National University of Computer and Emerging Sciences

Lab Manual 03
Artificial Intelligence Lab

Instructor: Mariam Nasim

Implementation of Blind Search Algorithm

Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental search algorithms used in Artificial Intelligence and Graph Theory. They are commonly used for pathfinding, connectivity checking, and solving problems in **grids**, **mazes**, and **graphs**.

1. Depth-First Search (DFS)

DFS explores as far as possible along one branch before backtracking. It is often implemented using recursion or a stack.

- DFS is useful for problems where you need to explore all possible paths (e.g., solving mazes, finding connected regions).
- It can be implemented using recursion (implicit stack) or an explicit stack.
- DFS is preferred when searching for a solution in deep structures (e.g., decision trees).
- DFS keeps moving deeper into one direction before backtracking.
- Time Complexity: O(N × M) (where N and M are grid dimensions).
- Space Complexity: O(N × M) (for visited matrix & recursion stack).

2. Breadth-First Search (BFS)

BFS explores all neighbors at the current level before moving to the next level. It is implemented using a queue (FIFO order).

- BFS is preferred for finding the shortest path in an unweighted grid.
- It is implemented using a queue to ensure all neighbors are explored before moving deeper.
- BFS is useful in pathfinding, connected component detection, and flood-fill algorithms.
- Time Complexity: O(N × M)
- Space Complexity: O(N × M)

Part 2: Introduction to Numpy

Scientific Python code uses a fast array structure, called the numpy array. Those who have worked in Matlab will find this very natural. For reference, the numpy documention can be found here.

Let's make a numpy array.

```
my_array = np.array([1, 2, 3, 4])
my_array
```

Numpy arrays are listy! Below we compute length, slice, and iterate.

```
print(len(my_array))
print(my_array[2:4])
for ele in my_array:
    print(ele)
```

In general you should manipulate numpy arrays by using numpy module functions (`np.mean`, for example). This is for efficiency purposes, and a discussion follows below this section.

You can calculate the mean of the array elements either by calling the method `.mean` on a numpy array or by applying the function np.mean with the numpy array as an argument.

```
print(my_array.mean())
print(np.mean(my_array))
```

The way we constructed the numpy array above seems redundant..after all we already had a regular python list. Indeed, it is the other ways we have to construct numpy arrays that make them super useful.

There are many such numpy array *constructors*. Here are some commonly used constructors. Look them up in the documentation.

```
np.ones(10) # generates 10 floating point ones
```

Numpy gains a lot of its efficiency from being typed. That is, all elements in the array have the same type, such as integer or floating point. The default type, as can be seen above, is a float of size appropriate for the machine (64 bit on a 64 bit machine).

```
np.dtype(float).itemsize # in bytes
```

```
np.ones(10, dtype='int') # generates 10 integer ones
```

```
np.zeros(10)
```

Often you will want random numbers. Use the `random` constructor!

```
np.random.random(10) # uniform on [0,1]
```

You can generate random numbers from a normal distribution with mean 0 and variance 1:

```
normal_array = np.random.randn(1000)
print("The sample mean and standard devation are %f and %f, respectively." %(np.mean(normal_array), np.std(normal_array)))
```

2D arrays

Similarly, we can create two-dimensional arrays.

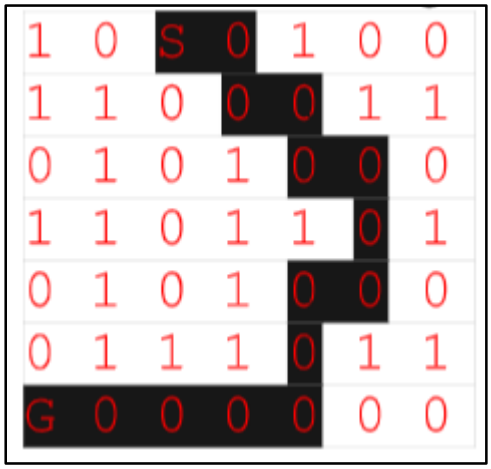
```
my_array2d = np.array([ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ])

# 3 x 4 array of ones
ones_2d = np.ones([3, 4])
print(ones_2d)
# 3 x 4 array of ones with random noise
ones_noise = ones_2d + .01*np.random.randn(3, 4)
print(ones_noise)
# 3 x 3 identity matrix
my_identity = np.eye(3)
print(my_identity)
```

Like lists, numpy arrays are 0-indexed. Thus we can access the nth row and the mth column of a two-dimensional array with the indices $[n-1, m-1]$.

You can read further on the following link:
https://www.w3schools.com/python/numpy/default.asp

Task 1

Your task is to implement a cube solver using blind search algorithms (**Depth-First Search-DFS** and **Breadth-First Search-BFS**). You are given a 2D array representing a cube, where 0s represent open spaces and 1s represent walls. You need to find a path from the starting point to the goal point, where the starting point is at the top left corner of the cube (0,0) and the goal point is at the bottom right corner of the cube (n-1, m-1) where n and m are the dimensions of the cube. Following is a sample Cube.



- **Reading the cube from a text file:** Write a function to read the cube from a text file. The function should take a filename as input and return a 2D array representing the cube. The cube can be represented using a binary matrix, where 0 represents an empty cell and 1 represents a wall. (5 marks)
- **Implementing DFS and BFS**: Write a function for DFS and BFS algorithms. Both functions should take the cube as input and return a path to the goal state if there exists one, and -1 if there is no path to the goal state. Students should use the appropriate data structures and algorithms to implement DFS and BFS. (8+8)
- **Returning the path to the goal state:** The function should return a list of coordinates representing the path from the starting point to the goal state. If there is no path, the function should return -1. (4)

Note: It is important to note that a cube can have multiple goal states, which means that the DFS and BFS algorithms may produce different paths to reach different goal states. As a result, it is possible that the DFS algorithm may output one path while the BFS algorithm outputs another.

Task 2 (5 marks)

Write a Python function using NumPy to detect all local minima in a 2D NumPy array. A local minima is defined as a value that is smaller than its four neighbors (up, down, left, and right).

- We use a 3×3 window to check every element (excluding borders).
- The center of the window is the element we check.
- We compare it with its 4 neighbors (top, bottom, left, right).
- If it's smaller than all four, it is a local minimum.

```
# Sample 2D NumPy array
matrix = np.array([
    [9, 8, 7, 6, 5],
    [5, 3, 4, 7, 8],
    [6, 7, 2, 8, 9],
    [8, 6, 5, 4, 3],
    [9, 7, 6, 5, 4]
])
```

Output:

Local minima positions : (2,2) (3,3)

Minima at position (2,2): 3

Minima at position (3,3) : 2

Submission Instructions:

- Rename your Jupyter notebook to your "roll number_Name" and download the notebook as .ipynb extension.
- To download the required file, go to File->Download .ipynb
- Only submit the .ipynb file. DO NOT zip or rar your submission file
- Submit this file on Google Classroom under the relevant assignment.
- All outputs should be displayed properly.
- Late submissions will NOT AT ALL be accepted.