**ARTIFICIAL INTELLIGENCE**

**EXPERIMENT NO 1 - TOY PROBLEM**

**Q1. What is a toy problem**

**Ans:**  It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms. A toy problem is useful to test and demonstrate methodologies.In the field of artificial intelligence, classical puzzles, games and problems are often used as toy problems. These include sliding-block puzzles, N-Queens problem, missionaries and cannibals problem, tic-tac-toe, chess,Hanoi tower and others.

**Name of Toy Problem - 8 Puzzle Problem**

**Q2. How many approaches do you have for solving the toy problem which you have taken?**

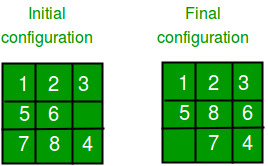
Ans. Here I'll show a Branch and Bound approach to solve the discussed problem. We can also use DFS and BFS to solve this toy problem.

It is the best one from other techniques. It is used to solve very complex problems

In this puzzle solution of 8 puzzle problems is discussed.

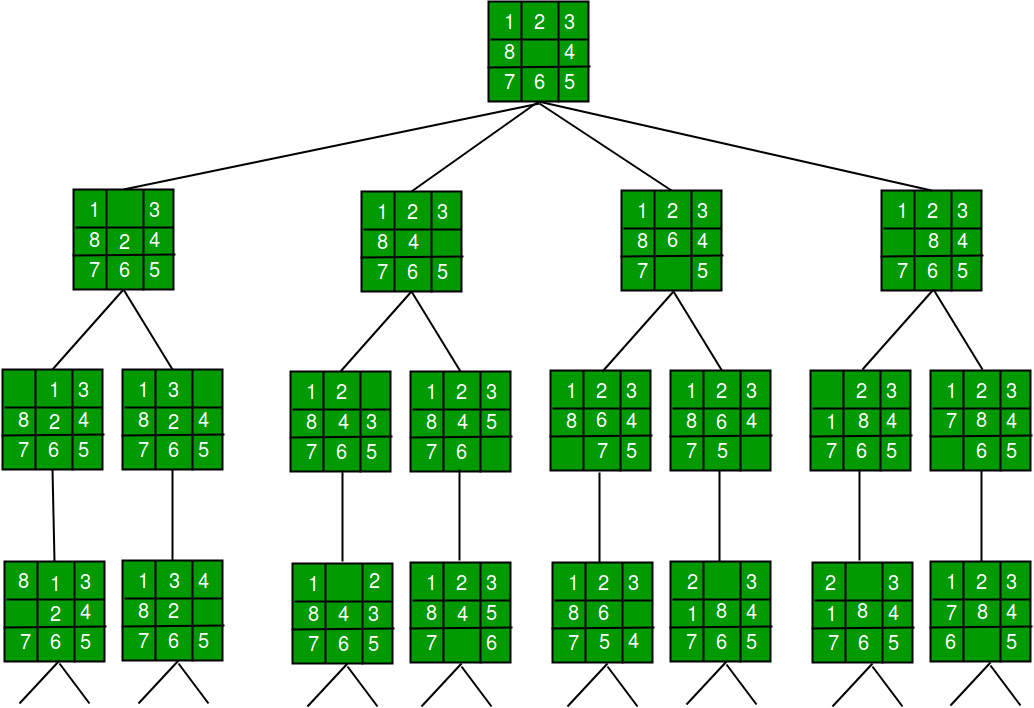
*Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.*

For example,



**1. DFS (Brute-Force)**

We can perform a depth-first search on state space (Set of all configurations of a given problem i.e. all states that can be reached from the initial state) tree.



In this solution, successive moves can take us away from the goal rather than bringing closer. The search of state space tree follows the leftmost path from the root regardless of the initial state. An answer node may never be found in this approach.

**2. BFS (Brute-Force)**

We can perform a Breadth-first search on the state space tree. This always finds a goal state nearest to the root. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

**3. Branch and Bound**

The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses BFS-like search.

There are basically three types of nodes involved in Branch and Bound

**1. Live node** is a node that has been generated but whose children have not yet been generated.

**2. E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

**3. Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Cost function:

Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

C(X) = g(X) + h(X) where

g(X) = cost of reaching the current node

from the root

h(X) = cost of reaching an answer node from X.

**Ideal Cost function for 8-puzzle Algorithm :**

We assume that moving one tile in any direction will have 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below:

c(x) = f(x) + h(x) where

f(x) is the length of the path from root to x

(the number of moves so far) and

h(x) is the number of non-blank tiles not in

their goal position (the number of mis-

-placed tiles). There are at least h(x)

moves to transform state x to a goal state

An algorithm is available for getting an approximation of h(x) which is an unknown value.

**Complete Algorithm:**

/\* Algorithm LCSearch uses c(x) to find an answer node

\* LCSearch uses Least() and Add() to maintain the list

of live nodes

\* Least() finds a live node with least c(x), deletes

it from the list and returns it

\* Add(x) adds x to the list of live nodes

\* Implement list of live nodes as a min-heap \*/

struct list\_node

{

list\_node \*next;

// Helps in tracing path when answer is found

list\_node \*parent;

float cost;

}

algorithm LCSearch(list\_node \*t)

{

// Search t for an answer node

// Input: Root node of tree t

// Output: Path from answer node to root

if (\*t is an answer node)

{

print(\*t);

return;

}

E = t; // E-node

Initialize the list of live nodes to be empty;

while (true)

{

for each child x of E

{

if x is an answer node

{

print the path from x to t;

return;

}

Add (x); // Add x to list of live nodes;

x->parent = E; // Pointer for path to root

}

if there are no more live nodes

{

print ("No answer node");

return;

}

// Find a live node with least estimated cost

E = Least();

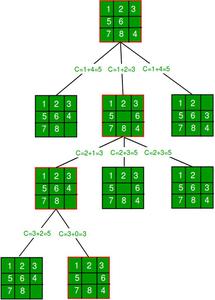
// The found node is deleted from the list of

// live nodes

}

}

Below diagram shows the path followed by the above algorithm to reach final configuration from the given initial configuration of 8-Puzzle. Note that only nodes having the least value of cost function are expanded.



C++ CODE:

#include <bits/stdc++.h>

using namespace std;

#define N 3

struct Node

{

Node\* parent;

int mat[N][N];

int x, y;

int cost;

int level;

};

int printMatrix(int mat[N][N])

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

printf("%d ", mat[i][j]);

printf("\n");

}

}

Node\* newNode(int mat[N][N], int x, int y, int newX,

int newY, int level, Node\* parent)

{

Node\* node = new Node;

node->parent = parent;

memcpy(node->mat, mat, sizeof node->mat);

swap(node->mat[x][y], node->mat[newX][newY]);

node->cost = INT\_MAX;

node->level = level;

node->x = newX;

node->y = newY;

return node;

}

int row[] = { 1, 0, -1, 0 };

int col[] = { 0, -1, 0, 1 };

int calculateCost(int initial[N][N], int final[N][N])

{

int count = 0;

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

if (initial[i][j] && initial[i][j] != final[i][j])

count++;

return count;

}

int isSafe(int x, int y)

{

return (x >= 0 && x < N && y >= 0 && y < N);

}

void printPath(Node\* root)

{

if (root == NULL)

return;

printPath(root->parent);

printMatrix(root->mat);

printf("\n");

}

struct comp

{

bool operator()(const Node\* lhs, const Node\* rhs) const

{

return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);

}

};

void solve(int initial[N][N], int x, int y,

int final[N][N])

{

priority\_queue<Node\*, std::vector<Node\*>, comp> pq;

Node\* root = newNode(initial, x, y, x, y, 0, NULL);

root->cost = calculateCost(initial, final);

pq.push(root);

while (!pq.empty())

{

Node\* min = pq.top();

pq.pop();

if (min->cost == 0)

{

printPath(min);

return;

}

for (int i = 0; i < 4; i++)

{

if (isSafe(min->x + row[i], min->y + col[i]))

{

Node\* child = newNode(min->mat, min->x,

min->y, min->x + row[i],

min->y + col[i],

min->level + 1, min);

child->cost = calculateCost(child->mat, final);

pq.push(child);

}

}

}

}

int main()

{

int initial[N][N] =

{

{1, 2, 3},

{5, 6, 0},

{7, 8, 4}

};

int final[N][N] =

{

{1, 2, 3},

{5, 8, 6},

{0, 7, 4}

};

int x = 1, y = 2;

solve(initial, x, y, final);

return 0;

}

OUTPUT:

1 2 3

5 6 0

7 8 4

1 2 3

5 0 6

7 8 4

1 2 3

5 8 6

7 0 4

1 2 3

5 8 6

0 7 4