A* Search Algorithm

This Algorithm is the advanced form of the BFS algorithm (Breadth-first search), which searches for the shorter path first than, the longer paths. It is a complete as well as an optimal solution for solving path and grid problems.

Optimal – find the least cost from the starting point to the ending point. *Complete* – It means that it will find all the available paths from start to end.

Basic concepts of A*

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

Where

g (n) : The actual cost path from the start node to the current node.

h (n) : The actual cost path from the current node to goal node.

f (n) : The actual cost path from the start node to the goal node.

For the implementation of A* algorithm we have to use two arrays namely OPEN and CLOSE.

OPEN:An array that contains the nodes that have been generated but have not been yet examined till yet.

CLOSE:An array which contains the nodes which are examined.

Algorithm

1: Firstly, Place the starting node into OPEN and find its f (n) value.

2: Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success.

3: Else remove the node from OPEN, and find all its successors.

4: Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

5: Goto Step-2.

6: Exit.

A* Search Algorithm – Python Implementation

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):

                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None


        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
```

```
            open_set.remove(n)
            closed_set.add(n)
        print('Path does not exist!')
        return None

    def get_neighbors(v):
        if v in Graph_nodes:
            return Graph_nodes[v]
        else:
            return None

    def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H_dist[n]

    Graph_nodes = {
        'A': [('B', 2), ('E', 3)],
        'B': [('A', 2), ('C', 1), ('G', 9)],
        'C': [('B', 1)],
        'D': [('E', 6), ('G', 1)],
        'E': [('A', 3), ('D', 6)],
        'G': [('B', 9), ('D', 1)]
    }

    aStarAlgo('A', 'G')
```