PROBLEM STATEMENT :

Implement A* approach for any suitable application.

OBJECTIVES :

• To develop problem solving abilities for gamifications.
• To apply algorithmic strategies while solving problems
• To develop time and space efficient algorithms

THEORY :

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. It is *complete* and will always find a solution if one exists. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

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

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost
of the cheapest path from n to the goal, we have
f (n) = estimated cost of the cheapest solution through n .

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

Optimality of A*:

The tree-search version of A ∗ is optimal if h(n) is admissible, while the graph-search version is optimal if h(n) is consistent.

Conditions for optimality: Admissibility and consistency

The first condition we require for optimality is that h(n) be an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal. Because g(n) is the actual cost to reach n along the current path, and f (n) = g(n) + h(n), we have as an immediate consequence that f (n) never overestimates the true cost of a solution along the current path through n.

A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A ∗ to graph search. A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n : h(n) ≤ c(n, a, n) + h(n) .

ALGORITHM :

First, we create lists open and closed. open is similar to bfs's queue, and closed is similar to the coloring technique we used before. However, we have to make a list because coloring doesn't store cost information.
So we pop the element with the least f off the open list, and generate its successors. If one of its

successors is the goal, we're golden and can stop the search.
For each successor, calculate its cost and add it to the open list. At the end of this iteration, put the node we popped off the open list on the closed list.

PSEUDO CODE:

```
1:   // A*
2:   initialize the open list
3:   initialize the closed list
-    put the starting node on the open list (you can leave its f at zero)
4:
5:   while the open list is not empty
6:       find the node with the least f on the open list, call it "q"
7:       pop q off the open list
8:       generate q's 8 successors and set their parents to q
9:       for each successor
10:          if successor is the goal, stop the search
11:          successor.g = q.g + distance between successor and q
12:          successor.h = distance from goal to successor
-            successor.f = successor.g + successor.h
13:
-            if a node with the same position as successor is in the OPEN list \
14:              which has a lower f than successor, skip this successor
-            if a node with the same position as successor is in the CLOSED list \
15:              which has a lower f than successor, skip this successor
16:          otherwise, add the node to the open list
17:      end
18:      push q on the closed list
     end
```

INPUT :

Goal State and Initial state.

EXPECTED OUTPUT :

Solution found in “n” steps.

MATHEMATICAL MODEL :

Let S be the solution perspective .
S={s, e, i, o, f, DD, NDD, success, failure}

s = { Initial state of the board consisting of all tiles at particular places }

I = Input of the system → { I1, I2 }

where I1 = { Initial position of tiles }
I2 = { Final position of tiles }

o = Output of the system → { O1 , O2 }

where O1 = { Goal state reached }
O2 = { Goal state not reached }

f = Functions used → { f1}

where f1 = { minimum f(n) ,where f(n) = g(n) + h(n)
where h(n) = no. of unmatched tiles }

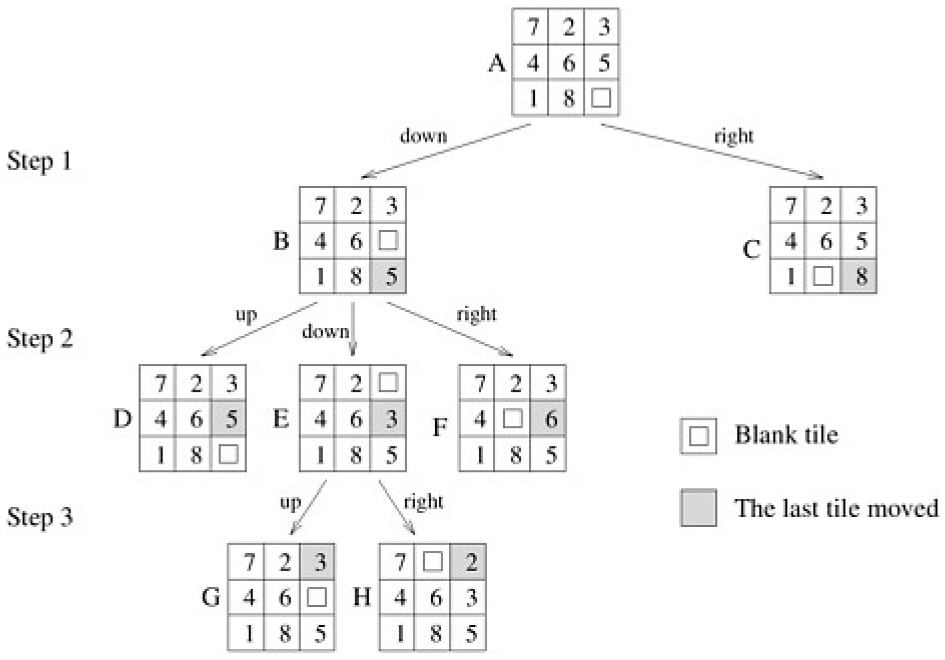
DD - Deterministic data → { Initial position of tiles , heuristic values }

NDD - Non deterministic data → { Number of moves }

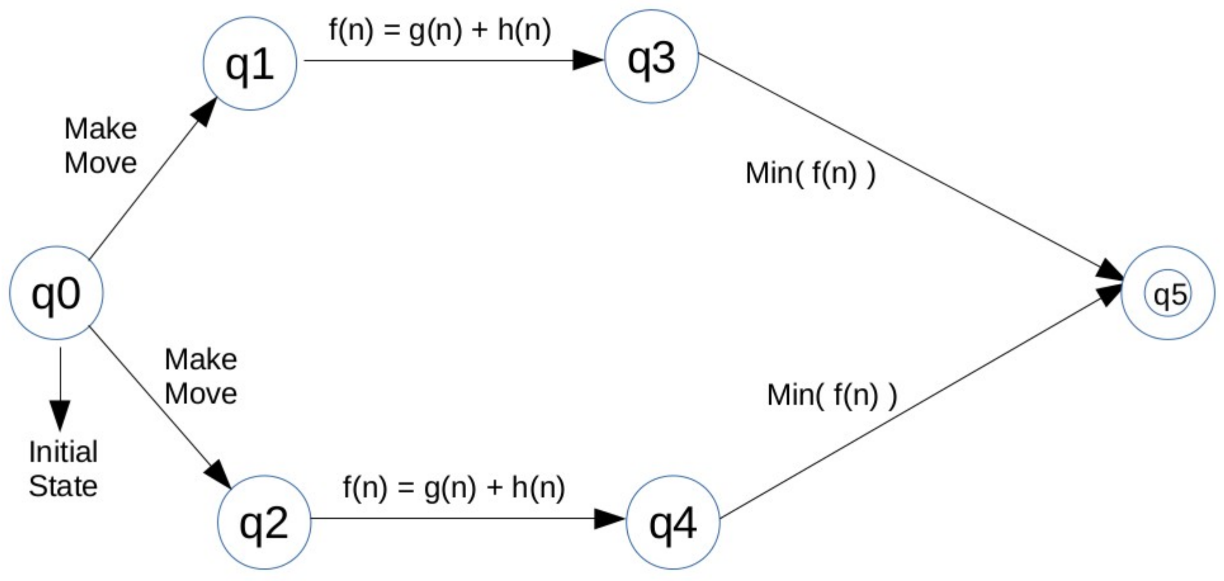
Success - Desired outcome generated → {Goal state reached }

Failure - Desired outcome not generated or forced exit due to system error.

GAME TREE :



STATE DIAGRAM :



TEST-CASES

TEST CASE	INPUT	EXPECTED OUTPUT	OUTPUT ACHIEVED	REMARKS
1.	1_3428657	Solution found in 9 steps	Solution found in 9 steps	Correct
2	1_3425678	Solution found in 9 steps	Solution found in 9 steps	Correct
3.	1234567890_ 1342	Invalid input	Wrong input .Enter in correct fashion	Correct

SPACE AND TIME COMPLEXITIES :

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the length of the solution (the shortest path) *d*: O(b), where *b* is the branching factor (average number of successors per state).This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate. The time complexity is polynomial when the search space is a tree, there is a single goal state, and the heuristic function *h* meets the following condition:

$$\mathbf{h(x) - h^*(x) = O(\log h^*(x))}$$

where h^* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from x to the goal.

CONCLUSION :

Hence we have successfully implemented the A-star algorithm to solve the n-queens problem.

OUTCOMES ACHIEVED :

COURSE OUTCOME	**ACHIEVED(√)**
Problem solving abilities for smart devices.	
Problem solving abilities for gamifications.	√
Problem solving abilities of pervasiveness,embedded security and NLP.	
To solve problems for multicore or distributed,concurrent/Parallel environments	