

Get started

Open in app



Follow

599K Followers



You have **2** free member-only stories left this month. [Sign up for Medium and get an extra one](#)



Image from Pixabay and created by Author

A-Star (A*) Search Algorithm

A-Star Search algorithm with step-by-step code



Baijayanta Roy · Sep 29, 2019 · 7 min read ★

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 heuristics 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:

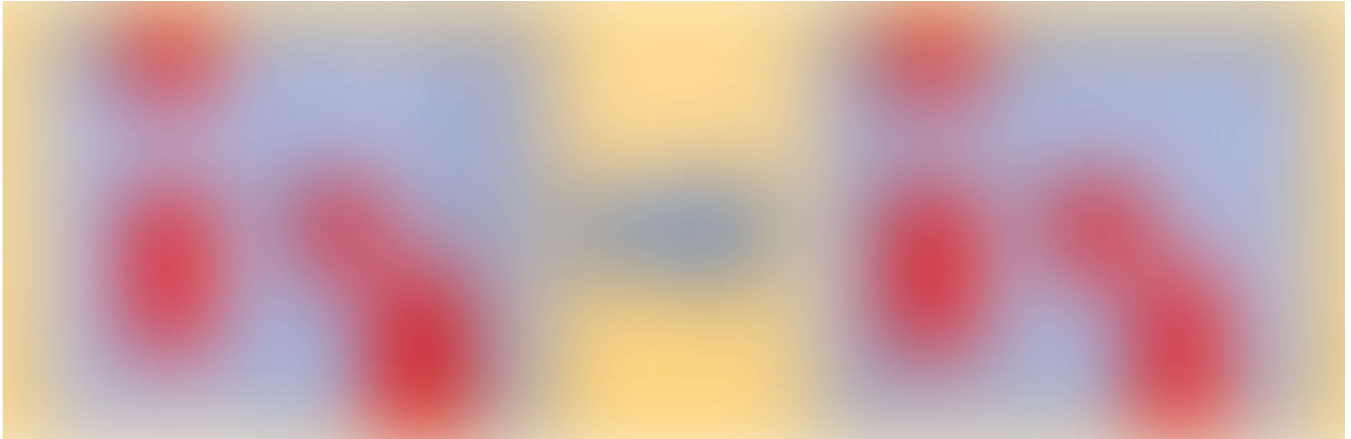
- **Node** (also called **State**) — All potential position or stops with a unique identification
- **Transition** — The act of moving between states or nodes.
- **Starting Node** — Where to start searching
- **Goal Node** — The target to stop searching.
- **Search Space** — A collection of nodes, like 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](#) and [this blog](#), 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.

1

```
import numpy as np

class Node:
    """
    A node class for A* Pathfinding
    parent is parent of the current Node
    position is current position of the Node in the maze
    g is cost from start to current Node
    h is heuristic based estimated cost for current Node to end Node
    f is total cost of present node i.e. : f = g + h
    """

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position
```

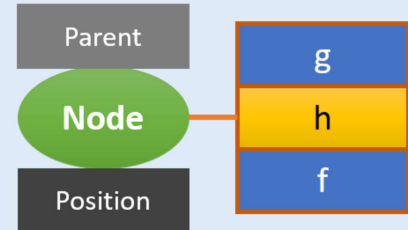


Image by Author

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).

2

```
#This function return the path of the search
def return_path(current_node, maze):
    path = []
    no_rows, no_columns = np.shape(maze)
    # here we create the initialized result maze with -1 in every position
    result = [[-1 for i in range(no_columns)] for j in range(no_rows)]
    current = current_node
    while current is not None:
        path.append(current.position)
        current = current.parent
    # Return reversed path as we need to show from start to end path
    path = path[::-1]
    start_value = 0
    # we update the path of start to end found by A-star search with every step incremented by 1
    for i in range(len(path)):
        result[path[i][0]][path[i][1]] = start_value
        start_value += 1
    return result
```

Image by Author

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.

3.1

```
def search(maze, cost, start, end):
    """
    Returns a list of tuples as a path from the given start to the given end in the given maze
    :param maze:
    :param cost
    :param start:
    :param end:
    :return:
    """

    # Create start and end node with initized values for g, h and f
    start_node = Node(None, tuple(start))
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, tuple(end))
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both yet_to_visit and visited List
    # in this list we will put all node that are yet_to_visit for exploration.
    # From here we will find the Lowest cost node to expand next
    yet_to_visit_list = []
    # in this list we will put all node those already explored so that we don't explore it again
    visited_list = []
    # Add the start node
    yet_to_visit_list.append(start_node)
```

Start Node (Initialized)

End Node (Initialized)

A list for nodes yet to visit (initialized)

A list of nodes visited so far (initialized)

Image by Author

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.

3.2

```
# Add the start node
yet_to_visit_list.append(start_node)

# Adding a stop condition. This is to avoid any infinite Loop and stop
# execution after some reasonable number of steps
outer_iterations = 0
max_iterations = (len(maze) // 2) ** 10

# what squares do we search . search movement is Left-right-top-bottom
# (4 movements) from every position

move = [[-1, 0], # go up
        [0, -1], # go left
        [1, 0], # go down
        [0, 1]] # go right
```

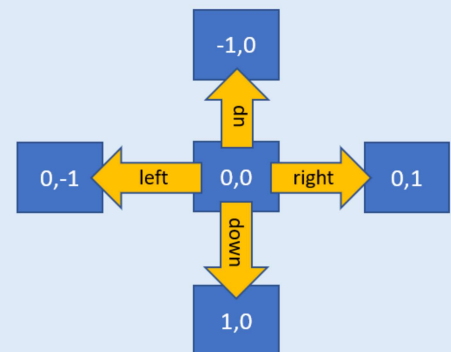


Image by Author

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)

3.3

```

#find maze has got how many rows and columns
no_rows, no_columns = np.shape(maze)

# Loop until you find the end

while len(yet_to_visit_list) > 0:

    # Every time any node is referred from yet_to_visit List, counter of limit operation incremented
    outer_iterations += 1

    # Get the current node
    current_node = yet_to_visit_list[0]
    current_index = 0
    for index, item in enumerate(yet_to_visit_list):
        if item.f < current_node.f:
            current_node = item
            current_index = index

    # if we hit this point return the path such as it may be no solution or
    # computation cost is too high
    if outer_iterations > max_iterations:
        print ("giving up on pathfinding too many iterations")
        return return_path(current_node,maze)

```

Image by Author

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.

3.4

```

# Pop current node out off yet_to_visit List, add to visited List
yet_to_visit_list.pop(current_index)
visited_list.append(current_node)

# test if goal is reached or not, if yes then return the path
if current_node == end_node:
    return return_path(current_node,maze)

```

Here we maintain which next node to visit and update visited list of nodes

Wow ...We have found the goal node . Terminate the search

Image by Author

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)

- check if a valid location exists (boundary wall will make few nodes invalid)
- if any node position is invalid (red square) then ignore that
- 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.

3.5

```
# Generate children from all adjacent squares
children = []

for new_position in move:

    # Get node position
    node_position = (current_node.position[0] + new_position[0],
                    current_node.position[1] + new_position[1])

    # Make sure within range (check if within maze boundary)
    if (node_position[0] > (no_rows - 1) or
        node_position[0] < 0 or |
        node_position[1] > (no_columns - 1) or
        node_position[1] < 0):
        continue

    # Make sure walkable terrain
    if maze[node_position[0]][node_position[1]] != 0:
        continue

    # Create new node
    new_node = Node(current_node, node_position)

    # Append
    children.append(new_node)
```

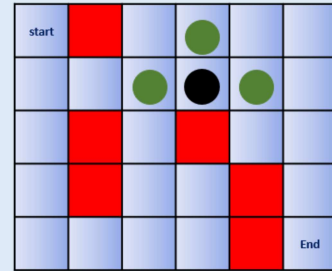


Image by Author

For all child node:

- if the child is in the visited list, then ignore it and try the next child node.
- 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.
- if the child in “yet to visit list” then ignore it else, move the child to “yet to visit list.”

3.6

```
# Loop through children
for child in children:

    # Child is on the visited list (search entire visited list)
    if len([visited_child for visited_child in visited_list if visited_child == child]) > 0:
        continue

    # Create the f, g, and h values
    child.g = current_node.g + cost
    ## Heuristic costs calculated here, this is using euclidean distance
    child.h = (((child.position[0] - end_node.position[0]) ** 2) +
               ((child.position[1] - end_node.position[1]) ** 2))

    child.f = child.g + child.h

    # Child is already in the yet_to_visit list and g cost is already lower
    if len([i for i in yet_to_visit_list if child == i and child.g > i.g]) > 0:
        continue

    # Add the child to the yet_to_visit list
    yet_to_visit_list.append(child)
```

Here Euclidean distance is used to calculate h cost. This is a major activity in real life problem to find appropriate heuristic.

Image by Author

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.

4

```

if __name__ == '__main__':
    maze = [[0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0],
            [0, 1, 0, 1, 0, 0],
            [0, 1, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0]]

    start = [0, 0] # starting position
    end = [4,5] # ending position
    cost = 1 # cost per movement

    path = search(maze, cost, start, end)
    print(path)

[[0, -1, -1, -1, -1, -1], [1, 2, 3, 4, 5, -1], [-1, -1, -1, -1, 6, 7], [-1, -1, -1, -1, -1, 8], [-1, -1, -1, -1, -1, 9]]

```

Final result path

```

print('\n'.join([''.join(["{: " >3d}".format(item) for item in row])
                  for row in path]))

```

```

0 -1 -1 -1 -1 -1
1 2 3 4 5 -1
-1 -1 -1 -1 6 7
-1 -1 -1 -1 -1 8
-1 -1 -1 -1 -1 9

```

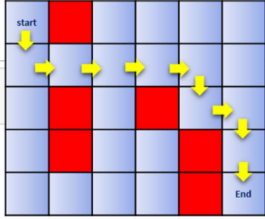


Image by Author

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.

You can find full code in [this](#) GitHub repository.

Thanks for reading. You can connect with me on [LinkedIn](#).

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

Get this newsletter