NAME:	Eshan Bhuse
UID:	2021300013
SUBJECT	Design and analysis of algorithm
EXPERIMENTNO:	8
AIM:	**Experiment based on branch and bound strategy (15 puzzle problem)**
ALGORITHM:	1. Define N as the size of the puzzle matrix. 2. Define a struct Node to represent each state of the puzzle. 3. Define a function printMat to print the puzzle matrix. 4. Define a function newChild to generate a new state of the puzzle by swapping two elements. 5. Define row and col arrays to represent possible moves in the puzzle. 6. Define a function checkCost to calculate the number of mis-placed tiles in the puzzle. 7. Define a function isSafe to check if a move is valid within the puzzle boundaries. 8. Define a function print to print the solution path. 9. Define a compare function to compare Nodes based on their cost and level. 10. Define a function solve to find the solution for the puzzle. 11. Initialize the priority queue pq with the root node of the puzzle. 12. While pq is not empty: a. Pop the node with the minimum cost from pq. b. If the cost of the popped node is 0, print the solution path and return. c. Generate child nodes by swapping elements in the puzzle matrix and calculate their cost. d. Push the child nodes into pq. 13. End the loop. 14. Define the initial and final state of the puzzle matrices. 15. Call the solve function with the initial matrix, starting position (x, y), and the final matrix.

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
#define N 4

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");
    }
    return 0 ;
}

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;
```

```
48
49      node->cost = INT_MAX;
50
51      node->level = level;
52
53      node->x = newX;
54      node->y = newY;
55
56      return node;
57  }
58
59  int row[] = { 1, 0, -1, 0 };
60  int col[] = { 0, -1, 0, 1 };
61
62  int calculateCost(int initial[N][N], int final[N][N])
63  {
64      int count = 0;
65      for (int i = 0; i < N; i++)
66      for (int j = 0; j < N; j++)
67          if (initial[i][j] && initial[i][j] != final[i][j])
68          count++;
69      return count;
70  }
71
72  int isSafe(int x, int y)
73  {
74      return (x >= 0 && x < N && y >= 0 && y < N);
75  }
76
77  void printPath(Node* root)
78  {
79      if (root == NULL)
80          return;
81      printPath(root->parent);
82      printMatrix(root->mat);
83
84      printf("\n");
85  }
86
87  struct comp
```

```
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);
```

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

                pq.push(child);
            }
        }
    }
}

int main()
{

    int initial[N][N] =
    {
        {1, 2, 3, 4},
        {5, 6, 0, 8},
        {9, 10, 7, 11},
        {13,14, 15, 12}
    };

    int final[N][N] =
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 0}
    };

    int x = 1, y = 2;

    solve(initial, x, y, final);
    cout<<"\n ...Final matrix";
    return 0;
}
```

```
1 2 3 4
5 6 0 8
9 10 7 11
13 14 15 12

1 2 3 4
5 6 7 8
9 10 0 11
13 14 15 12

1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0


 ...Final matrix

...Program finished with exit code 0
Press ENTER to exit console.
```

OBSERVATION:	In above program, we used the branch and bound strategy. To improve the efficiency of the algorithm, a "bound" is used to eliminate branches that are unlikely to lead to the goal state. The bound is a lower limit on the evaluation function value of any solution that can be found by exploring a particular branch. If the bound of a branch is higher than the evaluation function value of the best solution found so far, the branch can be pruned (i.e., not explored further). Time complexity: O(N^2 * N!) where N is the number of tiles in thepuzzle
CONCLUSION:	In this practical, I performed the practical of 15 puzzle problem, where 15 numbers puzzle is provided and there is a space which denoted as 0 and we have to arrange the numbers like final matrix. We solve this problem using branch and bound strategy.