8 puzzle algorithm code:

```
// Program to print path from root node to destination node
// for N*N -1 puzzle algorithm using Branch and Bound
// The solution assumes that instance of puzzle is solvable
#include <bits/stdc++.h>
using namespace std;
#define N 3

// state space tree nodes
struct Node
{
    // stores parent node of current node
    // helps in tracing path when answer is found
    Node* parent;

    // stores matrix
```

```
    int mat[N][N];

    // stores blank tile cordinates
    int x, y;

    // stores the number of misplaced tiles
    int cost;

    // stores the number of moves so far
    int level;
};

// Function to print N x N matrix
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("
");
    }
}

// Function to allocate a new node
Node* newNode(int mat[N][N], int x, int y, int newX,
              int newY, int level, Node* parent)
{
    Node* node = new Node;

    // set pointer for path to root
    node->parent = parent;

    // copy data from parent node to current node
    memcpy(node->mat, mat, sizeof node->mat);

    // move tile by 1 postion
    swap(node->mat[x][y], node->mat[newX][newY]);
```

```
    // set number of misplaced tiles
    node->cost = INT_MAX;

    // set number of moves so far
    node->level = level;

    // update new blank tile cordinates
    node->x = newX;
    node->y = newY;

    return node;
}

// botton, left, top, right
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };

// Function to calculate the the number of misplaced tiles
```

```
// ie. number of non-blank tiles not in their goal position
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;
}

// Function to check if (x, y) is a valid matrix cordinate
int isSafe(int x, int y)
{
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// print path from root node to destination node
void printPath(Node* root)
```

```
{
    if (root == NULL)
        return;
    printPath(root->parent);
    printMatrix(root->mat);

    printf("
");
}

// Comparison object to be used to order the heap
struct comp
{
    bool operator()(const Node* lhs, const Node* rhs)
const
    {
        return (lhs->cost + lhs->level) > (rhs->cost + rhs-
>level);
    }
```

```
};

// Function to solve N*N - 1 puzzle algorithm using
// Branch and Bound. x and y are blank tile coordinates
// in initial state
void solve(int initial[N][N], int x, int y,
           int final[N][N])
{
    // Create a priority queue to store live nodes of
    // search tree;
    priority_queue<Node*, std::vector<Node*>, comp>
pq;

    // create a root node and calculate its cost
    Node* root = newNode(initial, x, y, x, y, 0, NULL);
    root->cost = calculateCost(initial, final);

    // Add root to list of live nodes;
    pq.push(root);
```

```
// Finds a live node with least cost,
// add its childrens to list of live nodes and
// finally deletes it from the list.
while (!pq.empty())
{
    // Find a live node with least estimated cost
    Node* min = pq.top();

    // The found node is deleted from the list of
    // live nodes
    pq.pop();

    // if min is an answer node
    if (min->cost == 0)
    {
        // print the path from root to destination;
        printPath(min);
        return;
```

```
}

// do for each child of min
// max 4 children for a node
for (int i = 0; i < 4; i++)
{
    if (isSafe(min->x + row[i], min->y + col[i]))
    {
        // create a child node and calculate
        // its cost
        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);

        // Add child to list of live nodes
        pq.push(child);
    }
```

```
        }
    }
}

// Driver code
int main()
{
    // Initial configuration
    // Value 0 is used for empty space
    int initial[N][N] =
    {
        {1, 2, 3},
        {5, 6, 0},
        {7, 8, 4}
    };

    // Solvable Final configuration
    // Value 0 is used for empty space
    int final[N][N] =
```

```
    {
        {1, 2, 3},
        {5, 8, 6},
        {0, 7, 4}
    };

    // Blank tile coordinates in initial
    // configuration
    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
```