**Class Program:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Diagnostics;

namespace N\_Puzz

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("N Puzzle Problem:\n[1]Sample test cases\n[2] complete testing\n[3]V\_large");

Console.Write("\nEnter your choice [1-2-3]:");

char choice = (char)Console.ReadLine()[0];

switch (choice)

{

case '1':

bool return\_value0=false;

Console.WriteLine("\n[1] Solvable\n[2] Unsolvable");

Console.Write("\nEnter your choice [1-2]: ");

char choice1 = (char)Console.ReadLine()[0];

if(choice1 == '1')

{

Console.WriteLine("\n[1] manhatten\n[2] hamming");

Console.Write("\nEnter your choice [1-2]: ");

char m\_or\_h = (char)Console.ReadLine()[0];

if (m\_or\_h == '1')

return\_value0 = check("Sample Test/Solvable Puzzles/all\_test.txt", 1);

else if (m\_or\_h == '2')

return\_value0 = check("Sample Test/Solvable Puzzles/all\_test.txt", 2);

else

Console.WriteLine("Invalid Choice!");

}

else if(choice1 == '2')

return\_value0 = check("Sample Test/Unsolvable Puzzles/all\_test.txt",1);

else

Console.WriteLine("Invalid Choice!");

break;

case '2':

bool return\_value1 = false;

Console.WriteLine("N Puzzle Problem:\n[1] Solvable\n[2] Unsolvable");

Console.Write("\nEnter your choice [1-2]: ");

char choice3 = (char)Console.ReadLine()[0];

if (choice3 == '1')

{

Console.WriteLine("N Puzzle Problem:\n[1]Manhattan&Hamming\n[2]Manhattan Only");

Console.Write("\nEnter your choice [1-2]: ");

char choice4 = (char)Console.ReadLine()[0];

if (choice4 == '1')

{

Console.WriteLine("\n[1] manhatten\n[2] hamming");

Console.Write("\nEnter your choice [1-2]: ");

char m\_or\_h = (char)Console.ReadLine()[0];

if (m\_or\_h == '1')

return\_value1 = check("Complete TestCases/Solvable puzzles/Manhattan & Hamming/AllTests.txt",1);

else if (m\_or\_h == '2')

return\_value1 = check("Complete TestCases/Solvable puzzles/Manhattan & Hamming/AllTests.txt", 2);

else

Console.WriteLine("Invalid Choice!");

}

else if (choice4 == '2')

return\_value1 = check("Complete TestCases/Solvable puzzles/Manhattan Only/AllTests.txt", 1);

else

Console.WriteLine("Invalid Choice!");

}

else if (choice3 == '2') { return\_value1 = check("Complete TestCases/Unsolvable puzzles/AllTests.txt",1); }

if (return\_value1)

Console.WriteLine("\nCongratulations\n ");

break;

case '3':

bool return\_value2 = check("TEST.txt",1);

if (return\_value2)

{

Console.WriteLine("\nCongratulations\n ");

}

break;

}

}

public static bool check(string fileName , int m\_or\_h )

{

FileStream test\_file = new FileStream(fileName, FileMode.Open, FileAccess.Read);

StreamReader \_StreamReader = new StreamReader(test\_file);

int number\_cases = int.Parse(\_StreamReader.ReadLine());

int wrongAnswer = 0;

// O(T \* (n^2)) where T = number of test cases, n = size of matrix.

for (int counter = 0; counter < number\_cases; counter++)

{

int matrix\_size = int.Parse(\_StreamReader.ReadLine());

int[] matrix\_1d = new int[matrix\_size \* matrix\_size];

int k = 0;

for (int counter1 = 0; counter1 < matrix\_size; counter1++)

{

string new\_line = \_StreamReader.ReadLine();

string[] num\_splited = new\_line.Split(' ');

for (int counter2 = 0; counter2 < matrix\_size; counter2++)

{

int val = Int32.Parse(num\_splited[counter2]);

matrix\_1d[k++] = val;

}

}

int expectedResult = int.Parse(\_StreamReader.ReadLine());

long timeBefore = System.Environment.TickCount;

//Complexity time:

int receivedResult = ConsoleApp3.Solve.SolveNPuzzleAStar(matrix\_1d, matrix\_size , m\_or\_h);

long timeAfter = System.Environment.TickCount;

long time = timeAfter - timeBefore;

Console.WriteLine("3-the time of program in ms " + time);

Console.WriteLine("4-the time of program in Seconds " + time/1000);

Console.WriteLine("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

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

if (receivedResult == expectedResult)

\_StreamReader.ReadLine();

else

{

wrongAnswer++;

Console.WriteLine("wrong answer at case " + (counter + 1));

}

}

\_StreamReader.Close();

test\_file.Close();

if (wrongAnswer != 0)

{

Console.WriteLine(" wrong answer out of " + number\_cases);

return false;

}

return true;

}

}

}

// The whole class complexity = O(1)

**Class Solve:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp3

{

public class Solve

{

public static int index\_column\_of\_0;

public static int index\_row\_of\_0;

static int get\_inv\_count(int[] arr)

{

int inversion\_count = 0;

// O(S^2) where S = the puzzle size.

for (int i = 0; i < arr.Length - 1; i++)

{

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

for (int j = i + 1; j < arr.Length; j++)

{

if (arr[j] == 0)

continue;

else if (arr[i] > arr[j])

inversion\_count++;

}

}

return inversion\_count;

}

// O(S) where S = The size of the matrix

static int get\_0\_loc(int[] arr,int n)

{

int index\_0 = -1;

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

{

if (arr[i] == 0)

{

index\_0 = i / n;

break;

}

}

return index\_0;

}

static bool isSolvable(int n, int[] array)

{

int returned\_counts = get\_inv\_count(array);

int index\_0 = get\_0\_loc( array, n);

if (n % 2 == 1 && returned\_counts % 2 == 0)

return true;

else if (n % 2 == 0 && returned\_counts % 2 != 0 && index\_0 % 2 == 0)

return true;

else if (n % 2 == 0 && returned\_counts % 2 == 0 && index\_0 % 2 != 0)

return true;

return false;

}

public static int SolveNPuzzleAStar(int[] arr, int n ,int m\_or\_h)

{

int[,] arr1 = new int[n, n];

//Get The space index and to transform from 2d to 1d

for (int i = 0; i < n; i++) // O(n^2) where n = The size of the matrix.

{

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

{

arr1[i, j] = arr[i \* n + j];

if (arr1[i, j] == 0)

{

index\_column\_of\_0 = i;

index\_row\_of\_0 = j;

}

}

}

if (isSolvable(n, arr))

{

int level = 0;

// O(n^2)

int manhatten= ConsoleApp3.matrices\_operations.first\_manhatten(arr1, n);

// O(n^2)

int hamming= ConsoleApp3.matrices\_operations.first\_hamming(arr1,n);

node new\_node = new node(level, arr1, n, hamming, manhatten, m\_or\_h);

new\_node.zero\_index\_root\_i = ConsoleApp3.Solve.index\_column\_of\_0;

new\_node.zero\_index\_root\_j = ConsoleApp3.Solve.index\_row\_of\_0;

Console.WriteLine("1-Solvable");

// O(E)

A\_star\_algorithem Run\_Algo = new A\_star\_algorithem();

node x = Run\_Algo.A\_star(n, new\_node);

if(n == 3)

PrintPath(x, 3, new\_node);

Console.WriteLine("2-number of movments :" + " " + x.level);

return x.level;

}

Console.WriteLine("Not Solvable");

return -1;

}

public static void PrintPath(node top, int n, node root)

{

// O(M) where m = Total number of nodes in the Shortest Path

Stack<node> PrintNodes = new Stack<node>();

while (top.parent != null)

{

PrintNodes.Push(top);

top = top.parent;

}

PrintNodes.Push(root);

while (PrintNodes.Count != 0)

{

node NodePrinted = PrintNodes.Pop();

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

{

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

{

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

}

Console.WriteLine();

}

Console.WriteLine();

}

}

}

}

**Class node:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp3

{

public class node

{

public int manhattan\_cost;

public int hamming\_cost;

public int level;

public int Total\_F;

public int F\_cost;

public int F\_hamming;

public int F\_manhattan;

public int[,] matrix;

public int zero\_index\_root\_i, zero\_index\_root\_j;

public node parent;

public int m\_or\_h1;

public node(int g, int[,] matrix, int n ,int ham , int man ,int m\_or\_h)

{

manhattan\_cost = man ;

hamming\_cost = ham;

this.level = g;

m\_or\_h1 = m\_or\_h;

F\_hamming = level + hamming\_cost;

F\_manhattan = level + manhattan\_cost;

if (m\_or\_h == 1)

{

Total\_F = F\_manhattan;

F\_cost = manhattan\_cost;

}

else if(m\_or\_h == 2)

{

Total\_F = F\_hamming;

F\_cost = hamming\_cost;

}

this.matrix = matrix;

parent = null;

}

public void get\_child(int n, node node)

{

int NEXT\_LEVEL= node.level+1;

if (zero\_index\_root\_i + 1 < n)

{

if (parent == null || parent.zero\_index\_root\_i != node.zero\_index\_root\_i+1)

{

Swap(node.matrix, n, node.zero\_index\_root\_i, node.zero\_index\_root\_j,

node.zero\_index\_root\_i + 1, node.zero\_index\_root\_j, NEXT\_LEVEL,node);

}

}

if (node.zero\_index\_root\_j + 1 < n)

{

if (parent == null || parent.zero\_index\_root\_j != node.zero\_index\_root\_j+1 )

{

Swap(node.matrix, n, node.zero\_index\_root\_i, node.zero\_index\_root\_j,

node.zero\_index\_root\_i, node.zero\_index\_root\_j +1, NEXT\_LEVEL, node);

}

}

if (node.zero\_index\_root\_i - 1 >= 0)

{

if (parent == null || parent.zero\_index\_root\_i !=node.zero\_index\_root\_i-1)

{

Swap(node.matrix, n, node.zero\_index\_root\_i, node.zero\_index\_root\_j,node.zero\_index\_root\_i -1, node.zero\_index\_root\_j, NEXT\_LEVEL, node);

}

}

if (zero\_index\_root\_j - 1 >= 0)

{

if (parent == null || parent.zero\_index\_root\_j != node.zero\_index\_root\_j - 1)

{

Swap(node.matrix, n, node.zero\_index\_root\_i, node.zero\_index\_root\_j,

node.zero\_index\_root\_i, node.zero\_index\_root\_j - 1, NEXT\_LEVEL, node);

}

}

}

public void Swap(int[,] matrix, int n, int i, int j, int x, int y, int level, node node )

{

int[,] swapmatrix = new int[n, n];

Array.Copy(matrix, swapmatrix, n \* n);

//Swapping:

int tempswap = swapmatrix[i, j];

swapmatrix[i, j] = swapmatrix[x, y];

swapmatrix[x, y] = tempswap;

int new\_manhatten=ConsoleApp3.matrices\_operations.next\_manhatten(i, j, node.manhattan\_cost, swapmatrix[i, j], n, x, y);

int new\_hamming=ConsoleApp3.matrices\_operations.next\_hamming(i, j, node.hamming\_cost, swapmatrix[i, j], n ,x,y);

node curr = new node(level, swapmatrix, n, new\_hamming, new\_manhatten, m\_or\_h1);

curr.zero\_index\_root\_i = x;

curr.zero\_index\_root\_j = y;

curr.parent = this;

//Add node to priority Queue:

ConsoleApp3.A\_star\_algorithem.nodes\_list.enqueue(curr); // O(log V) where v = number of nodes in the priorityqueue

}

}

}

**Class matrices\_operations:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp3

{

public class matrices\_operations

{

public static int first\_manhatten (int [,] matrix,int n) // O(n^2)

{

int cost = 0;

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

{

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

{

int index\_column = 0;

if (matrix[i, j] % n != 0) { index\_column = matrix[i, j] % n - 1; }

else { index\_column = n - 1; }

double value = Convert.ToDouble(n);

double index\_row = Math.Ceiling(matrix[i, j] / value) - 1;

if (matrix[i, j] == 0)

continue;

cost +=Math.Abs(i- Convert.ToInt32(index\_row)) + Math.Abs(j - index\_column);

}

}

return cost;

}

public static int first\_hamming(int[,] matrix, int n) // O(n^2)

{

int counter = 0;

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

{

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

{

if (matrix[i, j] == 0)

continue;

int index\_column = 0;

if (matrix[i, j] % n != 0) { index\_column = matrix[i, j] % n - 1; }

else { index\_column = n - 1; }

double value = Convert.ToDouble(n);

double index\_row = Math.Ceiling(matrix[i, j] / value) - 1;

if (i != Convert.ToInt32(index\_row) || j != index\_column) {

counter++;

}

}

}

return counter;

}

// O(1)

public static int next\_manhatten(int new\_i ,int new\_j, int p\_cost ,int element, int n ,int i\_last,int j\_last)

{

int index\_column = 0;

if (element % n != 0) { index\_column = element % n - 1; }

else { index\_column = n - 1; }

double value = Convert.ToDouble(n);

double index\_row = Math.Ceiling(element / value) - 1;

p\_cost -= Math.Abs(i\_last - Convert.ToInt32(index\_row)) + Math.Abs(j\_last - index\_column);

p\_cost += Math.Abs(new\_i - Convert.ToInt32(index\_row)) + Math.Abs(new\_j - index\_column);

return p\_cost;

}

// O(1)

public static int next\_hamming(int new\_i, int new\_j, int p\_cost, int element, int n ,int i\_last, int j\_last)

{

int index\_column = 0;

if (element % n != 0)

index\_column = element % n - 1;

else

index\_column = n - 1;

double value = Convert.ToDouble(n);

double index\_row = Math.Ceiling(element / value) - 1;

if(new\_i== index\_row && new\_j== index\_column)

p\_cost--;

else if (index\_row== i\_last&& index\_column== j\_last)

p\_cost++;

return p\_cost;

}

}

}

**Class A\_star\_algorithem:**

using System;

namespace ConsoleApp3

{

public class A\_star\_algorithem

{

public static priorityqueue nodes\_list;

public node A\_star(int N,node start)

{

nodes\_list = new priorityqueue();

nodes\_list.enqueue(start);

node top = null;

// O(E log V) where E = number of moves, V number of nodes

while (!nodes\_list.empty()) // O(E)

{

top = nodes\_list.dequeue(); //Log(V)

if (top.F\_cost == 0)

return top;

top.get\_child(N, top);

}

return null;

}

}

}

**Class priorityqueue:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp3

{

public class priorityqueue

{

node[] Arr;

int length = 0;

public void enqueue(node val) // O(log V) where V = number of elements in the priorityqueue

{

length = length + 1;

Arr[length] = null; //assuming all the numbers greater than 0 are to be inserted in queue.

increase\_value(length, val); // O(log V) where V = number of elements in the priorityqueue

}

public node dequeue() // O(log V) where V = number of elements in the priorityqueue

{

if (length == 0)

throw new InvalidOperationException("Can’t remove element as queue is empty");

node min = Arr[1];

Arr[1] = Arr[length];//1

length = length - 1;//

min\_heapify(1, length); // O(log V) where V = number of elements in the priorityqueue

return min;

}

public node remove\_last() // O(1)

{

return Arr[1];

}

public node top() //O(1)

{

if (length == 0)

throw new InvalidOperationException("Can’t remove element as queue is empty");

return Arr[1];

}

public bool empty()

{

return (length == 0);

}

public priorityqueue()

{

Arr = new node[50000000];

}

public int count() { return length; }

public void increase\_value(int i, node val)

{

Arr[i] = val;

// O(log V) where V = number of elements in the priorityqueue

while (i > 1 && Arr[i / 2].Total\_F >= Arr[i].Total\_F)

{

swap(ref Arr[i / 2], ref Arr[i]);

i = i / 2;//1

}

}

void min\_heapify(int i, int N) // O(log V) where V = number of elements in the priorityqueue

{

// to get index of left child of node at index i

int left = 2 \* i;

// to get index of right child of node at index i

int right = 2 \* i + 1;

int smallest;

if (left <= N && Arr[left].Total\_F < Arr[i].Total\_F)

smallest = left;

else

smallest = i;

if (right <= N && Arr[right].Total\_F < Arr[smallest].Total\_F)//1

smallest = right;

if (smallest != i)//1

{

swap(ref Arr[i], ref Arr[smallest]);//1

min\_heapify(smallest, N);

}

}

void build\_minheap(int N)

{

for (int i = N / 2; i >= 1; i--)

min\_heapify(i, N);

}

void swap(ref node x, ref node y)//1

{

node t = x;

x = y;

y = t;

}

}

}

**Main Functions Analysis:**

1. getchild: O(logV)

V: number of nodes

1. Manhattan: **θ**‎(n2)
2. Hamming: **θ**‎(n2)
3. Next Manhattan: O(1)
4. Next Hamming : O(1)
5. IsSolvable: **θ** (S2)
6. A\* algorithm: O(E log V)

where E = number of moves, V number of nodes

**Comparison Between Hamming&manhattan:**

1. **Execution Time:**

|  |  |
| --- | --- |
| **Manhattan** | **Hamming** |
| 16 MS | 156 MS |
| 125 MS | 0 MS |
| 0 MS | 0 MS |
| 78 MS | 78 MS |

1. **Minimum Number of moves are EQUAL in both heuristic functions.**

As Manhattan&Hamming are heuristic functions doesn’t affect minimum number of moves.