Applied AI: First assignment

of Riccardo Storchi

Introduction:

The assignment consists in graph, graph search and implementation. As requested I have implemented three algorithms: BFS,DFS,A*.

Setup of the environment:

As first thing it I had to transform this graph into a matrix so an array of array. In this binary matrix the 1 represents walls and 0 a free space where it is possible to move. After that I made some checks on the starting point and ending point to see if they were inside the matrix and if they do not coincide.
To be efficient and respect the standards of programming I decided to divide the code into four files: main.py , BFS.py, DFS.py, A_star.py . in the first one in contained the validation checks of the given points and the functions for printing the result and the function calls, in the others there are the algorithms requested and that I look forward to explain in details in the next paragraphs.

Algorithms:

- **Breadth First Search Algorithm (BFS):** This algorithm finds a path between two nodes by going down one step and then immediately backtracking, so it explores the tree level by level and it guarantees the shortest path in an unweighted graph like the one we have.
  I handled the implementation by creating a queue using the library deque (double ended queue), that queue contains row, col, distance from the start and the path followed and it's processed in FIFO order (First-In-First-Out). Then I made a variable for keeping track of visited nodes. The algorithm then tries to understand if it has reached the end, if no it continues by looking at the neighbours to find a path until the queue is empty. As last thing there is the boundaries check to see if the values are inside the initial matrix "maze", is they are, the path is updated.
  The algorithm reaches the shortest path in 27 jumps.

- **Depth first search (DFS):** This algorithm explores in depth each branch before starting backtracking, that's why it can reach a nonoptimal solution before the optimal.
  It's important to point out that if we change the first movement in the neighbours exploration in the matrix, we reach different results like 47 or 31 jumps in this case. This algorithm uses a stack data structure that keeps track of visited nodes for efficiency. In the algorithm works as LIFO(Last-In-First-out) with the function pop from the stack. it checks is it has reached the end and in case returns the path. Then there is the neighbours exploration and boundaries checks in order to find out if the node is inside the matrix "maze" boundaries. If it is it returns a path, otherwise it returns -1 as in the other algorithms.
  The downside of this it's necessary to change the algorithm and make it remember the shortest path found by making continuous comparison with the new path found in order to discover the shortest one. But by doing this the algorithm increases its complexity both in time and space domain. In this implementation, this feature was not implemented.

- **A* search algorithm:** Compared to the Dijkastra's algorithm the A* algorithm finds the shortest path from a specified node to a specified goal instead of all possible goals. This algorithm uses a priority queue in which we store the nodes to be explored. It compares the cost of the jumps made united to a consistent and admissible heuristic: the Manhattan distance (which complexity is O(1)). This enables the algorithm to reach the optimal solution at a fist run.
  As first thing we define the directions in which we are doing the neighbours search as defined for the other algorithms, then the priority queue is defines in which each node is represented as a triplet: ((cost+Manhattan distance), the actual cost to reach the node from the start, the coordinates of the node).
  The visited set ensures that no node is revisited for efficiency.
  The came_form dictionary tracks the predecessor of each node; this is used to reconstruct the path once reached the end node as shown in the if current==end condition in the code. In this case the last instruction is to reverse the path as we reconstruct it from end to start, and we want it to be from start to end. If no path is found it returns -1.

Complexity table:

Definitions:

- Time complexity: it's a measure of how an algorithm's running time increases with the size of input. It provides a worst-case time required to execute an algorithm as function of the input size.
- Space complexity: measures the amount of memory used by an algorithm with respect to the input size. It represents the worst-case memory consumption as the input size increases.

Clarification:

- In the machine learning community it's common to visit only a shortest part of the graph while in the common algorithm community it's common to visit the graph completely, that's why V(V+E) results in an upper bound true but really wide. It's better to indicate in this case the complexity as $O(b^d)$ where **b** is the branching factor (the average number of successors per state) and **d** the depth of the solution.
  In this case the complexity is not linear in terms of b and d, but it remains linear in terms of V.

	BFS	DFS	A*
Time complexity	O(V+E) or $O(b^{d+1})$	O(V+E)	O(|E|log|V|) or $O(b^d)$
Space complexity	O(V)	O(V)	O(|V|) or $O(b^d)$

Note that the time complexity of A* is O(logV) due to each insertion (push) and removal(pop) from a priority queue. Then this factor is multiplied by E because in the worst-case scenario it must go through each node).

Comparison between the algorithms:

	Strengths	Weaknesses
DFS	- This algorithm is memory efficient, since	- It may not reach the optimal path (it is less

	it must remember only the current path. This results in a lower memory consumption respect the other two considered algorithms. - It has a simple implementation based on the stack. - It can converge to a solution faster than the other two algorithms	efficient than the other algorithms) - It has no heuristic compared to A* - In deep graphs it can get stuck exploring branches far away from the solution
BFS	- It has a systematic exploration - it guarantees the shortest path in an unweighted graph - it is effective in not too deep graphs (as the one in our case)	- It requires more memory than DFS because we need to insert in the queue all the nodes in the depth considered - The exploration of unnecessary nodes make it slows down in deep graphs
A*	- It always finds the shortest path - It is heuristic driven - Overall major efficiency compared to the other two algorithms considered - It is a flexible algorithm	- It has overheads due to the computation of the heuristic - The heuristic must be compliant with the graph to guarantee efficiency - It must store all the nodes in the memory, so like BFS it has a higher memory usage compared to DFS

Conclusions:

The comparison between the algorithms shows how BFS performs better than DFS algorithm that not always is able to find the shortest path even if from a memory point of view is more efficient than BFS. A* search algorithm is more efficient than BFS algorithm while maintaining the same space complexity. In this case the best algorithm in terms of space complexity, time complexity and overall efficiency is A* as studied in the theory.

Outputs :

```
----------BFS algorithm results----------

The path from (0, 1) to (9, 18) is 27 steps.

Path coordinates:
[(0, 1), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (4, 10), (4, 11), (4, 12), (5, 12), (6, 12), (6, 13), (6
, 14), (7, 14), (7, 15), (7, 16), (8, 16), (9, 16), (9, 17), (9, 18)]

Path:
0 * 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0
1 * * * * * 0 0 0 0 0 1 1 1 1 0 1 1 1 0
1 1 1 1 1 * 1 1 1 1 0 0 0 0 0 0 1 1 1 0
1 1 1 1 1 * 1 1 1 1 1 1 0 1 0 0 0 0 0 0
1 0 0 0 0 * * * * * * * * * 1 1 1 1 1 0 1
1 0 1 1 1 1 1 1 1 1 1 * 1 1 1 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 * * * 0 1 1 0 1
1 1 1 1 1 1 1 1 1 1 1 0 1 1 * * * 1 0 1
1 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 * 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 * * * 1


------------------------------------------------------------
----------DFS algorithm results----------

The path from (0, 1) to (9, 18) is 31 steps.

Path coordinates:
[(0, 1), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), (2, 10), (2, 11), (2, 12), (2, 13), (2, 14), (2, 15), (3, 15), (3, 16),
 (3, 17), (3, 18), (4, 18), (5, 18), (6, 18), (7, 18), (8, 18), (8, 17), (8, 16), (9, 16), (9, 17), (9, 18)]

Path:
0 * 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0
1 * * * * * * * * * * 1 1 1 1 0 1 1 1 0
1 1 1 1 1 0 1 1 1 1 * * * * * * 1 1 1 0
1 1 1 1 1 0 1 1 1 1 1 1 0 1 0 * * * * 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 * 1
1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 * 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 * 1
1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 * 1
1 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 * * * 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 * * * 1


------------------------------------------------------------
----------A* algorithm results----------

The path from (0, 1) to (9, 18) is 27 steps.

Path coordinates:
[(0, 1), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), (2, 10), (2, 11), (2, 12), (2, 13), (2, 14), (2, 15), (3, 15), (3, 16),
 (3, 17), (3, 18), (4, 18), (5, 18), (6, 18), (7, 18), (8, 18), (9, 18)]

Path:
0 * 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0
1 * * * * * * * * * * 1 1 1 1 0 1 1 1 0
1 1 1 1 1 0 1 1 1 1 * * * * * * 1 1 1 0
1 1 1 1 1 0 1 1 1 1 1 1 0 1 0 * * * * 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 * 1
1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 * 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 * 1
1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 * 1
1 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 * 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 * 1


------------------------------------------------------------
```

Sources:

https://cs.stackexchange.com/questions/64412/why-is-the-running-time-for-bfs-obd1
https://chatgpt.com/
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
https://en.wikipedia.org/wiki/A*_search_algorithm
https://www.geeksforgeeks.org/a-search-algorithm/
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
+ slides of the course