**1.Source Code**

using System;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Threading.Tasks;

using System.Diagnostics;

namespace ConsoleApp1

{

    internal class main

    {

        public class Node

        {

            public int Levels; **Θ🡪(1)**

            public int Costs; **Θ🡪(1)**

            public int[,] Matrix; **Θ🡪(1)**

            public Node Parent\_node; **Θ🡪(1)**

            public int X\_blank; **Θ🡪(1)**

            public int Y\_blank; **Θ🡪(1)**

  }

        static int index1 = 0, index2 = 0; **Θ🡪(1)**

**public static void Main(string[] args)**

        {

            int[,] Matrix\_Puzzle = Ret\_Mat(ReadFile\_GetPath());**Θ🡪()**

            Stopwatch stopwatch = new Stopwatch();

            int Size\_of\_Matrix = Matrix\_Puzzle.GetLength(0); **Θ🡪(1)**

            int[,] Goal\_OF\_Matrix = Get\_Matrix\_Goal(Size\_of\_Matrix);  **Θ🡪()**

            bool Solvable\_Or\_Not = IS\_Solvable\_Or\_Not(Matrix\_Puzzle,Size\_of\_Matrix);**Θ🡪()**

            char Choice1, Choice2; **Θ🡪(1)**

            if (Solvable\_Or\_Not) **Θ🡪(1)**

            {

                Get\_Zero(Matrix\_Puzzle); **Ω🡪(1) Ο🡪()**

                Console.WriteLine("[1]-A\*"); **Θ🡪(1)**

                Console.WriteLine("[2]-BFS"); **Θ🡪(1)**

                Choice1 = Console.ReadLine()[0]; **Θ🡪(1)**

                if (Choice1 == '1')

                {

                    Console.WriteLine("[1]-Hamming"); **Θ🡪(1)**

                    Console.WriteLine("[2]-Manhatten"); **Θ🡪(1)**

                    Choice2 = Console.ReadLine()[0]; **Θ🡪(1)**

                    if (Choice2 == '1') **Θ🡪(1)**

                    {

                        stopwatch.Start();

                        Solve\_A\_star(Matrix\_Puzzle, Goal\_OF\_Matrix,

index1, index2, Choice2); **Ο🡪(E log(V))**

                    }

                    else

                    {

                        stopwatch.Start();

                        Solve\_A\_star(Matrix\_Puzzle, Goal\_OF\_Matrix,

index1, index2, Choice2); **Ο🡪(E log(V))**

                    }

                }

                if (Choice1 == '2') **Θ🡪(1)**

                {

                    stopwatch.Start();

                    Solve\_BFS(Matrix\_Puzzle, Goal\_OF\_Matrix, index1, index2); **Ο🡪(E log(V))**

                }

            }

            else

            {

                Console.WriteLine("------------------------------------------------------------------------");

                Console.WriteLine("Can't Solve"); **Θ🡪(1)**

            }

            stopwatch.Stop();

            Console.WriteLine("Elapsed Time is {0} ms", stopwatch.ElapsedMilliseconds);

            Console.WriteLine();

            Main(args);

            Console.WriteLine();**Θ🡪(1)**

        }

**public static bool Is\_Same\_Parent(Node Nodes, int i\_index, int j\_index) Θ🡪(1)**

        {

            if (Nodes.Parent\_node != null) **Θ🡪(1)**

            {

                if (i\_index == Nodes.Parent\_node.X\_blank

&& j\_index == Nodes.Parent\_node.Y\_blank) **Θ🡪(1)**

                {

                    return true;

                }

            }

            return false;

        }

**public static void Solve\_BFS(int[,] intial\_Mat, int[,] Goal\_Mat,**

**int x\_index, int y\_index) Ο🡪(E log(V))**

    {

            int Size\_OF\_Directions = 4; **Θ🡪(1)**

            int[] Rows = { 1, 0, -1, 0 }; **Θ🡪(1)**

            int[] Cols = { 0, -1, 0, 1 }; **Θ🡪(1)**

            Queue<Node> queue = new Queue<Node>(); **Θ🡪(1)**

            Node First\_Node = new Node();

            First\_Node.Matrix = intial\_Mat; **Θ🡪(1)**

            First\_Node = Create\_NEW\_Node(intial\_Mat, x\_index, y\_index,

x\_index, y\_index, 0, null); **Θ🡪()**

            First\_Node.Costs = Hamming\_Distance(intial\_Mat, Goal\_Mat); **Θ🡪()**

            queue.Enqueue(First\_Node); **Ο🡪(log(n))**

            while (queue.Count != 0) **Ο🡪(E**)

            {

                Node node = queue.Dequeue();**Ο🡪(log(V))**

                if (node.Costs == 0) **Θ🡪(1)**

                {

                    Print\_Path\_node(node);🡪 **Θ(1)+Θ()+T(N)**

                    Console.WriteLine("# OF Movements = " + node.Levels); **Θ🡪(1)**

                    return;

                }

                for (int i = 0; i < Size\_OF\_Directions; i++) **Θ🡪(1)**

                {

                    if (Safe\_MAtrix(node.X\_blank + Rows[i], node.Y\_blank + Cols[i], node.Matrix.GetLength(0))) **Θ🡪(1)**

                    {

                        Node child = Create\_NEW\_Node(node.Matrix, node.X\_blank,

                        node.Y\_blank, node.X\_blank + Rows[i],

                        node.Y\_blank + Cols[i],

                        node.Levels + 1, node); **Θ🡪()**

                        child.Costs = Hamming\_Distance(child.Matrix, Goal\_Mat); **Θ🡪()**

                        if (!(Is\_Same\_Parent(node, child.X\_blank, child.Y\_blank))) **Θ🡪(1)**

                        {

                            queue.Enqueue(child); **Ο🡪(log(V))**

                        }

                    }

                }

            }

        }

**public static void Get\_Zero(int[,] Matrix) Ω🡪(1) Ο🡪()**

        {

            for (int i = 0; i < Matrix.GetLength(0); i++)

            {

                for (int j = 0; j < Matrix.GetLength(1); j++)

                {

                    if (Matrix[i, j] == 0) **Θ🡪(1)**

                    {

                        index1 = i; **Θ🡪(1)**

                        index2 = j; **Θ🡪(1)**

                        break;

                    }

                }

            }

        }

**public static int[,] Ret\_Mat(string File\_Path) total=#of itrations\* Θ(Body)Θ🡪()**

        {

            StreamReader sr = new StreamReader(File\_Path);

            int N\_size = int.Parse(sr.ReadLine());

            int[,] matrix = new int[N\_size, N\_size];

            string firstline = sr.ReadLine();

            for (int i = 0; i < N\_size; i++) **Θ🡪()**

            {

                string[] s;

                if (firstline != "") **Θ🡪(1)**

                {

                    s = firstline.Split(' ');

                    firstline = "";

                }

                Else

{

                    s = sr.ReadLine().Split(' ');

}

                for (int j = 0; j < N\_size; j++) **Θ🡪(n)**

                {

                    matrix[i, j] = int.Parse(s[j]); **Θ🡪(1)**

                }

            }

            sr.Close();

            return matrix;

        }

**public static bool IS\_Solvable\_Or\_Not(int[,] matrix,**

**int Size\_of\_Matrix) total=#of itrations\* Θ(Body)=Θ🡪()**

        {

            int Count\_Solve = 0; **Θ🡪(1)**

            int Index\_OF\_Zero = 0;

            int Counter = 0;

            int[] arr\_Matrix = new int[Size\_of\_Matrix \* Size\_of\_Matrix];

            for (int i = 0; i < Size\_of\_Matrix; i++) **Θ🡪()**

            {

                for (int j = 0; j < Size\_of\_Matrix; j++) **Θ🡪(n)**

                {

                    arr\_Matrix[Counter++] = matrix[i, j];

                }

            }

            for (int i = 0; i < arr\_Matrix.Length; i++) **Θ🡪()**

            {

                if (arr\_Matrix[i] == 0) **Θ🡪(1)**

                {

                    Index\_OF\_Zero = i / Size\_of\_Matrix;

                    continue;

                }

                int j = i + 1;

                for (int j=i+1; j < arr\_Matrix.Length; j++) **Θ🡪(n)**

                {

                    if (arr\_Matrix[i] > arr\_Matrix[j]) **Θ🡪(1)**

                    {

                        if (arr\_Matrix[j] != 0)

                        {

                            Count\_Solve++;

                        }

                    }

                }

            }

            if (Size\_of\_Matrix % 2 != 0 && Count\_Solve % 2 == 0) **Θ🡪(1)**

            {

                return true;

            }

            else if (Size\_of\_Matrix % 2 == 0 && Count\_Solve % 2 != 0) **Θ🡪(1)**

            {

                if (Index\_OF\_Zero % 2 == 0)

                {

                    return true;

                }

                return false;

            }

            else if (Size\_of\_Matrix % 2 == 0 && Count\_Solve % 2 == 0) **Θ🡪(1)**

            {

                if (Index\_OF\_Zero % 2 != 0) **Θ🡪(1)**

                {

                    return true;

                }

                return false;

            }

            else

                return false;

        }

**public static int[,] Get\_Matrix\_Goal(int SizeofMatrix) Θ🡪()**

        {

            int[,] Goal\_OF\_Matrix = new int[SizeofMatrix, SizeofMatrix]; **Θ🡪(1)**

            int Counter = 1; **Θ🡪(1)**

            int i = 0;

            while (i < SizeofMatrix) **total=#of itrations\* Θ(Body)=Θ🡪()**

            {

                int j = 0;

                while (j < SizeofMatrix) **Θ🡪(n)**

                {

                    if (j == SizeofMatrix - 1 && i == SizeofMatrix - 1) **Θ🡪(1)**

                    {

                        Goal\_OF\_Matrix[i, j] = 0;

                        break;

                    }

                    Goal\_OF\_Matrix[i, j] = Counter;

                    Counter++;

                    j++;

                }

                i++;

            }

            return Goal\_OF\_Matrix;

        }

**public static void Print\_Matrices(int[,] MATRIX,**

**int Mat\_Size)total=#of itrations\* Θ(Body)=Θ🡪()**

        {

            for (int i = 0; i < Mat\_Size;i++) **Θ🡪(n)**

            {

                for (int j = 0; j < Mat\_Size j++) **Θ🡪(n)**

                {

                    Console.Write(MATRIX[i, j] + " ");

                }

                Console.WriteLine();

            }

        }

**public static int Hamming\_Distance(int[,] Intial\_Matrix,**

**int[,] Goal\_Matrix) total=#of itrations\* Θ(Body)=Θ🡪()**

        {

            int Count\_Distance = 0; **Θ🡪(1)**

            for (int i = 0;< Intial\_Matrix.GetLength(0); i++) **Θ🡪()**

            {

                for (int j = 0; j < Intial\_Matrix.GetLength(1); j++) **Θ🡪(n)**

                {

                    if (!(Goal\_Matrix[i, j] == Intial\_Matrix[i, j])) **Θ🡪(1)**

                    {

                        if (Intial\_Matrix[i, j] != 0) **Θ🡪(1)**

                        {

                            Count\_Distance++;

                        }

                    }

                }

            }

            return Count\_Distance;

        }

**public static int Manhaten\_Dist(int[,] Matrix,**

**int size) total=#of itrations\* Θ(Body)=Θ🡪()**

        {

            int manhaten\_distance = 0;

            for (int i = 0; i < size; i++) **Θ🡪()**

            {

                for (int j = 0; j < size; j++) **Θ🡪(n)**

                {

                    int Col, Row; **Θ🡪(1)**

                    int mat = Matrix[i, j] - 1; **Θ🡪(1)**

                    Row = (mat / size) - i;

                    Col = (mat % size) - j;

                    int len = (i \* size + j) + 1; **Θ🡪(1)**

                    if (Matrix[i, j] != 0 && (len != Matrix[i, j])) **Θ🡪(1)**

                    {

                        manhaten\_distance += Math.Abs(Row) + Math.Abs(Col); **Θ🡪(1)**

                    }

                }

            }

            return manhaten\_distance;

        }

**public static Node Create\_NEW\_Node(int[,] Matrix, int x\_index, int y\_index, int New\_x\_index, int New\_y\_index,**

**int NumOFlevels, Node Parent\_nodes) total=#of itrations\* Θ(Body)=Θ🡪()**

        {

            Node node = new Node();

            if (Parent\_nodes == null) **Θ🡪(1)**

            {

                node.Matrix = Matrix;

                node.Parent\_node = null;

            }

            else

            {

                node.Parent\_node = Parent\_nodes; **Θ🡪(1)**

                //node.matrix = mat;

                //node.matrix = mat

                node.Matrix = new int[Parent\_nodes.Matrix.GetLength(0), Parent\_nodes.Matrix.GetLength(1)]; **Θ🡪(1)**

                for (int i = 0; i < Matrix.GetLength(0); i++) **Θ🡪()**

                {

                    for (int j = 0; j < Matrix.GetLength(1); j++) **Θ🡪(n)**

                    {

                        node.Matrix[i, j] = Matrix[i, j]; **Θ🡪(1)**

                    }

                }

            }

            node.Matrix = SwapOFindices(node.Matrix, x\_index, y\_index,

New\_x\_index, New\_y\_index); **Θ🡪(1)**

            node.Costs = int.MaxValue; **Θ🡪(1)**

            node.Levels = NumOFlevels; **Θ🡪(1)**

            node.X\_blank = New\_x\_index; **Θ🡪(1)**

            node.Y\_blank = New\_y\_index; **Θ🡪(1)**

            return node;

        }

**public static int[,] SwapOFindices(int[,] Matrix, int x\_index, int y\_index, int Newx\_index, int Newy\_index) Θ🡪(1)**

        {

            int temp\_Swap;

            temp\_Swap = Matrix[x\_index, y\_index]; **Θ🡪(1)**

            Matrix[x\_index, y\_index] = Matrix[Newx\_index, Newy\_index]; **Θ🡪(1)**

            Matrix[Newx\_index, Newy\_index] = temp\_Swap; **Θ🡪(1)**

            return Matrix;

        }

**public static bool Safe\_MAtrix(int x\_index, int y\_index,**

**int Size\_OF\_Matrix) Θ🡪(1)**

        {

            if (x\_index >= 0 && x\_index < Size\_OF\_Matrix) **Θ🡪()**

            {

                if (y\_index >= 0 && y\_index < Size\_OF\_Matrix) **Θ🡪(1)**

                {

                    return true;

                }

                else

                {

                    return false;

                }

            }

            else

            {

                return false;

            }

        }

**public static void Print\_Path\_node(Node Elements\_node) Θ(1)+Θ()+T(N)**

        {

            if (Elements\_node == null)

            {

                return;

            }

            Print\_Path\_node(Elements\_node.Parent\_node); **T(N)**

            Print\_Matrices(Elements\_node.Matrix, Elements\_node.Matrix.GetLength(0)); **Θ()**

            Console.WriteLine();

        }

**public static void Solve\_A\_star(int[,] Intial\_Matrix, int[,] Goal, int x\_index, int y\_index, char Method) Ο🡪(E log(V))**

    {

            int Size\_OF\_Directions = 4; **Θ🡪(1)**

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

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

            PriorityQueue<Node, int> queue = new PriorityQueue<Node, int>(); **Θ🡪(1)**

            Node First\_node = new Node();

            First\_node.Matrix = Intial\_Matrix;

            First\_node = Create\_NEW\_Node(Intial\_Matrix, x\_index, y\_index,

x\_index, y\_index, 0, null); **Θ()**

            if (Method == '1') **Θ🡪(1)**

            {

                First\_node.Costs = Hamming\_Distance(Intial\_Matrix, Goal); **Θ()**

            }

            else

            {

                First\_node.Costs = Manhaten\_Dist(Intial\_Matrix, Intial\_Matrix.GetLength(0)); **Θ()**

            }

            queue.Enqueue(First\_node, First\_node.Costs + First\_node.Levels); **Ο(log(V))**

            while (queue.Count != 0) **Ο(E)**

            {

                Node node = queue.Dequeue(); 🡪 **Ο(log(V))**

                if (node.Costs == 0) **🡪Θ(1)**

                {

                    Print\_Path\_node(node); **Θ(1)+Θ()+T(N)**

                    Console.WriteLine("# of movements = " + node.Levels);

                    return;

                }

                int i = 0;

                while (i < Size\_OF\_Directions) **🡪 Ο(1)🡪 total=#of itrations\* Θ(Body)=Θ🡪()**

                {

                    if (Safe\_MAtrix(node.X\_blank + Rows[i], node.Y\_blank + Cols[i], node.Matrix.GetLength(0))) **🡪Θ(1)**

                    {

                        Node Child\_node = Create\_NEW\_Node(node.Matrix, node.X\_blank, node.Y\_blank, node.X\_blank + Rows[i],  node.Y\_blank + Cols[i], node.Levels + 1, node); **🡪Θ()**

                        if (Method == '1')

                        {

                            Child\_node.Costs = Hamming\_Distance(Child\_node.Matrix, Goal); **🡪Θ()**

                        }

                        else

                        {

                            Child\_node.Costs = Manhaten\_Dist(Child\_node.Matrix, Child\_node.Matrix.GetLength(0)); **🡪Θ()**

                        }

                        if (!(Is\_Same\_Parent(node, Child\_node.X\_blank, Child\_node.Y\_blank))) **🡪Θ(1)**

                        {

                            queue.Enqueue(Child\_node, Child\_node.Costs + Child\_node.Levels);**🡪Θ(log(V))**

                        }

                    }

                    i++;

                }

            }

        }

    }

}

***2.Comparing Table***

|  |
| --- |
| Sample Test Cases |
| |  |  |  |  | | --- | --- | --- | --- | | Case Number | Manhatten | Hamming | Movement | | #Case 1 | **19 ms** | **15 ms** | **8** | | #Case 2 | **65 ms** | **70 ms** | **20** | | #Case 3 | **35 ms** | **44 ms** | **14** | |  |  |  |  | | #Case 4 | **25 ms** | **29 ms** | **5** | | #Case 5 | **52 ms** | **56 ms** | **11** | | #Case 6 | **145 ms** | **68 ms** | **24** | |

|  |
| --- |
| Complete Hamming & Manhattan Test Cases |
| |  |  |  |  | | --- | --- | --- | --- | | Case Number | Manhatten | Hamming | Movement | | #Case 1 | **1562 ms** | **1831 ms** | **18** | | #Case 2 | **111 ms** | **104 ms** | **18** | | #Case 3 | **222 ms** | **228 ms** | **38** | |  |  |  |  | | #Case 4 | **1540 ms** | **1514 ms** | **4** | |
| **Complete Manhattan ONLY Test Cases** |
| |  |  |  | | --- | --- | --- | | Case Number | Manhatten | Movement | | #Case 1 | **3797 ms** | **46** | | #Case 2 | **1416 ms** | **38** | | #Case 3 | **528 ms** | **44** | |  |  |  | | #Case 4 | **24381 ms** | **45** | |

|  |
| --- |
| Complete Very Large Test Cases |
| |  |  |  | | --- | --- | --- | | Case Number | Manhatten | Movement | | #Case 1 | **ms** | **56** | |

A picture containing map

Description automatically generated