N-Puzzle Project

# Source Code and analysis:

class Node

{

// stores the parent node of the current node

// helps in tracing path when the answer is found

public Node node;

// stores matrix

public int[,] array;

// stores blank tile coordinates

public int x, y;

// stores the number of misplaced tiles

public int cost;

// stores the number of moves so far

public int level;

};

class Program

{

static bool isSolvable(int[,] arr, int[] puzzle) //total = ϴ(N^2)

{

int inv = getInvCount(puzzle); // ϴ(N^2)

if ((puzzle.Length % 2 != 0) && (inv % 2 == 0))

{

return true; ϴ(1)

}

else

{

int pos = findXPosition(arr); // ϴ(N^2)

if (pos % 2 == 0 && inv % 2 != 0)

return true; ϴ(1)

else if (pos % 2 != 0 && inv % 2 == 0)

return true; ϴ(1)

else

return false; ϴ(1)

}

}

static int getInvCount(int[] arr) //total = ϴ(N^2)

{

int count = 0;

for (int i = 0; i < (arr.Length \* arr.Length) - 1; i++) //N

{

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

{

if (arr[j] != 0 && arr[i] != 0 && arr[i] > arr[j])

{

count++; ϴ(1)

}

}

}

return count; ϴ(1)

}

static int findXPosition(int[,] puzzle) //total = ϴ(N^2)

{

for (int i = puzzle.GetLength(0) - 1; i >= 0; i--) //N

for (int j = puzzle.GetLength(1) - 1; j >= 0; j--) //N

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

return puzzle.GetLength(1) - i;

return 0;

}

static Node newNode(int[,] mat, int x, int y, int newx, int newy, int levelNode, Node parent, int num) ϴ(1)

{

Node node = new Node(); ϴ(1)

node.node = parent; ϴ(1)

node.array = new int[num, num]; ϴ(1)

node.array = mat; ϴ(1)

int temp = node.array[x, y]; ϴ(1)

node.array[x, y] = node.array[newx, newy]; ϴ(1)

node.array[newx, newy] = temp; ϴ(1)

node.x = newx; ϴ(1)

node.y = newy; ϴ(1)

node.level = levelNode; ϴ(1)

//cost in Astar

return node; ϴ(1)

}

static (int, int) findBlankTile(int[,] mat, int num) //total = O(N^2)

{

int x1 = -1, y1 = -1;

for (int i = 0; i < num; i++) // N

for (int j = 0; j < num; j++) // N

{

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

{

x1 = i; ϴ(1)

y1 = j; ϴ(1)

break; ϴ(1)

}

}

return (x1, y1);

}

static void printMatraix(int[,] mat, int num) //total = ϴ(N^2)

{

for (int i = 0; i < num; i++) // N

{

for (int j = 0; j < num; j++) // N

{

Console.Write(mat[i, j] + " "); ϴ(1)

}

Console.WriteLine(); ϴ(1)

}

}

T(N) = T(N-1) + ϴ(N^2)

T(Base) = ϴ(1)

1. Draw the tree for at least three levels : T(N) 🡪 T(N-1) 🡪 T(N-2) 🡪 T(N-3)
2. Calculate number of levels: It's an **balanced** tree : N,N-1,N-2,N-3,…,N-L where L: # levels L= N
3. Calculate complexity of each level: = from the above tree
4. last level contains only 1 leaf with O(1) FROM BASE CASE T(Base) = 1
5. Total = sum of all levels

Final Order : ϴ(N^3)

static int numberSteps;

static unsafe void showPath(Node path, int num)

{

if (path.node == null)

return;

numberSteps++;

showPath(path.node, num;

Console.WriteLine("###############################################");

printMatraix(path.array, num);

}

static int Hamming\_method(Node n, int[,] goal, int num) // total = ϴ(N^2)

{

int count = 0;

for (int i = 0; i < num; i++) // N

for (int j = 0; j < num; j++) // N

if (n.array[i, j] != goal[i, j] && n.array[i, j] != 0)

{

count++; ϴ(1)

}

return count; ϴ(1)

}

static int manhattan(Node initial, int[,] goal, int N) // total = ϴ(N^2)

{

int manhattanDistance = 0; ϴ(1)

int[] arr = new int[N \* N]; ϴ(1)

int count = 0;

for (int i = 0; i < N; i++) // N

{

for (int j = 0; j < N; j++) // N

{

arr[count] = initial.array[i, j]; ϴ(1)

count++; ϴ(1)

}

}

for (int i = 0; i < N \* N; i++) // iterations = Size of puzzle(S)

{

if (arr[i] == 0)

continue; ϴ(1)

int vertical = Math.Abs((i / N) - ((arr[i] - 1) / N)); ϴ(1)

int horizontal = Math.Abs((i % N) - ((arr[i] - 1) % N)); ϴ(1)

manhattanDistance += vertical + horizontal; ϴ(1)

}

return manhattanDistance; ϴ(1)

}

**Final Order = O(E log(V))**

static void Astar(int[,] initial, int[,] goal,int n,string s)

{

int[] row = { 1, 0, -1, 0 }; ϴ(1)

int[] col = { 0, -1, 0, 1 }; ϴ(1)

Node root; ϴ(1)

PriorityQueue<Node,int> openlist=new PriorityQueue<Node, int>();

PriorityQueue<Node, int> path = new PriorityQueue<Node, int>();

Node parent=new Node();ϴ(1)

parent.node = null; ϴ(1)

var blank=findBlankTile(initial,n); O(N^2)

root=newNode(initial, blank.Item1, blank.Item2, blank.Item1, blank.Item2,0,parent,n); ϴ(1)

root.node.x = -1; ϴ(1)

root.node.y = -1; ϴ(1)

if(s=="1") ϴ(1)

root.cost = Hamming\_method(root, goal, n); ϴ(N^2)

else

root.cost = manhattan(root, goal, n); ϴ(N^2)

openlist.Enqueue(root, 0); (build funcation in c# O(Log n))

# iterations = n (number node in open list), Body = MAX(1,N^2, N^3)= N^3

Final Order is N^3

while(openlist.Count > 0)

{

Node node=openlist.Dequeue(); (build funcation in c# O(Log n))

if(node.cost==0) ϴ(1)

{

showPath(node,n); ϴ(N^3)

Console.WriteLine("Number of steps : "+(numberSteps-1)); ϴ(1)

return; ϴ(1)

}

int xnew,ynew; ϴ(1)

# iterations = 4, Body = MAX(1, Log(N), N^2)= N^2

Final Order is N^2

//create 4 child for parent node

for(int i=0;i<4;i++)

{

Node child =new Node();

xnew= node.x + row[i]; ϴ(1)

ynew= node.y + col[i]; ϴ(1)

if(xnew>=0&& xnew<n&&ynew>=0&& ynew<n&&(node.node.x!= xnew||node.node.y!= ynew)) ϴ(1)

{

int[,] newmat=(int[,])node.array.Clone();

child = newNode(newmat, node.x, node.y, xnew, ynew, (node.level + 1),node,n); ϴ(1)

if(s=="1") ϴ(1)

child.cost = Hamming\_method(child, goal, n); ϴ(N^2)

else

child.cost = manhattan(child, goal, n); ϴ(N^2)

int proiraty = child.cost + child.level; ϴ(1)

openlist.Enqueue(child, proiraty); (build funcation in c# O(Log n))

}

}

}

}

static void Main(string[] args)

{

Stopwatch stopwatch = new Stopwatch();

bool check; ϴ(1)

//read from file

String input = File.ReadAllText(@"D:\project\Visual\nPuzzle\Test Case\TEST.txt");

int row=0,col=0; ϴ(1)

string[] Lines = input.Split('\n');

Lines=Lines.Where(l => l!="\r").ToArray();

int number=int.Parse(Lines[0]); ϴ(1)

int[,] test = new int[number, number]; ϴ(1)

int[,] test2=new int[number, number]; ϴ(1)

int[] arr = new int[number\*number]; ϴ(1)

int checker = 0; ϴ(1)

int count = 1; ϴ(1)

Final oreder = N^2

for(int i=1; i<Lines.Length; i++) //N

{

col = 0;

foreach (var num in Lines[i].Trim().Split(' ')) //N

{

if (num.Trim() == "")

continue; ϴ(1)

test[row,col]=int.Parse(num.Trim()); ϴ(1)

arr[checker]= int.Parse(num.Trim()); ϴ(1)

checker++; ϴ(1)

col++; ϴ(1)

}

row++; ϴ(1)

}

Total order = N^2

for(int i=0; i<number; i++) //N

{

for(int j=0; j<number; j++) //N

{

test2[i, j] = count; ϴ(1)

count++; ϴ(1)

}

}

test2[number - 1, number - 1] = 0; ϴ(1)

check=isSolvable(test,arr); ϴ(N^2)

if(check)

{

Console.WriteLine("Solvable"); ϴ(1)

Console.WriteLine("Write number [1] if you want run by Hamming priority function"); ϴ(1)

Console.WriteLine("Write number [2] if you want run by Manhattan priority function"); ϴ(1)

string select=Console.ReadLine();

stopwatch.Start();

Astar(test, test2, number, select); O(E log(V))

stopwatch.Stop();

Console.WriteLine("Elapsed Time is {0} ms", (stopwatch.ElapsedMilliseconds)); ϴ(1)

}

else

{

Console.WriteLine("Unsolvable"); ϴ(1)

}

}

}

}

# Comparison between the two distances (Hamming vs. Manhattan)

|  |  |  |  |
| --- | --- | --- | --- |
| Hamming vs. Manhattan | | | |
| Sample Test Case | Hamming Execution Time (ms) | Manhattan Execution Time (ms) | Minimum Distance |
| 8 Puzzle (1).txt | 15 | 13 | 8 |
| 8 Puzzle (2).txt | 41 | 40 | 20 |
| 8 Puzzle (3).txt | 27 | 25 | 14 |
| 15 Puzzle - 1.txt | 11 | 15 | 5 |
| 24 Puzzle 1.txt | 41 | 37 | 11 |
| 24 Puzzle 2.txt | 89 | 87 | 24 |

|  |  |  |  |
| --- | --- | --- | --- |
| Manhattan & Hamming | | | |
| Complete Test Case | Hamming Execution Time (ms) | Manhattan Execution Time (ms) | Minimum Distance |
| 50 Puzzle.txt | 3072 | 2975 | 18 |
| 99 Puzzle - 1.txt | 167 | 167 | 18 |
| 99 Puzzle - 2.txt | 340 | 338 | 38 |
| 9999 Puzzle.txt | 2980 | 2996 | 4 |
| Manhattan Only | | | |
| 15 Puzzle 1.txt | - | 4122 | 46 |
| 15 Puzzle 3.txt | - | 1506 | 38 |
| 15 Puzzle 4.txt | - | 604 | 44 |
| 15 Puzzle 5.txt | - | 26071 | 45 |
| V. Large test case | | | |
| TEST.txt | - | 20106 | 56 |