A* search algorithm

By: vivek vittal biragoni(211Ai041)

An overview of the algorithm:

A* search is a popular algorithm used in pathfinding and graph traversal. It is an extension of Dijkstra's algorithm and uses heuristic function to guide the search towards the goal.

The algorithm works by maintaining a priority queue of nodes to be explored. Each node in the queue has an associated cost, which is the sum of the cost of the path from the start node to that node and an estimate of the cost to reach the goal node. The estimate is calculated using a heuristic function, which provides an approximation of the remaining distance to the goal.

The algorithm starts by initializing the queue with the start node, and the cost to reach the start node is set to zero. The algorithm then dequeues the node with the lowest cost and examines its neighbors. For each neighbor, the algorithm calculates the total cost to reach that neighbor (by adding the cost of the current node to the cost of the edge connecting the two nodes) and the estimate of the remaining distance to the goal.

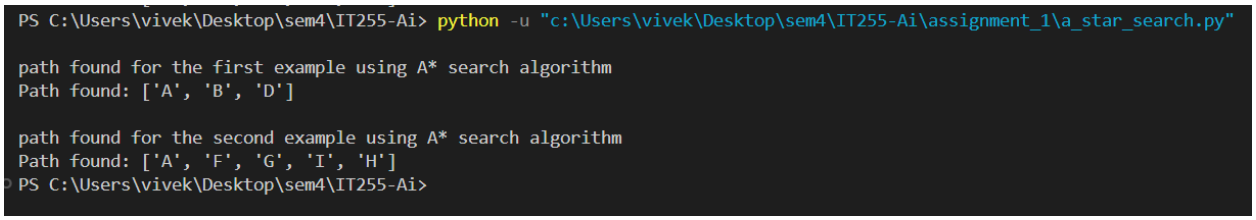
If the neighbor node is not in the queue, it is added to the queue with its cost and estimate. If the neighbor node is already in the queue, its cost and estimate are updated if the new values are lower than the current ones.

The algorithm continues this process, dequeuing nodes from the priority queue and exploring their neighbors until the goal node is reached or the queue is empty.

A* search guarantees to find the optimal path from the start node to the goal node as long as the heuristic function used is admissible, meaning it never overestimates the distance to the goal, and consistent, meaning the estimated distance from a node to the goal is always less than or equal to the sum of the cost from the node to its neighbor and the estimated distance from the neighbor to the goal.

Overall, A* search combines the best of both worlds by using both informed search (heuristic function) and un-informed search (Dijkstra's algorithm) techniques to efficiently find the shortest path between two points in a graph.

Screenshot of the output



Code

```
from collections import deque


class Graph:



    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list


    def get_neighbors(self, v):
        return self.adjacency_list[v]


    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
```

```
            'C': 1,
            'D': 1,
            'E': 1,
            'F': 1,
            'G': 1,
            'H': 1,
            'I': 1,
            'J': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been
visited, but who's neighbors
        # haven't all been inspected, starts off with the
start node
        # closed_list is a list of nodes which have been
visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to
all other nodes
```

```
        # the default value (if it's not found in the
map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() -
evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] +
self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
```

```
            # then we begin reconstructin the path from
it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()

                print('Path found:
{}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both
open_list and closed_list
                # add it to open_list and note n as it's
parent
                if m not in open_list and m not in
closed_list:
```

```
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to
first visit n, then m
                # and if it is, update parent data and g
data
                # and if the node was in the closed_list,
move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to
closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)
```

```
        print('Path does not exist!')
        return None


adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
print()
print("path found for the first example using A* search
algorithm")
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')



#Describe your graph here
adjacency_list = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
```

```
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
print()
print("path found for the second example using A* search
algorithm")
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'H')
```

Code explanation in short

This code implements the A* search algorithm to find the shortest path between two nodes in a graph.

The graph is represented using an adjacency list, which is a dictionary where the keys are the nodes of the graph and the values are lists of tuples representing the edges and their weights. For example, the adjacency_list dictionary {'A': [('B', 1), ('C', 3), ('D', 7)], ...} represents a graph where node A is connected to nodes B, C, and D with edge weights of 1, 3, and 7, respectively.

The A* algorithm uses a heuristic function h(n) to estimate the cost of the cheapest path from node n to the goal node. In this implementation, the heuristic function returns a constant value of 1 for all nodes.

The algorithm keeps track of two sets of nodes: the open_list, which are nodes that have been visited but whose neighbors have not all been inspected, and the closed_list, which are nodes that have been visited and whose neighbors have all been inspected. It starts with only the start_node in the open_list, and keeps expanding it by examining the neighbors of the nodes in the open_list.

The algorithm computes the cost g(n) of the cheapest path from the start_node to each node n that it visits. It also keeps track of the parent of each node n, which is the node from which it was first visited.

When the algorithm visits the goal node, it reconstructs the path from the start_node to the goal node by following the parent links backwards. It returns the path as a list of nodes.

If the algorithm exhausts the open_list without finding the goal node, it concludes that there is no path between the start_node and the goal node.