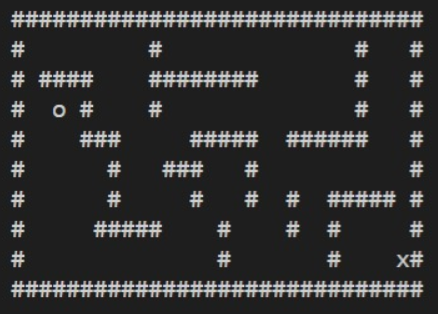
“Python Maze”

**Maze Problem**

The problem we want to solve is to find the path from one **initial state(o)** to a **goal state(x)** out of a virtual maze. In our problem we will assume that our initial state is dropped down somewhere into the middle of the maze and must find its way out to the goal state. Look at Figure 1 to get an idea of where we are going in this section.

We implement 2 methods of solving out this problem, which we will explain above in this document.

  
  
Fig 1. Maze Problem

To make it easier for us we will assume that our maze is divided up into “squares.” Each square of the maze is either open or occupied by a section of wall (#). The turtle can only pass through the spaces of the maze. If the state(o) bumps into a wall it must try a different direction. The state(o) will require a systematic procedure to find its way out of the maze.

**Environment, States and Movement rules**

The environment for this problem is simple, is a maze that is delimited by walls (#) and you can manually indicate where you wanted to be your initial state and your goal state. The maze is also form by some walls, and you can manually edit it, in order to make a little bit more complex to find the solution.

States

The states are defined by a pair of points *x, y.* Where we can find one **initial state** defined by the o symbol and one **goal state**, defined by the x symbol.

Rules

To go from one state to another, we need to validate only that the next pair of point (x, y) is not on a wall (#). Beyond that, the next state, is acceptable for the algorithm.

**A\***

To solve this problem, we will use the A\* Searching Algorithm. This algorithm implements a heuristic function to find the optimal path to our goal state.

A\* is an informed algorithm as it uses a heuristic to guide the search. The algorithm starts from an initial start node, expands neighbors, and updates the full path cost of each neighbor. It selects the neighbor with the lowest cost and continues until it finds a goal node, this can be implemented with a priority queue or by sorting the list of open nodes in ascending order. It is important to select a good heuristic to make A\* fast in searches, a good heuristic should be close to the actual cost but should not be higher than the actual cost.

A\* is complete and optimal, it will find the shortest path to the goal. A good heuristic can make the search very fast, but it may take a long time and consume a lot of memory in a large search space. The time complexity is O(n) in a grid and O(b^d) in a graph/tree with a branching factor (b) and a depth (d). The branching factor is the average number of neighbor nodes that can be expanded from each node and the depth is the average number of levels in a graph/tree.

**Cost Function**

The function we define in order to achieve the cost of the algorithm is the following:

Where x = cost of i state.

Where the cost will be confirmed by the following rules:

* Regular Cost = 1.0, for “up”, “down”, “left” and “right” moves.

Giving this, the cost function is the sum of all the moves from the initial state(o) to the goal state(x).

**Heuristic Function**

To use A\* Informed Search Algorithm, we declare a heuristic that will help us find the optimal path to our goal state. Which is the following:

Where:  
, y is one state

is the goal state

**Brute Force Algorithm**

This are straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.

The time complexity of brute force is O(mn), which is sometimes written as O(n\*m). So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n \* m tries.

We wanted also to solve this problem, with one brute force algorithm. This to solve the problem and compare it with an informed Search Algorithm. So, what the code does is to try all paths possible, respecting the rules of the maze and then, starts evaluating when we can.

References:

* A\* Search Algorithm in Python | A Name Not Yet Taken AB. (2020). Retrieved 1 October 2020, from <https://www.annytab.com/a-star-search-algorithm-in-python/>
* PacktPublishing/Artificial-Intelligence-with-Python. (2020). Retrieved 1 October 2020, from <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python/blob/master/Chapter%2007/code/maze.py>
* Brute Force Algorithms Explained. (2020). Retrieved 1 October 2020, from <https://www.freecodecamp.org/news/brute-force-algorithms-explained/#:~:text=Brute%20Force%20Algorithms%20are%20exactly,%2C%20each%20from%200%2D9>.