EARIN MINI-PROJECT 1

solves a maze using A* algorithm

Authors:	Das Devansh
	Xin Yang
Supervisor:	Dr. Marczak Daniel
Date:	2023/3/19

Introduction:

The A* algorithm could be used for finding the shortest path between two points in a graph. In this lab, we implemented the A* algorithm to solve mazes. The algorithm uses a heuristic function to guide its search, which makes it more efficient than other search algorithms, which is characterized by the definition of valuation function. For general heuristic graph search, the node with the lowest f value of the evaluation function is always selected as the extension node. Therefore, f estimates nodes from the point of view of finding a path of minimum cost. Therefore, the value of the evaluation function of each node n can be considered as two components: the actual cost from the start node to node n and the estimated cost from node n to the target node.

Method:

We have implemented the A* algorithm in Python and used it to solve the maze problem.

First, we read the maze from a text file and stored it in a 2D list. We then defined the start and end positions in the maze using a position function.

We use a heuristic function to estimate the cost of reaching the goal from the current position.

At this point, we have two different heuristic functions to choose from to calculate the distance between two points: the Manhattan distance or the Euclidean distance. We then use the A* algorithm to find the shortest path between the start and end positions.

We also define a print_maze function to print the maze and the path to the goal. This function takes the maze and a set of positions as input (the icon for the start position is ▸ and the icon until the end is ★) and prints the maze and the path to the goal marked with the "x" character, the explored area with o, the available space with □ and the wall marked with ■

The main components are as follows：

1. Import the necessary libraries,
    - the code uses the sleep function from the time library to control the display time of the maze
    - the ndarray class from the numpy library to store a two-dimensional array of the maze

    - the heap data structure from the heapq library to manage the open set of the A* algorithm.
    - The clear_output function from the IPython.display library is also used to clear the output and achieve a real-time refresh of the maze state.
2. 2. Define the wall and space symbols in the maze file and set the display time and the different heuristic algorithm [H(n)]. We also define a number of different mazes, including the coordinates of the start and end points.
3. Define node classes that store the coordinates, g-values(the cost of the path from the start node to the current node), h-values(the estimated cost of the path from the current node to the goal node), f-values(the sum of the g-value and the h-value), and parent nodes of a maze lattice.
4. Define a function to read the maze file, read the maze information from the file, and check whether it is legal.
5. Defines a position, converts a coordinate to a node object and checks that the node is valid (i.e. in a maze and not a wall). The function first extracts the x and y coordinates from the coordinates argument and then checks that these coordinates are within the maze boundary and that the node pointed to is not a wall.
6. Define two different computational heuristic functions which calculate the heuristic estimate between two nodes using either the Manhattan distance or the Euclidean distance.
7. Define a function to display the maze, which is used to dynamically display the direction of the maze with different symbol

```
START_SYMBOL = "►"
END_SYMBOL = "★"
PATH_SYMBOL = "x"
CHECKED_SYMBOL = "o"
OPEN_SPACE_SYMBOL = "□"
WALL_SYMBOL = "■"
```

8. main part
   Define the main algorithm to find the shortest path from the start to the end point in a maze.

It takes as input a maze (represented as a NumPy array), a start point, and an end point, and returns a path from the start point to the end point (if one exists).

- start and goal are Node objects for the start and end points, respectively.
- display_time controls how quickly the maze is displayed while the algorithm is running.
- h_type specifies the type of the heuristic function, which can be a Manhattan distance or a Euclidean distance.

The specific algorithm is implemented as follows:

1. First, the starting point is added to the open list, which represents the node to explore. Then, loop through the following steps:
2. Pop the node with the lowest F value from the open list, mark it as explored, and add it to the checked set.
3. If the current node is the end point, it is marked as part of the path, and the final maze matrix and path are output.
4. Otherwise, the children of the current node are generated and added to the children list.
5. Loop over the children in the children list and skip them if they have already been explored. Otherwise, calculate the value of the evaluation function of the child to the end point, update the F value of the child, add it to the open list, and add it to the checked set.
6. Until the open list is empty, or an end point is found, the algorithm stops and returns. The final output path is a sequence of node coordinates in the form of tuples, and the correct path order needs to be output in reverse order.

Steps:

1. Initialize the start and end nodes
2. Create an empty checked set and a to-check list containing only the starting point.
3. The node with the smallest F-value was selected as the current node, and it was removed from the to-be-checked list and added to the checked set.
4. If the current node is the destination, then the path is returned.
5. Otherwise, the neighbor nodes in the four directions of the current node are traversed, if the neighbor node has not been checked, its g value, h value and f value are calculated, and it is added to the list to be checked.
6. Read the maze file, get the starting point and the ending point, and call the A* algorithm function to solve the maze problem.

Differences between two heuristic functions:

一种极端情况，如果 h(n)是 0，则只有 g(n)起作用，此时 A* 算法演变成 Dijkstra 算法，就能保证找到最短路径。

如果 h(n)总是比从 n 移动到目标的代价小（或相等），那么 A* 保证能找到一条最短路径。h(n)越小，A* 需要扩展的点越多，运行速度越慢。

如果 h(n)正好等于从 n 移动到目标的代价，那么 A* 将只遵循最佳路径而不会扩展到其他任何结点，能够运行地很快。尽管这不可能在所有情

况下发生，但你仍可以在某些特殊情况下让 h(n)正好等于实际代价值。只要所给的信息完善，A* 将运行得很完美。
如果 h(n)比从 n 移动到目标的代价高，则 A* 不能保证找到一条最短路径，但它可以运行得更快。
另一种极端情况，如果 h(n)比 g(n)大很多，则只有 h(n)起作用，同时 A* 算法演变成贪婪最佳优先搜索算法（Greedy Best-First-Search）。

In the case where h(n) is 0, only g(n) is used and A* algorithm becomes Dijkstra's algorithm, which guarantees finding the shortest path.

If h(n) is always smaller (or equal) than the cost of moving from n to the goal, A* guarantees finding a shortest path. The smaller h(n) is, the more nodes A* needs to expand, and the slower it runs.

If h(n) is exactly equal to the cost of moving from n to the goal, A* will only follow the optimal path and not expand any other nodes, which can run very quickly. Although this is not possible in all cases, you can still make h(n) exactly equal to the actual cost in some special cases. As long as the given information is complete, A* will run perfectly.

If h(n) is greater than the cost of moving from n to the goal, A* cannot guarantee finding the shortest path, but it can run faster.

In the other extreme case where h(n) is much greater than g(n), only h(n) is used and A* algorithm becomes Greedy Best-First-Search algorithm.

启发式函数可以控制 A*的行为： 一种极端情况，如果 h(n)是 0，则只有 g(n)起作用，此时 A*演变成 Dijkstra 算法，这保证能找到最短路径。 如果 h(n)经常都比从 n 移动到目标的实际代价小（或者相等），则 A*保证能找到一条最短路径。h(n)越小，A*扩展的结点越多，运行就得越慢。 如果 h(n)精确地等于从 n 移动到目标的代价，则 A*将会仅仅寻找最佳路径而不扩展别的任何结点，这会运行得非常快。尽管这不可能在所有情况下发生，你仍可以在一些特殊情况下让它们精确地相等（译者：指让 h(n)精确地等于实际值）。只要提供完美的信息，A*会运行得很完美，认识这一点很好。 如果 h(n)有时比从 n 移动到目标的实际代价高，则 A*不能保证找到一条最短路径，但它运行得更快。 另一种极端情况，如果 h(n)比 g(n)大很多，则只有 h(n)起作用，A*演变成 BFS 算法。 所以我们得到一个很有趣的情况，那就是我们可以决定我们想要从 A*中获得什么。理想情况下（注：原文为 At exactly the right point），我们想最快地得到最短路径。如果我们的目标太低，我们仍会得到最短路径，不过速度变慢了；如果我们的目标太高，那我们就放弃了最短路径，但 A*运行得更快。 在游戏中，A*的这个特性非常有用。例如，你会发现在某些情况下，你希望得到一条好的路径（"good" path）而不是一条完美的路径（"perfect" path）。为了权衡 g(n)和 h(n)，你可以修改任意一个。 注:在学术上，如果启发式函数值是对实际代价的

低估，A*算法被称为简单的 A 算法（原文为 simply A）。然而，我继续称之为 A*，因为在实现上是一样的，并且在游戏编程领域并不区别 A 和 A*。

Heuristic functions can control the behaviour of A*:

In an extreme case, if h(n) is 0, only g(n) is used, and A* becomes Dijkstra's algorithm, which ensures finding the shortest path. If h(n) frequently underestimates (or equals) the actual cost of moving from n to the goal, A* guarantees finding the shortest path. The smaller h(n), the more nodes A* expands, and the slower it runs. If h(n) exactly equals the cost of moving from n to the goal, A* will only search for the optimal path without expanding any other nodes, which runs very quickly. Although this is not possible in all cases, you can make them exactly equal in some special cases (i.e., make h(n) exactly equal to the actual value). A* will run perfectly if perfect information is provided, and it is good to know this. If h(n) sometimes overestimates the actual cost of moving from n to the goal, A* cannot guarantee finding the shortest path, but it runs faster. In another extreme case, if h(n) is much larger than g(n), only h(n) is used, and A* becomes BFS algorithm.

Therefore, we have an interesting situation where we can decide what we want to get from A*. Ideally, we want to get the shortest path as quickly as possible. If our goal is too low, we will still get the shortest path, but the speed will be slower. If our goal is too high, we give up the shortest path, but A* runs faster.

In games, this feature of A* is very useful. For example, in some cases, you may want to get a good path rather than a perfect path. You can modify either g(n) or h(n) to balance them.

Note: In academic terms, if the heuristic function value is an underestimate of the actual cost, the A* algorithm is called simply A. However, I still refer to it as A*, because it is the same in implementation, and there is no difference between A and A* in game programming.