N Puzzle A* Search Solver

Implemented Code & Detailed Analysis Documentation

Team: T028

Department: SC

Adnan Ahmad Dalain - 20191700804 - SC

Rawan Mostafa Ramadan - 20191700800 - SC

Demiana Esam Kamel - 201191700234 - SC

Functions Details & Code Analysis

1) Loading GUI Form: setting Test file names in the Compo_box **O(1)**

Function Code:

```
private void Form1_Load(object sender, EventArgs e)
        {
            shortest_path = new List<game_State>();
            GUI_puzzle_btns = new List<Button>();

            file_path= "99 Puzzle - 1.txt";
            //Sample - Solvable
            puzzle_path_textBox.Text = file_path;
            puzzle_path_textBox.Items.Add("--Sample Tests - Solvable--");
            puzzle_path_textBox.Items.Add("8 Puzzle (1).txt");
            puzzle_path_textBox.Items.Add("8 Puzzle (2).txt");
            puzzle_path_textBox.Items.Add("8 Puzzle (3).txt");
            puzzle_path_textBox.Items.Add("15 Puzzle - 1.txt");
            puzzle_path_textBox.Items.Add("24 Puzzle 1.txt");
            puzzle_path_textBox.Items.Add("24 Puzzle 2.txt");
            //Sample - unsolvable
            puzzle_path_textBox.Items.Add("--Sample Tests - Unsolvable--");
            puzzle_path_textBox.Items.Add("8 Puzzle - Case 1.txt");
            puzzle_path_textBox.Items.Add("8 Puzzle(2) - Case 1.txt");
            puzzle_path_textBox.Items.Add("8 Puzzle(3) - Case 1.txt");
            puzzle_path_textBox.Items.Add("15 Puzzle - Case 2.txt");
            puzzle_path_textBox.Items.Add("15 Puzzle - Case 3.txt");
            //Complete solvable Manhattan & Hamming
            puzzle_path_textBox.Items.Add("--Complete Tests - Solvable--");
            puzzle_path_textBox.Items.Add("50 Puzzle.txt");
            puzzle_path_textBox.Items.Add("99 Puzzle - 1.txt");
            puzzle_path_textBox.Items.Add("99 Puzzle - 2.txt");
            puzzle_path_textBox.Items.Add("9999 Puzzle.txt");
            //Complete solvable Manhattan Only
            puzzle_path_textBox.Items.Add("--Complete Tests - Manhattan Only--
                                                                    ");
            puzzle_path_textBox.Items.Add("15 Puzzle 1.txt");
            puzzle_path_textBox.Items.Add("15 Puzzle 3.txt");
            puzzle_path_textBox.Items.Add("15 Puzzle 4.txt");
            puzzle_path_textBox.Items.Add("15 Puzzle 5.txt");
            //Complete - unsolvable
            puzzle_path_textBox.Items.Add("--Complete Tests - Unsolvable--");
            puzzle_path_textBox.Items.Add("15 Puzzle 1 - Unsolvable.txt");
            puzzle_path_textBox.Items.Add("99 Puzzle - Unsolvable Case 1.txt");
            puzzle_path_textBox.Items.Add("99 Puzzle - Unsolvable Case 2.txt");
            puzzle_path_textBox.Items.Add("9999 Puzzle  - Unsolvable Case.txt");
            //Complete - v Large
            puzzle_path_textBox.Items.Add("--Complete Tests - V.Large--");
            puzzle_path_textBox.Items.Add("TEST.txt");

            groupBox1.Controls.Add(this.Hamming_radio);
            groupBox1.Controls.Add(this.Manhattan_radio);
            Manhattan_radio.Checked = true;
                                                    }
```

2) Game State Class

Data Structure object which holds each state of the puzzle to be used by A* Search Algorithm during the solving process from Initial state to the Goal state

Class Code:

```
class game_State
   {
        public game_State parent;
        public int[,] game_matrix;
        public int x_blank;
        public int y_blank;
        public int Heuristic_Cost; //Total cost (sum of Hamming & Manhattan)
        public int Expansion_Level;
        public int from;
        private int Hamming_priority;
        private int Manhattan_priority;
        private int size;

        //implementation of Swap function
        static void Swap(ref int x, ref int y)   O(1)
        {
            int temp = x;
            x = y;
            y = temp;
        }

        //Root constructor (without X & Y blank of child)                    O(1)
        public game_State(int size, game_State parent, int[,] game_matrix,
                          int x_blank, int y_blank, int Expansion_order)
        {
            this.from = -1;
            this.size = size;
            this.parent = parent;
            this.game_matrix = game_matrix.Clone() as int[,];
            this.x_blank = x_blank;
            this.y_blank = y_blank;
            this.Expansion_Level = Expansion_order; //Level
            this.Hamming_priority = -1;
            this.Manhattan_priority = -1;
            this.Heuristic_Cost = int.MaxValue;
        }

        //New game states Constructor (to be used during search with new X & Y Blank values)
        public game_State(int size, game_State parent, int[,] game_matrix, int x_blank, int y_blank,
                          int x_blank_child, int y_blank_child, int Expansion_order)
        {
            this.from = -1;
            this.size = size;
            this.parent = parent;
            this.game_matrix = new int[size, size];
            this.Expansion_Level = Expansion_order;
            this.Hamming_priority = -1;
            this.Manhattan_priority = -1;
            this.Heuristic_Cost = int.MaxValue;
```

```
            //copying Game matrix from parameters to the object variable
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    this.game_matrix[i, j] = game_matrix[i, j];
                }
            }

            //sliding a value to generate the new state of puzzle
            Swap(ref this.game_matrix[x_blank, y_blank], ref this.game_matrix[x_blank_child,
                                                                              y_blank_child]);

            //Update the indices of blank value in the new game state object
            this.x_blank = x_blank_child;
            this.y_blank = y_blank_child;
        }

        //calculate and set Total (Heuristic) cost
        public void set_Heuristic_Cost(int Hamming, int Manhattan, int priority_fn)
        {
            this.Hamming_priority = Hamming;
            this.Manhattan_priority = Manhattan;

            if (priority_fn == 1)
            {
                Heuristic_Cost = Hamming + Expansion_Level;
            }
            else
            {
                Heuristic_Cost = Manhattan + Expansion_Level;
            }
        }


        public bool is_solved()
        {
            if (Hamming_priority == 0 || Manhattan_priority == 0)
            {
                return true;
            }
            return false;
        }
        public void display_state()
        {
            for (int i = 0; i < size; i++)
            {
                Console.Write("[");
                for (int j = 0; j < size; j++)
                {
                    if (j == (size - 1))
                    {
                        Console.Write(game_matrix[i, j] + " ");
                    }
                    else
                        Console.Write(game_matrix[i, j] + " ,");
                }
                Console.Write("]");
                Console.WriteLine();
            }
        }

    }
}
```

O(S)

O(1)

O(1)

O(S)

3) Is Solvable() & Calculate Inversion() $O(S^2)$

(Where "S" is the puzzle size)

We can determined if a puzzle is solvable or not by calculating Inversions and then check if it's odd or even

Function Code:

```
static bool is_solvable(int[,] game_matrix)
        {
            int Count = calc_Inversions(game_matrix);


            if (Matrix_Size % 2 != 0 && Count % 2 == 0)
                return true;

                                                    O(1)
            Else
            {
                int pos = Matrix_Size - x_blank;
                if (pos % 2 != 0)
                    return Count % 2 == 0;
                else
                    return Count % 2 != 0;
            }


        }
static int calc_Inversions(int[,] game_matrix)
        {

            List<int> one_D_arr = new List<int>();
            for (int i = 0; i < Matrix_Size; i++)       O(S)
            {
                for (int j = 0; j < Matrix_Size; j++)
                {
                    if (game_matrix[i, j] != 0)
                    {
                        one_D_arr.Add(game_matrix[i, j]);
                    }
                }
            }

            int Count = 0 , list_size = one_D_arr.Count;
            for (int i = 0; i < list_size - 1; i++)     O(S^2)
            {
                for (int j = i + 1; j < list_size; j++)
                {
                    if (one_D_arr[i] > one_D_arr[j])
                    {
                        Count++;
                    }
                }
            }
            return Count;
        }
```

4) Calculate Heuristic Cost() O(S)

Calculating Hamming or Manhattan Priority of each Game State to be used in f(n)= h + g as a Heuristic_Cost

Function Code:

```
static void calculate_Heuristic_Cost(game_State current_state)
       {
          int Manhattan = int.MaxValue;
          int Wrong_Positions = int.MaxValue;

          if (funtion_selected == 1)
          {
              Wrong_Positions = 0;
              // Hamming
              for (int i = 0; i < Matrix_Size; i++)
              {
                  for (int j = 0; j < Matrix_Size; j++)
                  {
                      if (current_state.game_matrix[i, j] != Goal[i, j] &&
                          current_state.game_matrix[i, j] != 0)
                      {
                          Wrong_Positions++;
                      }
                  }
              }
          }
          if(funtion_selected == 2)
          {
              Manhattan = 0;
              //manhattan
              for (int k = 0; k < Matrix_Size; k++)
              {
                  for (int n = 0; n < Matrix_Size; n++)
                  {
                      if (current_state.game_matrix[k, n] != Goal[k, n] &&
                           current_state.game_matrix[k, n] != 0)
                      {

                          int real_J = (current_state.game_matrix[k, n] - 1) % Matrix_Size;
                          int real_i = (current_state.game_matrix[k, n] - 1) / Matrix_Size;
                          Manhattan += (Math.Abs(real_i - k) + Math.Abs(real_J - n));
                      }
                  }
              }
          }

           current_state.set_Heuristic_Cost(Wrong_Positions, Manhattan, funtion_selected);
       }
```

5) Solve() O(E log(V))

(Where "E" is the total # of moves & "V" is states # to Goal)

This function makes the actual solving by applying Modified version of A* Search Algorithm with Hamming or Manhattan Priority to Initiate a Graph that simulates the states of games and trying to solve it using Heuristic_Cost function to reach the selected Goal State

Function Code:

```
static void solve(int[,] initial_mat, int Matrix_Size, int x_blank, int y_blank)
        {
            PriorityQueue<game_State, int> qu = new PriorityQueue<game_State, int>();

            game_State initial_State = new game_State(Matrix_Size, null, initial_mat, x_blank,
                                                                        y_blank, 0);
            calculate_Heuristic_Cost(initial_State);

            qu.Enqueue(initial_State, initial_State.Heuristic_Cost);

            // bottom, left, top, right
            int[] move_Directions = { 1, 0, -1, 0, 0, -1, 0, 1 };
            int x_move_safty = 0;
            int y_move_safty = 0;

                                                                        O(E log(V))
            while (qu.TryDequeue(out game_State current_Game_state, out int H_cost))
            {
                if (current_Game_state.is_solved())
                {
                    solver_Stopwatch.Stop();
                    printPath(current_Game_state);
                    Console.WriteLine("Number of moves: " + (moves_counter - 1));
                    CreateFile();
                    return;
                }
                for (int i = 0; i < 4; i++)
                {                       O(1)
                    x_move_safty = current_Game_state.x_blank + move_Directions[i];
                    y_move_safty = current_Game_state.y_blank + move_Directions[i + 4];
                    if (x_move_safty >= 0 && x_move_safty < Matrix_Size && y_move_safty >= 0 &&
                          y_move_safty < Matrix_Size && current_Game_state.from != i)
                    {
                        game_State next_Move = new game_State(Matrix_Size, current_Game_state,
                                                current_Game_state.game_matrix,
                                                current_Game_state.x_blank,
                                                current_Game_state.y_blank, x_move_safty,
                                            y_move_safty, (current_Game_state.Expansion_Level + 1));
                        next_Move.from = (i + 2) % 4;
                        calculate_Heuristic_Cost(next_Move);

                        qu.Enqueue(next_Move, next_Move.Heuristic_Cost);        O(log(V))

                    }     }     }     }
```

6) printPath() O(M S)

(Where “S” is the puzzle size & M # of S-Path movments)

After completing the solving process and reaching Goal state, This function is called to print the shortest path from the initial state to the goal, move by move as a result in the console in case of using “Console Solver” or preparing the path as a list to be displayed in the GUI Form in case of using “GUI Solver”

Also it calculates the # of moves of shortest path

Function Code:

```
static void printPath(game_State goal_State)
        {
            if (goal_State == null)
                return;
            //Print the path from root to Goal
            printPath(goal_State.parent);

            Console.WriteLine("Move #" + moves_counter);
            if (is_Console)
            {
                goal_State.display_state();     O(S)
            }
            shortest_path.Add(goal_State);      O(1)
            moves_counter++;
            Console.WriteLine();
        }
```

7) CreateFile() O(M S)

(Where “S” is the puzzle size & M # of S-Path movments)

After printing the shortest path on the screen the result solution & solving time in ms and sec will be stored in a text file as “Solutions\fileName solution.txt”

Function Code:

```
static void CreateFile()
        {
            string solved_file_path = @"Solutions\"+file_path.Remove(file_path.Length - 4)
+ " solution.txt";
            try
            {
                FileStream sb = new FileStream(solved_file_path, FileMode.OpenOrCreate);
                using (StreamWriter sr = new StreamWriter(sb))
                {

                    int create_counter = 0;
                    foreach (game_State s in shortest_path)
                    {

                        sr.WriteLine("Move #" + create_counter);
                        for (int i = 0; i < Matrix_Size; i++)          O(S)
                        {
                            for (int j = 0; j < Matrix_Size; j++)
                            {
                                sr.Write(s.game_matrix[i, j] + " ");
                            }
                            sr.WriteLine();
                        }
                        create_counter++;
                        sr.WriteLine();
                    }
                    sr.WriteLine("Total number of moves: " + (create_counter - 1));
                    sr.WriteLine("Solving time: " + solver_Stopwatch.ElapsedMilliseconds +
                                " ms      /      " + (solver_Stopwatch.ElapsedMilliseconds
                                                        / (float)1000) + " Sec");
                }
            }
            catch
            {
                File.Delete(solved_file_path);
                MessageBox.Show("Error in saving Solution File");
            }
        }
```

8) GUI Solver or Console Solver functions() $O(S^2 + E \log(V))$

They share the same purpose but in different way of displaying the results

This function can be considered as the main or entering point to the solver as it's Reading N-Puzzle from text file then find blank block position & Generate The Goal state Puzzle

Then, Check if the puzzle is solvable or not before calling Solve() function to solve the puzzle and Printing moves path & Solving Time In sec & ms

Console Function Code:

```
private void Console_solver_btn_Click(object sender, EventArgs e)
        {
            [DllImport("kernel32.dll", SetLastError = true)]
            [return: MarshalAs(UnmanagedType.Bool)]
            static extern bool AllocConsole();


            is_Console = true;
            GUI_solver_btn.Enabled = false;

            file_path = puzzle_path_textBox.Text;
            try
            {
                FileStream file = new FileStream(file_path, FileMode.Open, FileAccess.Read); O(1)

                StreamReader s_Reader = new StreamReader(file);      O(1)
                string line;

                if (s_Reader == null) //if its not open
                {
                    throw new Exception("Unable to open file.");
                }
                //Open Console
                AllocConsole();
                //Read First Line to get Matrix Size
                line = s_Reader.ReadLine();
                Matrix_Size = int.Parse(line);
                Console.WriteLine("Matrix size: " + Matrix_Size + "\n");

                //To skip the empty line
                s_Reader.ReadLine();

                //declaring 2D array
                int[,] game_mat = new int[Matrix_Size, Matrix_Size];

                // reading data from text file
```

```
    string[] temp_row = new string[Matrix_Size];

for (int i = 0; i < Matrix_Size; i++)      O(S)
    {
        line = s_Reader.ReadLine();
        temp_row = line.Split(" ");
        for (int j = 0; j < Matrix_Size; j++)
        {
            game_mat[i, j] = int.Parse(temp_row[j]);
        }
    }

    //Closing file & stream reader
    s_Reader.Close();
    file.Close();

    //Generate Goal matrex
    Goal = new int[Matrix_Size, Matrix_Size];
    int temp_counter = 1;

    for (int i = 0; i < Matrix_Size; i++)   O(S)
    {
        for (int j = 0; j < Matrix_Size; j++)
        {
            if (i == (Matrix_Size - 1) && j == (Matrix_Size - 1))
            {
                //to set the last value in the matrix (Blank value) = 0
                Goal[i, j] = 0;
                break;
            }
            Goal[i, j] = temp_counter;
            temp_counter++;
        }
    }

    //Finding Blank value index in initial matrix
    bool position_found = false;

    for (int i = 0; i < Matrix_Size; i++)   O(S)
    {
        for (int j = 0; j < Matrix_Size; j++)
        {
            if (game_mat[i, j] == 0)
            {
                x_blank = i;
                y_blank = j;
                position_found = true;
                break;
            }
        }
        if (position_found)
            break;
    }

    if (Hamming_radio.Checked)                          O(1)
    {
        funtion_selected = 1;
    }
    else
    {

        funtion_selected = 2;
    }
    if (funtion_selected == 1)
    {
        Console.WriteLine("Selected Priority function: Hamming");
    }
```

```
            else
            {
                Console.WriteLine("Selected Priority function: Manhattan");
            }

            //Time Measure to solve the puzzle using A* algorithm
            solver_Stopwatch = Stopwatch.StartNew();
            if (is_solvable(game_mat))
            {
                Console.WriteLine("\nSolving Started: \n");
                solve(game_mat, Matrix_Size, x_blank, y_blank);
            }
            else
            {
                Console.WriteLine("Puzzle is not solvable");
                return;
            }
            Console.WriteLine("Solving Time is: " + (solver_Stopwatch.ElapsedMilliseconds) + "
                 ms / " + (solver_Stopwatch.ElapsedMilliseconds / (float)1000) + " sec");

        }
        catch
        {
            MessageBox.Show("Invalid path, Please try again (.../fileName.txt)");
        }
    }
```

$O(S^2)$

$O(E \log(V) + M S)$

The Remaining function is out of scope

as they are a part of GUI bones

GUI Form Code:

```
private void GUI_solver_btn_Click(object sender, EventArgs e)
        {
            file_path = puzzle_path_textBox.Text;
            try
            {
                FileStream file = new FileStream(file_path, FileMode.Open, FileAccess.Read); O(1)
                StreamReader s_Reader = new StreamReader(file);      O(1)
                string line;

                if (s_Reader == null) //if its not open
                {
                    throw new Exception("Unable to open file.");
                }
                //Read First Line to get Matrix Size
                line = s_Reader.ReadLine();
                Matrix_Size = int.Parse(line);
                Console.WriteLine("Matrix size: " + Matrix_Size + "\n");

                if (Matrix_Size > 10)
                {
                    s_Reader.Close();
                    file.Close();
                    MessageBox.Show("Sorry, The Puzzle is too large to display- Please use Console
                                                                                        Solver.");
                    Application.Restart();
                    Environment.Exit(0);

                }
                //To skip the empty line
                s_Reader.ReadLine();

                //declaring 2D array
                int[,] game_mat = new int[Matrix_Size, Matrix_Size];

                // reading data from text file
                string[] temp_row = new string[Matrix_Size];
                for (int i = 0; i < Matrix_Size; i++)  O(S)
                {
                    line = s_Reader.ReadLine();
                    temp_row = line.Split(" ");
                    for (int j = 0; j < Matrix_Size; j++)
                    {
                        game_mat[i, j] = int.Parse(temp_row[j]);
                    }
                }

                //Closing file & stream reader
                s_Reader.Close();
                file.Close();

                //Generate Goal matrex
                Goal = new int[Matrix_Size, Matrix_Size];
                int temp_counter = 1;
                for (int i = 0; i < Matrix_Size; i++)  O(S)
                {
```

```
    for (int j = 0; j < Matrix_Size; j++)
    {
        if (i == (Matrix_Size - 1) && j == (Matrix_Size - 1))
        {
            //to set the last value in the matrix (Blank value) = 0
            Goal[i, j] = 0;
            break;
        }
        Goal[i, j] = temp_counter;
        temp_counter++;
    }
}

//Finding Blank value index in initial matrix
bool position_found = false;
for (int i = 0; i < Matrix_Size; i++)   O(S)
{
    for (int j = 0; j < Matrix_Size; j++)
    {
        if (game_mat[i, j] == 0)
        {
            x_blank = i;
            y_blank = j;
            position_found = true;
            break;
        }
    }
    if (position_found)
        break;
}

//set puzzle block size (Dynamically)
int button_size = 470 / Matrix_Size;
//selecting Priority function based on user choice
if (Hamming_radio.Checked)
{
    funtion_selected = 1;
    priority_label.Text = "Hamming priority";
}




else
{
    funtion_selected = 2;
    priority_label.Text = "Manhattan priority";
}
solver_Stopwatch = Stopwatch.StartNew();
if (is_solvable(game_mat))
{
    solve(game_mat, Matrix_Size, x_blank, y_blank);
}
else
{
    MessageBox.Show("This Puzzle is unsolvable");
    Application.Restart();
}


for (int i = 0; i < Matrix_Size; i++)   O(S)
{
    for (int j = 0; j < Matrix_Size; j++)
    {
        Create_block(((j * button_size) + 52), ((i * button_size) + 163),
                                                button_size, game_mat[i, j]);
    }
}
```

O(1)

```
            solving_T_label.Text = (solver_Stopwatch.ElapsedMilliseconds /
                                    (float)1000).ToString() + " sec";
            solving_T_millS_label.Text = (solver_Stopwatch.ElapsedMilliseconds).ToString() + "
                                                                                    ms";
            total_moves_label.Text = (moves_counter - 1).ToString();
            welcome_panel.Hide();
        }
        catch
        {
            MessageBox.Show("Invalid path, Please try again (.../fileName.txt)");
        }

    }
```

Create block() Code:

This function creates a dynamic # of puzzle blocks base on Puzzle size

```
private void Create_block(int pos_x,int pos_y,int size,int text)
    {
        Button puzzle_Block = new Button();
        GUI_puzzle_btns.Add(puzzle_Block);

        puzzle_Block.Height = size;
        puzzle_Block.Width = size;
        puzzle_Block.BackColor = Color.AliceBlue;
        puzzle_Block.ForeColor = Color.Black;
        //puzzle_Block.BackColor = Color.Black;
        //puzzle_Block.ForeColor = Color.White;
        puzzle_Block.Location = new Point(pos_x, pos_y);
        if(text!=0)
            puzzle_Block.Text = text.ToString();
        puzzle_Block.Name = "Dynamic_Puzzle_Button"+text;
        puzzle_Block.Font = new Font("Georgia", 12);

        // Add Button to the Form. Placement of the Button
        // will be based on the Location and Size of button
        game_panel.Controls.Add(puzzle_Block);

    }
```

Restart btn() Code:

Used in the GUI as a restart Button to solve another N Puzzle

```
private void restart_btn_Click(object sender, EventArgs e)
    {
        Application.Restart();
    }
```

Next Move () Code:

Used in GUI Form to enable the user to trace the movements of solution step by step in a visual way

```
static int num_clicks = 1;
        private void Next_Move_Click(object sender, EventArgs e)
        {
            if (num_clicks < moves_counter)
            {
                int counter = 0;
                for (int i = 0; i < Matrix_Size; i++)
                {
                    for (int j = 0; j < Matrix_Size; j++)
                    {
                        GUI_puzzle_btns[counter].Text = shortest_path[num_clicks].game_matrix[i,
                            j] != 0 ? shortest_path[num_clicks].game_matrix[i, j].ToString() : "";
                        game_panel.Controls.Add(GUI_puzzle_btns[counter]);
                        counter++;
                    }

                }
                moves_num.Text = (num_clicks).ToString();
                num_clicks++;
            }
            else
            {
                solved_label.Text = "Puzzle Solved Successfully";
            }

        }
```

Auto solving () Code:

This Button is similar to the previous one but goes from initial state to Goal state step by step in a visual way automatically

```
        Timer auto_solver_timer = new Timer();
        private void auto_solving_Click(object sender, EventArgs e)
        {
            auto_solver_timer.Interval = 250;
            EventHandler ev = new EventHandler(auto_Next_move_Tick);
            auto_solver_timer.Tick += ev;
            auto_solver_timer.Start();
        }
        private void auto_Next_move_Tick(object sender, EventArgs e)
        {
            if (num_clicks == (moves_counter))
            {
                auto_solver_timer.Enabled = false;
            }
            Next_Move.PerformClick();
        }
```

Hamming Priority Vs Manhattan Priority

Manhattan Priority

Test File Name	Min number of moves	Execution time
50 Puzzle	18	30 ms / 0.03 sec
99 Puzzle - 1	18	3 ms / 0.003 sec
99 Puzzle - 2	38	3 ms / 0.003 sec
9999 Puzzle	4	201 ms / 0.201 sec

Hamming Priority

Test File Name	Min number of moves	Execution time
50 Puzzle	18	270 ms / 0.27 sec
99 Puzzle - 1	18	3 ms / 0.003 sec
99 Puzzle - 2	38	4 ms / 0.004 sec
9999 Puzzle	4	199 ms / 0.199 sec

As shown in the comparison and another Puzzle test: Manhattan Priority is more powerful and effective than Hamming as it's applicable on almost all solvable N-Puzzle which Hamming priority fails sometimes in solving them.

Solvable Sample & Complete Tests Results

Samples with Hamming Priority

Test File Name	Min number of moves	Execution time
8 Puzzle (1)	8	3 ms / 0.003 sec
8 Puzzle (2)	20	9 ms / 0.009 sec
8 Puzzle (3)	14	3 ms / 0.003 sec
15 Puzzle - 1	5	3 ms / 0.003 sec
24 Puzzle 1	11	3 ms / 0.003 sec
24 Puzzle 2	24	3 ms / 0.003 sec

Complete with Manhattan Priority

Test File Name	Min number of moves	Execution time
50 Puzzle	18	30 ms / 0.03 sec
99 Puzzle - 1	18	3 ms / 0.003 sec
99 Puzzle - 2	38	3 ms / 0.003 sec
9999 Puzzle	4	201 ms / 0.201 sec
15 Puzzle 1	46	3115 ms / 3.115 sec
15 Puzzle 3	38	1100 ms / 1.1 sec
15 Puzzle 4	44	374 ms / 0.374 sec
15 Puzzle 5	45	14458 ms/ 14.458 sec

V Large Test

Test File Name	Min number of moves	Execution time
TEST	56	11291 ms / 11.291 sec