**N-Puzzle**

**Team 54:**

Members:

Mazen Mohamed Bakr - 20191700479

Waleed Mohamed Mohamed - 20191700747

Mohamed Ibrahim Mohamed - 20191700490

**Description:**

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.

The goal of the game is to arrange the numbers ascendingly and a blank tile in the end position.

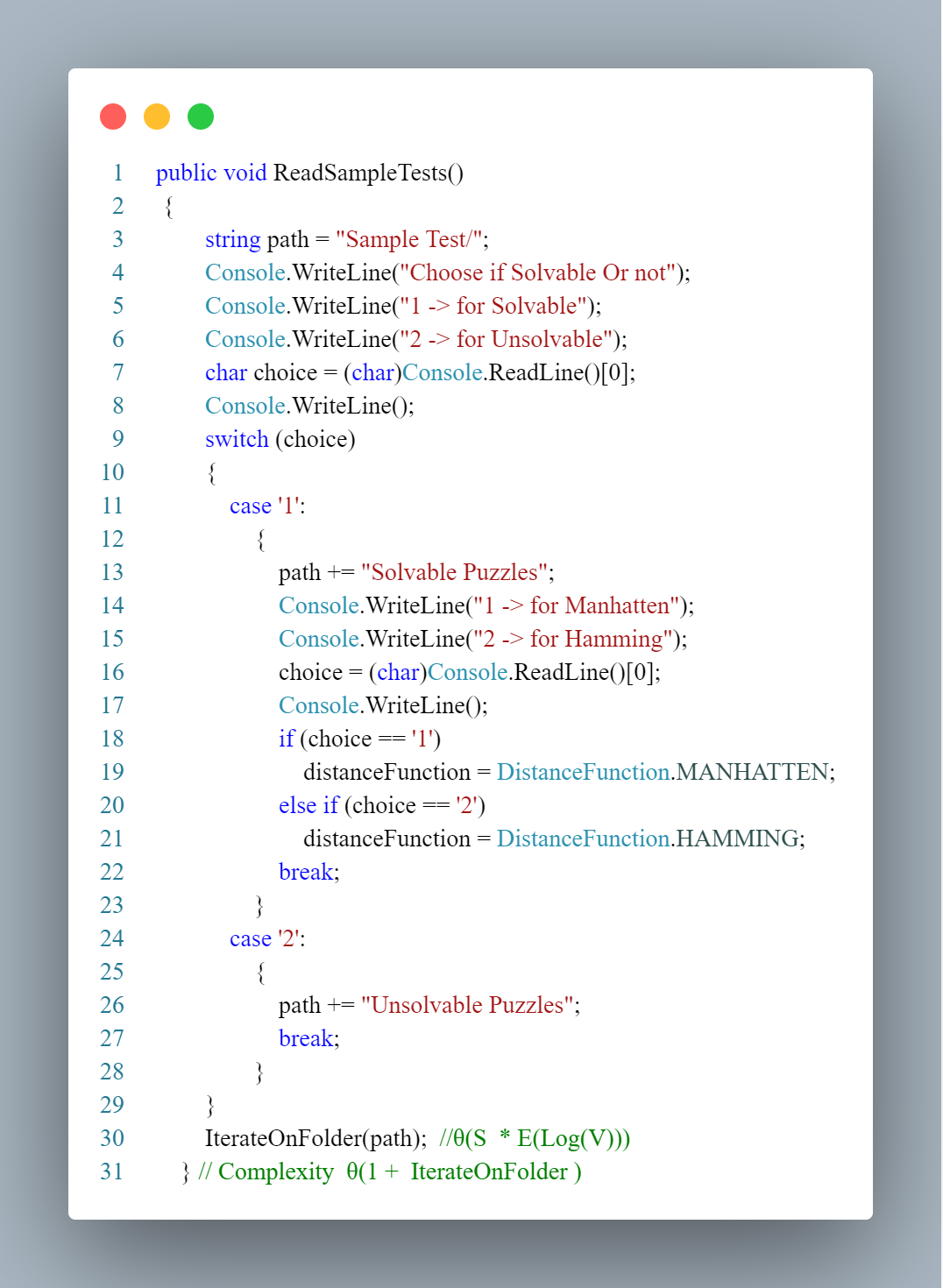
This could be done using A\* Algorithm to find the shortest path. There’re multiple ways to solve the puzzle, one way is using priority queue to sort the possible paths according to a heuristic function   
- Hamming or Manhattan etc. -

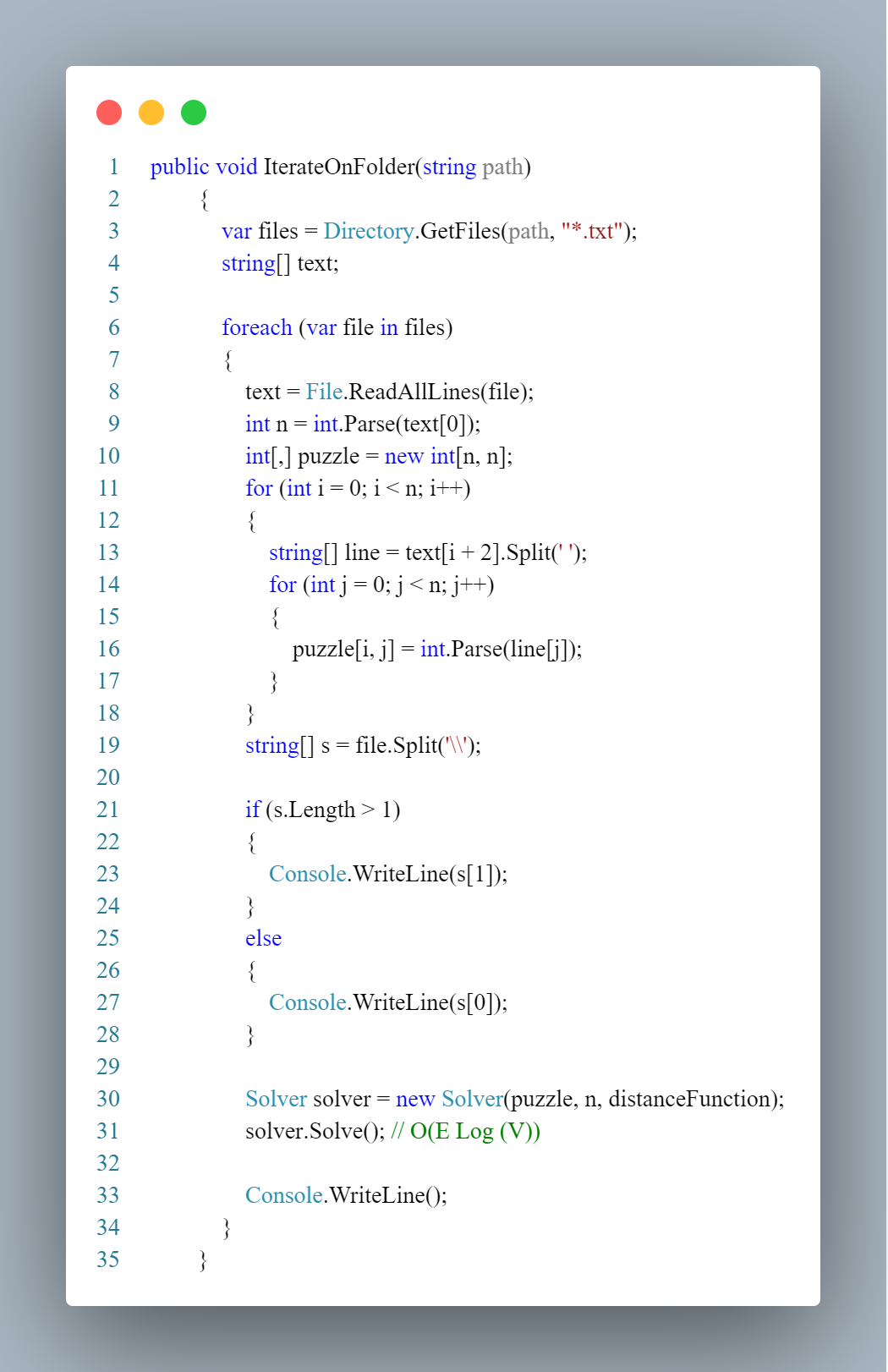
So, first thing first, we read the puzzle as follows ... We ask the user to choose the test cases to be solved ...



**Complexity:** The time taken for solve 1 puzzle \* time taken for read 1 puzzle

**Getting right folder ex.:**

****

**Reading each test case in folder and solve it:**

**Complexity:** The time taken for solve 1 puzzle \* time taken for read 1 puzzle

**Solving Algorithm:**

After reading the puzzle we solve it using the A\* algorithm as follows:

We initialize our Solver to solve the puzzle.Graphical user interface, text, application

Description automatically generated

Global Static Helper Fields

We Initialize the Open and Closed set to help us solving.

Graphical user interface, text, application

Description automatically generated

Open set: Priority queue that used for sorting the nodes using their distances.

**Priority Queue:**

Priority Queue is a Data Stucture used for arranging the nodes according to their heuristic cost.

**Text

Description automatically generated with low confidence** -Priority Queue insertion.

**Complexity: O(Log(V)).**

-Priority Queue Deletion.

**Graphical user interface, text, application

Description automatically generated**

**Complexity: O(Log(V)).**

-ExtraPriority Queue Funtions.

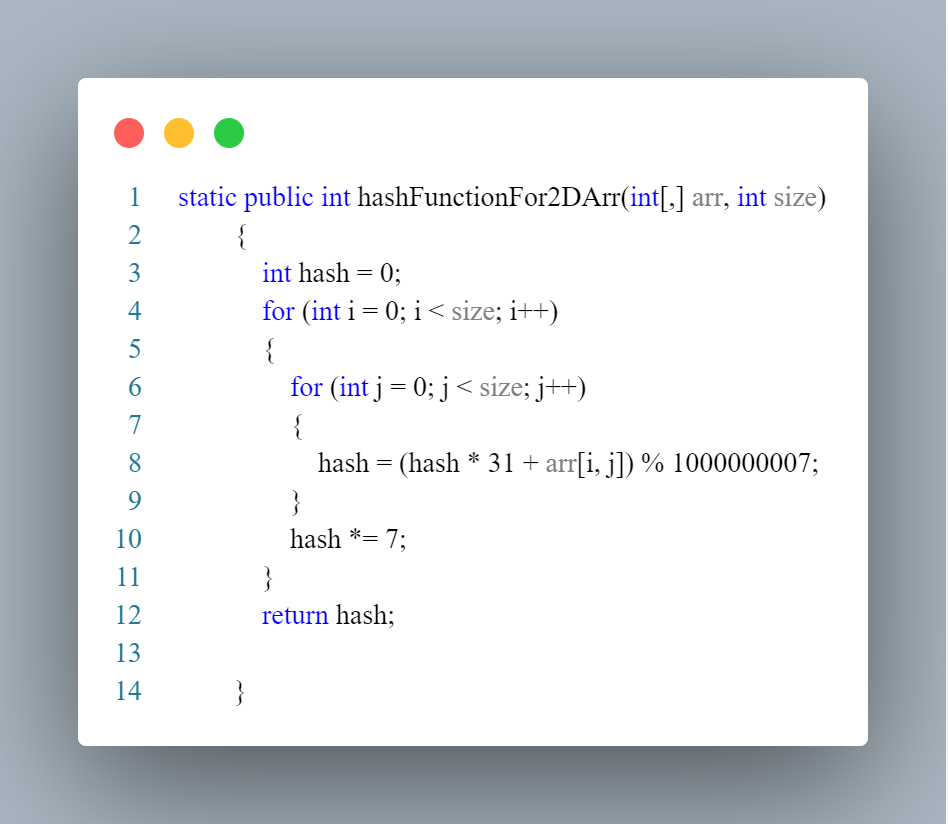
**Graphical user interface

Description automatically generated with low confidence**

**Complexity: (θ(1)) Each.**

Closed set: A Hash set that’s used for saving a unique hash number for a the node so that we don’t push a puzzle that we already generated.

The way of hashing, we just loop over the puzzle and get a unique number that can’t be repeated for more one puzzle.

the hash function: 

Now, once we are ready, we start solving …

we start a stopwatch to calculate the time

Graphical user interface, text, application

Description automatically generated

We Initialize The initial puzzle using the first constructor of the node class and push it in the PQ.

**Node:**The node is the core thing in the project, each node contains 2D array -which is the puzzle-, size, zero position, pointer to the parent node, cost, level or depth, the heuristic function Type, and a hash code to give an id for the puzzle so we can define the unique nodes.

There’s two Constructors.

First Constructor is to define the Initial State

**Graphical user interface, text, application, email

Description automatically generated**

InitBoard is used for cloning the puzzle and define the zero index.

The Second Constructor is used for Initializing each child.

Graphical user interface, text, application

Description automatically generated

The difference between the first and second constructor is the first one initialize the parent node and save every properity so we can use it later in the children constructor.

And calculate the distances for the Initial board as follows:

Graphical user interface, text, application, email

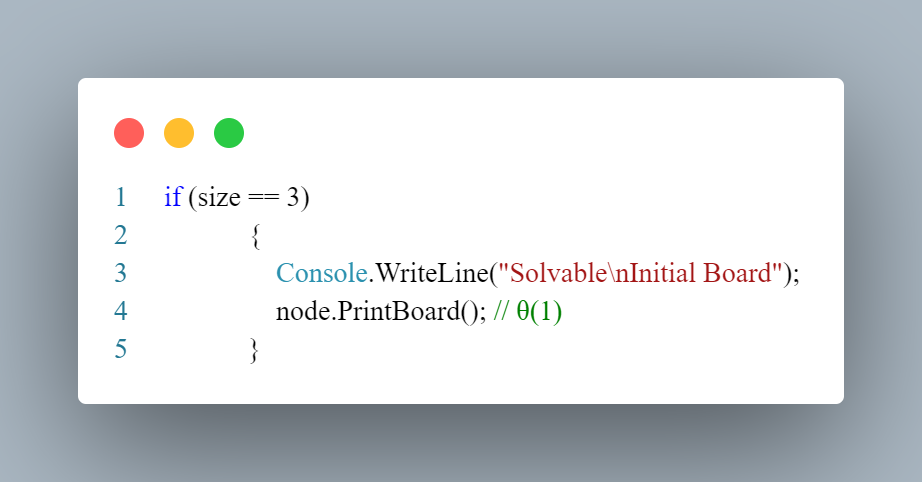
Description automatically generated

**Compexity for hamming or Manhattan:  θ(S)**

Then we check the solvability of the puzzle using the isSolvable() method

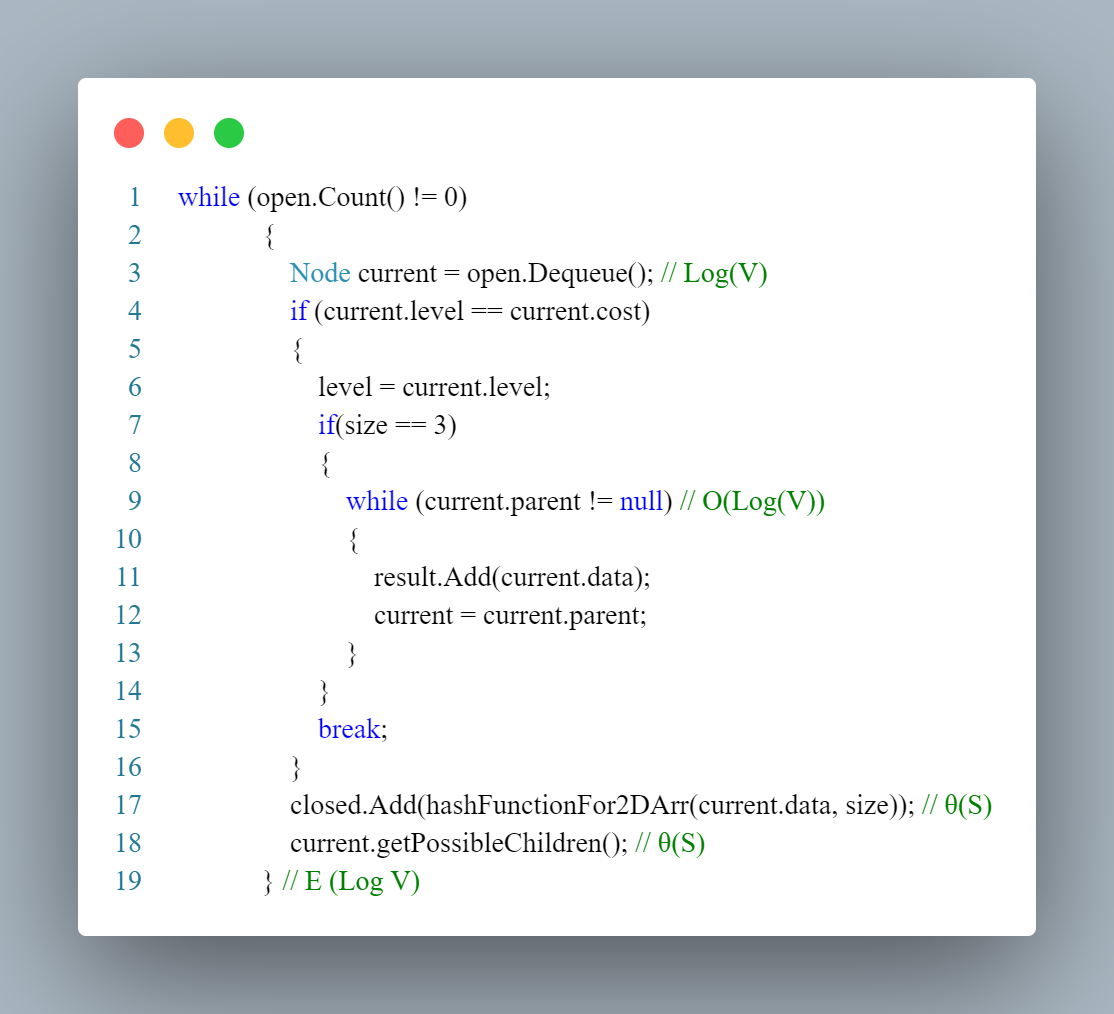
Graphical user interface, text

Description automatically generated with medium confidence

If it wasn’t solvable, we return. Else we Print the Initial board if it’s 3\*3 board and start solving.

While the PQ isn’t empty …

We Deque the minimum cost node from the PQ to start check wheather it was the goal state or not,  
if it’s goal we story the solution branch in result list, else we add the function to the closed set then generate the children till we find the solution.



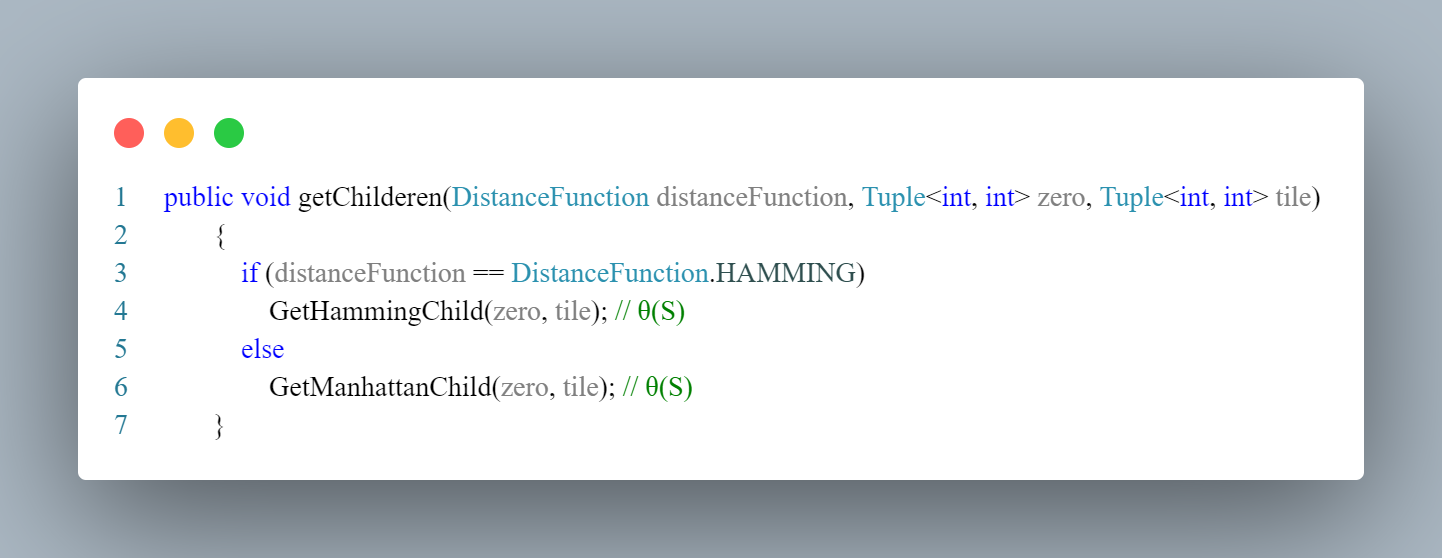
**Complexity:  θ(E Log (V)).**

**Generation of the children:**

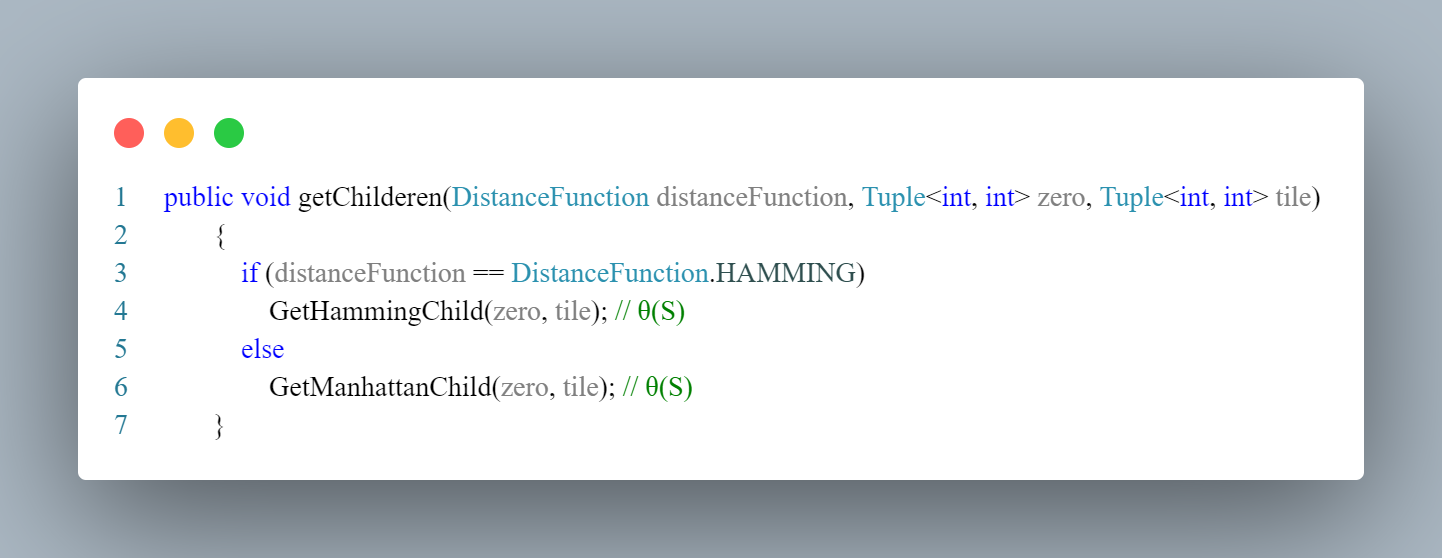
We move the zero in possible directions of the current node …

The distance function is calculated at the first time only with complexity θ(N²) then it’s calculated while generating the child as θ(S).



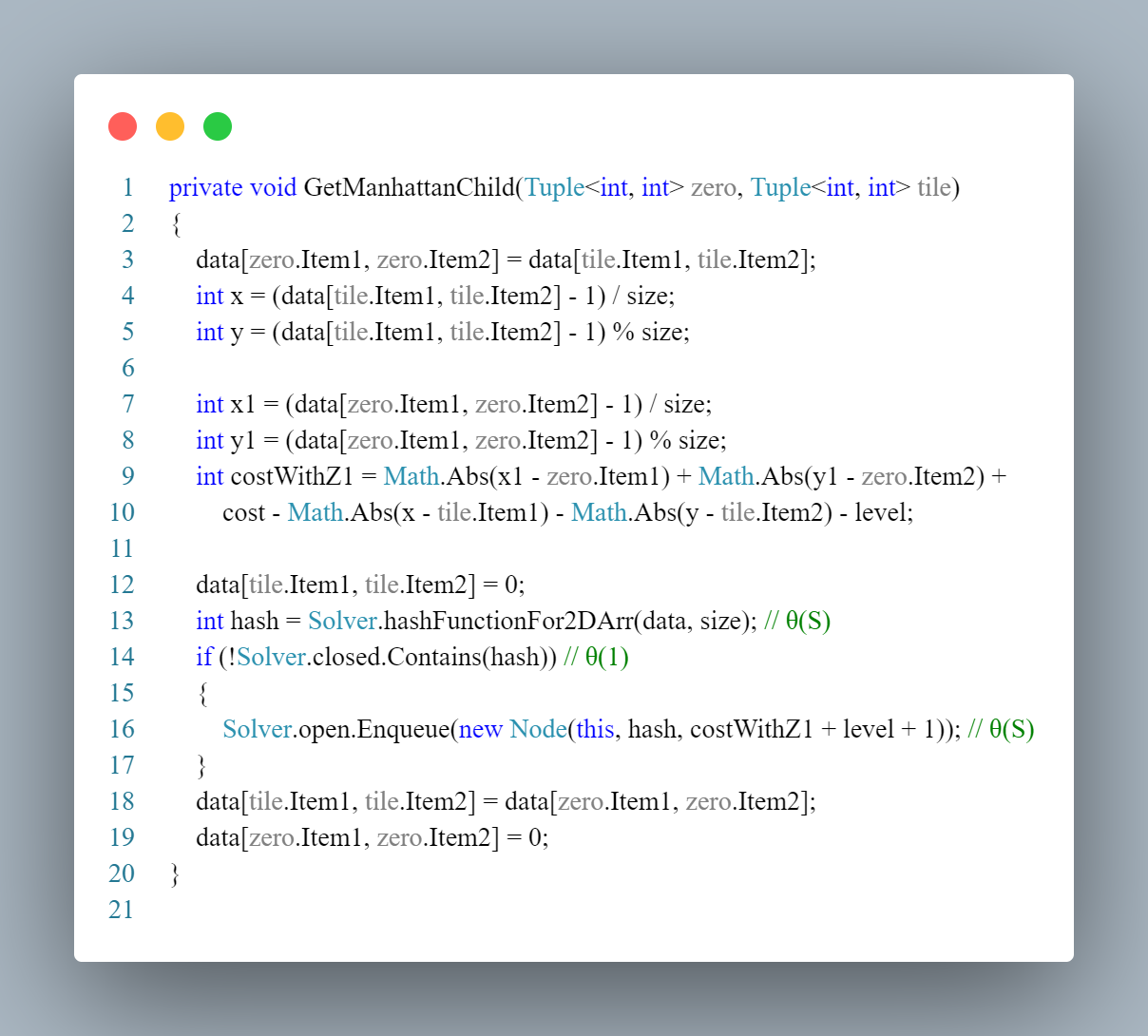


After calcualting the distance for the first time, we take O(1) to calculate it using the parent distance.



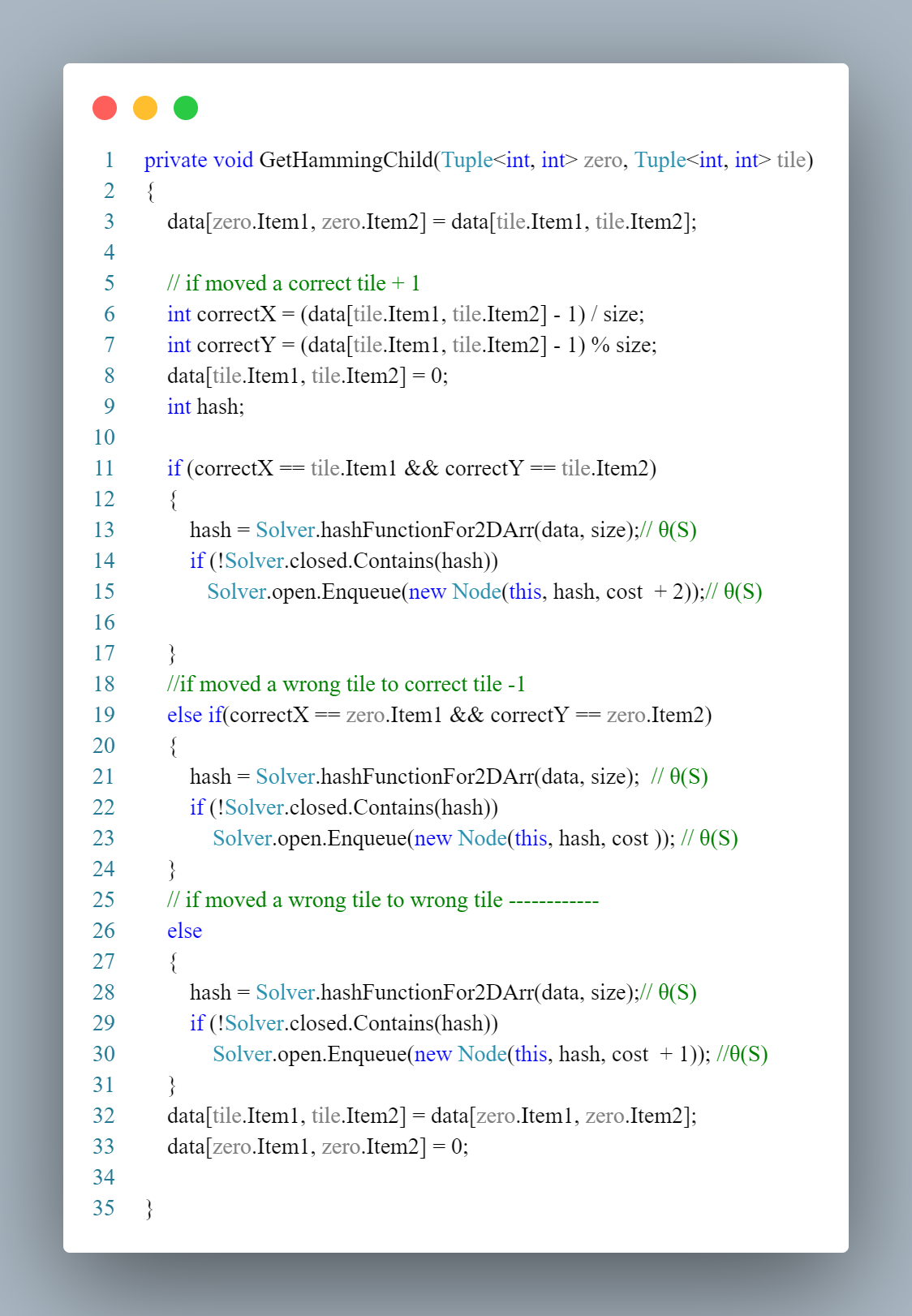
After we decide which heuristic function, we calculate the distances as follows:

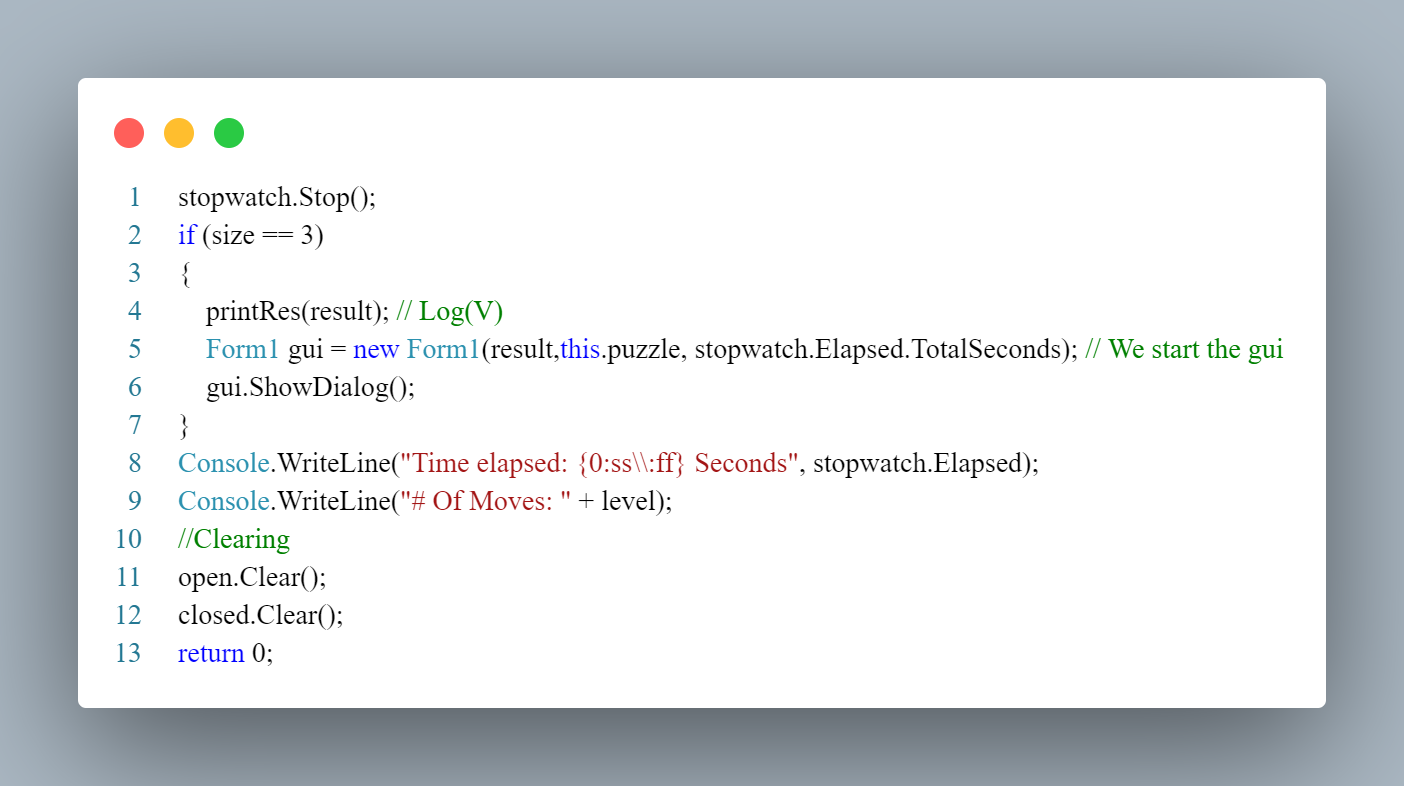
We swap the zero position with the selected tile and we use the parent cost to get the child cost but before generate and pushing the child to the PQ we check the hash for it, if it was unique we push it else we ignore it.

For Manhattan child: 

**Complexity:** θ(S)

For Hamming Child:

  
**Complexity:** θ(S)

Finaly, we stop the clock, starting the GUI, printing the time, minimum number of moves used for solving the puzzle and the solution sequence if it was 3\*3.

**Conclusion:**

Manhattan is more accurate in terms of getting the next move but slower in calculating the costs, so in large puzzles Hamming is better, but if we can get the costs in θ(1) then Manhattan is much better, and here’s a Comparison between them in brief as we get the costs in θ(1) in Manhattan, In Time:

-In Manhattan & Hamming TestCases:

|  |  |  |
| --- | --- | --- |
| Tests | Manhattan | Hamming |
| 50 Puzzle | 04 MS | 37 MS |
| 99 Puzzle - 1 | 00 MS | 00 MS |
| 99 Puzzle - 2 | 00 MS | 00 MS |
| 9999 Puzzle | 19 MS | 19 MS |
| V. Large TEST | 17:59 S | ----- |

-In Manhattan Only TestCases:

|  |  |  |
| --- | --- | --- |
| Tests | Manhattan | Hamming |
| 15 Puzzle 1 | 2:80 S | ----- |
| 15 Puzzle 3 | 69 MS | ----- |
| 15 Puzzle 4 | 83 MS | ----- |
| 15 Puzzle 5 | 10:63 S | ----- |

In terms of the moves, Manhattan gets the solution faster than the Hamming because it calculate the costs more accurate than the Hamming.

In terms of the MINIMUM number of moves, both will eventually get the solution in the same number, but it could take more time using Hamming.