**N-Puzzle**

**Team (T116)**

**Team Members :**

|  |  |
| --- | --- |
| **20191700177** | **بافلي ماجد وديع** |
| **20191700542** | **محمد صبري يوسف عبدالظاهر** |
| **20191700478** | **مازن خالد إبراهيم عبدالشافي** |

**Description :**

The N-Puzzle is known in finite versions such as the 8-puzzle (a 3x3 board) and the 15-puzzle (a 4x4 board), and with various names like "sliding block", "tile puzzle", etc. The N-Puzzle is a board game for a single player. It consists of (N) numbered squared tiles in random order, and one blank space ("a missing tile").

The objective of the puzzle is to rearrange the tiles in order by making sliding moves that use the empty space, using the fewest moves. Moves of the puzzle are made by sliding an adjacent tile into the empty space. Only tiles that are horizontally or vertically adjacent to the blank space (not diagonally adjacent) may be moved.

The n puzzle is a classical problem for modelling algorithms involving heuristics. Commonly used heuristics for this problem include counting the number of misplaced tiles and finding the sum of the taxicab distances between each block and its position in the goal configuration.

**Program Flow:**

* Program.cs

1. void Main(string[] args) 🡺 main method enables user to choose test type to be committed (sample test / complete test / very large test) and preferred distance method (Hamming / Manhattan) then based on the user choices it calls the corresponding test function.
2. void SampleTest() 🡺 has all sample test file names and passes them to ReadFile() method to read samples from them and check whether they’re solvable or not solvable then calls solve method located in NPuzzle class.
3. void CompleteTest() 🡺 has all complete test file names and passes them to ReadFile() method to read samples from them and check whether they’re solvable or not solvable then calls solve method located in NPuzzle class.
4. void VeryLargeTest() 🡺 has all very large test file names and passes them to ReadFile() method to read samples from them and check whether they’re solvable or not solvable then calls solve method located in NPuzzle class.
5. NPuzzle ReadFile(string filename , int methodtype) 🡺reads file and checks solvability by calling isSolvable method.

* NPuzzle.cs

1. bool isSolvable(List<int> lis) 🡺 checks solvability by calculating number of inversions and checking if:
   1. inversions and dimensions are odd then its unsolvable.
   2. dimensions are even:
      1. checks position of the row which contains zero whether its even and number of inversions is even or its odd and number of inversions is odd then its not solvable
   3. otherwise its solvable
2. Node solve() 🡺 Creates parent Node then applies A\* algorithm to get the closest children by calling childFactory() method.
3. void childFactory(Node parent) 🡺 calls permissionToGo() method that checks if the direction is valid to move to or not, if its valid then it will call Swap() method to get the new state of the matrix and it will give it a unique state value by calling setState() method and checks the existence of this state if not it will generate a new child by calling genrate() method and calculate its distance by calling method() method then pushing this new node(child) to the priority queue.
4. int Swap(int[,] mat , int x , int y , Node parent, int depth) 🡺 moves zero to any of the 4 directions and calculates Manhattan distance.
5. string setState(int[,] matrix) 🡺 sets a unique state for each matrix.
6. Node generate(int x, int y, int x1, int y1, string state,Node node) 🡺 generates a child and sets its parent and attributes.
7. int method(int type, Node node, int cost = 0) 🡺 calls either manhattan() or hamming() distance method depending on entered type.
8. int manhattan(int[,] mat , int cost) 🡺 calculates Manhattans distance.
9. public int hamming(int[,] mat) 🡺 calculates Hamming distance.
10. void Print(int[,] matrix) 🡺 prints matrix.
11. void printroot(Node root) 🡺 prints the sequence of solving the puzzle.

* PriorityQueue.cs

1. void push(Node value) 🡺 Adds a node to priority queue.
2. public Node pop() 🡺 Removes last node in the priority queue and return its content.
3. void heapifyMin(int index) 🡺 Rebuilds the priority queue.
4. void Swap(int first , int second) 🡺 Swaps two nodes in priority queue.
5. int getSize() 🡺 returns priority queue’s size.

**Main Program :** (program.cs)

| Source Code | Analysis |
| --- | --- |
| static int DISTANCE\_TYPE;  static bool solve;  static void Main(string[] args)//O(Elog(V) + S^2)  {  Console.WriteLine("SampleTest [1] : \nCompleteTest [2] : \nVeryLargeTest[3] :");  int diff = int.Parse(Console.ReadLine());  Console.WriteLine("Hamming [0] : \nManhattan [1] :");  DISTANCE\_TYPE = int.Parse(Console.ReadLine());  switch (diff)  {  case 1 :  SampleTest();//O(Elog(V) + S^2)  break;  case 2:  CompleteTest();//O(Elog(V) + S^2)  break;  default:  VeryLargeTest();//O(Elog(V) + S^2)  break;  }  } | O() |
| static void SampleTest()//O(Elog(V) + S^2)  {  Console.WriteLine("Sample : ");    string[] samplefilenames = { "8 Puzzle (1)", "8 Puzzle (2)", "8 Puzzle (3)", "15 Puzzle - 1", "24 Puzzle 1", "24 Puzzle 2" };  foreach (string name in samplefilenames)  {  string solvedSample = @"Sample Test/Solvable Puzzles/" + name + @".txt";  NPuzzle nPuzzle = ReadFile(solvedSample, DISTANCE\_TYPE); //Θ(S^2)  Stopwatch watch = new Stopwatch();  watch.Start();  if (solve)  {  Node parent = nPuzzle.solve(); //O(Elog(V))  if (nPuzzle.n == 3)  nPuzzle.printroot(parent); //Θ(#of correct moves)  Console.WriteLine("steps :" + parent.depth);  watch.Stop();  Console.WriteLine($"Execution time : {watch.Elapsed.TotalSeconds} sec ");  Console.WriteLine();  }  }  string[] samplefilenamesnot = { "8 Puzzle - Case 1", "8 Puzzle(2) - Case 1", "8 Puzzle(3) - Case 1", "15 Puzzle - Case 2", "15 Puzzle - Case 3" };  foreach (string name in samplefilenamesnot)  {  string unsolvedSample = @"Sample Test/Unsolvable Puzzles/" + name + @".txt";  ReadFile(unsolvedSample);  Console.WriteLine();  }    } | O() |

| static void CompleteTest()//O(Elog(V) + S^2)  {  Console.WriteLine("Complete : ");    string[] completefilenames = { "15 Puzzle 1", "15 Puzzle 3", "15 Puzzle 4", "15 Puzzle 5" };  string[] completefilenames2 = { "50 Puzzle", "99 Puzzle - 1", "99 Puzzle - 2", "9999 Puzzle" };    foreach (string name in completefilenames)  {  string solvedComplete = @"Complete Test/Solvable puzzles/Manhattan Only/" + name + @".txt";  NPuzzle nPuzzle = ReadFile(solvedComplete, 1);//Θ(S^2)  Stopwatch watch = new Stopwatch();  watch.Start();  if (solve)  {  Node parent = nPuzzle.solve();//O(Elog(V))  if (nPuzzle.n == 3)  nPuzzle.printroot(parent);//Θ(#of correct moves)  Console.WriteLine("steps :" + parent.depth);  watch.Stop();  Console.WriteLine($"Execution time : {watch.Elapsed.TotalSeconds} sec ");  Console.WriteLine();  }  }  foreach (string name in completefilenames2)  {  string solvedComplete = @"Complete Test/Solvable puzzles/Manhattan & Hamming/" + name + @".txt";  NPuzzle nPuzzle = ReadFile(solvedComplete, DISTANCE\_TYPE);//Θ(S^2)  Stopwatch watch = new Stopwatch();  watch.Start();  if (solve)  {  Node parent = nPuzzle.solve();//O(Elog(V))  if (nPuzzle.n == 3)  nPuzzle.printroot(parent);//Θ(#of correct moves)  Console.WriteLine("steps :" + parent.depth);  watch.Stop();  Console.WriteLine($"Execution time : {watch.Elapsed.TotalSeconds} sec ");  Console.WriteLine();  }  }  string[] completefilenamesnot = { "15 Puzzle 1 - Unsolvable", "99 Puzzle - Unsolvable Case 1", "99 Puzzle - Unsolvable Case 2", "9999 Puzzle" };  foreach (string name in completefilenamesnot)  {  string unsolvedComplete = @"Complete Test/Unsolvable puzzles/" + name + @".txt";  ReadFile(unsolvedComplete);  Console.WriteLine();  }    } | O() |
| --- | --- |

| static void VeryLargeTest()//O(Elog(V) + S^2)  {  string veryLarge = @"Complete Test/V. Large test case/TEST.txt";  NPuzzle nPuzzle = ReadFile(veryLarge, 1);//Θ(S^2)  Stopwatch watch = new Stopwatch();  watch.Start();  if (solve)  {  Node parent = nPuzzle.solve();//O(Elog(V))  if (nPuzzle.n == 3)  nPuzzle.printroot(parent);//Θ(#of correct moves)  Console.WriteLine("steps :" + parent.depth);  watch.Stop();  Console.WriteLine($"Execution time : {watch.Elapsed.TotalSeconds} sec ");  Console.WriteLine();  }  } | O() |
| --- | --- |
| static NPuzzle ReadFile(string fileName, int methodType = 1) //Θ(S^2)  {  List<int> lis = new List<int>();  string[] s = File.ReadAllLines(fileName);//Θ(n^2)  int n = int.Parse(s[0]);  NPuzzle nPuzzle = new NPuzzle(n, methodType);  int c = 0;  int i = 0;  if (s[1] == "") i = 2;  else i = 1;  for (; i < s.Length; i++) //Θ(n^2)  {  string[] k = s[i].Split(' ');  for (int j = 0; j < k.Length; j++)  {  if (!k[j].Equals(""))  {  int result = 0;  int.TryParse(k[j], out result);  lis.Add(result);  nPuzzle.matrix[c, j] = result;  if (nPuzzle.matrix[c, j] == 0)  {  nPuzzle.x0 = c;  nPuzzle.y0 = j;  }  }  }  c++;  }  solve = nPuzzle.isSolvable(lis); //Θ(S^2)  Console.WriteLine( solve ? "Solvable" : "not Solvable");  return nPuzzle;  } | Θ() |

**Puzzle solver :** (NPuzzle.cs)

| Source Code | Analysis |
| --- | --- |
| public bool isSolvable(List<int> lis) //Θ(S^2)  {  int inversion = 0;  for (int x = 0; x < n \* n - 1; x++) //Θ(S^2)  {  for (int y = x + 1; y < n \* n; y++)  {  if (lis[y] != 0 && lis[x] != 0 && (lis[x] > lis[y]))  {  inversion++;  }  }  }  if (n % 2 != 0)  {  if (inversion % 2 != 0) return false; //Θ(1)  }  else  {  int index = 0;  for (int i = n - 1; i >= 0; i--) //Θ(n^2)  for (int j = n - 1; j >= 0; j--)  if (matrix[i, j] == 0)  index = n - i;  if ((index % 2 == 0 && inversion % 2 == 0) || (index % 2 != 0 && inversion % 2 != 0)) return false;  }  return true;  } | Θ() |
| public Node solve() //O(Elog(V) + n^2)  {  Node parent = new(0,"",null,n);  parent.mat = matrix;  parent.state = setState(parent.mat); //Θ(n^2)  parent.x0 = x0;  parent.y0 = y0;  parent.hValue = method(methodType, parent); //O(n^2)  set.Add(parent.state);  priorityQueue.push(parent); //O(log(V))  while (priorityQueue.getSize() != 0) //Θ(1)  {  if (parent.hValue == parent.depth)  break;  parent = priorityQueue.pop(); //O(log(V))  childFactory(parent);  }  return parent;  } | O() |

|  |  |
| --- | --- |
| void childFactory(Node parent) //O(log(V))  {  for (int i = 0; i < 4; i++)  {  if (permissionToGo(parent.x0 + aroundX[i], parent.y0 + aroundY[i])) //Θ(1)  {  int[,] temp = parent.mat.Clone() as int[,]; //Θ(n^2)  int cost = Swap(temp,parent.x0 + aroundX[i], parent.y0 + aroundY[i], parent , parent.depth); //Θ(1)  string s = setState(temp); //Θ(n^2)  if (!set.Contains(s))  {  Node newNode = generate(parent.x0, parent.y0, parent.x0 + aroundX[i], parent.y0 + aroundY[i], s, parent); //Θ(n^2)  set.Add(newNode.state); //Θ(1)  newNode.hValue = method(methodType, newNode, cost);//O(n^2)  priorityQueue.push(newNode);//O(log(V))  }  }  }  } | O() |
| bool permissionToGo(int x , int y) //Θ(1)  {  if (x < 0 || y < 0 || x > n - 1 || y > n - 1) return false;  return true;  } | Θ(1) |
| int Swap(int[,] mat , int x , int y , Node parent, int depth) // Θ(1)  {  mat[parent.x0, parent.y0] = mat[x, y];  int rightX = (mat[x, y] - 1) / n;  int rightY = (mat[x, y] - 1) % n;  int miss = parent.hValue - Math.Abs(rightX - x) - Math.Abs(rightY - y);  int newX = (mat[parent.x0, parent.y0] - 1) / n;  int newY = (mat[parent.x0, parent.y0] - 1) % n;  int cost = miss + (Math.Abs(newX - parent.x0) + Math.Abs(newY - parent.y0)) - depth;  mat[x, y] = 0;  return cost;  } | Θ(1) |
| public string setState(int[,] matrix) //Θ(n^2)  {  string state = "";  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  {  state += matrix[i, j];  }  }  return state;  } | Θ() |

|  |  |
| --- | --- |
| Node generate(int x, int y, int x1, int y1, string state,Node node) //Θ(n^2)  {  Node newNode = new(node.depth + 1, state, node, n);  newNode.mat = node.mat.Clone() as int[,]; //Θ(n^2)  newNode.mat[x, y] = newNode.mat[x1, y1];  newNode.mat[x1, y1] = 0;  newNode.x0 = x1;  newNode.y0 = y1;  return newNode;  } | Θ() |
| int method(int type, Node node, int cost = 0) //Θ(n^2)  {  int value = 0;  if(type == 0){  value = hamming(node.mat) + node.depth; //Θ(n^2)  }  else{  value = manhattan(node.mat , cost) + node.depth; //O(n^2)  }  return value;  } | Θ() |
| public int hamming(int[,] mat) //Θ(n^2)  {  int misPlaced = 0;  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  {  if (mat[i, j] != 0 && mat[i, j] != (1 + j + (i \* n)))  misPlaced++;  }  }  return misPlaced;  } | Θ() |
| int manhattan(int[,] mat , int cost) //O(n^2)  {  int miss = 0;  if (first) {  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  {  if (mat[i, j] != 0)  {  int x = (mat[i, j] - 1) / n;  int y = (mat[i, j] - 1) % n;  miss += Math.Abs(x - i) + Math.Abs(y - j);  }  }  first = false;  }  }  else  {  miss = cost;  }  return miss;  } | O() |

|  |  |
| --- | --- |
| void printroot(Node root) //Θ(#of correct moves))  {  if (root == null)  {  return;  }  printroot(root.parent);  Print(root.mat); //Θ(n^2)  } | Θ() |
| void Print(int[,] matrix) //Θ(n^2)  {  for (int i = 0; i < n; i++)  {  for (int j = 0; j < n; j++)  {  Console.Write(matrix[i, j] + " ");  }  Console.WriteLine();  }  Console.WriteLine();  } | Θ() |
| public NPuzzle(int n, int methodType)  {  this.n = n;  this.methodType = methodType;  matrix = new int[n, n];  } | Θ(1) |

**Priority Queue :**

|  |  |
| --- | --- |
| Source Code | Analysis |
| public void push(Node value) // Olog(V)  {  items.Add(value);  size++;  int index = size - 1 ;  while (index >= 0 && items[(index - 1) / 2].hValue > items[index].hValue) //log(V)  {  Swap(index, (index - 1) / 2);  index = (index - 1) / 2;  }  } | O() |
| public Node pop() //log(V)  {  Node value = items[0];  items[0] = items[size-1];  items.RemoveAt(size-1); //O(1)  size--;  heapifyMin(0); // log(V)  return value;  } | O() |
| private void heapifyMin(int index) // log(V)  {  int left = index \* 2 + 1;  int right = index \* 2 + 2;  int highest = index;  if (left < size && items[highest].hValue > items[left].hValue)  highest = left;  if (right < size && items[highest].hValue > items[right].hValue)  highest = right;  if (highest != index)  {  Swap(highest, index);  heapifyMin(highest);  }  } | O() |
| void Swap(int first , int second) //Θ(1)  {  Node temp = items[first];  items[first] = items[second];  items[second] = temp;  } | Θ(1) |
| public int getSize()//Θ(1)  {  return size ;  } | Θ(1) |
| int size = 0;  List<Node> items = new List<Node>();  public PriorityQueue()  {  items = new List<Node>();  } | Θ(1) |

**Node :** (Node.cs)

|  |
| --- |
| public class Node  {  public int[,] mat;  public int hValue;  public int x0 = 0;  public int y0 = 0;  public int depth;  public string state;  public int n;  public Node parent;  public Node(int depth, string state, Node parent, int n) //Θ(1)  {  this.depth = depth;  this.state = state;  this.parent = parent;  this.n = n;  }  } |

Comparison :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File Name | Manhattan | | Hamming | |
| Comp. | Execution time | Min. number of moves | Execution time | Min. number of moves |
| 15 Puzzle 1  (Manhattan only) | 2 seconds | 46 | - | - |
| 15 Puzzle 3  (Manhattan only) | 1 second | 38 | - | - |
| 15 Puzzle 4  (Manhattan only) | 0.6 second | 44 | - | - |
| 15 Puzzle 5  (Manhattan only) | 13 seconds | 45 | - | - |
| 50 Puzzle  (Manhattan & Hamming) | 0.8 second | 18 | 10 seconds | 18 |
| 99 Puzzle – 1  (Manhattan & Hamming) | 0.0006 second | 18 | 0.0005 seconds | 18 |
| 99 Puzzle – 2  (Manhattan & Hamming) | 0.0012 second | 38 | 0.005 seconds | 38 |
| 9999 Puzzle  (Manhattan & Hamming) | 0.4 second | 4 | 0.4 seconds | 4 |
| TEST.txt  (V. Large test case) | 14 second | 56 | - | - |