**ID:L201826100121**

**NAME:MOLLICK SHAHRIAR HOSSAIN**

**LAB4:A Star Search**

1. **Objective of this lab**

Be familiar with the process of solving search problems. Be familiar with the evaluation metrics of search algorithms. Understand the theory of A Star Search. Understand the theory and techniques of designing heuristic functions for A Star Search. Be able to solve maze problems with A Star Search.

1. **A Star Search theory**

A star search is an important informative search algorithm. It always select from the open table the node with the highest estimated cost as the new state. For any state , its cost estimation is defined by:



Above  is the paid cost from the starting point to state , which is a known value. Taking the maze problem as an instance,  is exactly the number of steps you have moved from the starting state to .  is called the heuristic function, which estimates the future cost from s to the goal state. Let  denote the minimum cost from  to the goal state ( is typically unknown beforehand). If  satisfies

,

we say  is **admissible**. Then we have an important theorem for A Star Search: A Star Search with an admissible heuristic is guaranteed to find the minimum cost for the search problem. Note for our maze problem in this lab, the heuristic based on either Euclidean or Manhattan distance is always admissible.

1. **Experimental results and analysis**

Reaching a destination via the shortest route is a daily activity we all do. A-star (also referred to as A\*) is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path cost and also uses [heuristic](https://en.wikipedia.org/wiki/Heuristic_(computer_science))s to find the solution.

In this article, I will focus on how to build A-star (A\*) search algorithm using a simple python code. I found many articles and blogs focus heavily on theory but not much information on the program. I am trying here to present the code step-by-step with easy to understand details.

First, let’s have a bit of theory to warm-up.

A\* achieve ***optimality*** and ***completeness***, two valuable property of search algorithms.

*When a search algorithm has the property of****optimality****, it means it is****guaranteed****to find the****best possible solution****. When a search algorithm has the property of****completeness****, it means that if a solution to a given problem****exists****, the algorithm is****guaranteed****to find it.*

Now to understand how A\* works, first we need to understand a few terminologies:

* **Node** (also called **State**) — All potential position or stops with a unique identification
* **Transition** — The act of moving between states or nodes.
* **Starting Node —**Whereto start searching
* **Goal Node —**The target to stop searching.
* **Search Space** — A collection of nodes, like all board positions of a board game
* **Cost**— Numerical value (say distance, time, or financial expense) for the path from a node to another node.
* **g(n)** — this represents the ***exact cost*** of the path from the **starting node** to any node **n**
* **h(n)** — this represents the heuristic ***estimated cost*** from node **n** to the goal node.
* **f(n)**— lowest cost in the neighboring node n

Each time A\* enters a node, it calculates the cost, f(n)(n being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of f(n).

These values we calculate using the following formula:

f(n) = g(n) + h(n)

Here we will solve a maze problem. We have to find the shortest path in the maze from a start node to the end node.

1) Left: Maze problem 2) Right: Position of every node (2D NumPy array positions) of the Maze (Image by Author)

For this problem, there are four moves (left, right, up, and down) from a maze position provided a valid step is available. In red square positions, no movement is allowed (like in start position only down motion is available since up and left move are blocked by the wall while for the right is a red square position thus no movement allowed).

I have adopted code from [here](https://gist.github.com/ryancollingwood/32446307e976a11a1185a5394d6657bc) and [this blog](https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2), which is an excellent source of information. If you need more clear theory and explanation, read the article on A\* from Patrick Lester.

First, we will create below class and helping function :

(1) **Class ‘Node’** that can be used to create an object for every node with information as the parent node, current position in the maze, and cost values (g, h & f).

(2) We need to define a **path function** that will return the path from start to end node that A\*

We will establish a **search function** which will be the drive the code logic:

(3.1) Initialize all variables.

(3.2) Add the starting node to the “yet to visit list.” Define a stop condition to avoid an infinite loop. Define movement in terms of relative position.

Repeat the following till stop criteria have met:

(3.3) Look for the lowest f cost square on the “yet to visit list.” This square becomes the current square. We also check max iteration reached or not

(3.4) Check if the current square is the same as target square (meaning we have found the path)

(3.5) Use the current square and check four squares adjacent to this current square to update the children node. If it is not movable or if it is on the “visited list,” ignore it. Otherwise, create the new node with the parent as the current node and update the position of the node.

(3.7) Check all the children node created to see

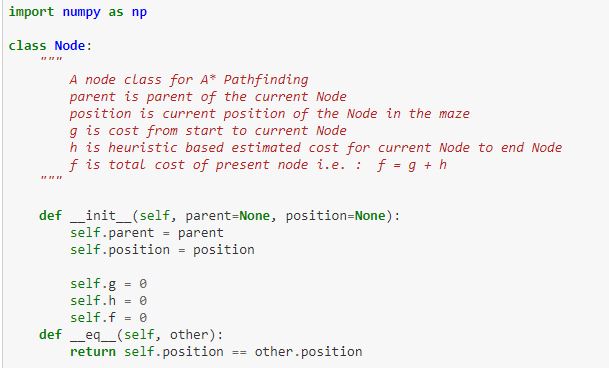
· If it isn’t on the “yet to visit list,” add it to the “yet to visit list.” Make the current square the parent of this square. Record the f, g, and h costs of the square.

· If it is in the “yet to visit list” already, check to see if this path to that square is better, using g cost as the measure. A lower g cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the g and f scores of the square.

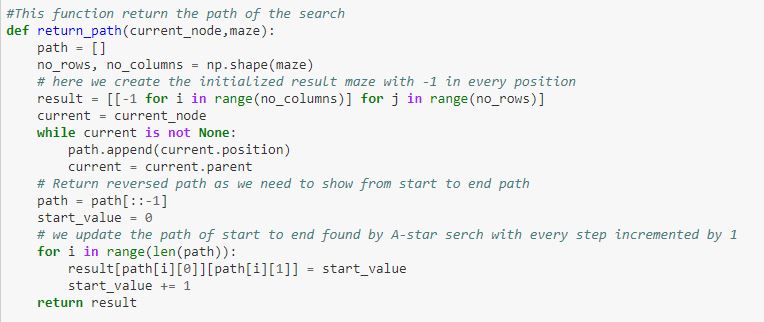
(4) **Main Program:** We will define maze, start, and end position. Then we will use the search function, and if a path exists, we can print the path from path function.

Now I will go through the code step by step about the steps mentioned above (refer the bracketed number ).

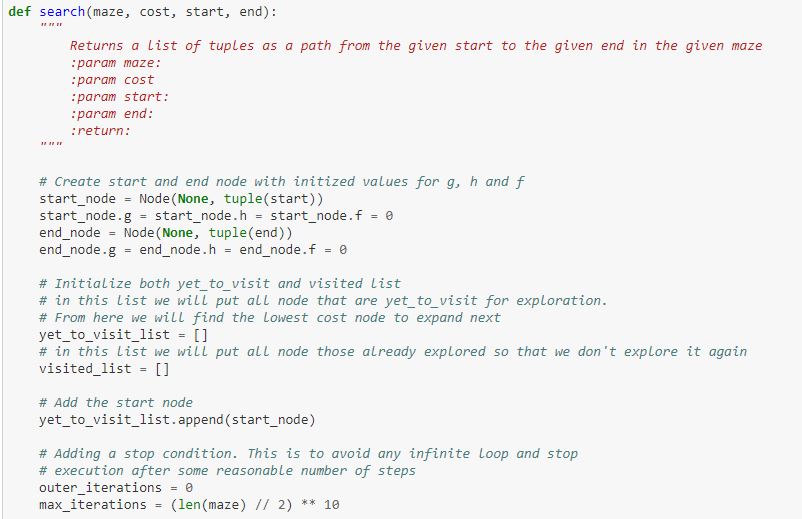
First, we will create a class for a node that will contain all the attributes associated with the node like the parent of the node, position of the node, and all three costs (g,h & f) for the node. We initialize the node and build a method for checking the equality of the node with another node.



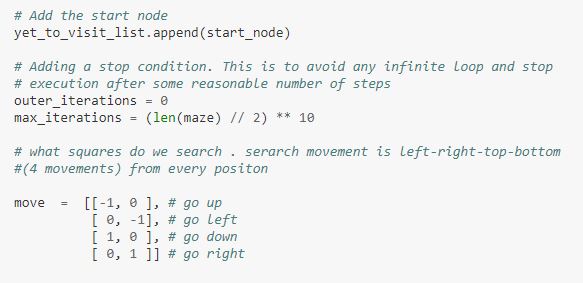
Now we will build the path function, which will be used to return the path from the start node to the target node (end node).



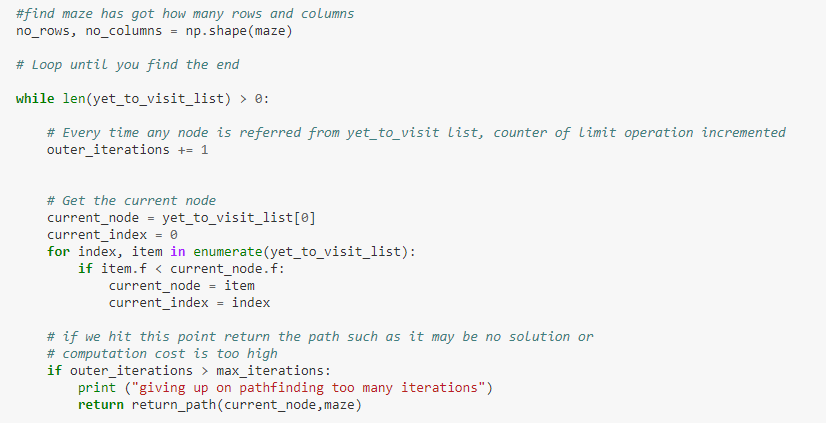
Now we will define the search function, which has multiple steps. The first step will be to initialize nodes and lists that we will use in the function.



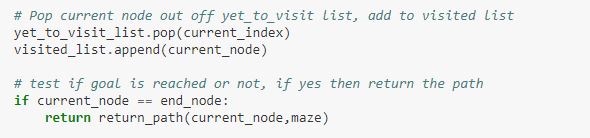
Add the starting node to the “yet to visit list.” Define a stop condition to avoid an infinite loop. Define movement in terms of relative position, which will be used to find the child node and other relative positions.



Now we use the current node by comparing all f cost and selecting the lowest cost node for further expansion. We also check max iteration reached or not, Set a message and stop execution (avoid infinite loop)

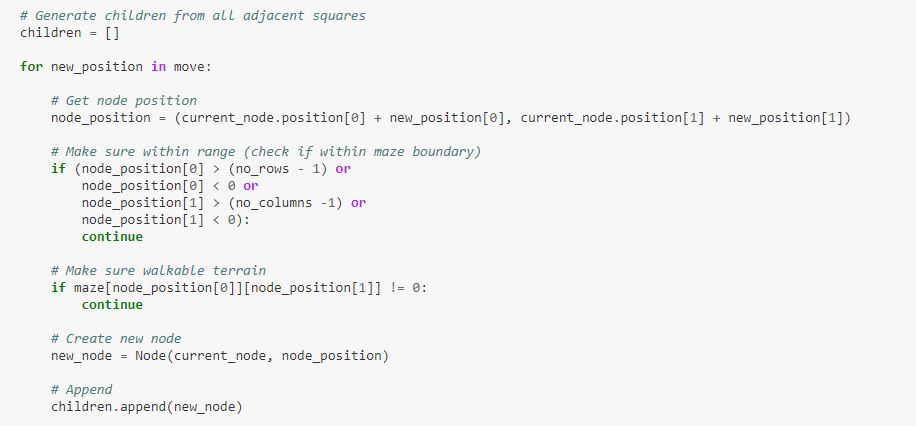


Remove the selected node from “yet to visit list” and add this node to the visited list. Now we put a check if we found the target square. If we have located the target square, then call the path function and return.

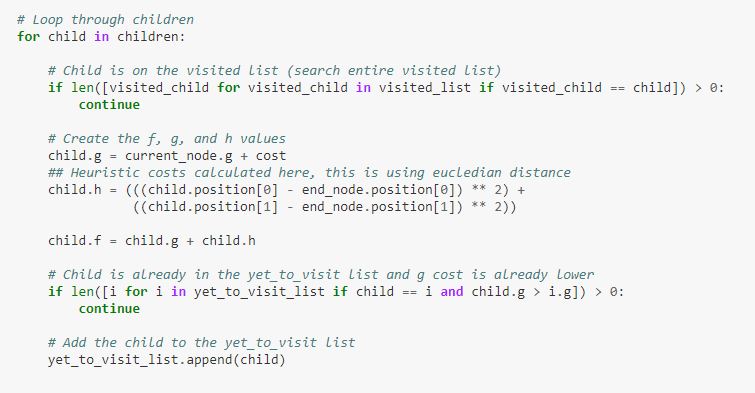


For the selected node, find out all children (use the move to find children). Get the current position for the selected node (this becomes the parent node for the children)  
a) check if a valid location exists (boundary wall will make few nodes invalid)  
b) if any node position is invalid (red square) then ignore that  
c) add to valid children node list for the selected parent

Here in the diagram, we show the black circle node is the current node, and green circle nodes are correct children node.



For all child node:  
a) if the child is in the visited list, then ignore it and try the next child node.  
b) calculate child node g, h, and f values. For h heuristic for the cost to reach the goal node for the current node is calculated here using euclidean distance.  
c) if the child in “yet to visit list” then ignore it else, move the child to “yet to visit list.”



Now finally, we will run the program from the main with the maze and obtain the path. Refer to the path also shown using the arrow.

**Explanation why these two heuristic functions (manhattan and euclidean) are admissible.**

A heuristic function hh is admissible, if it never overestimates the cost for any given node. Formally speaking, let h∗h∗ map each node to its true cost of reaching the goal. The heuristic function hh is admissible, if for all nodes nn in the search tree the following inequality holds:

h(n)≤h∗(n).(⋆)(⋆)h(n)≤h∗(n).

That means for checking whether a given heuristic function hh is admissible, we have to verify that the inequality (⋆)(⋆) holds by either  
  (a) calculating the real cost h∗h∗ for each node and comparing the values, or  
  (b) proving it by using additional information available of the heuristic.

For example, we know that the eucledian distance is admissible for searching the shortest path (in terms of actual distance, not path cost). Note also that any [consistent heuristic](http://en.wikipedia.org/wiki/Consistent_heuristic) is admissible (but not always vice-versa).

For your example, there is no additional information available regarding the two heuristics. Thus you have to calculate the real cost h∗h∗ for each node, and then check whether the inequality (⋆)(⋆) holds (I leave this task to you).

By checking the total cost you can **neither prove** that a heuristic is **admissible nor** that a heuristic is **not admissible**. The problem with this idea is that on the one hand you sum up the costs of the edges, but on the other hand you sum up the path cost (the heuristic values). For example, consider the following search tree with start node AA and goal node CC

enter image description here

and the following heuristic functions h1h1 and h2h2:

h1(A)=20;h1(B)=10;h1(C)=0;h2(A)=8h2(B)=11h2(B)=0h1(A)=20;h2(A)=8h1(B)=10;h2(B)=11h1(C)=0;h2(B)=0

The sum of the total cost of the search graph is 10+10=2010+10=20. The sum of the heuristic values of h1h1 is equal to 20+10+0=3020+10+0=30, which is larger than 2020 although h1h1 is admissible. The sum of the heuristic values of h2h2 is equal to 8+11+0=198+11+0=19, which is smaller than 2020, but h2h2 is **not admissible**, since h2(B)=11≰h∗(B)=10h2(B)=11≰h∗(B)=10.

**Compare the performance of manhattan and euclidean as heuristic functions in terms of the numbers expanded nodes in solving mazes.**

Euclidean distance is the shortest path between source and destination which is a straight line but Manhattan distance is sum of all the real distances between source(s) and destination(d) and each distance are always the straight lines .So, there can be more than one possible Manhattan distances in a maze and Ms. Pac-Man follows any of the path among them in a maze. The distance metrics itself can not be considered as terrain sensitive or not. The use of potential field and influence map in the three different distance metrics make them either terrain sensitive or not. Potential field is not terrain sensitive i.e. it does not take terrain of map into account meaning it is unable to recognize the presence of wall between the source and destination. This facilitates the propagation possible through or over the walls. In the other hand, influence map is terrain sensitive i.e. the presence of wall between source and destination affect the propagation of influences .

Since we are comparing A\* distance metric in influence map and Euclidean and Manhattan in potential field in research question 2, this makes A\* terrain sensitive but Euclidean and Manhattan not.

A\* Search algorithm Representation to destination (d) may be substantially greater than Euclidean distance between s and d since Euclidean distance calculates the shortest path between s and d .

1. **Conclusion**

A-star (A\*) is a mighty algorithm in Artificial Intelligence with a wide range of usage. However, it is only as good as its heuristic function( which can be highly variable considering the nature of a problem). A\* is the most popular choice for pathfinding because it’s reasonably flexible.

It has found applications in many software systems, from Machine Learning and search Optimization to game development where characters navigate through complex terrain and obstacles to reach the player.