two lists to keep track of the explored and frontier cells, and two dictionaries to store the paths from each visited cell to its parent cell and from the starting cell to each visited cell. The function also marks the cells that have more than one possible direction to go as decision points.

The function iterates through the frontier cells using a stack data structure (last-in first-out), and checks if the current cell is the goal cell. If not, it adds the adjacent cells that are not walls and not explored to the frontier and explored lists, and updates the path dictionaries. The function returns the list of cells visited during the search, and the two path dictionaries.



Total cells : 100

Created By : MohamedIsmail, MohamedSamy, YassminAhmed, SalwaAhmed

and : ShaimaAbdelHamid

Figure 2 DFS

DFS Pseudocode

Order = W -> N -> S -> E

```
1. Push the start cell onto the Frontier with a priority equal to the heuristic distance.
2. Initialise an empty Explored list.
3. Repeat until the goal is reached or the Frontier is empty:
    A. Pop the cell with the highest priority from the Frontier and set it as the current cell.
    B. If the current cell is the goal cell, return the path to the goal and the number of steps taken by the agent.
    C. Add the current cell to the Explored list.
    D. For each direction (ESNW):
        a. Compute the next possible cell in the given direction.
        b. If the next possible cell is a wall, skip to the next direction.
        c. If the next possible cell is already in the Explored list, skip to the next direction.
        d. Update the cost of the path from the start to the next possible cell.
        e. Calculate the heuristic distance of the next possible cell to the goal using the Manhattan distance formula.
        f. Set the priority of the next possible cell in the Frontier to the sum of the cost and the heuristic distance.
        g. Update the parent of the next possible cell to the current cell.
4. If the goal was not found, return the list of cells visited during the search and the number of steps taken by the agent.
```

Breadth-First Search (BFS) Algorithm

This is a Python program that implements the Breadth-First Search (BFS) algorithm on a maze. The program uses the pyamaze module to create and display the maze, the agent, and the text labels.

How it works

The program defines a function called BFS that takes two arguments: a maze object and an optional starting cell coordinates. If the starting cell is not given, it defaults to the bottom-right corner of the maze.

The function uses two lists to keep track of the explored and frontier cells, and two dictionaries to store the paths from each visited cell to its parent cell and from the starting cell to each visited cell. The function also marks the cells that have more than one possible direction to go as decision points.

The function iterates through the frontier cells using a queue data structure (first-in first-out), and checks if the current cell is the goal cell. If not, it adds the adjacent cells that are not walls and not explored to the frontier and explored lists, and updates the path dictionaries. The function returns the list of cells visited during the search, and the two path dictionaries.

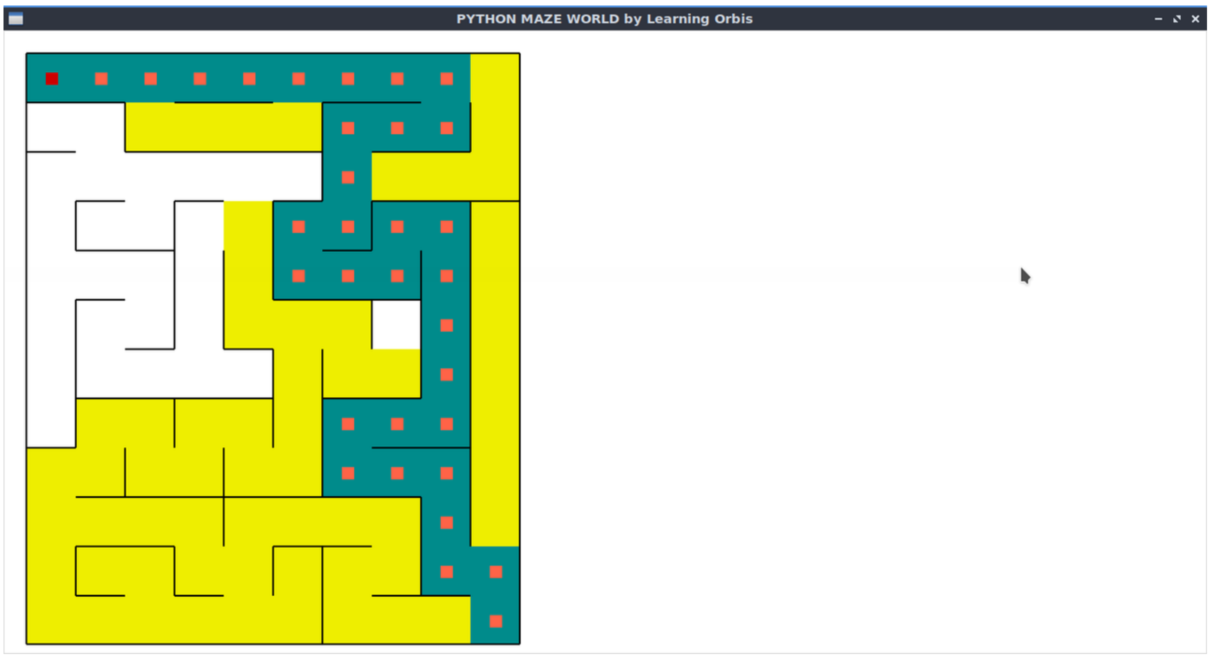


Figure 3 BFS

BFS Pseudocode

Order = W -> N -> S -> E

```
1. Add the start cell to the Frontier queue.
2. Initialise an empty Explored set.
3. Repeat until the goal is reached or the Frontier is empty:
   A. Dequeue the first cell in the Frontier and set it as the current cell.
   B. If the current cell is the goal cell, return the path to the goal and
      the number of steps taken by the agent.
   C. Add the current cell to the Explored set.
   D. For each direction (ESNW):
      a. Compute the next possible cell in the given direction.
      b. If the next possible cell is a wall, skip to the next direction.
      c. If the next possible cell is already in the Explored set, skip
         to the next direction.
      d. If the next possible cell is not in the Frontier queue, add it to
         the Frontier queue.
      e. Update the parent of the next possible cell to the current
         cell.
4. If the goal was not found, return the list of cells visited during the search
   and the number of steps taken by the agent.
```

Uniform-Cost Search Algorithm

This is a Python program that implements the Uniform-Cost Search (UCS) algorithm on a maze. The program uses the pyamaze module to create and display the maze, the agent, and the text labels.

How it works

The program defines a function called UCS that takes two arguments: a maze object and an optional starting cell coordinates. If the starting cell is not given, it defaults to the bottom-right corner of the maze.

The function uses two lists to keep track of the explored and frontier cells, and two dictionaries to store the paths from each visited cell to its parent cell and the cost to reach the visited cell. The function also marks the cells that have more than one possible direction to go as decision points.

The function iterates through the frontier cells using a priority queue data structure, which sorts the cells by their accumulated cost. The accumulated cost of a cell is the cost to reach its parent cell plus the cost to move from the parent cell to the current cell. If a cell has already been visited and the new accumulated cost is lower than the stored cost, the function updates the path dictionary and the accumulated cost.

The function continues to explore the frontier cells until it reaches the goal cell or until there are no more cells to explore. The function returns the list of cells visited during the search, and the two path dictionaries.

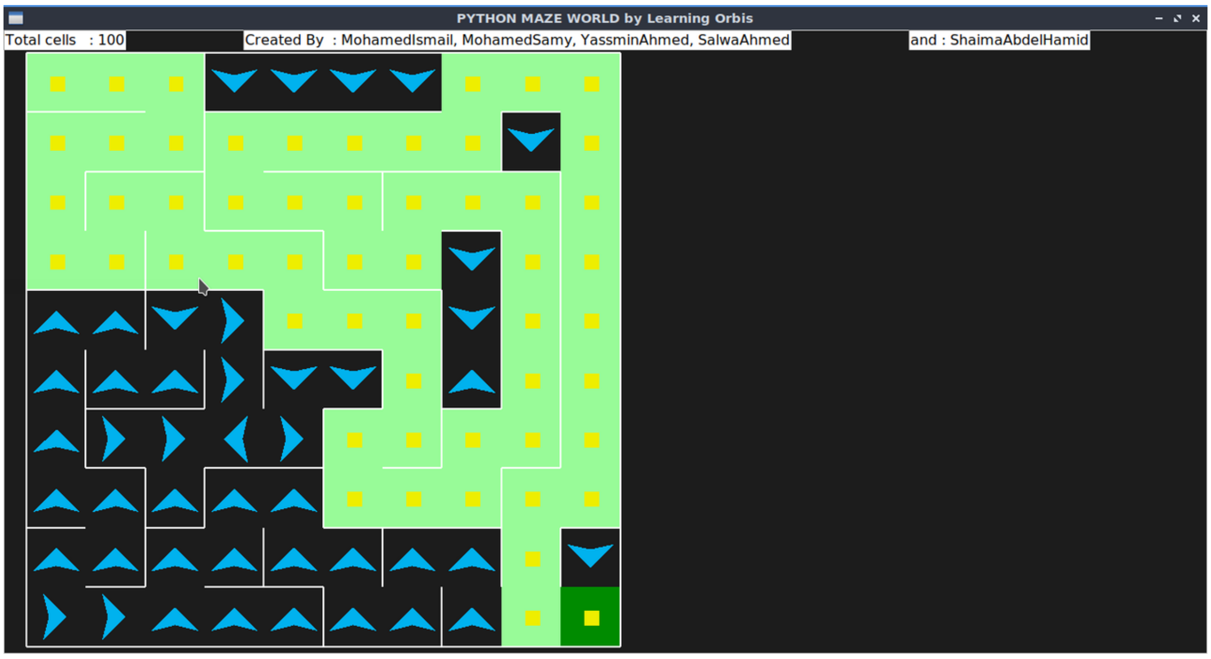

PYTHON MAZE WORLD by Learning Orbis
Total cells : 100
Created By : MohamedIsmail, MohamedSamy, YassminAhmed, SalwaAhmed
and : ShaimaAbdelHamid

Figure 4 UCS

UCS Pseudocode

```
1. Initialize Frontier with the start cell and its path cost 0.
2. Initialise an empty Explored list.
3. While Frontier is not empty:
    A. current_cell, current_cost = Frontier.pop the cell with the lowest cost.
    B. If current_cell is the goal cell, return the path from the start to the goal.
    C. Add current_cell to the Explored list.
    D. For each direction (ESNW):
        a. child_cell = Next possible cell
        b. child_cost = Cost to move from current_cell to child_cell
        c. total_cost = current_cost + child_cost
        d. If child_cell is already in the Explored list with a lower cost, do nothing.
        e. Otherwise, if child_cell is not in the Frontier or total_cost is lower than the stored cost for child_cell:
            i. Add child_cell to the Frontier with its total_cost.
            ii. Update the path from the start to child_cell with the path from the start to current_cell plus the move from current_cell to child_cell.
4. Return None if no path is found.
```

A* Search Algorithm

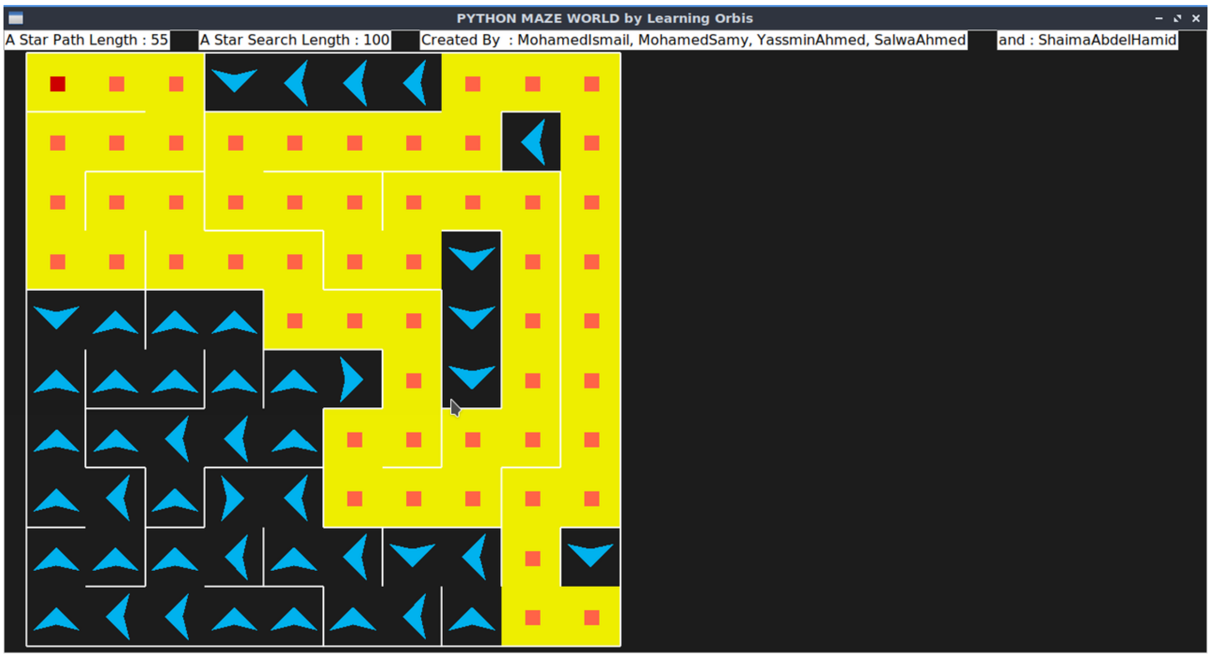
This is a Python program that implements the A* search algorithm on a maze. The program uses the pyamaze module to create and display the maze, the agent, and the text labels.

How it works

The program defines a function called A_star that takes two arguments: a maze object and an optional starting cell coordinates. If the starting cell is not given, it defaults to the bottom-right corner of the maze.

The function uses two lists to keep track of the explored and frontier cells, and two dictionaries to store the paths from each visited cell to its parent cell and from the starting cell to each visited cell. The function also calculates the heuristic distance of each cell to the goal cell using the Manhattan distance formula.

The function iterates through the frontier cells using a priority queue data structure, sorted by the sum of the cost of the path from the start to the current cell and the estimated cost of the path from the current cell to the goal cell. If the current cell is the goal cell, the function returns the path from the start to the goal, and the number of steps taken by the agent. If not, it adds the adjacent cells that are not walls and not explored to the frontier and explored lists, and updates the path and heuristic dictionaries. The function returns the list of cells visited during the search, and the two path dictionaries.



```
A Star Path Length : 55   A Star Search Length : 100   Created By  : MohamedIsmail, MohamedSamy, YassminAhmed, SalwaAhmed   and : ShaimaAbdelHamid
```

Figure 5 A Star

A* Pseudocode

```
1. Set the order of exploration to W, N, S, E.
2. Push the start cell into the Frontier with priority equal to its heuristic distance to the goal.
3. Initialise an empty Explored list.
4. Repeat until the goal is reached or the Frontier is empty:
    A. Pop the cell with the highest priority from the Frontier and set it as the current cell.
    B. If the current cell is the goal cell, return the path to the goal and the number of steps taken by the agent.
    C. Add the current cell to the Explored list.
    D. For each direction in the order of exploration (ESNW):
        a. Compute the child cell in that direction.
        b. If the child cell is a wall, skip to the next direction.
        c. If the child cell is already in the Explored list, skip to the next direction.
        d. Update the cost of the path from the start to the child cell as the cost of the path from the start to the current cell plus the cost of the step from the current cell to the child cell.
        e. Calculate the heuristic distance of the child cell to the goal using the Manhattan distance formula.
        f. Set the priority of the child cell in the Frontier to the sum of the cost and the heuristic distance.
        g. Update the parent of the child cell to the current cell.
5. Return the list of cells visited during the search, and the number of steps taken by the agent if the goal was not found.
```

Conclusion

Pyamaze is a Python module that allows for the creation and display of mazes, agents, and labels. It is a useful tool for visualising various search algorithms and can aid in the understanding of their mechanics.

`Depth-First Search (DFS)`, **`Breadth-First Search (BFS)`**, **`Uniform-Cost Search (UCS)`**, and **`A*`** are all popular search algorithms used in computer science. DFS and BFS are uninformed search algorithms, which means that they do not use any heuristic information to guide the search. UCS is a variant of BFS that takes into account the cost of each step, while A* combines both the cost and heuristic information to make informed decisions during the search.

Each algorithm has its own strengths and weaknesses and is suited for different types of problems. DFS is useful for searching through large, unstructured spaces and finding paths that are deep in the search tree. BFS is best for finding the shortest path between two points, while UCS is useful for finding the path with the lowest cost. A* is generally considered the best search algorithm as it combines the benefits of both uninformed and informed search.

Overall, understanding the differences between these search algorithms and when to use them can be a valuable skill for solving complex problems in computer science.