N-PUZZLE Project.

Team 142

Hadeel Osama Hajar Ahmed Manar Mahmoud

Introduction:

The N-Puzzle is known in finite versions, in this project have many versions: (3*3), (4*4), (5*5) and so on.

To solve this problem, we use search strategies to choose the smallest path:

- Informed Search:
  - A * search Algorithm by Hamming and Manhattan Priority.
- Uninformed Search:
  - Depth first search.
  - Breadth first search.

Analysis functions:

- `Main ()` -> best case bounded by O(1), worst case bounded by O(E s^2 + V)

To detect whether the board is solvable or not, we use more than function (`checkngSolvability` class):

- `numOfinver(int size, int[,] arr)`➔`to know number of inversions`
  Bounded by O(s^2).
- `FindEmptyIndex(int size)`➔ `to know the empty index or blank`
  Bounded by O(s^2).
- `convertArray( int size, int[,] arr)`➔`to convert array from 2D to 1D`
  Bounded by O(s^2).
- `fillArr(int size,int[,] puzzle)`➔`to fill array with values of puzzle`
  Bounded by O(s^2)
- `solvabilityOdd(int arrSize, int[] array)`➔`to check the number of inversions is odd or even` in odd puzzle
  Bounded by O(s^2).
- `solvabilityEven(int arrSize, int[] array)`➔`to check the number of inversions is odd or even` in even puzzle

Bounded by O(s^2)

- `solvability(int arrSize, int[] puzzle)` ➔`to check the number of array size is odd or even`

  Bounded by O(1)

If puzzle is solvable, to solve puzzle and find the minimum number of movements using this functions (`SolvingPUZZLE` class):

- **`AStar(State startState)`** -> to solve puzzle by two priorities with minimum number of iterations.

  Bounded by O(E log V).

- **`makePuzzleStates(State newState, int size)`** -> to generate children of parent state.

  Bounded by O(1)

- **`swap(int raw1, int col1, int raw2, int col2, int[,] arr)`** -> to swap to elements in array. O(1)

- **`priorityType(int[,] arr, int size, int NR, int NC, int R, int C, int dis)`** -> to determine type of priority.

  Bounded by O(1)

- **`calculateDiffHamming (int NR, int NC, int R, int C, int [,]arr, int size)`** -> to check if the tile in its place or not. Bounded by O(1).
- **`calculateDiffmanhatten (int NR, int NC, int R, int C, int [,]arr, int size)`** -> to check if the tile is closed to its place. Bounded by O(1).

- **calculateDistanceManhatten (int[,] arr, int size)** -> to calculate Manhattan priority. Bounded by O(s^2)

- **calculateDistanceHamming (int[,] arr, int size)** -> to calculate Hamming priority. Bounded by O(s^2)

- **BFSTree(State startState)** -> this search strategy to solve the puzzle by breadth first search, search on the goal state level by level. Bounded by O(E s^2+V).

- **DFSTree(State startState)** -> this search strategy to solve the puzzle by depth first search, search on the goal state by all edges (depth). Bounded by O(E s^2+V).

- **CompareArrays(int[,] arr, int[,] goal, int size)** -> to compare 2 arrays, bounded by O(s^2)

- **goalState(int size)** -> function to prepare the goal state. Bounded by O(s^2)
- **getZeroIndex(int[,] arr, int size, ref int row, ref int col)** `-> bounded by O(numOfMoves*s^2).`
- **printCases(State st)** -> bounded by O(s^2).

And using priority queue, we determine any path should take it depends on priority function, so the functions in priority queue(`PriorityQueue` class):

- **`Enqueue(State newState)`** -> to add the new state. Bounded by O(Log N), where N is the size of list.

- **`siftUp()`** -> to put the new state in it's right place. Bounded by O(Log N), where N is the size of list.

- **`Dequeue()`** -> Bounded by O(Log N), where N is the size of list.

- **`siftDown()`** -> Bounded by O(Log N), where N is the size of list.

- **`SwapInt(int X, int Y)`** -> Bounded by O(1).

Comparison between Hamming and Manhattan priorities:

	Hamming priority function	**Manhattan priority function**
Definition	The number of blocks in the wrong position+ the number of moves made so far to get to the state.	Manhattan priority function: The sum of the distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus, the number of moves made so far to get to the state.
Order	**O(S^2)**	**O(S^2)**
Time-compleate-50	0.046	0.016
Complete -99-1	0	0 s
complete 99-2	0	0 s
Complete 9999	0	0 s
M15-1	-	9 s
M15-3	-	2 s
M15-4	-	0.8 s
M15-5	-	30 s
sample-8-1	0.031	-
sample-8-2	0.063	-
sample-8-3	0.031	-
sample-15-1	0.016	-
sample-24-1	0	-
sample-24-2	0	-
BFS-8-1	0	
BFS-8-2	0.344	
BFS-8-3	0.032	
BFS-15-1	0	
BFS-24-1	0.062	
DFS-8-1	0	
DFS-8-2	0.875	
DFS-8-3	0.032	

DFS-15-1	0
DFS-24-1	0.125
Large test	93 s

Comparison between A* Search & BFS & DFS:

	A * Search	**BFS**	**DFS**
Definition	It finds the shortest path through the search space using the heuristic function and expand less search tree. It chooses the path to take it with the **lowest priority**. If the priority is zero, so this state is the goal state.	The search is a ***level-by-level order*** traversal. Discovers all vertices starting the initial state and generate it's children and iterate till reach to goal state	Search as **deep** as possible first. Discover the start state and generate its children and pick the first child to discover its children and so on, when all edges of the start state have been explored, backtrack to explore the other edges. The search will be stop if the current state become the goal state.
Order	**O (E Log V)**	**O (E+V)**	**O (E+V)**
Time	Sure this strategy is fastest than (BFS & DFS)	Depends on the cases, but always it's slower than A* search.	Depends on the cases, but always it's slower than A* search.

Source Code:

SolvingPUZZLE class

```
using System;
using System.Collections;
using System.Collections.Generic;


namespace N_PUZZLE
{
    class SolvingPUZZLE
    {
        public int moves;
        public string type;
        int Col, Raw;


        public SolvingPUZZLE(string type)
        {
            this.type = type;
        }

        public State AStar(State startState)
        {
            moves = 0;
            List<State> children;
            State currentState;


            PriorityQueue prQueue = new PriorityQueue();
            prQueue.Enqueue(startState);
            while (true)
            {
                currentState = prQueue.Dequeue();


                Raw = currentState.raw;
                Col = currentState.col;
                children = makePuzzleStates(currentState,
currentState.size);
                foreach (State child in children)
                {
                    prQueue.Enqueue(child);
                }

                if (currentState.levels-currentState.distance == 0)
                {
                     return currentState;
                }
            }
        }

        public State BFSTree(State startState)
        {
            moves = 0;
            List<State> children;
            State currentState;
            Queue<State> queue = new Queue<State>();
            queue.Enqueue(startState);
            int[,] goal = goalState(startState.size);
            while (true)
            {
                currentState = queue.Dequeue();
                Raw = currentState.raw;
                Col = currentState.col;
                children = makePuzzleStates(currentState,
currentState.size);
                foreach (State child in children)
                {
                    queue.Enqueue(child);
                }
                currentState.visitState();
                if (CompareArrays(currentState.arr, goal,
startState.size))
                {
                    return currentState;
                }
            }
        }
        public State DFSTree(State startState)
        {
            moves = 0;
            List<State> children;
            List<State> children2;
            State currentState;
            Queue<State> queue = new Queue<State>();
            queue.Enqueue(startState);
            int[,] goal = goalState(startState.size);
            children = makePuzzleStates(startState, startState.size);
            foreach (State child in children)
            {
                queue.Enqueue(child);

                while (queue.Count != 0)
                {
                    currentState = queue.Dequeue();
                    Raw = currentState.raw;
                    Col = currentState.col;
                    children2 = makePuzzleStates(currentState,
currentState.size);
                    foreach (State child2 in children2)
                    {
                        queue.Enqueue(child2);
                    }

                    if (CompareArrays(currentState.arr, goal,
startState.size))
                    {
                        return currentState;
                    }
                }
            }
            return null;
        }
        public List<State> makePuzzleStates(State newState, int size)
        {
            List<State> children = new List<State>();
            int dis;
            int[,] up = (int[,])newState.arr.Clone(), left =
(int[,])newState.arr.Clone()
                , down = (int[,])newState.arr.Clone(), right =
(int[,])newState.arr.Clone();
            int raw = Raw, col = Col;
            moves = newState.levels;
            if (raw - 1 >= 0&&newState.previosMovement!="down")//up
            {
                up = swap(raw, col, raw - 1, col, up);
                dis = priorityType(up, size, raw, col, raw-1, col,
newState.distance - moves+1) + moves;
                State Up = new State(up, size, "up", dis, moves + 1,
newState,  raw-1, col);
                children.Add(Up);
            }
            if (col + 1 < size && newState.previosMovement !=
"left")//right
            {
                right = swap(raw, col, raw, col + 1, right);
                dis = priorityType(right, size, raw, col, raw, col+1,
newState.distance-moves + 1) + moves;
                State Right = new State(right, size, "right", dis,
moves + 1, newState, raw, col+1);
                children.Add(Right);
            }
            if (col - 1 >= 0 && newState.previosMovement !=
"right")//left
            {
                left = swap(raw, col, raw, col - 1, left);
                dis = priorityType(left, size, raw, col, raw, col-1,
newState.distance - moves + 1) + moves;
                State Left = new State(left, size, "left", dis, moves
+ 1, newState, raw, col-1);
                children.Add(Left);

            }
            if (raw + 1 < size && newState.previosMovement != "up")
//down
            {
                down = swap(raw, col, raw + 1, col, down);
                dis = priorityType(down, size, raw, col, raw+1, col,
newState.distance - moves + 1) + moves;
                State Down = new State(down, size, "down", dis, moves
+ 1, newState, raw+1, col);
                children.Add(Down);
            }

            return children;
        }
        public int[,] swap(int raw1, int col1, int raw2, int col2,
int[,] arr)
        {
            int temp = arr[raw1, col1];//0
            arr[raw1, col1] = arr[raw2, col2];
            arr[raw2, col2] = temp;
            return arr;
        }
        public int priorityType(int[,] arr, int size, int NR, int NC,
int R, int C, int dis)
        {
            if (this.type == "H")
```

```
                return dis - calculateDiffHamming(NR, NC, R, C, arr,
size);
            //return calculateDistanceH(arr, size);
            else if (this.type == "M")
                return dis - calculateDiffmanhatten(NR, NC, R, C, arr,
size);

            //return calculateDistanceManhatten(arr, size);
            else
                return 0;
        }
        //hamming
        public int calculateDistanceHamming(int[,] arr, int size)
        {
            int pos = 0;
            int c = 0;
            for (int i = 0; i < size; i++)
            {
                for (int ii = 0; ii < size; ii++)
                {
                    c = c + 1;
                    if (arr[i, ii] != c && arr[i, ii] != 0)
                        pos++;
                }
            }

            return pos;
        }

        public int calculateDiffHamming(int NR, int NC, int R, int C,
int [,]arr, int size)
        {
            int raw = (arr[NR, NC] - 1) / size;
            int col = (arr[NR, NC] - 1) % size;
            if (raw == NR && col == NC)
                return 1;
            else if (raw == R && col == C)
                return -1;
            else
                return 0;
        }
        //manhatten
        public int calculateDiffmanhatten(int NR, int NC, int R, int
C, int[,] arr, int size)
        {
            int raw = (arr[NR, NC] - 1) / size;
            int col = (arr[NR, NC] - 1) % size;
            int newx1 = Math.Abs(NR - raw);
            int newy1 = Math.Abs(NC - col);
            int newx2 = Math.Abs(R - raw);
            int newy2 = Math.Abs(C - col);
            int manh1 = newx1 + newy1;//after
            int manh2 = newx2 + newy2;//before
            return manh2- manh1;
        }
        public int calculateDistanceManhatten(int[,] arr, int size)
        {

            int menh_sum = 0;


            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    if (arr[i, j] != 0)
                    {

                        int raw = (arr[i, j] - 1) / size;
                        int col = (arr[i, j] - 1) % size;


                        int newx = Math.Abs(i - raw);//0
                        int newy = Math.Abs(j - col);//0

                        menh_sum += newx + newy;

                    }

                }
            }


            return menh_sum;
        }

        public void getZeroIndex(int[,] arr, int size, ref int row, 
    ref int col)
        {
            for (int i = 0; i < size; i++)
            {
                for (int ii = 0; ii < size; ii++)
                {
                    if (arr[i, ii] == 0)
                    {
                        row = i;
                        col = ii;

                    }
                }
            }
        }
        public int[,] goalState(int size)
        {
            int[,] arr = new int[size, size];
            int c = 0;
            for (int i = 0; i < size; i++)
            {
                for (int ii = 0; ii < size; ii++)
                {
                    if (ii == size - 1 && i == size - 1)
                        break;
                    c++;
                    arr[i, ii] = c;

                }
            }
            return arr;
        }

        public bool CompareArrays(int[,] arr, int[,] goal, int size)
        {
            for (int i = 0; i < size; i++)
            {
                for (int ii = 0; ii < size; ii++)
                {
                    if (arr[i, ii] != goal[i, ii])
                        return false;
                }
            }
            return true;
        }

        public void printCases(State st)
        {
            if (st == null)
                return;
            printCases(st.parent);
            for (int i = 0; i < st.size; i++)
            {
                for (int ii = 0; ii < st.size; ii++)
                {
                    Console.Write(st.arr[i, ii]);
                    Console.Write(" ");

                }
                Console.Write("\n");

            }
            Console.Write("\n");

        }

    }
    }
```

State Class

```
using System;
using System.Collections.Generic;
using System.Text;

namespace N_PUZZLE
{
    class State
    {
        public string previosMovement;
        public string visit;
        public int levels;
        public int [,] arr;
        public int size;
        public int distance;
        public int raw;
        public int col;
        public State parent;
        public State(int [,] arr, int size, string prevMove, int 
distance, int levelNum, State parent, int raw, int col)
        {
            this.previosMovement = prevMove;
            this.arr = arr;
            this.size = size;
            this.distance = distance;
            this.raw = raw;
            this.col = col;
            this.levels = levelNum;
            this.visit = "no";
            this.parent = parent;
        }
        public void visitState()
        {
            visit = "yes";
        }
    }
}
```

Program Class

```
using System;
using System.Collections.Generic;
using System.IO;

namespace N_PUZZLE
{
    class Program
    {

        static void Main()
        {

            FileStream file;
            int size;
            StreamReader sr;
            string line;
            string[] Sampel = {"8 Puzzle1", "8 Puzzle2", "8 Puzzle3",
"15 Puzzle1",
            "24 Puzzle1", "24 Puzzle2"};
            string[] CompleteH_M = {"50 Puzzle","99 Puzzle1","99
Puzzle2",
            "9999 Puzzle"};
            string[] CompleteM_Only = { "15 Puzzle 1M","15 Puzzle
3M","15 Puzzle 4M","15 Puzzle 5M"};
            string[] BFStests = {"8 Puzzle1", "8 Puzzle2", "8
Puzzle3", "15 Puzzle1",
            "24 Puzzle1"};
            string[] DFStests = {"8 Puzzle1", "8 Puzzle2", "8
Puzzle3", "15 Puzzle1",
            "24 Puzzle1"};
            string[] unSolvableComplete = { "un1", "un2", "un3", "un4"
};
```

```
            string[] unSolvableSample = { "uns1", "uns2", "uns3", 
"uns4", "uns5" };
            Console.WriteLine("\nChoose: \n1-unSolvableSample\n2-
unSolvableComplete1 \n3-SampleTest  \n4-Manhattan & Hamming" +
                "\n5-ManhattanOnly\n6-BFS\n7-DFS\n8-veryLarge");
            string input = Console.ReadLine();
            int raw = 0, col = 0;
            long timeBefore1, timeBefore;
            long timeAfter;
            long timeAfter1;
            int dis; State state, start;
            if (input == "1")
            {
                int Case = 0;
                foreach (string FileName in unSolvableSample)
                {
                    file = new FileStream(FileName + ".txt", 
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();

                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }

                    timeBefore = System.Environment.TickCount;

                    int[] a = checkngSolvability.convertArray(size, 
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable = 
checkngSolvability.solvability(size, a);
                    Case++;
                    if (!solvable)
                    {
                        Console.WriteLine("Case " + Case + " :");
                        Console.WriteLine("Puzzle is Not Solvable");
                    }

                }

            }
            else if (input == "2")
            {
                int Case = 0;

                foreach (string FileName in unSolvableComplete)
                {
                    file = new FileStream(FileName + ".txt",
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();

                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }


                    int[] a = checkngSolvability.convertArray(size,
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable =
checkngSolvability.solvability(size, a);
                    Case++;
                    if (!solvable)
                    {
                        Console.WriteLine("Case " + Case + " :");
                        Console.WriteLine("Puzzle is Not Solvable");
                    }
                }

            }
            else if (input == "3")
            {
               foreach (string FileName in Sampel)
               {
                    file = new FileStream(FileName+".txt",
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();

                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }

                    timeBefore = System.Environment.TickCount;

                    int[] a = checkngSolvability.convertArray(size,
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable =
checkngSolvability.solvability(size, a);

                    if (!solvable)
                    {
                        Console.WriteLine("Puzzle is Not Solvable");
                        return;
                    }
                    Console.WriteLine(FileName+":");
                    Console.WriteLine("Puzzle is Solvable\n");
                    SolvingPUZZLE hamming = new SolvingPUZZLE("H");
                    hamming.getZeroIndex(arr, size, ref raw, ref col);
                    dis = hamming.calculateDistanceHamming(arr, size);
                    start = new State(arr, size, "", dis, 0, null,
raw, col);
                    state = hamming.AStar(start);
                    if (size == 3)
                        hamming.printCases(state);
                    Console.WriteLine("Hamming ");
                    Console.WriteLine("{0} # of movements.",
state.levels);
                    timeAfter = System.Environment.TickCount;
                   // totalTime += timeAfter - timeBefore;
                    Console.WriteLine("total time: {0} ms{1}\n",
(timeAfter - timeBefore), " -  " + (float)(timeAfter - timeBefore) /
1000 + " s.\n");

                }
               // Console.WriteLine("___\nTotal Time: {0} ms{1}\n",
totalTime, " -  " + (float)totalTime / 1000 + " s.\n");

            }
            else if (input == "4")
            {
                foreach (string FileName in CompleteH_M)
                {
                    file = new FileStream(FileName + ".txt",
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();

                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }
                    int[] a = checkngSolvability.convertArray(size, 
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable = 
checkngSolvability.solvability(size, a);

                    if (!solvable)
                    {
                        Console.WriteLine("Puzzle is Not Solvable");
                        return;
                    }
                    Console.WriteLine(FileName + ":");

                    Console.WriteLine("Puzzle is Solvable\n");
                    timeBefore1 = System.Environment.TickCount;

                    SolvingPUZZLE hamming = new SolvingPUZZLE("H");
                    hamming.getZeroIndex(arr, size, ref raw, ref col);
                    dis = hamming.calculateDistanceHamming(arr, size);
                    start = new State(arr, size, "", dis, 0, null, 
raw, col);
                    state = hamming.AStar(start);
                    timeAfter1 = System.Environment.TickCount;
                    timeBefore = System.Environment.TickCount; ;
                    SolvingPUZZLE manhatten = new SolvingPUZZLE("M");

                    dis = manhatten.calculateDistanceManhatten(arr, 
size);
                    start = new State(arr, size, "", dis, 0, null, 
raw, col);
                    State endM = manhatten.AStar(start);

                    Console.WriteLine("Hamming & Manhatten: ");
                    Console.WriteLine("{0} # of movements.", 
endM.levels);
                    timeAfter = System.Environment.TickCount;
                    //totalTime += timeAfter - timeBefore;

                    Console.WriteLine("total time hamming: {0} ms{1}", 
(timeAfter1 - timeBefore1), " -  " + (float)(timeAfter1 - timeBefore1) 
/ 1000 + " s.");
                    Console.WriteLine("total time Manhattan: {0} 
ms{1}\n", (timeAfter - timeBefore), " -  " + (float)(timeAfter - 
timeBefore) / 1000 + " s.");

                }
                //Console.WriteLine("___\nTotal Time: {0} ms{1}\n",
totalTime, " -  " + (float)totalTime / 1000 + " s.\n");


            }
            else if (input == "5")
            {
                foreach (string FileName in CompleteM_Only)
                {
                    file = new FileStream(FileName + ".txt",
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();

                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }

                    timeBefore = System.Environment.TickCount;

                    int[] a = checkngSolvability.convertArray(size,
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable =
checkngSolvability.solvability(size, a);

                    if (!solvable)
                    {
                        Console.WriteLine("Puzzle is Not Solvable");
                        return;
                    }
                    Console.WriteLine(FileName + ":");

                    Console.WriteLine("Puzzle is Solvable\n");
                SolvingPUZZLE manhatten = new SolvingPUZZLE("M");
                manhatten.getZeroIndex(arr, size, ref raw, ref col);
                dis = manhatten.calculateDistanceManhatten(arr, size);
                start = new State(arr, size, "", dis, 0, null, raw,
col);
                State endM = manhatten.AStar(start);
                timeAfter = System.Environment.TickCount;

                Console.WriteLine("Manhatten:");
                Console.WriteLine("{0} # of movements.", endM.levels);
                   // totalTime += timeAfter - timeBefore;

                    Console.WriteLine("total time: {0} ms{1}\n",
                    (timeAfter - timeBefore), " -  " +
(float)(timeAfter - timeBefore) / 1000 + " s.\n");

                }
                //Console.WriteLine("___\nTotal Time: {0} ms{1}\n",
totalTime, " -  " + (float)totalTime / 1000 + " s.\n");

            }
            //-------------------------
            else if (input == "6")
            {
                foreach (string FileName in BFStests)
                {
                    file = new FileStream(FileName + ".txt",
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();

                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }

                    timeBefore = System.Environment.TickCount;

                    int[] a = checkngSolvability.convertArray(size,
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable =
checkngSolvability.solvability(size, a);

                    if (!solvable)
                    {
                        Console.WriteLine("Puzzle is Not Solvable");
                        return;
                    }
                    Console.WriteLine(FileName + ":");

                    Console.WriteLine("Puzzle is Solvable\n");
                    SolvingPUZZLE BFS = new SolvingPUZZLE("");
                    BFS.getZeroIndex(arr, size, ref raw, ref col);
                    start = new State(arr, size, "", 0, 0, null, raw,
col);
                    State endB = BFS.BFSTree(start);
                    timeAfter = System.Environment.TickCount;
                    Console.WriteLine("BFS:");
                    Console.WriteLine("{0} # of movements.",
endB.levels);
                    Console.WriteLine("total time: {0} ms{1}\n",
(timeAfter - timeBefore), " -  " + (float)(timeAfter - timeBefore) /
1000 + " s.\n");

                }
            }
            else if (input == "7")
            {
                foreach (string FileName in DFStests)
                {
                    file = new FileStream(FileName + ".txt",
FileMode.Open, FileAccess.Read);
                    sr = new StreamReader(file);
                    size = int.Parse(sr.ReadLine());
                    int[,] arr = new int[size, size];
                    for (int i = 0; i < size; i++)
                    {
                        line = sr.ReadLine();
                        string[] tiles = line.Split(' ');
                        if (tiles[0] == "")
                        {
                            i--;
                            continue;
                        }
                        for (int ii = 0; ii < size; ii++)
                        {
                            arr[i, ii] = int.Parse(tiles[ii]);
                        }
                    }

                    timeBefore = System.Environment.TickCount;

                    int[] a = checkngSolvability.convertArray(size, 
arr);
                    checkngSolvability.fillArr(size, arr);
                    bool solvable = 
checkngSolvability.solvability(size, a);

                    if (!solvable)
                    {
                        Console.WriteLine("Puzzle is Not Solvable");
                        return;
                    }
                    Console.WriteLine(FileName + ":");

                    Console.WriteLine("Puzzle is Solvable\n");
                    SolvingPUZZLE DFS = new SolvingPUZZLE("");
                    //timeBefore = System.Environment.TickCount;
                    DFS.getZeroIndex(arr, size, ref raw, ref col);
                    start = new State(arr, size, "", 0, 0, null, raw, 
col);
                    State endD = DFS.DFSTree(start);
                    timeAfter = System.Environment.TickCount;
                    Console.WriteLine("DFS:");
                    Console.WriteLine("{0} # of movements.", 
endD.levels);
                    Console.WriteLine("total time: {0} ms{1}\n", 
(timeAfter - timeBefore), " -  " + (float)(timeAfter - timeBefore) / 
1000 + " s.\n");

                }

            }
            else if (input == "8")
            {
                file = new FileStream("TEST" + ".txt", FileMode.Open,
FileAccess.Read);
                sr = new StreamReader(file);
                size = int.Parse(sr.ReadLine());
                int[,] arr = new int[size, size];
                for (int i = 0; i < size; i++)
                {
                    line = sr.ReadLine();

                    string[] tiles = line.Split(' ');
                    if (tiles[0] == "")
                    {
                        i--;
                        continue;
                    }
                    for (int ii = 0; ii < size; ii++)
                    {
                        arr[i, ii] = int.Parse(tiles[ii]);
                    }
                }


                int[] a = checkngSolvability.convertArray(size, arr);
                checkngSolvability.fillArr(size, arr);
                bool solvable = checkngSolvability.solvability(size,
a);

                if (!solvable)
                {
                    Console.WriteLine("Puzzle is Not Solvable");
                    return;
                }

                Console.WriteLine("Puzzle is Solvable\n");

                timeBefore = System.Environment.TickCount;

                SolvingPUZZLE manhatten = new SolvingPUZZLE("M");
                //timeBefore = System.Environment.TickCount;
                manhatten.getZeroIndex(arr, size, ref raw, ref col);
                dis = manhatten.calculateDistanceManhatten(arr, size);
                start = new State(arr, size, "", dis, 0, null, raw,
col);
                State endM = manhatten.AStar(start);
                timeAfter = System.Environment.TickCount;

                Console.WriteLine("TEST:");
                Console.WriteLine("{0} # of movements.", endM.levels);
                Console.WriteLine("total time: {0} ms{1}\n",
                    (timeAfter - timeBefore), " -  " +
(float)(timeAfter - timeBefore) / 1000 + " s.\n");


            }

            else
            {
                Console.WriteLine("Invalid Input :((((((((((((((");
                return;
            }
            Console.WriteLine("\n\nDone :))))))))))))");

        }

    }


}
```

PriorityQueue Class:

```
using System.Collections.Generic;

namespace N_PUZZLE
{
    class PriorityQueue
    {

        private List<State> States;
        HashSet<int[,]> openList;


        public PriorityQueue()
      {
            States = new List<State>();
            openList = new HashSet<int[,]>();
        }

        public void Enqueue(State newState)//8,7,6
        {
            if(!openList.Contains(newState.arr))
            {
                 States.Add(newState);
                 siftUp();
            }
        }

        public State Dequeue()
        {
            State min = States[0];
            States[0] = States[States.Count - 1];
            States.RemoveAt(States.Count - 1);
            siftDown();

            return min;
        }

        void siftUp()
        {

            int ci = States.Count - 1;
            while (ci > 0)
            {
                int pi = (ci - 1) / 2; // parent index

                // If child item is larger than (or equal) parent so
    we're done
                if (States[ci].distance > States[pi].distance)
                    break;
                // Else swap parent & child
                SwapInt(ci, pi);

                ci = pi;
            }

        }




        void siftDown()
        {
            int li = States.Count - 1; //last index (after removal)

            int pi = 0; // parent index. start at front of pq
            while (true)
            {
                int ci = pi * 2 + 1; // left child index of parent
                if (ci > li)
                    break;  // no children so done
                int rc = ci + 1;     // right child
                // if there is a right child , and it is smaller than
left child, use the rc instead
                if (rc <= li &&
States[rc].distance.CompareTo(States[ci].distance) < 0)
                    ci = rc;
                // If parent is smaller than (or equal to) smallest
child so done
                if (States[pi].distance < States[ci].distance)
                    break;
                // Else swap parent and child
                SwapInt(pi, ci);

                pi = ci;
            }


        }

        void SwapInt(int X, int Y)
        {
            State tmpS;
            tmpS = States[X];
            States[X] = States[Y];
            States[Y] = tmpS;
        }



    }
}
```