Source Code

```
using System;
using System.Diagnostics;
namespace MyApp // Note: actual namespace depends on the project name.
{
   internal class Program
   {

      public  struct node
      {
         public int last_move;
         public int distance;
         public int index_of_zero;
         public List<int> grid;
         public node(int last_move, int distance, int index_of_zero, List<int> grid)
         {
            this.last_move = last_move;
            this.index_of_zero = index_of_zero;
            this.distance = distance;
            this.grid = grid;
         }
      }
      static int getInvCount(int[,] ar)
      {
         int inv_count = 0;
         List<int> arr = new List<int>();
         for (int i = 0; i < N; i++)
```

```
    {
        for (int j = 0; j < N; j++)
        {
            arr.Add(ar[i, j]);
        }
    }
    for (int i = 0; i < N * N - 1; i++)
    {
        for (int j = i + 1; j < N * N; j++)
        {

            if (arr[j] != 0 && arr[i] != 0 && arr[i] > arr[j])
                inv_count++;
        }
    }
    return inv_count;
}
static int findXPosition(int[,] puzzle)
{

    for (int i = N - 1; i >= 0; i--)
        for (int j = N - 1; j >= 0; j--)
            if (puzzle[i, j] == 0)
                return N - i;
    return 0;
}
static bool isSolvable(int[,] puzzle)
{
```

```
    int invCount = getInvCount(puzzle);



    if (N % 2 != 0)
        return !(invCount % 2 != 0);

    else
    {
        int pos = findXPosition(puzzle);
        if (pos % 2 != 0)
            return !(invCount % 2 != 0);
        else
            return invCount % 2 != 0;
    }
}

static bool have_solution(List<int> grid)
{
    int[,] g = new int[M, M];

    int x = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {

            g[i, j] = grid[x++];
        }
    }
```

```
    return (isSolvable(g));
}

static bool valid(int x)
{
    return x >= 0 && x < N * N;
}
static void pre()
{
    goal = new List<int>();
    for (int i = 0; i < N * N - 1; i++)
        goal.Add(i + 1);
    goal.Add(0);
}
static void print(List<int> v)
{
    int x = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            Console.Write(v[x++] + " ");
        }
        Console.WriteLine();
    }
}

static int calc_hamming(List<int> grid)
```

```
{
    int c = 1;
    int res = 0;
    for (int i = 0; i < N * N - 1; i++)
    {
        if (grid[i] != c)
            res++;
        c++;

    }
    return res;
}
static int calc_Manhattan(List<int> grid)
{
    int c = 1;
    int res = 0;
    int cx = 0, cy = 0;
    for (int i = 0; i < N * N; i++)
    {
        if (grid[i] != c && grid[i] != 0)
        {
            int x = (grid[i] - 1) / (N);
            int y = Math.Abs(grid[i] - (1 + (x * N)));
            res += Math.Abs(cx - x);
            res += Math.Abs(cy - y);
        }
        cy++;
        if (cy == N)
        {
```

```
            cy = 0;
            cx++;
        }
        c++;
    }
    return res;
}
static int calc_Heuristic(List<int> grid, int option)
{
    if (option == 1)
    {
        return calc_hamming(grid);
    }
    else
    {
        return calc_Manhattan(grid);
    }
}
static void get_solution(List<int> cur, List<int> term_node, Dictionary<List<int>, List<int>> parent,
int dis)
{

    Console.WriteLine("solution is found");
    Console.WriteLine("**********************");
    Console.WriteLine("number of moves = " + dis);
    if (N == 3)
    {
        List<int> cc = cur;
        List<List<int>> ans = new List<List<int>>();
```

```
        int sz = 0;
        while (cc != term_node)
        {
            ans.Add(cc);
            sz++;
            cc = parent[cc];
        }
        ans.Reverse();
        foreach (var vec in ans)
        {
            print(vec);
            Console.WriteLine("==============================");
        }

    }
}
static void A_star_algorithm(List<int> grid, int option)
{
    int ix = -1;
    PriorityQueue<node,int>  not_vis = new PriorityQueue<node, int>();
    for (int i = 0; i < N * N; i++)
    {
        if (grid[i] == 0)
        {
            ix = i;
            break;
        }
    }
    Dictionary<List<int>, int> vis = new Dictionary<List<int>, int>();
```

```
node n = new node(0,0,ix,grid);
not_vis.Enqueue(n, 0);
Dictionary<List<int>, List<int>> parent = new Dictionary<List<int>, List<int>>();
List<int> term = new List<int>();
term.Add(-1);
parent[grid] = term;
int c = calc_Heuristic(grid, option);
if (c == 0)
{
   Console.WriteLine("Time taken: " + before.ElapsedMilliseconds + "ms");
   get_solution(grid, term, parent, 0);
   return;
}
while (true)
{
   var ccc = not_vis.Dequeue();
   List<int> cur = ccc.grid;
   int idx_of_zero = ccc.index_of_zero;
   int vs = ccc.last_move; // move
   int d = ccc.distance;// distence
   List<int> v = new List<int>();
   if (idx_of_zero + N < N * N)
      v.Add(N);
   if (idx_of_zero % N != 0)
      v.Add(-1);
   if ((idx_of_zero + 1) % N != 0)
      v.Add(1);
   if (idx_of_zero - N >= 0)
      v.Add(-1 * N);
```

```
    foreach (var j in v)
    {
        if (j == vs * -1)
        {
            continue;
        }
        int from = idx_of_zero;
        int to = from + j;
        List<int> next = new List<int>();
        next.AddRange(cur);
        int tmp = next[from];
        next[from] = next[to];
        next[to] = tmp;
        int cost = calc_Heuristic(next, option);//O(N*N)
        if (N == 3)
            parent[next] = cur;
        if (cost == 0)
        {
            float after = int.Parse(before.ElapsedMilliseconds.ToString());
            Console.WriteLine("Time taken: " + after/1000.0 + " Second");
            get_solution(next, term, parent,d +1);
            return;
        }
        node nn = new node(j, d+1, j + idx_of_zero, next);
        not_vis.Enqueue(nn, cost + d+1);
    }
}
```

```
}
static void calc_BFS(List<int> grid)
{
   return;
}
static Stopwatch before = Stopwatch.StartNew();
static List<int> grid, GD, goal;
static int N = 0, M = 111;
static void Main(string[] args)
{
   while (true)
   {
      string name= @"C:\Users\EslamSaeed\Desktop\all\";
      Console.WriteLine("Unsolvable puzzles\n");
      Console.WriteLine("[1] 15 Puzzle 1 - Unsolvable");
      Console.WriteLine("[2] 99 Puzzle - Unsolvable Case 1");
      Console.WriteLine("[3] 99 Puzzle - Unsolvable Case 2");
      Console.WriteLine("[4] 9999 Puzzle Unsolvable");
      Console.WriteLine("\nSolvable puzzles\n");
      Console.WriteLine("\nManhattan & Hamming\n");
      Console.WriteLine("[5] 50 Puzzle");
      Console.WriteLine("[6] 99 Puzzle - 1");
      Console.WriteLine("[7] 99 Puzzle - 2");
      Console.WriteLine("[8] 9999 Puzzle");
      Console.WriteLine("\nManhattan Only\n");
      Console.WriteLine("[9] 15 Puzzle 1");
      Console.WriteLine("[10] 15 Puzzle 3");
      Console.WriteLine("[11] 15 Puzzle 4");
      Console.WriteLine("[12] 15 Puzzle 5");
```

```
Console.WriteLine("\nV. Large test case\n");
Console.WriteLine("[13] TEST");
int test = int.Parse(Console.ReadLine());
if (test == 1)
{
   name += "15 Puzzle 1 - Unsolvable";
}
else if (test == 2)
{
   name += "99 Puzzle - Unsolvable Case 1";
}
else if (test == 3)
{
   name += "99 Puzzle - Unsolvable Case 2";
}
else if (test == 4)
{
   name += "9999 Puzzle Unsolvable";
}
else if (test == 5)
{
   name += "50 Puzzle";
}
else if (test == 6)
{
   name += "99 Puzzle - 1";
}
else if (test == 7)
{
```

```
    name += "99 Puzzle - 2";
}
else if (test == 8)
{
    name += "9999 Puzzle";
}
else if (test == 9)
{
    name += "15 Puzzle 1";
}
else if (test == 10)
{
    name += "15 Puzzle 3";
}
else if (test == 11)
{
    name += "15 Puzzle 4";
}
else if (test == 12)
{
    name += "15 Puzzle 5";
}
else if (test == 13)
{
    name += "TEST";
}
else
{
    Console.WriteLine("Enter size of grid");
```

```
int n = int.Parse(Console.ReadLine());
N = n;
grid = new List<int>();
Console.WriteLine("GRID ? ");
for (int i = 0; i < n; i++)
{
   int[] a = Array.ConvertAll(Console.ReadLine().Split(' '), int.Parse);
   for (int j = 0; j < n; j++)
      grid.Add(a[j]);

}
GD = grid;
if (have_solution(grid) == true)
{
   Console.WriteLine("this grid have a solution");
   Console.WriteLine("[1] use A_star_algorithm");
   Console.WriteLine("[2] use BFS_algorithm");
   int opt = int.Parse(Console.ReadLine());
   if (opt == 1)
   {
      Console.WriteLine("[1] use hamming");
      Console.WriteLine("[2] use Manhattan");

      opt = int.Parse(Console.ReadLine());
      Console.WriteLine("Running.....");
      before = Stopwatch.StartNew();
      A_star_algorithm(grid, opt);

   }
```

```
        else
        {
            Console.WriteLine("Running.....");
            before = Stopwatch.StartNew();
            calc_BFS(grid);
        }
    }
    else
        Console.WriteLine("there is no solution for this grid");
}
if (test > 13)
    continue;
name += ".txt";
var file_read = File.ReadLines(name);
grid = new List<int>();
N = 0;
foreach(var line in file_read)
{
    if (line.Length == 0)
        continue;
    if (N == 0)
        N = int.Parse(line);
    else
    {
        int[] a = Array.ConvertAll(line.Split(' '), int.Parse);
        for (int j = 0; j < a.Length; j++)
            grid.Add(int.Parse(a[j].ToString()));
    }
}
```

```
            GD = grid;
            if (have_solution(grid) == true)
            {
                Console.WriteLine("this grid have a solution");
                Console.WriteLine("[1] use A_star_algorithm");
                Console.WriteLine("[2] use BFS_algorithm");
                int opt = int.Parse(Console.ReadLine());
                if (opt == 1)
                {
                    Console.WriteLine("[1] use hamming");
                    Console.WriteLine("[2] use Manhattan");
                    opt = int.Parse(Console.ReadLine());
                    Console.WriteLine("Running.....");
                    before = Stopwatch.StartNew();
                    A_star_algorithm(grid, opt);
                }
                else
                {
                    Console.WriteLine("Running.....");
                    before = Stopwatch.StartNew();
                    calc_BFS(grid);
                }
            }
            else
                Console.WriteLine("there is no solution for this grid");
        }
        before.Stop();
    }
}}
```

Analysis of CODE

`A_star_algorithm:`	O(E*log(V)), E=total number of moves, V=number of states to reach the solution.
Insertion of Priority Queue	O(log(N)),for 1 element. O(Nlog(N)) for n elements.
`calc_hamming`	O(N^2)
`calc_Manhattan`	O(N^2)
Getting available moves	O(1)
GetSolution(Base Case)	O(1),in case n!=3 O(V),in case n==3,such that : V=number of states to reach the solution.
Removing from priority queue	O(log(N)),for 1 element.
Reading files	O(N^2)
Calculate BFS	O(E+V)
Have solution	O(S^2),S=size of grid

Comparison Between Hamming and Manhattan using A* Algorithm

Test Name	type	solvable	MIN Number of moves:	Execution time in ms
[1] 15 Puzzle 1	----------	NO	----------	----------
[2] 99 Puzzle	----------	NO	----------	----------
[3] 99 Puzzle	----------	NO	----------	----------
[4] 9999 Puzzle	----------	NO	----------	----------
[5] 50 Puzzle	Hamming	Yes	18	0.153
	Manhattan		18	0.054
[6] 99 Puzzle - 1	Hamming	Yes	18	0
	Manhattan		18	0
[7] 99 Puzzle - 2	Hamming	Yes	38	0
	Manhattan		38	0
[8] 9999 Puzzle	Hamming	Yes	4	0
	Manhattan		4	0.001
[9] 15 Puzzle 1	Manhattan	Yes	46	4.218
[10] 15 Puzzle 3	Manhattan	Yes	38	0.941
[11] 15 Puzzle 4	Manhattan	Yes	44	1.422
[12] 15 Puzzle 5	Manhattan	Yes	45	30.319
[13] V. Large test case	Hamming	NO	56	22.159
	Manhattan	Yes		

Comparison Between A* Algorithm and BFS Algorithm

A* Algorithm	BFS Algorithm
It is considered to be the best algorithm for N PUZZLE. As it is the fastest one that is able to solve any large puzzle, because it uses priority queue, that makes it a greedy algorithm that reach solution directly . And it neglects all other solutions and focuses on the fastest solution that has minimum number of moves to solve the puzzle.	It has a limit to solve the N puzzle ,it can solve only 3*3 puzzle and can't solve what above. As it tries to reach as many as possible of solutions, and that's what makes it slow.