**A\* Algorithm-**

* A\* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
* A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
* It is essentially a best first search algorithm.

**Working-**

A\* Algorithm works as-

* It maintains a tree of paths originating at the start node.
* It extends those paths one edge at a time.
* It continues until its termination criterion is satisfied.

A\* Algorithm extends the path that minimizes the following function-

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

Here,

* ‘n’ is the last node on the path
* g(n) is the cost of the path from start node to node ‘n’
* h(n) is a heuristic function that estimates cost of the cheapest path from node ‘n’ to the goal node

**Algorithm-**

* The implementation of A\* Algorithm involves maintaining two lists- OPEN and CLOSED.
* OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
* CLOSED contains those nodes that have already been visited.

The algorithm is as follows-

**Step-01:**

* Define a list OPEN.
* Initially, OPEN consists solely of a single node, the start node S.

**Step-02:**

If the list is empty, return failure and exit.

**Step-03:**

* Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
* If node n is a goal state, return success and exit.

**Step-04:**

Expand node n.

**Step-05:**

* If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
* Otherwise, go to Step-06.

**Step-06:**

For each successor node,

* Apply the evaluation function f to the node.
* If the node has not been in either list, add it to OPEN.

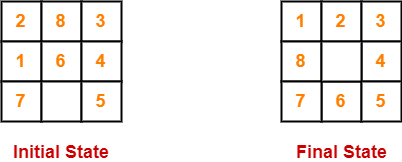
**Step-07:**

Go back to Step-02.

**PRACTICE PROBLEMS BASED ON A\* ALGORITHM-**

**Problem-01:**

Given an initial state of a 8-puzzle problem and final state to be reached-

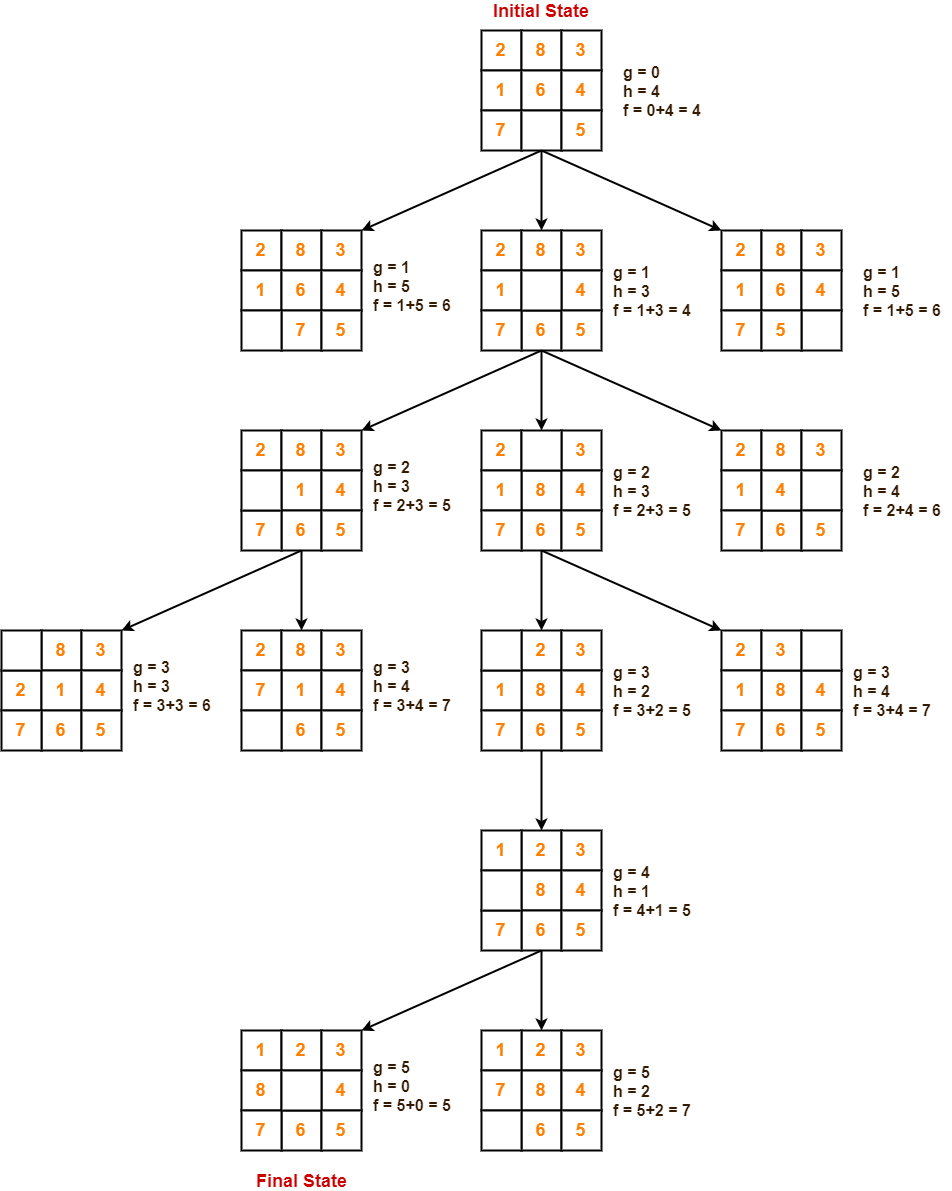


Find the most cost-effective path to reach the final state from initial state using A\* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

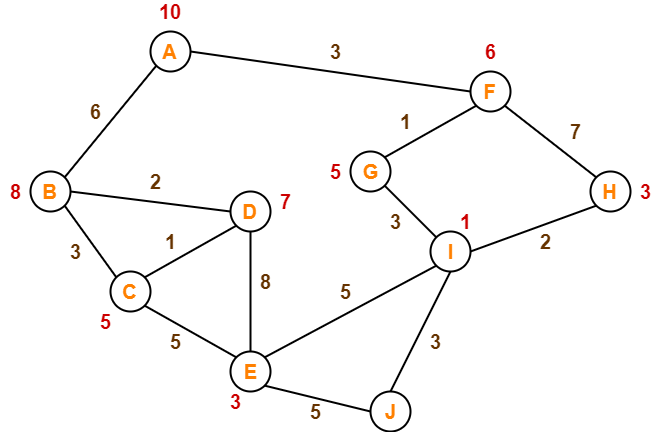
**Solution-**

* A\* Algorithm maintains a tree of paths originating at the initial state.
* It extends those paths one edge at a time.
* It continues until final state is reached.



**Problem-02:**

Consider the following graph-



The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

Find the most cost-effective path to reach from start state A to final state J using A\* Algorithm.

**Solution-**

**Step-01:**

* We start with node A.
* Node B and Node F can be reached from node A.

A\* Algorithm calculates f(B) and f(F).

* f(B) = 6 + 8 = 14
* f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

**Path- A → F**

**Step-02:**

Node G and Node H can be reached from node F.

A\* Algorithm calculates f(G) and f(H).

* f(G) = (3+1) + 5 = 9
* f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node G.

**Path- A → F → G**

**Step-03:**

Node I can be reached from node G.

A\* Algorithm calculates f(I).

f(I) = (3+1+3) + 1 = 8

It decides to go to node I.

**Path- A → F → G → I**

**Step-04:**

Node E, Node H and Node J can be reached from node I.

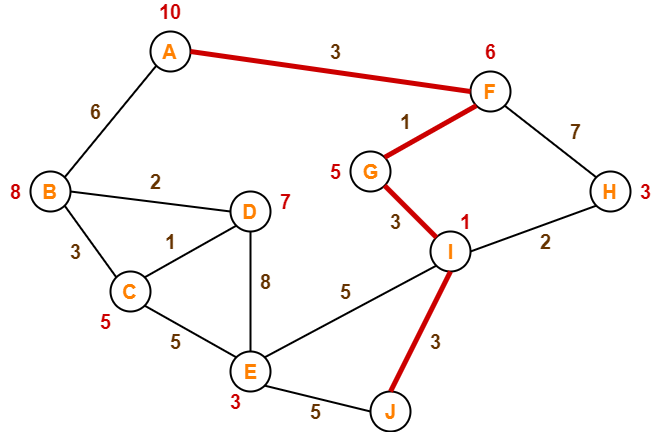
A\* Algorithm calculates f(E), f(H) and f(J).

* f(E) = (3+1+3+5) + 3 = 15
* f(H) = (3+1+3+2) + 3 = 12
* f(J) = (3+1+3+3) + 0 = 10

Since f(J) is least, so it decides to go to node J.

**Path- A → F → G → I → J**

This is the required shortest path from node A to node J.



**Important Note-**

It is important to note that-

* A\* Algorithm is one of the best path finding algorithms.
* But it does not produce the shortest path always.
* This is because it heavily depends on heuristics.

**Advantages:**

1. A\* search algorithm is the best algorithm than other search

algorithms.

2. A\* search algorithm is optimal and complete.

3. This algorithm can solve very complex problems.

**Disadvantages:**

1. It does not always produce the shortest path as it mostly

based on heuristics and approximation.

2. A\* search algorithm has some complexity issues.

3. The main drawback of A\* is memory requirement as it keeps

all generated nodes in the memory, so it is not practical for

various large-scale problems.

**Implementation**

def aStarAlgo(start\_node, stop\_node):

    open\_set = set(start\_node)

    closed\_set = set()

    g = {}               #store distance from starting node

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

    #distance of starting node from itself is zero

    g[start\_node] = 0

    #start\_node is root node i.e it has no parent nodes

    #so start\_node is set to its own parent node

    parents[start\_node] = start\_node

    while len(open\_set) > 0:

        n = None

        #node with lowest f() is found

        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):

                #nodes 'm' not in first and last set are added to first

                #n is set its parent

                if m not in open\_set and m not in closed\_set:

                    open\_set.add(m)

                    parents[m] = n

                    g[m] = g[n] + weight

                #for each node m,compare its distance from start i.e g(m) to the

                #from start through n node

                else:

                    if g[m] > g[n] + weight:

                        #update g(m)

                        g[m] = g[n] + weight

                        #change parent of m to n

                        parents[m] = n

                        #if m in closed set,remove and add to open

                        if m in closed\_set:

                            closed\_set.remove(m)

                            open\_set.add(m)

        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:

            path = []

            while parents[n] != n:

                path.append(n)

                n = parents[n]

            path.append(start\_node)

            path.reverse()

            print('Path found: {}'.format(path))

            return path

        # remove n from the open\_list, and add it to closed\_list

        # because all of his neighbors were inspected

        open\_set.remove(n)

        closed\_set.add(n)

    print('Path does not exist!')

    return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

        #for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

    H\_dist = {

        'A': 11,

        'B': 6,

        'C': 5,

        'D': 7,

        'E': 3,

        'F': 6,

        'G': 5,

        'H': 3,

        'I': 1,

        'J': 0

    }

    return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

    '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)],

}

aStarAlgo('A', 'J')

Output :

Path found: ['A', 'F', 'G', 'I', 'J']

['A', 'F', 'G', 'I', 'J']

