A Name Not Yet Taken AB

Halmstad, Sweden

A* Search Algorithm in Python

by Administrator / Computer Science / January 22, 2020 / 13 Comments

I will show you how to implement an A* (Astar) search algorithm in this tutorial, the algorithm will be used solve a grid problem and a graph problem by using Python. The A* search algorithm uses the full path cost as the heuristic, the cost to the starting node plus the estimated cost to the goal node.

A* is an informed algorithm as it uses an 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.

Grid problem (maze)

I have created a simple maze (download it) with walls, a start (@) and a goal ($). The goal of the A* algorithm is to find the shortest path from the starting point to the goal point as fast as possible. The full path cost (f) for each node is calculated as the distance to the starting node (g) plus the distance to the goal node (h). Distances is calculated as the manhattan distance (taxicab geometry) between nodes.

```
# This class represents a node
class Node:

    # Initialize the class
```

```
    def __init__(self, position:(), parent:()):
        self.position = position
        self.parent = parent
        self.g = 0 # Distance to start node
        self.h = 0 # Distance to goal node
        self.f = 0 # Total cost

    # Compare nodes
    def __eq__(self, other):
        return self.position == other.position

    # Sort nodes
    def __lt__(self, other):
         return self.f < other.f

    # Print node
    def __repr__(self):
        return ('({0},{1})'.format(self.position, self.f))

# Draw a grid
def draw_grid(map, width, height, spacing=2, **kwargs):
    for y in range(height):
        for x in range(width):
            print('%%-%ds' % spacing % draw_tile(map, (x, y), kwargs), end='
        print()

# Draw a tile
def draw_tile(map, position, kwargs):

    # Get the map value
    value = map.get(position)

    # Check if we should print the path
    if 'path' in kwargs and position in kwargs['path']: value = '+'

```

```
    # Check if we should print start point
    if 'start' in kwargs and position == kwargs['start']: value = '@'

    # Check if we should print the goal point
    if 'goal' in kwargs and position == kwargs['goal']: value = '$'

    # Return a tile value
    return value

# A* search
def astar_search(map, start, end):

    # Create lists for open nodes and closed nodes
    open = []
    closed = []

    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)

    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:

        # Sort the open list to get the node with the lowest cost first
        open.sort()

        # Get the node with the lowest cost
        current_node = open.pop(0)

        # Add the current node to the closed list
        closed.append(current_node)

```

```
        # Check if we have reached the goal, return the path
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.position)
                current_node = current_node.parent
            #path.append(start)
            # Return reversed path
            return path[::-1]

        # Unzip the current node position
        (x, y) = current_node.position

        # Get neighbors
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

        # Loop neighbors
        for next in neighbors:

            # Get value from map
            map_value = map.get(next)

            # Check if the node is a wall
            if(map_value == '#'):
                continue

            # Create a neighbor node
            neighbor = Node(next, current_node)

            # Check if the neighbor is in the closed list
            if(neighbor in closed):
                continue

            # Generate heuristics (Manhattan distance)
            neighbor.g = abs(neighbor.position[0] - start_node.position[0])
```

```
            neighbor.h = abs(neighbor.position[0] - goal_node.position[0]) +
            neighbor.f = neighbor.g + neighbor.h

            # Check if neighbor is in open list and if it has a lower f valu
            if(add_to_open(open, neighbor) == True):
                # Everything is green, add neighbor to open list
                open.append(neighbor)

    # Return None, no path is found
    return None

# Check if a neighbor should be added to open list
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f >= node.f):
            return False
    return True

# The main entry point for this module
def main():

    # Get a map (grid)
    map = {}
    chars = ['c']
    start = None
    end = None
    width = 0
    height = 0

    # Open a file
    fp = open('data\\maze.in', 'r')

    # Loop until there is no more lines
    while len(chars) > 0:

```

```python
        # Get chars in a line
        chars = [str(i) for i in fp.readline().strip()]

        # Calculate the width
        width = len(chars) if width == 0 else width

        # Add chars to map
        for x in range(len(chars)):
            map[(x, height)] = chars[x]
            if(chars[x] == '@'):
                start = (x, height)
            elif(chars[x] == '$'):
                end = (x, height)

        # Increase the height of the map
        if(len(chars) > 0):
            height += 1

    # Close the file pointer
    fp.close()

    # Find the closest path from start(@) to end($)
    path = astar_search(map, start, end)
    print()
    print(path)
    print()
    draw_grid(map, width, height, spacing=1, path=path, start=start, goal=en
    print()
    print('Steps to goal: {0}'.format(len(path)))
    print()

# Tell python to run main method
if __name__ == "__main__": main()
```

```
############################################################################
#.#...#....$....#...................#...#.........#.......#.............#...
#.#.#.#.###+###.#########.#########.#.#####.#####.#####.#.#.#######.###.#.##
#...#.....#+++#.#.........#.#.....#.#...#...#...#.......#.#.#.......#.#.#.#.
#############+#.#.#########.#.###.#.###.#.###.#.#.#######.###.#######.#.#.#.
#+++++++++++#+#...#.#.....#...#...#...#.#.#.#.#...#...#.......#.......#.#.#.
#+#########+#+#####.#.#.#.#.###.#####.#.#.#.#.#####.#.#########.###.###.###.
#+#........+#+++#...#.#.#.#...#.....#.#.#.#...#.#...#.......#.....#.#...#...
#+#########+#.#+###.#.#.#####.###.#.#.#.#.#.###.#.#########.#####.#.#.###.##
#+#+++++++#+#.#+++#...#.#.....#.#.#.#...#.#.....#.#.....#.#...#...#.......#.
#+#+#####+#+#.###+#####.#.#####.#.#.###.#.#######.###.#.#.###.#.###########.
#+++#+++#+#+#...#+++++#.#.......#.#.#...#.....#...#...#.....#.#.#...#...#...
#####+#+#+###########+#.#######.#.###.#######.#.###.#########.###.#.#.#.#.##
#+++++#+++#+#+++++++++#.......#.#...#.#.#.....#.#.....#.......#...#.#.#.#.#.
#+#########+#+#########.###.###.###.#.#.#.###.#.#.###.#.#######.###.#.###.#.
#+++#.#+++++#+++#.....#.#.#...#.#.#.....#...#.#.#...#.#...#...#...#.#.#...#.
###+#.#+#####.#+#.#.###.#.###.#.#.#####.###.###.#####.###.#.#.#.###.#.#.####
#+++#+++#.....#+#.#.#...#...#.....#...#.#...#...........#.#.#...#...#.......
#+###+#########+#.#.#.###.#.#####.#.#.###.###.###########.#.#####.#########.
#+#..+++++++++++#.#.......#.#...#.#.#...#.#...#.#.......#.......#.#...#.....
#+#.#############.#########.#.#.###.###.#.#.###.#.#####.#.#######.#.#.#.####
#+#.#+++++++++++#.#.#.#.....#.#.....#...#.#.....#...#.#.#.#.#...#.#.#.#.#...
#+###+#########+#.#.#.#######.#######.###.#####.###.#.#.#.#.###.#.#.#.#.####
#+++++#+++#+++++#...#.........#.....#...#.....#...#...#.#.....#.#...#.#.#...
#.#####+#+#+#######.###########.#######.#.#######.###.#.###.###.#####.#.#.##
#.....#+#+#+++#...#.#+++++++#.........#.#...#.......#.#.#...#...#.....#.#.#.
#######+#+###+#.###.#+#####+#.#####.###.#.#.#.#######.#.#####.###.#####.#.##
#+++++++#+#+++#.....#+#...#+#...#.#.....#.#.#.#.#.....#...#...#...#.....#...
#+#######+#+#.#####.#+###.#+###.#.#######.#.#.#.#.#######.#.###.#.###.#####.
#+#.#+++++#+#.#+++#.#+++#.#+++#...#.#...#.#...#.#.....#.#...#...#...#.......
#+#.#+#####+#.#+#+#####+#.###+###.#.#.#.#.#####.#####.#.#####.#####.########
#+#..+#..+++#.#+#+#+++#+++#.#+#...#...#.#.#...#.....#...#.#...#...#.....#...
#+###+###+#.###+#+#+#+###+#.#+#.#######.#.#.#.#####.###.#.#.###.#.#####.###.
#+++#+++#+#.#+++#+#+#+++#+#.#+#.#.......#...#.........#.#...#...#.#...#...#.
```

```
#.#+###+#+#.#+###+#+###+#+#.#+#.###.###.###########.###.#.###.###.###.###.#.
#.#+++#+#+#.#+++#+++#+++#+#.#+#.....#...#...#.....#.#...#.....#.....#.#...#.
#.###+#+#+#####+#####+#.#+#.#+#######.###.#.#####.#.#.#############.#.#.###.
#...#+#+++#+++#+++++#+#.#+#.#+#+++#...#.#.#......#.#.#...#...#...#...#.#.#.
###.#+#####+#+#####+#+###+#.#+#+#+#.###.#.#########.#.#.#.#.#.#.#.#####.#.#.
#...#+++++++#+++++++#+++++..#+++#+++++++@..........#...#...#...#.......#...
############################################################################

Steps to goal: 339
```

Graph problem

The goal of this graph problem is to find the shortest path between a starting location and destination location. A map has been used to create a graph with actual distances between locations. The A* algorithm uses a Graph class, a Node class and heuristics to find the shortest path in a fast manner. Heuristics is calculated as straight-line distances (air-travel distances) between locations, air-travel distances will never be larger than actual distances.

```python
# This class represent a graph
class Graph:

    # Initialize the class
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    # Create an undirected graph by adding symmetric edges
    def make_undirected(self):
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist

```

```
    # Add a link from A and B of given distance, and also add the inverse li
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    # Get neighbors or a neighbor
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

    # Return a list of nodes in the graph
    def nodes(self):
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items
        nodes = s1.union(s2)
        return list(nodes)

# This class represent a node
class Node:

    # Initialize the class
    def __init__(self, name:str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0 # Distance to start node
        self.h = 0 # Distance to goal node
        self.f = 0 # Total cost

    # Compare nodes
    def __eq__(self, other):
        return self.name == other.name
```

```

    # Sort nodes
    def __lt__(self, other):
         return self.f < other.f

    # Print node
    def __repr__(self):
        return ('({0},{1})'.format(self.name, self.f))

# A* search
def astar_search(graph, heuristics, start, end):

    # Create lists for open nodes and closed nodes
    open = []
    closed = []

    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)

    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:

        # Sort the open list to get the node with the lowest cost first
        open.sort()

        # Get the node with the lowest cost
        current_node = open.pop(0)

        # Add the current node to the closed list
        closed.append(current_node)

```

```
        # Check if we have reached the goal, return the path
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.name + ': ' + str(current_node.g))
                current_node = current_node.parent
            path.append(start_node.name + ': ' + str(start_node.g))
            # Return reversed path
            return path[::-1]

        # Get neighbours
        neighbors = graph.get(current_node.name)

        # Loop neighbors
        for key, value in neighbors.items():

            # Create a neighbor node
            neighbor = Node(key, current_node)

            # Check if the neighbor is in the closed list
            if(neighbor in closed):
                continue

            # Calculate full path cost
            neighbor.g = current_node.g + graph.get(current_node.name, neigh
            neighbor.h = heuristics.get(neighbor.name)
            neighbor.f = neighbor.g + neighbor.h

            # Check if neighbor is in open list and if it has a lower f valu
            if(add_to_open(open, neighbor) == True):
                # Everything is green, add neighbor to open list
                open.append(neighbor)

    # Return None, no path is found
    return None
```

```python

# Check if a neighbor should be added to open list
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f > node.f):
            return False
    return True

# The main entry point for this module
def main():

    # Create a graph
    graph = Graph()

    # Create graph connections (Actual distance)
    graph.connect('Frankfurt', 'Wurzburg', 111)
    graph.connect('Frankfurt', 'Mannheim', 85)
    graph.connect('Wurzburg', 'Nurnberg', 104)
    graph.connect('Wurzburg', 'Stuttgart', 140)
    graph.connect('Wurzburg', 'Ulm', 183)
    graph.connect('Mannheim', 'Nurnberg', 230)
    graph.connect('Mannheim', 'Karlsruhe', 67)
    graph.connect('Karlsruhe', 'Basel', 191)
    graph.connect('Karlsruhe', 'Stuttgart', 64)
    graph.connect('Nurnberg', 'Ulm', 171)
    graph.connect('Nurnberg', 'Munchen', 170)
    graph.connect('Nurnberg', 'Passau', 220)
    graph.connect('Stuttgart', 'Ulm', 107)
    graph.connect('Basel', 'Bern', 91)
    graph.connect('Basel', 'Zurich', 85)
    graph.connect('Bern', 'Zurich', 120)
    graph.connect('Zurich', 'Memmingen', 184)
    graph.connect('Memmingen', 'Ulm', 55)
    graph.connect('Memmingen', 'Munchen', 115)
    graph.connect('Munchen', 'Ulm', 123)
```

```
    graph.connect('Munchen', 'Passau', 189)
    graph.connect('Munchen', 'Rosenheim', 59)
    graph.connect('Rosenheim', 'Salzburg', 81)
    graph.connect('Passau', 'Linz', 102)
    graph.connect('Salzburg', 'Linz', 126)

    # Make graph undirected, create symmetric connections
    graph.make_undirected()

    # Create heuristics (straight-line distance, air-travel distance)
    heuristics = {}
    heuristics['Basel'] = 204
    heuristics['Bern'] = 247
    heuristics['Frankfurt'] = 215
    heuristics['Karlsruhe'] = 137
    heuristics['Linz'] = 318
    heuristics['Mannheim'] = 164
    heuristics['Munchen'] = 120
    heuristics['Memmingen'] = 47
    heuristics['Nurnberg'] = 132
    heuristics['Passau'] = 257
    heuristics['Rosenheim'] = 168
    heuristics['Stuttgart'] = 75
    heuristics['Salzburg'] = 236
    heuristics['Wurzburg'] = 153
    heuristics['Zurich'] = 157
    heuristics['Ulm'] = 0

    # Run the search algorithm
    path = astar_search(graph, heuristics, 'Frankfurt', 'Ulm')
    print(path)
    print()

# Tell python to run main method
if __name__ == "__main__": main()
```

```
['Frankfurt: 0', 'Wurzburg: 111', 'Ulm: 294']
```

Tags: AI Python

13 thoughts on "A* Search Algorithm in Python"

ALEXANDRE THIAULT
May 3, 2020 at 9:42 pm

Hello,

I think there's a mistake, just before the comment "# Check if neighbor is in open list and if it has a lower f value" it should be break, not continue, and then that last line "open.append(neighbor)" should be inside a "else:", using the syntax for;break;else.

Reply

ADMINISTRATOR
May 4, 2020 at 8:29 am

Hi,

thank you very much for your comment. I have updated the code according to your suggestions.

Reply

GUILHERME OLIVEIRA
June 9, 2020 at 6:01 pm

Hi, great post!

Is there any restrictions or licenses upon this code? What’s the correct procedure to reproduce it on a graduation project?

Regards

Reply

ADMINISTRATOR
June 10, 2020 at 6:10 am

Hi,

you are free to use the code however you want. If you use it in a graduation project, add a reference to this page and apply the code to other problems. You can also make some minor changes in the code in your graduation project.

Reply

GUILHERME
June 11, 2020 at 9:01 pm

Thanks, it will be referenced and I would also like to send you a copy if you want to see it!

Reply

ADMINISTRATOR
June 12, 2020 at 5:26 am

Thank you, you can find my email address in the bottom right corner of this website.

Reply