A picture containing text

Description automatically generated

**N-Puzzle**

Algorithm Documentation

Icon

Description automatically generated with medium confidence

Team-ID: T-050

Team Member:

1. Ahmed Osama Mansour
2. Muhammed Ashraf Said
3. Yara El-Sayed Ali

* **Important Function:**

1. Show Steps (State):

Using Stack Data Structure and Pointer in Parent to Print Solution,

Push The Final State in stack and use pointer on parent to push the parent in stack (in While loop and stop looping when I reach in the initial state (parent of initial state is null) and print solution on Console).

1. Hamming (Matrix):

Compare The Number in The Right Index In Puzzle if True counter+=0 ---- Else counter +=1;

1. Manhattan (Matrix, length of Puzzle):

Show Number of Steps to Reach Goal Index for Each Index in matrix.

Using Reminder and divide option to solve it   
Reminder: ((Value of Index -1) % Size) – Column Index

Divide: (Value of Index -1) / Size – Row Index

and sum Positive number Of Reminder, Divide.

1. Is Solvable or Not (Matrix (1-Dimension, Size of Puzzle)):

Calculate number of inversions Based on Index By Index from first to last if index is greater than any index after it number of inversions++ Based on Number that are less than it and when the loop finish check if Size of puzzle odd and num of inversions is even then return true, OR if size of puzzle is even and number of inversions is odd and place of zero is in even place then return true, OR if size is even and number of inversions is even and place of zero is odd place then return true Else Return False "Not Solvable".

**Introduction Of Code:**

1. First Reading Puzzle From File And Save Matrix in "Two\_D\_Puzzle" and Save Index Of Zero In " indexofZero\_X" X-Dimension, "indexofZero\_Y" Y- Dimension
2. In Switch Case It have 4 Option:
3. Solve Puzzle in Hamming
4. Solve Puzzle in Manhattan
5. Go To Next Puzzle in Array " File Path++ "
6. Exit Program
7. In First Option in Switch Case (Hamming):

Create Object From A-star And Send Parameter (2D-Matrix, size of Puzzle, True If Hamming, Index of Zero in X, Y Dimension)

"new astar(Two\_D\_Puzzle, size, true, indexofZero\_X, indexofZero\_Y);"

-----------------------------------------------------------------------------------------------------

In Second Option in Switch Case (Manhattan):

Create Object From A-star And Send Parameter (2D-Matrix, size of Puzzle, False If Manhattan, Index of Zero in X, Y Dimension)

"new astar(Two\_D\_Puzzle, size, false, indexofZero\_X, indexofZero\_Y);"

1. In Constructor A-star:

Save Data of Puzzle and using function" Optimal\_N\_puzzel"

To Save The Goal That I want to reach (Stop Condition) and Create Object from Priority Queue (Data Structure) and Call Function " initialState '' To create The First Initial Puzzle

" initialState(start, ref old\_Of\_X, ref old\_Of\_Y, size)"

1. Convert 2-Dimension Matrix in One Dimension using function " ONE\_D\_Matrix " To Check If Puzzle Solvable or Not
2. In Solvable Function " is\_Solvable\_OR\_Not "
3. If Not Solvable Print in Console " Not Solvable!!! ", If Solvable Print in Console " Solvable " and go to create the initial node Based on Hamming or Manhattan Function and Check If the Matrix is the goal or not Based On H-Of-X (Hamming or Manhattan Function) and Push the Initial State in Priority-Queue

(P-Q Have on Element Now It is the Initial State) and Go to Resolve Function to Solve Puzzle.

1. In Resolve Function " Resolve ":

Create Temp Key Value Pair That equal Each Element in Dequeue from Priority-Queue and Push The Unique-Int "push"

In Hash Set.

1. To Create Children:

If Index of zero can Move Down:

Create The Second Constructor In State Class and pass parameter (Parent of State , Matrix , Level of children, new Index Of Zero in X – Y Dimension, Size Of Puzzle)

" new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x + 1, Temp.Key.New\_y, this.size); " and check If Zero Move Down That it Is The Same With Parent Of Parent If True: Skip ----- If False: Calculate H-Of-X Based On Hamming or Manhattan and check if It the Goal call function "ShowSteps(new\_Down\_State);" To Show Steps, and check If Down Puzzle In Hash Set "Visited" Push it In Priority Queue Else Not Push it In P-Q.

If Index of zero can Move Left:

Create The Second Constructor In State Class and pass parameter (Parent of State, Matrix, Level of children, new Index Of Zero in X – Y Dimension, Size Of Puzzle)

" new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x, Temp.Key.New\_y - 1, this.size)"

and check If Zero Move Left That it Is The Same With Parent Of Parent If True: Skip ----- If False: Calculate H-Of-X Based On Hamming or Manhattan and check if It the Goal call function "ShowSteps(new\_Down\_State);" To Show Steps, and check If Left Puzzle In Hash Set "Visited" Push it In Priority Queue Else Not Push it In P-Q.

If Index of zero can Move Up:

Create The Second Constructor in State Class and pass parameter (Parent of State, Matrix, Level of children, new Index of Zero in X – Y Dimension, Size of Puzzle)

" new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x - 1, Temp.Key.New\_y, Temp.Key.size) " and check If Zero Move up That it Is The Same With Parent Of Parent If True: Skip ----- If False: Calculate H-Of-X Based On Hamming or Manhattan and check if It the Goal call function "ShowSteps(new\_Down\_State);" To Show Steps, and check If Up Puzzle In Hash Set "Visited" Push it In Priority Queue Else Not Push it In P-Q.

If Index of zero can Move Right:

Create The Second Constructor In State Class and pass parameter (Parent of State, Matrix, Level of children, new Index Of Zero in X – Y Dimension, Size Of Puzzle)

" new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x, Temp.Key.New\_y + 1, this.size) " and check If Zero Move Right That it Is The Same With Parent Of Parent If True: Skip ----- If False: Calculate H-Of-X Based On Hamming or Manhattan and check if It the Goal call function "ShowSteps(new\_Down\_State);" To Show Steps, and check If Right Puzzle In Hash Set "Visited" Push it In Priority Queue Else Not Push it In P-Q.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace N\_Puzzle**

**{**

**class Priority\_Queue**

**{**

**List<KeyValuePair<State, int>> Tree;**

**public Priority\_Queue() Total = θ(1)**

**{**

**Tree = new List<KeyValuePair<State, int>>();**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public bool IsEmpty() Total= θ(1)**

**{**

**if (Tree.Count == 0)**

**{**

**return true;**

**}**

**else**

**{**

**return false;**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**private void Mini\_Heapify(int x) Total= O(log(N))**

**{**

**// Because it is zero Index**

**int Right\_term = (2 \* x) + 2; -> θ(1)**

**int Parent\_of\_Node = x / 2; -> θ(1)**

**int left\_term = (2 \* x) + 1; -> θ(1)**

**int Lowest = x; -> θ(1)**

**if (left\_term < Tree.Count && Tree[left\_term].Value < Tree[Lowest].Value) -> θ(1)**

**{**

**Lowest = left\_term; -> θ(1)**

**}**

**if (Right\_term < Tree.Count && Tree[Right\_term].Value < Tree[Lowest].Value) -> θ(1)**

**{**

**Lowest = Right\_term; -> θ(1)**

**}**

**if (Lowest != x) -> θ(1)**

**{**

**KeyValuePair<State, int> Temp = Tree[x];**

**Tree[x] = Tree[Lowest]; -> θ(1)**

**Tree[Lowest] = Temp; -> θ(1)**

**Mini\_Heapify(Lowest) ; -> O(log(N))**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public void Mini\_Heap\_Insert(KeyValuePair<State, int> Key)**

**Total= O(log(N))**

**{**

**Tree.Add(Key);**

**int Length = Tree.Count - 1;**

**while (Length > 0)**

**{**

**int len = (Length - 1) / 2;**

**if (Tree[len].Value > Tree[Length].Value)**

**{**

**KeyValuePair<State, int> Temp = Tree[Length];**

**Tree[Length] = Tree[len];**

**Tree[len] = Temp;**

**Length = len; //parent = i/2**

**}**

**else**

**{**

**break;**

**}**

**} }**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**private KeyValuePair<State, int> Heap\_Extract\_Mini() Total= O(log(N))**

**{**

**if (Tree.Count==0)**

**{**

**KeyValuePair<State, int> newp = new KeyValuePair<State, int>(null,-1);**

**return newp;**

**}**

**KeyValuePair<State, int> mini\_Value = Tree[0];**

**Tree[0] = Tree[Tree.Count - 1];**

**Tree.RemoveAt(Tree.Count - 1);**

**Mini\_Heapify(0);**

**return mini\_Value;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public void Enqueue(KeyValuePair<State, int> new\_node)**

**Total= O(log(N))**

**{**

**Mini\_Heap\_Insert(new\_node);**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public KeyValuePair<State, int> Dequeue() Total= O(log(N))**

**{**

**return Heap\_Extract\_Mini();**

**}**

**}**

**}**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**using System;**

**using System.Collections.Generic;**

**using System.Security.Cryptography;**

**using System.Text;**

**namespace N\_Puzzle**

**{**

**class State**

**{**

**public State parient; -> θ(1)**

**public int[,] Matrix; -> θ(1)**

**public int Old\_X; -> θ(1)**

**public int Old\_y; -> θ(1)**

**public int New\_x; -> θ(1)**

**public int New\_y; -> θ(1)**

**public int G\_Of\_X; -> θ(1) // number of Movement or levels**

**public int H\_Of\_X; -> θ(1) // cost based on Manhatten or Hamming**

**public int cost; -> θ(1)**

**public int size; -> θ(1)**

**public int path; -> θ(1)**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public State(int[,] Temp,int old\_Of\_X,int old\_Of\_Y, int size)**

**{**

**this.size = size; -> θ(1)**

**this.Matrix = new int[size, size]; -> θ(1)**

**for (int start = 0; start < size; start++) -> θ(N^2)**

**{**

**for (int start2 = 0; start2 < size; start2++)**

**{**

**this.Matrix[start, start2] = Temp[start, start2];**

**}**

**}**

**//index of Zero**

**this.Old\_X = -1; -> θ(1)**

**this.Old\_y = -1; -> θ(1)**

**this.New\_x = old\_Of\_X; θ(1)**

**this.New\_y = old\_Of\_Y; -> θ(1)**

**//---------> Because it is the first initial state <---------**

**this.G\_Of\_X = 0 ; θ(1)**

**MD5 md5Hasher = MD5.Create(); O(N)**

**var hashed = md5Hasher.ComputeHash(Encoding.UTF8.GetBytes(ConvertMatrixToString(Temp, size)));**

**var ivalue = BitConverter.ToInt32(hashed, 0);**

**this.path = ivalue; -> θ(1)**

**this.parient = null; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public State (State new\_P,int[,]Temp,int Old\_G\_Of\_X,int old\_Of\_X,int old\_Of\_Y,int New\_Of\_x,int New\_Of\_y,int size)**

**{**

**this.size = size; -> θ(1)**

**this.G\_Of\_X = Old\_G\_Of\_X + 1; -> θ(1)**

**this.Matrix = new int[size, size]; -> θ(1)**

**for (int start = 0;start<size;start++) -> O(N^2)**

**{**

**for (int start2=0;start2<size;start2++) {**

**this.Matrix[start, start2] = Temp[start, start2];**

**}**

**}**

**this.New\_x = New\_Of\_x; -> θ(1)**

**this.New\_y = New\_Of\_y; -> θ(1)**

**this.Old\_X = old\_Of\_X; -> θ(1)**

**this.Old\_y = old\_Of\_Y; -> θ(1)**

**int temp = this.Matrix[New\_Of\_x, New\_Of\_y]; -> θ(1)**

**this.Matrix[New\_Of\_x, New\_Of\_y]= this.Matrix[old\_Of\_X, old\_Of\_Y]; -> θ(1)**

**this.Matrix[old\_Of\_X, old\_Of\_Y] = temp; -> θ(1)**

**this.path = ConvertMatrixToString(Temp, size).GetHashCode();-> θ(1)**

**this.parient = new\_P; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public int Total\_Cost() { Total = θ(1)**

**this.cost = G\_Of\_X + H\_Of\_X;**

**return this.cost;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public string ConvertMatrixToString(int[,] arr, int length) Total = θ(N^2)**

**{**

**string StringMatrix = ""; -> θ(1)**

**for (int i = 0; i < length; i++) -> { θ(N^2)**

**for (int j = 0; j < length; j++)**

**{**

**StringMatrix += arr[i, j].ToString();**

**}**

**}**

**return StringMatrix; -> θ(1)**

**}}}**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace N\_Puzzle**

**{**

**class astar Total = θ(1)**

**{**

**int size;**

**int[] One\_D;**

**int[,] Goal;**

**Priority\_Queue pqueue;**

**bool Is\_Hamming\_OR\_Manhatten;**

**bool Compare\_With\_Parient;**

**HashSet<int> Visited = new HashSet<int>();**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public static int[] ONE\_D\_Matrix(int[,] Two\_D\_Puzzle, int size) Total = O(N^2)**

**{**

**int length = (size) \* (size); -> θ(1)**

**int[] arr1D = new int[length];**

**//int indexofZero\_X = -1, indexofZero\_Y = -1;**

**int x = 0, y = 0; -> θ(1)**

**for (int i = 0; i < length; i++) -> O(N^2)**

**{**

**arr1D[i] = Two\_D\_Puzzle[x, y]; -> θ(1)**

**y++; -> θ(1)**

**if (y == size) -> θ(1)**

**{**

**x++; -> θ(1)**

**y = 0; -> θ(1)**

**}**

**}**

**return arr1D; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**private static bool is\_Solvable\_OR\_Not(int[] Matrix, int length) Total = O(S^2)**

**{**

**int Number\_Of\_inversions = 0; -> θ(1)**

**int spaceIndex = -1; -> θ(1)**

**//compare from first cell to the pre last**

**for (int i = 0; i < Matrix.Length - 1; i++) -> O(N^2)**

**{**

**if (Matrix[i] == 0) -> θ(1)**

**{**

**spaceIndex = i / length; -> θ(1)**

**continue; -> θ(1)**

**}**

**//compare with the cell after i cell till the last cell**

**for (int j = i + 1; j < Matrix.Length; j++) -> O(N^2)**

**{**

**if (Matrix[j] == 0) -> θ(1)**

**{**

**continue; -> θ(1)**

**}**

**else if (Matrix[i] > Matrix[j]) -> θ(1)**

**{**

**Number\_Of\_inversions++; -> θ(1)**

**}**

**}**

**}**

**if (length % 2 != 0 && Number\_Of\_inversions % 2 == 0) -> θ(1)**

**{**

**return true; -> θ(1)**

**}**

**else if (length % 2 == 0 && Number\_Of\_inversions % 2 != 0 && spaceIndex % 2 == 0) -> θ(1)**

**{**

**return true; -> θ(1)**

**}**

**else if (length % 2 == 0 && Number\_Of\_inversions % 2 == 0 && spaceIndex % 2 != 0) -> θ(1)**

**{**

**return true; -> θ(1)**

**}**

**return false; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public bool check\_Parient\_Step(State child) Total = θ(1)**

**{**

**bool check = false; -> θ(1)**

**if (child.parient.Old\_X == child.New\_x && child.parient.Old\_y == child.New\_y)**

**-> θ(1)**

**{**

**return true; -> θ(1)**

**}**

**return check; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**private void ShowSteps(State S) Total = O(N^2\*……)**

**{**

**int CounterOfSteps = 1; -> θ(1)**

**Stack<State> Save\_Steps = new Stack<State>(); -> θ(1)**

**while (S.parient != null) ->** **O(……)**

**{**

**Save\_Steps.Push(S); \*\*\*\*-> θ(1)**

**S = S.parient; -> θ(1)**

**}**

**// push initial state in stack**

**Save\_Steps.Push(S); -> θ(1)**

**// pop initial state**

**State TempIteration = Save\_Steps.Pop();-> θ(1)**

**while (Save\_Steps.Count > 0) -> O(N^2\*……)**

**{**

**// print Board Of Puzzle for each step**

**for (int i = 0; i < TempIteration.size; i++) -> O(N^2)**

**{**

**for (int j = 0; j < TempIteration.size; j++)**

**{**

**Console.Write(TempIteration.Matrix[i, j] + " ");**

**}**

**Console.WriteLine();**

**}**

**Console.WriteLine();**

**TempIteration = Save\_Steps.Pop();-> θ(1)**

**Console.WriteLine("Step Number : " + CounterOfSteps); -> θ(1)**

**CounterOfSteps++; -> θ(1)**

**}**

**for (int i = 0; i < S.size; i++) -> O(N^2)**

**{**

**for (int j = 0; j < S.size; j++)**

**{**

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

**}**

**Console.WriteLine();**

**}**

**return;**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**private int Manhattan(int[,] Matrix, int length) Total = O(N^2)**

**{**

**int Reminder = 0, divide = 0, counter = 0; -> θ(1)**

**for (int i = 0; i < length; i++) -> O(N^2)**

**{**

**for (int j = 0; j < length; j++)**

**{**

**if (Matrix[i, j] != 0 && Matrix[i, j] != ((i \* length) + j + 1)) -> θ(1)**

**{**

**divide = (Matrix[i, j] - 1) / length; -> θ(1)**

**Reminder = (Matrix[i, j] - 1) % length; -> θ(1)**

**int check1 = divide - i; -> θ(1)**

**int check2 = Reminder - j; -> θ(1)**

**if (check1 < 0) -> θ(1)**

**{**

**check1 \*= -1; -> θ(1)**

**}**

**if (check2 < 0) -> θ(1)**

**{**

**check2 \*= -1; -> θ(1)**

**}**

**counter += check1 + check2; -> θ(1)**

**}**

**}**

**}**

**return counter; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**private int Hamming(int[,] Matrix) Total = O(N^2)**

**{**

**int counter = 0; -> θ(1)**

**for (int i = 0; i <this.size ; i++) -> O(N^2)**

**{**

**for (int j = 0; j < this.size; j++)**

**{**

**if (Matrix[i, j] != Goal[i,j] && Matrix[i, j] != 0) -> θ(1)**

**{**

**counter++; -> θ(1)**

**}**

**}**

**}**

**return counter; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public int[,] Optimal\_N\_puzzel(int length) Total = O(****N^2)**

**{**

**int counter = 0; -> θ(1)**

**int[,] Matrix = new int[length, length]; -> O(N^2)**

**for (int i = 0; i < length; i++) -> O(N^2)**

**{**

**for (int j = 0; j < length; j++)**

**{**

**counter++; -> θ(1)**

**Matrix[i, j] = counter; -> θ(1)**

**if (i == (length - 1) && j == (length - 1)) -> θ(1)**

**{**

**Matrix[i, j] = 0; -> θ(1)**

**}**

**}**

**}**

**return Matrix; -> θ(1)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public astar(int[,] start, int size, bool Is\_Hamming\_OR\_Manhatten, int old\_Of\_X, int old\_Of\_Y)**

**{**

**this.Goal = Optimal\_N\_puzzel(size); -> θ(N^2)**

**this.size = size; -> θ(1)**

**this.Is\_Hamming\_OR\_Manhatten = Is\_Hamming\_OR\_Manhatten; -> θ(1)**

**pqueue = new Priority\_Queue();-> θ(1)**

**//if true it is same with parient**

**initialState(start, ref old\_Of\_X, ref old\_Of\_Y, size); -> θ(S^2)**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public void initialState(int[,] Temp, ref int old\_Of\_X, ref int old\_Of\_Y, int size)**

**{ Total = O(E S^2 Log(V))**

**int x = 0; -> θ(1)**

**int y = 0; -> θ(1)**

**One\_D = ONE\_D\_Matrix(Temp, size); -> θ(N^2)**

**x = Environment.TickCount; -> θ(1)**

**if (is\_Solvable\_OR\_Not(One\_D, size)) -> θ(S^2)**

**{**

**Console.WriteLine("Solvable"); -> θ(1)**

**State initialMatrix = new State(Temp, old\_Of\_X, old\_Of\_Y, size); -> θ(1)**

**if (Is\_Hamming\_OR\_Manhatten) -> θ(1)**

**{**

**initialMatrix.H\_Of\_X = Hamming(Temp); -> θ(N^2)**

**}**

**else**

**{**

**initialMatrix.H\_Of\_X = Manhattan(initialMatrix.Matrix, size); -> θ(N^2)**

**}**

**if (initialMatrix.H\_Of\_X == 0) -> θ(1)**

**{**

**Console.Write("Number Of Movement = " + initialMatrix.G\_Of\_X); -> θ(1)**

**Console.WriteLine();**

**return; -> θ(1)**

**}**

**pqueue.Enqueue(new KeyValuePair<State, int>(initialMatrix, initialMatrix.Total\_Cost())); O(N^2)**

**Resolve(); O(E Log(V))**

**y = Environment.TickCount; -> θ(1)**

**Console.WriteLine("Time Taken Is : " + (y - x)+ " Ms "); θ(1)**

**}**

**else**

**{**

**Console.WriteLine("Not Solvable!!!"); θ(1)**

**return; θ(1)**

**}**

**}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**public void Resolve() Total = O(E Log(V))**

**{**

**KeyValuePair<State, int> Temp; θ(1)**

**while (!pqueue.IsEmpty())//pqueue.Count!=0**

**{**

**Temp = pqueue.Dequeue(); O(Log N)**

**Visited.Add(Temp.Key.path);**

**if (Temp.Key.New\_x < this.size - 1)//Temp.Key.Move.down**

**{**

**State new\_Down\_State = new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x + 1, Temp.Key.New\_y, this.size); θ(N^2)**

**Compare\_With\_Parient = check\_Parient\_Step(new\_Down\_State); θ(1)**

**if (!Compare\_With\_Parient) θ(1)**

**{**

**if (Is\_Hamming\_OR\_Manhatten) θ(1)**

**{**

**new\_Down\_State.H\_Of\_X = Hamming(new\_Down\_State.Matrix); θ(N^2)**

**}**

**else**

**{**

**new\_Down\_State.H\_Of\_X = Manhattan(new\_Down\_State.Matrix, this.size); θ(N^2)**

**}**

**if (new\_Down\_State.H\_Of\_X == 0) θ(1)**

**{**

**ShowSteps(new\_Down\_State);**

**return;**

**}**

**if (!Visited.Contains(new\_Down\_State.path)) θ(1)**

**{**

**pqueue.Enqueue(new KeyValuePair<State, int>(new\_Down\_State, new\_Down\_State.Total\_Cost())); O(LogN)**

**}**

**}**

**}**

**if (Temp.Key.New\_y > 0)//Temp.Key.Move.left θ(1)**

**{**

**State new\_Left\_State = new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x, Temp.Key.New\_y - 1, this.size); θ(N^2)**

**Compare\_With\_Parient = check\_Parient\_Step(new\_Left\_State); θ(1)**

**if (!Compare\_With\_Parient) θ(1)**

**{**

**if (Is\_Hamming\_OR\_Manhatten) θ(1)**

**{**

**new\_Left\_State.H\_Of\_X = Hamming(new\_Left\_State.Matrix); θ(N^2)**

**}**

**else**

**{**

**new\_Left\_State.H\_Of\_X = Manhattan(new\_Left\_State.Matrix, this.size); θ(N^2)**

**}**

**if (new\_Left\_State.H\_Of\_X == 0) θ(1)**

**{**

**ShowSteps(new\_Left\_State);**

**return; θ(1)**

**}**

**if (!Visited.Contains(new\_Left\_State.path)) θ(1)**

**{**

**pqueue.Enqueue(new KeyValuePair<State, int>(new\_Left\_State, new\_Left\_State.Total\_Cost())); O(LogN)**

**}**

**}**

**}**

**if (Temp.Key.New\_x > 0) //Temp.Key.Move.up θ(1)**

**{**

**State New\_Up\_State = new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x - 1, Temp.Key.New\_y, Temp.Key.size); θ(N^2)**

**Compare\_With\_Parient = check\_Parient\_Step(New\_Up\_State); θ(1)**

**if (!Compare\_With\_Parient) θ(1)**

**{**

**if (Is\_Hamming\_OR\_Manhatten) θ(1)**

**{**

**New\_Up\_State.H\_Of\_X = Hamming(New\_Up\_State.Matrix); θ(N^2)**

**}**

**else**

**{**

**New\_Up\_State.H\_Of\_X = Manhattan(New\_Up\_State.Matrix, this.size); θ(N^2)**

**}**

**if (New\_Up\_State.H\_Of\_X == 0) θ(1)**

**{**

**//Console.WriteLine("Number of Movements = " + New\_Up\_State.G\_Of\_X); θ(1)**

**ShowSteps(New\_Up\_State);**

**return; θ(1)**

**}**

**if (!Visited.Contains(New\_Up\_State.path)) θ(1)**

**{**

**pqueue.Enqueue(new KeyValuePair<State, int>(New\_Up\_State, New\_Up\_State.Total\_Cost())); O(LogN)**

**}**

**}**

**}**

**if (Temp.Key.New\_y < size - 1) //Temp.Key.Move.right θ(1)**

**{**

**State new\_Right\_State = new State(Temp.Key, Temp.Key.Matrix, Temp.Key.G\_Of\_X, Temp.Key.New\_x, Temp.Key.New\_y, Temp.Key.New\_x, Temp.Key.New\_y + 1, this.size); θ(N^2)**

**Compare\_With\_Parient = check\_Parient\_Step(new\_Right\_State); θ(1)**

**if (!Compare\_With\_Parient) θ(1)**

**{**

**if (Is\_Hamming\_OR\_Manhatten) θ(1)**

**{**

**new\_Right\_State.H\_Of\_X = Hamming(new\_Right\_State.Matrix); θ(N^2)**

**}**

**else**

**{**

**new\_Right\_State.H\_Of\_X = Manhattan(new\_Right\_State.Matrix, this.size); θ(N^2)**

**}**

**if (new\_Right\_State.H\_Of\_X == 0) θ(1)**

**{**

**ShowSteps(new\_Right\_State);**

**return;**

**}**

**if (!Visited.Contains(new\_Right\_State.path)) θ(1)**

**{**

**pqueue.Enqueue(new KeyValuePair<State, int>(new\_Right\_State, new\_Right\_State.Total\_Cost())); O(Log N)**

**}**

**}**

**}**

**}**

**}}}**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | puzzle | HAMMING/ Milli-sec | HAMMING / sec | MANHATTEN / sec | MANHATTEN / Milli-sec | #Movemnt |
| 1- | **8 Puzzle (1)** | **46** | **0.046** | **0.047** | **47** | **8** |
| 2- | **8 Puzzle (2)** | **109** | **0.109** | **0.109** | **109** | **20** |
| 3- | **8 Puzzle (3)** | **63** | **0.063** | **0.063** | **63** | **14** |
| 4- | **15 Puzzle - 1** | **47** | **0.047** | **0.047** | **47** | **5** |
| 5- | **24 Puzzle 1** | **78** | **0.078** | **0.094** | **94** | **11** |
| 6- | **24 Puzzle 2** | **172** | **0.172** | **0.172** | **172** | **24** |
| 7- | **50 Puzzle** | **4955** | **4.955** | **4.903** | **4903** | **18** |
| 8- | **99 Puzzle - 1** | **266** | **0.266** | **0.25** | **25** | **18** |
| 9- | **99 Puzzle - 2** | **531** | **0.531** | **0.516** | **516** | **38** |
| 10- | **9999 Puzzle** | **461** | **4.61** | **4.609** | **4609** | **4** |
| 11- | **15 Puzzle 1** | **none** | **none** | **4.781** | **4781** | **46** |
| 12- | **15 Puzzle 3** | **none** | **none** | **1.953** | **1953** | **38** |
| 13- | **15 Puzzle 4** | **none** | **none** | **1.203** | **1203** | **44** |
| 14- | **15 Puzzle 5** | **none** | **none** | **20.56** | **20560** | **45** |
| 15- | **TEST** | **none** | **none** | **24.547** | **24547** | **56** |
| 16- | **8 Puzzle - Case 1** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 17- | **8 Puzzle(2) - Case 1** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 18- | **8 Puzzle(3) - Case 1** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 19- | **15 Puzzle - Case 2** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 20- | **15 Puzzle - Case 3** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 21- | **15 Puzzle 1 - Unsolvable** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 22- | **99 Puzzle - Unsolvable Case 1** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 23- | **99 Puzzle - Unsolvable Case 2** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |
| 24- | **9999 Puzzle** | **#NS** | **#NS** | **#NS** | **#NS** | **#NS** |