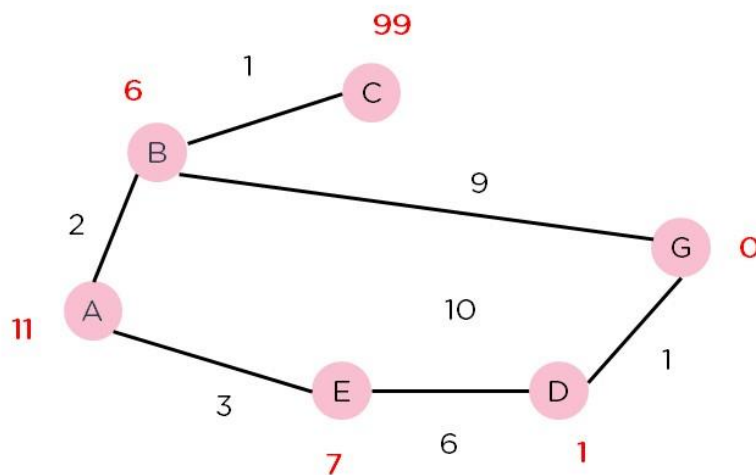


A* SEARCH ALGORITHM

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes, n , and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If $f(n)$ represents the final cost, then it can be denoted as : $f(n) = g(n) + h(n)$, where : $g(n)$ = cost of traversing from one node to another. This will vary from node to node $h(n)$ = heuristic approximation of the node's value. This is not a real value but an approximation cost.



CODE:

```
def astar(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  
  
        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': [(('C', 1), ('G', 9))],  
        'C': None,  
        'E': [(('D', 6))],  
        'D': [(('G', 1))],  
    }  
    astar('A', 'G')
```

OUTPUT:



The screenshot displays a Jupyter Notebook window titled '220701055.ipynb'. The interface includes a top menu bar with options like File, Edit, View, Insert, Runtime, Tools, and Help. Below the menu, there's a toolbar with icons for running code, saving, and other functions. The main area shows a code cell with the following Python code:

```
'E' : [('D',6)],  
'D' : [('G',1)],  
}  
astar('A','G')
```

Below the code cell, the output is displayed: 'Path found: ['A', 'E', 'D', 'G']'. The notebook interface also shows a sidebar on the left with icons for file management and a search bar.

RESULT: thus the A* searching problem using python is executed successfully.