**N-PUZZLE Project**

**Team members**

**اندريه عادل ابراهيم (1**

**أندرو ناصف امين (2**

**مينا سامى انور (3**

**Sections**

**1- Detailed analysis of code**

**2- Hamming vs. Manhattan over “Complete Test” comparison**

**3- Entire Source Code (without GUI)**

**4- All test cases (time and number of moves)**

1) Detailed analysis of code:

Note: any line that is not analyzed means that it is Θ(1)

Abbreviations:

S size of puzzle = (dimension \* dimension) (ex: 8-puzzle , S = 3\*3 = 9).

E Total number of moves to get to the solution.

V Number of states till reaching the solution.

M Min number of moves to solve the puzzle.

In Node class:

public Node(int[] puzzle\_1d, int puzzle\_dimension, Node parent, int cost\_so\_far) Θ(S)

{

this.puzzle\_1d = new int[puzzle\_dimension \* puzzle\_dimension];

puzzle\_1d.CopyTo(this.puzzle\_1d, 0); Θ(S)

this.puzzle\_dimension = puzzle\_dimension;

this.cost\_so\_far = cost\_so\_far;

this.parent = parent;

}

public void calculate\_manhattan\_distance() Θ(S)

{

int distance = 0;

int puzzle\_size = puzzle\_dimension \* puzzle\_dimension;

for (int i = 0; i < puzzle\_size; i++) Θ(S)

{

int v = puzzle\_1d[i];

if (v == 0)

{

continue;

}

v = v - 1;

int goal\_x = v % puzzle\_dimension;

int goal\_y = v / puzzle\_dimension;

int x = i % puzzle\_dimension;

int y = i / puzzle\_dimension;

int manhatten\_cost = Math.Abs(x - goal\_x) + Math.Abs(y - goal\_y);

distance += manhatten\_cost;

}

heuristic\_value = distance;

}

public void calculate\_hamming\_distance() Θ(S)

{

for (int i = 0; i < puzzle\_dimension \* puzzle\_dimension; i++) Θ(S)

{

if (puzzle\_1d[i] != i + 1 && puzzle\_1d[i] != 0)

heuristic\_value++;

}

}

public int priority() Θ(1)

{

return heuristic\_value + cost\_so\_far;

}

public void modify\_manhattan\_distance(int old\_index, int new\_index) Θ(1)

{

int v = puzzle\_1d[old\_index];

v = v - 1;

int goal\_x = v % puzzle\_dimension;

int goal\_y = v / puzzle\_dimension;

int x = new\_index % puzzle\_dimension;

int y = new\_index / puzzle\_dimension;

int manhatten\_cost = Math.Abs(x - goal\_x) + Math.Abs(y - goal\_y);

heuristic\_value -= manhatten\_cost;

v = puzzle\_1d[old\_index];

v = v - 1;

goal\_x = v % puzzle\_dimension;

goal\_y = v / puzzle\_dimension;

x = old\_index % puzzle\_dimension;

y = old\_index / puzzle\_dimension;

manhatten\_cost = Math.Abs(x - goal\_x) + Math.Abs(y - goal\_y);

heuristic\_value += manhatten\_cost;

}

public void modify\_hamming\_distance(int old\_index , int new\_index) Θ(1)

{

if (puzzle\_1d[old\_index] == old\_index + 1)

{

heuristic\_value--;

}

else if (puzzle\_1d[old\_index] == new\_index + 1)

{

heuristic\_value++;

}

}

In Program class:

public static bool is\_puzzle\_solvable(int puzzle\_dimension, int[] puzzle\_1d\_array) O(S^2)

{

int no\_of\_inversions = 0;

int blank\_space\_pos = 0;

for (int i = 0; i < puzzle\_dimension \* puzzle\_dimension; i++) Θ(S)\*O(S) = O(S^2)

{

if (puzzle\_1d\_array[i] == 0)

{

blank\_space\_pos = i / puzzle\_dimension + 1;

continue;

}

for (int j = i + 1; j < puzzle\_dimension \* puzzle\_dimension; j++) O(S)

{

if (puzzle\_1d\_array[i] > puzzle\_1d\_array[j] && puzzle\_1d\_array[j] != 0)

{

no\_of\_inversions++;

}

}

}

//if N is even

if (puzzle\_dimension % 2 == 0)

{

if (no\_of\_inversions % 2 == 0 && (puzzle\_dimension - blank\_space\_pos) % 2 == 0 ||

no\_of\_inversions % 2 != 0 && (puzzle\_dimension - blank\_space\_pos) % 2 != 0)

{

return true;

}

}

//if N is odd

else

{

if (no\_of\_inversions % 2 == 0)

{

return true;

}

}

return false;

}

public static Node A\_Star\_Search(Node node) O(E log(V))

{

total\_number\_of\_moves = 0;

Node tmp\_node = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far); Θ(S)

while (node.heuristic\_value != 0) Θ(E)

{

total\_number\_of\_moves++;

if (node.heuristic\_value == 0)

{

return node;

}

Node node1;

if (total\_number\_of\_moves > 30000000 && Hamming\_or\_Manhattan == "2")

{

MessageBox.Show("Puzzle is not solvable with hamming distance");

break;

}

// is up node

if (node.zero\_pos - node.puzzle\_dimension >= 0)

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos - node.puzzle\_dimension;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1); Θ(S)

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos - node.puzzle\_dimension];

node1.puzzle\_1d[node.zero\_pos - node.puzzle\_dimension] = 0;

node1.zero\_pos = node.zero\_pos - node.puzzle\_dimension;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority()); O(log(V))

}

}

// is down node

if ((node.zero\_pos + node.puzzle\_dimension) < (node.puzzle\_dimension \* node.puzzle\_dimension))

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos + node.puzzle\_dimension;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1); Θ(S)

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos + node.puzzle\_dimension];

node1.puzzle\_1d[node.zero\_pos + node.puzzle\_dimension] = 0;

node1.zero\_pos = node.zero\_pos + node.puzzle\_dimension;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority()); O(log(V))

}

}

// is right node

if ((node.zero\_pos % node.puzzle\_dimension) != (node.puzzle\_dimension - 1))

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos + 1;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1); Θ(S)

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos + 1];

node1.puzzle\_1d[node.zero\_pos + 1] = 0;

node1.zero\_pos = node.zero\_pos + 1;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority()); O(log(V))

}

}

// is left node

if ((node.zero\_pos % node.puzzle\_dimension) != 0)

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos - 1;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1); Θ(S)

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos - 1];

node1.puzzle\_1d[node.zero\_pos - 1] = 0;

node1.zero\_pos = node.zero\_pos - 1;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority()); O(log(V))

}

}

node = priorityQueue.Dequeue(); O(log(V))

}

return (node);

}

A\_star\_search function analysis:

-Order = E [ ( 4 \* S )+ 5 log(V) ] + S

-Order = (12 \* E \* S)+ (5 \* E \* log(V)) + S

-Order = (E \* S ) + ( E \* log (V))

-Assumption: in large test cases V is a very large

Value so log(V) > (S)

Order = E Log(V)

public static void Solve\_puzzle(string h\_or\_m, int puzzle\_dimension,

int[] puzzle\_1d\_array) O(Elog(V))

{

Hamming\_or\_Manhattan = h\_or\_m;

min\_number\_of\_moves = 0;

if (is\_puzzle\_solvable(puzzle\_dimension, puzzle\_1d\_array)) O(S^2)

{

Node root = new Node(puzzle\_1d\_array, puzzle\_dimension, null, 0); Θ(S)

if (Hamming\_or\_Manhattan == "1" )

root.calculate\_manhattan\_distance(); Θ(S)

else if (Hamming\_or\_Manhattan == "2")

root.calculate\_hamming\_distance(); Θ(S)

for (int i = 0; i < puzzle\_dimension \* puzzle\_dimension; i++) Θ(S)

{

if (root.puzzle\_1d[i] == 0)

{

root.zero\_pos = i;

}

}

priorityQueue.Enqueue(root, root.priority()); O(log(V))

stopwatch.Start();

Node goal = A\_Star\_Search(priorityQueue.Dequeue()); E\*log(V)+log(v)=O(E\*Log(V))

stopwatch.Stop();

MessageBox.Show("time taken to solve the puzzle = " + stopwatch.Elapsed);

stopwatch.Reset();

while (goal.parent != null) Θ(M) O(M^2)

{

Solving\_Moves.Insert(0, goal.puzzle\_1d); O(M)

goal = goal.parent;

min\_number\_of\_moves++;

}

Solving\_Moves.Insert(0, goal.puzzle\_1d); O(M)

}

else

{

MessageBox.Show("Puzzle is not Solvable");

}

}

Solve\_puzzle function analysis:

-Order = S^2 + 3S + log(V) + E\*log(V) + M^2 + M.

-Order = S^2 + S + log(V) + E\*log(V) + M^2 + M.

-Order = S^2 + E\*log(V) + M^2.

-Asssumption: in large test cases E and V are very

Large values compared to the S and M so

E\*log(V) > S^2 & E\*log(V) > M^2

-Order = E \* log(V)

2) **Hamming vs. Manhattan over “Complete Test” comparison**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| test case name | expected output | Manhattan execution Time(sec) | Hamming execution Time (sec) | Manhattan number of moves | Hamming number of moves |
| 50 Puzzle.txt | 18 | 00:0085166 | 00:0448060 | 136 | 2171 |
| 99 Puzzle - 1.txt | 18 | 00:0000265 | 00:0000236 | 18 | 18 |
| 99 Puzzle - 2.txt | 38 | 00:0000651 | 00:0001727 | 67 | 202 |
| 9999 Puzzle.txt | 4 | 00:0000382 | 00:0000607 | 4 | 4 |

Conclusion:

1. Hamming number of moves is always greater than or equal to Manhattan number of moves.
2. Manhattan formula always finds the puzzle solution in the fewest moves possible compared to hamming.
3. For complicated puzzles, Manhattan is better than Hamming to solve it (Takes less moves and less time).

**3- Entire Source Code (without GUI)**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace N\_Puzzle\_game

{

class Node

{

public int[] puzzle\_1d;

public int puzzle\_dimension;

public int heuristic\_value = 0;

public int cost\_so\_far;

public int zero\_pos;

public Node parent;

public Node(int[] puzzle\_1d, int puzzle\_dimension, Node parent, int cost\_so\_far)

{

this.puzzle\_1d = new int[puzzle\_dimension \* puzzle\_dimension];

puzzle\_1d.CopyTo(this.puzzle\_1d, 0);

this.puzzle\_dimension = puzzle\_dimension;

this.cost\_so\_far = cost\_so\_far;

this.parent = parent;

}

public void calculate\_manhattan\_distance()

{

int distance = 0;

int puzzle\_size = puzzle\_dimension \* puzzle\_dimension;

for (int i = 0; i < puzzle\_size; i++)

{

int v = puzzle\_1d[i];

if (v == 0)

{

continue;

}

v = v - 1;

int goal\_x = v % puzzle\_dimension;

int goal\_y = v / puzzle\_dimension;

int x = i % puzzle\_dimension;

int y = i / puzzle\_dimension;

int manhatten\_cost = Math.Abs(x - goal\_x) + Math.Abs(y - goal\_y);

distance += manhatten\_cost;

}

heuristic\_value = distance;

}

public void calculate\_hamming\_distance()

{

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

{

if (puzzle\_1d[i] != i + 1 && puzzle\_1d[i] != 0)

heuristic\_value++;

}

}

public int priority()

{

return heuristic\_value + cost\_so\_far;

}

public void modify\_manhattan\_distance(int old\_index, int new\_index)

{

int v = puzzle\_1d[old\_index];

v = v - 1;

int goal\_x = v % puzzle\_dimension;

int goal\_y = v / puzzle\_dimension;

int x = new\_index % puzzle\_dimension;

int y = new\_index / puzzle\_dimension;

int manhatten\_cost = Math.Abs(x - goal\_x) + Math.Abs(y - goal\_y);

heuristic\_value -= manhatten\_cost;

v = puzzle\_1d[old\_index];

v = v - 1;

goal\_x = v % puzzle\_dimension;

goal\_y = v / puzzle\_dimension;

x = old\_index % puzzle\_dimension;

y = old\_index / puzzle\_dimension;

manhatten\_cost = Math.Abs(x - goal\_x) + Math.Abs(y - goal\_y);

heuristic\_value += manhatten\_cost;

}

public void modify\_hamming\_distance(int old\_index , int new\_index)

{

if (puzzle\_1d[old\_index] == old\_index + 1)

{

heuristic\_value--;

}

else if (puzzle\_1d[old\_index] == new\_index + 1)

{

heuristic\_value++;

}

}

}

}

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Diagnostics;

namespace N\_Puzzle\_game

{

internal static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

// To customize application configuration such as set high DPI settings or default font,

// see https://aka.ms/applicationconfiguration.

ApplicationConfiguration.Initialize();

Application.Run(new Main\_Menu());

}

public static PriorityQueue<Node, int> priorityQueue = new PriorityQueue<Node, int>(/\*new DataComparer()\*/);

public static int min\_number\_of\_moves;

public static long total\_number\_of\_moves = 0;

public static string Hamming\_or\_Manhattan;

public static List<int[]> Solving\_Moves = new List<int[]>();

public static Stopwatch stopwatch = new Stopwatch();

public static bool is\_puzzle\_solvable(int puzzle\_dimension, int[] puzzle\_1d\_array)

{

int no\_of\_inversions = 0;

int blank\_space\_pos = 0;

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

{

if (puzzle\_1d\_array[i] == 0)

{

blank\_space\_pos = i / puzzle\_dimension + 1;

continue;

}

for (int j = i + 1; j < puzzle\_dimension \* puzzle\_dimension; j++)

{

if (puzzle\_1d\_array[i] > puzzle\_1d\_array[j] && puzzle\_1d\_array[j] != 0)

{

no\_of\_inversions++;

}

}

}

//if N is even

if (puzzle\_dimension % 2 == 0)

{

if (no\_of\_inversions % 2 == 0 && (puzzle\_dimension - blank\_space\_pos) % 2 == 0 ||

no\_of\_inversions % 2 != 0 && (puzzle\_dimension - blank\_space\_pos) % 2 != 0)

{

return true;

}

}

//if N is odd

else

{

if (no\_of\_inversions % 2 == 0)

{

return true;

}

}

return false;

}

public static Node A\_Star\_Search(Node node)

{

total\_number\_of\_moves = 0;

Node tmp\_node = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far);

while (node.heuristic\_value != 0)

{

total\_number\_of\_moves++;

if (node.heuristic\_value == 0)

{

return node;

}

Node node1;

if (total\_number\_of\_moves > 30000000 && Hamming\_or\_Manhattan == "2")

{

MessageBox.Show("Puzzle is not solvable with hamming distance");

break;

}

// is up node

if (node.zero\_pos - node.puzzle\_dimension >= 0)

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos - node.puzzle\_dimension;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1);

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos - node.puzzle\_dimension];

node1.puzzle\_1d[node.zero\_pos - node.puzzle\_dimension] = 0;

node1.zero\_pos = node.zero\_pos - node.puzzle\_dimension;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority());

}

}

// is down node

if ((node.zero\_pos + node.puzzle\_dimension) < (node.puzzle\_dimension \* node.puzzle\_dimension))

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos + node.puzzle\_dimension;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1);

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos + node.puzzle\_dimension];

node1.puzzle\_1d[node.zero\_pos + node.puzzle\_dimension] = 0;

node1.zero\_pos = node.zero\_pos + node.puzzle\_dimension;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority());

}

}

// is right node

if ((node.zero\_pos % node.puzzle\_dimension) != (node.puzzle\_dimension - 1))

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos + 1;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1);

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos + 1];

node1.puzzle\_1d[node.zero\_pos + 1] = 0;

node1.zero\_pos = node.zero\_pos + 1;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority());

}

}

// is left node

if ((node.zero\_pos % node.puzzle\_dimension) != 0)

{

tmp\_node.parent = node;

tmp\_node.zero\_pos = node.zero\_pos - 1;

tmp\_node.cost\_so\_far = node.cost\_so\_far + 1;

if (tmp\_node.cost\_so\_far < 2 ||

(tmp\_node.cost\_so\_far >= 2 && tmp\_node.zero\_pos != tmp\_node.parent.parent.zero\_pos))

{

node1 = new Node(node.puzzle\_1d, node.puzzle\_dimension, node, node.cost\_so\_far + 1);

node1.puzzle\_1d[node.zero\_pos] = node1.puzzle\_1d[node.zero\_pos - 1];

node1.puzzle\_1d[node.zero\_pos - 1] = 0;

node1.zero\_pos = node.zero\_pos - 1;

node1.heuristic\_value = node.heuristic\_value;

if (Hamming\_or\_Manhattan == "1")

node1.modify\_manhattan\_distance(node.zero\_pos, node1.zero\_pos);

else if (Hamming\_or\_Manhattan == "2")

node1.modify\_hamming\_distance(node.zero\_pos, node1.zero\_pos);

priorityQueue.Enqueue(node1, node1.priority());

}

}

node = priorityQueue.Dequeue();

}

return (node);

}

public static void Solve\_puzzle(string h\_or\_m, int puzzle\_dimension, int[] puzzle\_1d\_array)

{

Hamming\_or\_Manhattan = h\_or\_m;

min\_number\_of\_moves = 0;

if (is\_puzzle\_solvable(puzzle\_dimension, puzzle\_1d\_array))

{

Node root = new Node(puzzle\_1d\_array, puzzle\_dimension, null, 0);

if (Hamming\_or\_Manhattan == "1")

root.calculate\_manhattan\_distance();

else if (Hamming\_or\_Manhattan == "2")

root.calculate\_hamming\_distance();

priorityQueue.Enqueue(root, root.priority());

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

{

if (root.puzzle\_1d[i] == 0)

{

root.zero\_pos = i;

}

}

stopwatch.Start();

Node goal = A\_Star\_Search(priorityQueue.Dequeue());

stopwatch.Stop();

MessageBox.Show("time taken to solve the puzzle = " + stopwatch.Elapsed);

stopwatch.Reset();

while (goal.parent != null)

{

Solving\_Moves.Insert(0, goal.puzzle\_1d);

goal = goal.parent;

min\_number\_of\_moves++;

}

Solving\_Moves.Insert(0, goal.puzzle\_1d);

}

else

{

MessageBox.Show("Puzzle is not Solvable");

}

}

}

/\* public class DataComparer : IComparer<int>

{

public int Compare(int a, int b)

{

if (a < b)

{

return -1;

}

if (a >= b)

{

return 1;

}

return 0;

}

}\*/

}

**4-All test cases (time and number of moves)**

