**N*-*puzzel *Document***

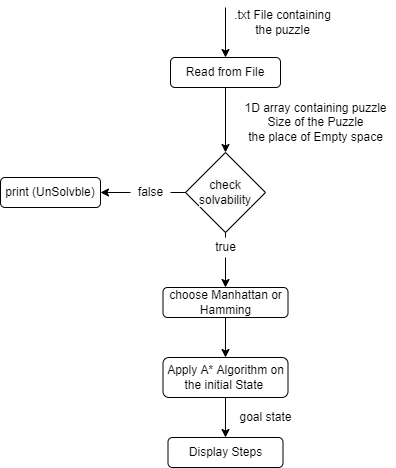
**1)Introduction:**

N-Puzzle is a board game. It consists of (N) numbered squared tiles in random order, and one blank space The goal is to rearrange the tiles in order by making sliding moves that use the empty space, using the fewest moves.

Our Program is an N-Puzzle Solver to read a Puzzle from a text file then check if the puzzle has a solution or not.

If it has a solution we create a node for our initial state and start solving it using A\* search algorithm, our program has two types of heuristic value which are Manhattan and hamming distance the user can choose either of them to be the heuristic value of A\*, and if it doesn’t have a solution print “Unsolvable” then the program terminates.

***2)*pipeline*:***



**readFromFile():**

**Input**: A text file containing the puzzle.

**Output**: A 2D array with the puzzle ,1D array with the puzzle, Row and column of the blank space and the puzzle size.

**Description:** This function takes the content of the text file and parse it to integers so we can solve it in our program.

……………………………………………………………………………………………………

**calculateInvCount():**

**Input:** 1D array with the puzzle and the puzzle size.

**Output**: The inversion counts of the given puzzle.

**Description**: This function takes the puzzle and iterate over each element if the we find the number after a given item smaller than it we increment the inversion count.

……………………………………………………………………………………………………

**cheackSolvability():**

**Input:** 1D array of the puzzle and the puzzle size and the row of the blank space.

**Output:** True if the puzzle has a solution and False if it doesn’t.

**Description:** It checks if the size of the puzzle is odd and the inversion count is even then it has a solution, else if the puzzle size is even we check for the row given in input if it’s odd and inversion count is even and vice versa then the puzzle is solvable if not one of these cases then it doesn’t have a solution.

……………………………………………………………………………………………………

**AstarAlgorithm():**

**Input**: Initial state of the puzzle.

**Output**: Goal state of the puzzle.

**Description:** This function searches for the solution in a Priority Queue it add the initial state then generate the possible moves (using **generateNextMoves()**) according to the position of the blank space, it iterates till we find a state with a heuristic value of zero which means no misplaced numbers and that’s the goal state.

**generateNextMoves():**

input: State Node

output: Void

Description: This function takes a node as parent and checks for the possibility for a new move with each move available it creates a new node calculate its heuristic value and add to the priority queue for next inspection in **AstarAlgorithm()**

……………………………………………………………………………………………………

**calculateHaming:**

**Input:** 2D array with the puzzle and the puzzle size.

**Output:** hamming cost

**Description:**hamming distance is The number of blocks in the wrong position

it compare the goal matrix with the matrixes if they aren`t equal increase the hamming cost

by 1.

……………………………………………………………………………………………………

**calculateManhattan**

**Input:** 1D array with the puzzle and the puzzle size.

**Output:** Manhattan Distance

**Description:** sum of the distances of each tile from its goal position

……………………………………………………………………………………………………

***3)The* source *code:***

* **The Main Function:**

using System;

using System.Diagnostics;

using System.IO;

using System.Linq;

using System.Text.RegularExpressions;

namespace Npuzzle

{

    class Program

    {

        static void Main(string[] args)

        {

            int size = 0; θ(1)

            int blankRow = 0, blankCol = 0; θ(1)

            int[] puzzle1D; θ(1)

            int[,] puzzle2D = readFromFile("TEST.txt", out puzzle1D, out size, out blankRow,

out blankCol); //TBD

            //Main menue

            if (cheackSolvability(puzzle1D, size, blankRow))

            {

                Console.WriteLine("Solvable"); θ(1)

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

                Console.WriteLine("Choose the heuristic Value type");

                Console.WriteLine("[1] Hamming Distance");

                Console.WriteLine("[2] Manhattan Distance");

                int type = Convert.ToInt32(Console.ReadLine()); θ(1)

                int heruicValue = 0; θ(1)

                if (type == 1)

                {

                    Stopwatch stopwatch = new Stopwatch();

                    // Begin timing

                    stopwatch.Start();

                    System.Threading.Thread.Sleep(500);

                    Console.WriteLine();

                    heruicValue = AstarFunctions.calculateHamming(puzzle2D, size); O(S^2)

                    State\_Node initState = new State\_Node(puzzle2D, size, blankRow, blankCol,

heruicValue, 0, null, 1); θ(1)

  State\_Node goal = AstarFunctions.AstarAlgorithm(initState); O(E log(V))

                    int minMoves = goal.currentSteps; θ(1)

                    if (size == 3)

                        AstarFunctions.displaySteps(goal);

                    Console.WriteLine("Minimum # Of Moves = " + minMoves); θ(1)

                    Console.WriteLine();

                    // Stop timing

                    stopwatch.Stop();

                    Console.WriteLine("Time : {0}", stopwatch.Elapsed);

                }

                else if (type == 2)

                {

                    Stopwatch stopwatch2 = new Stopwatch();

                    // Begin timing

                    stopwatch2.Start();

                    System.Threading.Thread.Sleep(500);

                    Console.WriteLine();

                    heruicValue = AstarFunctions.calculateManhattan(puzzle2D, size); O(S^2)

                    State\_Node initState = new State\_Node(puzzle2D, size, blankRow, blankCol,

heruicValue, 0, null, 2); θ(1)

                    State\_Node goal = AstarFunctions.AstarAlgorithm(initState); O(E log(V))

                    int minMoves = goal.currentSteps; θ(1)

                    if (size == 3)

                        AstarFunctions.displaySteps(goal);

                    Console.WriteLine("Minimum # Of Moves = " + minMoves); θ(1)

                    Console.WriteLine();

                    // Stop timing

                    stopwatch2.Stop();

                    Console.WriteLine("Time : {0}", stopwatch2.Elapsed);

                }

            }

            else

                Console.WriteLine("UnSolvable"); θ(1)

            // Stop timing

        }

        //Total of the function = O(N^2)

        public static int[,] readFromFile(string path, out int[] p, out int s, out int

blankRow, out int blankCol)

        {

            /\*This function reads the puzzle from the file and Parse it into 1D AND 2D array

and we save the position of blank cell to for later Operations\*/

            string contet = File.ReadAllText(path);

            contet = Regex.Replace(contet, @"\s+",",");

            if (contet[contet.Length - 1] == ',') θ(1)

                contet = contet.Remove(contet.Length - 1);

            int[] puzzle1D = contet.Split(' ', ',').Select(z =>

Convert.ToInt32(z)).ToArray();

            int size = puzzle1D[0]; θ(1)

            int bR = 0, bC = 0; θ(1)

            puzzle1D = puzzle1D.Skip(1).ToArray();

            int[,] puzzle2D = new int[size, size]; θ(1)

            int c = 0; θ(1)

            //Total = θ(N^2) \* θ(1) = θ(N^2)

            for (int x = 0; x < size; x++)

            {

                for (int y = 0; y < size; y++)

                {

                    puzzle2D[x, y] = puzzle1D[c];

                    if (puzzle2D[x, y] == 0)

                    {

                        bR = x;

                        bC = y;

                    }

                    c++;

                }

            }

            blankRow = bR; θ(1)

            blankCol = bC; θ(1)

            s = size; θ(1)

            p = puzzle1D; θ(1)

            return puzzle2D θ(1)

        }

        //Total = //θ(S^2)

        static int calculateInversionCount(int[] puzzle, int size)

        {

            //Calculate invCount for all numbers except blank space

            int inversionCount = 0; θ(1)

            //θ(S^2)

            for (int i = 0; i < size \* size - 1; i++)

            {

                for (int j = i + 1; j < size \* size; j++)

                {

                    if (puzzle[i] != 0 && puzzle[j] != 0 && puzzle[i] > puzzle[j]) θ(1)

                        inversionCount++; θ(1)

                }

            }

            return inversionCount;

        }

        //Total = θ(S^2)

        static bool cheackSolvability(int[] puzzle, int size, int blankIndex)

        {

            int inversionCount = calculateInversionCount(puzzle, size);//θ(S^2)

            //checking if the puzzle has a solution

            if (size % 2 != 0) θ(1)

            {

                if (inversionCount % 2 == 0) θ(1)

                    return true θ(1)

            }

            else if (size % 2 == 0) θ(1)

            {

                if (inversionCount % 2 != 0 && blankIndex % 2 == 0)

                    return true; θ(1)

                else if (inversionCount % 2 == 0 && blankIndex % 2 != 0)

                    return true; θ(1)

            }

            return false; θ(1)

        }

    }

}

* **The PriorityQ Class:**

using System;

namespace Npuzzle

{

    class PriorityQ

    {

        State\_Node[] heapArr; **θ(1)**

        int length = 0; **θ(1)**

**Total = θ(1)**

        public PriorityQ()

        {

            heapArr = new State\_Node[100000000];

        }

**θ(1)**

        public void insertKey(State\_Node key)

        {

            length = length + 1; **θ(1)**

            heapArr[length] = null; **θ(1)**

            increaseKey(length, key); **O(log(V))**

        }

**//Total = O(log(V))**

        public void increaseKey(int i, State\_Node key)

        {

            heapArr[i] = key; **θ(1)**

            while (i > 1 && heapArr[i / 2].cost >= heapArr[i].cost)

            {

                exchange(ref heapArr[i / 2], ref heapArr[i]); **θ(1)**

                i = i / 2; **θ(1)**

            }

        }

**// Total = O(log(V))**

        public State\_Node extractMin()

        {

            if (length == 0) **θ(1)**

            {

                throw new InvalidOperationException("Queue is Empty");

            }

            State\_Node minimum = heapArr[1]; **θ(1)**

            heapArr[1] = heapArr[length]; **θ(1)**

            length = length - 1; **θ(1)**

            minHeapify(1, length); **O(log(V))**

            return minimum; **θ(1)**

        }

**// Total = θ(1)**

        public bool isEmpty()

        {

            if (length == 0)

                return true;

            return false;

        }

**// Total = O(log(V))**

        void minHeapify(int i, int heapSize)

        {

            int left = 2 \* i; **θ(1)**

            int right = 2 \* i + 1; **θ(1)**

            int least; **θ(1)**

            if (left <= heapSize && heapArr[left].cost < heapArr[i].cost) **θ(1)**

                least = left;

            else **θ(1)**

                least = i;

            if (right <= heapSize && heapArr[right].cost < heapArr[least].cost) **θ(1)**

                least = right;

            if (least != i) **θ(1)**

            {

                exchange(ref heapArr[i], ref heapArr[least]); **θ(1)**

                minHeapify(least, heapSize); **O(log(V))**

            }

        }

**// Total = θ(1)**

        void exchange(ref State\_Node x, ref State\_Node y)

        {

            State\_Node t = x;

            x = y;

            y = t;

        }

    }

}

* **Node class:**

namespace Npuzzle

{

    class State\_Node

    {

**θ(1) - for all lines**

        public int[,] state;

        public int huristicType;

        public State\_Node parent = null;

        public int blankRow;

        public int blankCol;

        public int currentSteps;

        public int cost;

        public int heruicValue;

        public int puzzleSize;

        public bool down, up, left, right;

**// Total = θ(1)**

        public State\_Node(int[,] state, int puzzleSize, int blankRow, int blankCol, int

heruicValue, int level, State\_Node parent, int huristicType)

        {

            this.state = state;

            this.puzzleSize = puzzleSize;

            this.parent = parent;

            this.blankRow = blankRow;

            this.blankCol = blankCol;

            this.currentSteps = level;

            this.heruicValue = heruicValue;

            this.huristicType = huristicType;

            /\*We assign the moves state according to the position of the blank cell in row

and coloumn so we know what moves can be done\*/

            if (this.blankRow > 0) θ(1)

                this.up = true;

            if (this.blankRow < this.puzzleSize - 1) **θ(1)**

                this.down = true;

            if (this.blankCol > 0) **θ(1)**

                this.left = true;

            if (this.blankCol < this.puzzleSize - 1) **θ(1)**

                this.right = true;

        }

    }

}

* **AstarFunctions class:**

using System;

namespace Npuzzle

{

    class AstarFunctions

    {

        public static PriorityQ Q = new PriorityQ(); **θ(**1)

**Total = O(E log (V))**

        public static State\_Node AstarAlgorithm(State\_Node initailState)

        {

            Q.insertKey(initailState); **O(log(V))**

            State\_Node min; **θ(1)**

**//Total = O(E) \* O(Log(V)) = O(E log(V))**

            while (!Q.isEmpty())

            {

                //Extracting the top of the heap which has the minimum cost;

                min = Q.extractMin(); **O(log(V))**

                /\*if the heuristic value = 0 that means no misplaced values and that's the

goal state\*/

                if (min.heruicValue == 0) **θ(1)**

                    return min; **θ(1)**

                //getting the possibilites of the moves then add it to the Queue

                AstarFunctions.generateNextMoves(min); **Total = O(log (V))**

            }

            return null; **θ(1)**

        }

**//Total = O(log (V))**

        public static void generateNextMoves(State\_Node Parent)

        {

            //What I need to do here is to get the child Nodes then add it to the Queue

            /\*Create a new node with each new state and add it to the queue if it's diffrent

from its GrandParent\*/

            //Calculate Hamming OR Manhattan for every new state

            if (Parent.up) **θ(1)**

            {

                //This means that (blankCell) can move up

                if (Parent.huristicType == 1) **θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length);

**//θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow - 1, Parent.blankCol]); **θ(1)**

                    int hammingDistance = calculateHamming(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow - 1, Parent.blankCol, hammingDistance,

Parent.currentSteps + 1, Parent, 1); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

                else if (Parent.huristicType == 2) **θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length); **θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow - 1, Parent.blankCol]); **θ(1)**

                    int ManhattanDistance = calculateManhattan(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow - 1, Parent.blankCol, ManhattanDistance,

Parent.currentSteps + 1, Parent, 2); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(v))**

                    }

                }

            }

            if (Parent.down) **// Total = θ(1)**

            {

                //This means that (blankCell) can move down

                if (Parent.huristicType == 1**)//θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length);**//θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow + 1, Parent.blankCol]); **θ(1)**

                    int hammingDistance = calculateHamming(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow + 1, Parent.blankCol, hammingDistance,

Parent.currentSteps + 1, Parent, 1);

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

                else if (Parent.huristicType == 2) **Total = θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length);**//θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow + 1, Parent.blankCol]); **θ(1)**

                    int ManhattanDistance = calculateManhattan(newState, Parent.puzzleSize);

**θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow + 1, Parent.blankCol, ManhattanDistance,

Parent.currentSteps + 1, Parent, 2); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) θ(1)

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

            }

            if (Parent.left**)// Total = θ(1)**

            {

                //This means that (blankCell) can move left

                if (Parent.huristicType == 1) **θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length); **θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow, Parent.blankCol - 1]); **θ(1)**

                    int hammingDistance = calculateHamming(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow, Parent.blankCol - 1, hammingDistance,

Parent.currentSteps + 1, Parent, 1); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

                else if (Parent.huristicType == 2) **θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length);**θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow, Parent.blankCol - 1]); **θ(1)**

                    int ManhattanDistance = calculateManhattan(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow, Parent.blankCol - 1, ManhattanDistance,

Parent.currentSteps + 1, Parent, 2); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

            }

            if (Parent.right) **Total = θ(1)**

            {

                //This means that (blankCell) can move right

                if (Parent.huristicType == 1) **θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length);**//θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow, Parent.blankCol + 1]); **θ(1)**

                    int hammingDistance = calculateHamming(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow, Parent.blankCol + 1, hammingDistance,

Parent.currentSteps + 1, Parent, 1); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

                else if (Parent.huristicType == 2) **Tatal = θ(1)**

                {

                    int[,] newState = new int[Parent.puzzleSize, Parent.puzzleSize]; **θ(1)**

                    Array.Copy(Parent.state, newState, Parent.state.Length); **θ(state.Length)**

                    swapBlank(ref newState[Parent.blankRow, Parent.blankCol], ref

newState[Parent.blankRow, Parent.blankCol + 1]); **θ(1)**

                    int ManhattanDistance = calculateManhattan(newState,

Parent.puzzleSize); **θ(N^2)**

                    State\_Node newNode = new State\_Node(newState, Parent.puzzleSize,

Parent.blankRow, Parent.blankCol + 1, ManhattanDistance,

Parent.currentSteps + 1, Parent, 2); **θ(1)**

                    if (!((Parent.parent != null && newNode.blankRow ==

Parent.parent.blankRow && newNode.blankCol ==

Parent.parent.blankCol))) **θ(1)**

                    {

                        newNode.cost = newNode.currentSteps + newNode.heruicValue; **θ(1)**

                        Q.insertKey(newNode); **O(log(V))**

                    }

                }

            }

        }

**//Total = θ(N^2)**

        public static int calculateHamming(int[,] puzzle2D, int size)

        {

            /\*This function gets the hamming distance by comparing the puzzle with a sample

of its goal when we find a diffrence in place we increament the distance\*/

            int[,] gaolSample = new int[size, size]; **θ(1)**

            int hamming = 0, s = 1; **θ(1)**

**//Total = θ(1) \* θ(N^2)**

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

            {

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

                {

                    gaolSample[i, j] = s; **θ(1)**

                    if (puzzle2D[i, j] != 0 && puzzle2D[i, j] != gaolSample[i, j]) **θ(1)**

                        hamming++; **θ(1)**

                    s++;

                }

            }

            return hamming; **θ(1)**

        }

**//Total =θ(N^2)**

        public static int calculateManhattan(int[,] Puzzle2D, int size)

        {

            /\*This function gets the manhattan distance by calculating the horizontal and

vertical distance between the current cell and the correct one in the goal\*/

            int manhattanDistance = 0, correctRow = 0, correctCol = 0; **θ(1)**

            int currentRow = 0, currentCol = 0; **θ(1)**

**//Total = θ(S^2)**

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

            {

                currentRow = i / size; **θ(1)**

                currentCol = i % size; **θ(1)**

                if (Puzzle2D[currentRow, currentCol] != 0) **θ(1)**

                {

                    correctRow = (Puzzle2D[currentRow, currentCol] - 1) / size; **θ(1)**

                    correctCol = ((Puzzle2D[currentRow, currentCol] - 1) % size); **θ(1)**

                    manhattanDistance += Math.Abs(correctRow - currentRow) +

Math.Abs(correctCol - currentCol);**θ(1)**

                }

            }

            return manhattanDistance;

        }

**//Total = θ(1)**

        static void swapBlank(ref int oldBlank, ref int newBlank)

        {

            //Noraml Swap

            int tmp = oldBlank;

            oldBlank = newBlank;

            newBlank = tmp;

        }

**//Total = θ(1) + θ(1) + T(N^2)**

        public static void displaySteps(State\_Node s)

        {

            //To display the movements we took to reach goal state in 8-Puzzle

            if (s == null)

                return;

            displaySteps(s.parent);

            for (int i = 0; i < 3; i++)

            {

                for (int j = 0; j < 3; j++)

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

                Console.WriteLine();

            }

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

        }

    }

}

**4)ComparingTable*:***

***Hamming & Manhattan***

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Manhattan time** | **Hamming Time** | **Number Of Moves** |
| 50 Puzzle.txt | 00:00:00.6170554 | 00:00:01.0008608 | 18 |
| 99 Puzzle – 1.txt | 00:00:00.5550396 | 00:00:00.5412025 | 18 |
| 99 Puzzle – 2.txt | 00:00:00.5390743 | 00:00:00.5365375 | 38 |
| 9999 Puzzle.txt | 00:00:00.5637809 | 00:00:00.5349893 | 4 |

***V large test***

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Manhattan time** | **Hamming Time** | **Number Of Moves** |
| Test.txt | 00:00:37.3268703 | 00:01:22.4732168 | 56 |