**8-Puzzle Problem**

Initially, the puzzle will have random numbers with 1 single empty tile.

However, there could be unsolvable puzzles when the numbers are shuffled, also the code must make sure each randomized initial state is solvable. This is crucial to have the program work seamlessly with no infinite loops.

Since this is a 3x3 or NxN table, we will work with a 2D matrix data type.

|  |  |  |
| --- | --- | --- |
| {0,0} | {0,1} | {0.2} |
| {1,0} | {1,1} | {1,2} |
| {2,0} | {2,1} | {2,2} |

1. Node.java – represent a state/node in the 8-puzzle problem.

This class is shared among BFS, DFS, and GUI.java files. Node.java initializes all the variables needed for this project.

* Int [][] board 🡪 refers to 2D array for the puzzle state.
* Parent 🡪 A custom Node variable. Refers to first or parent node. This will be used to reconstruct the solution path for the GUI.
* Row and column instance 🡪 represents the “0” or the empty tile.
* Initialize and define a private goal 2D array 🡪 private so the state never changes as all solutions will be the same as the goal state.
* Initialize and define a private movements array 🡪 up, down, left, right movements of the tile will refer to all.

Based on the initialized movements the queue will store in order of: up,down,left,right. Once a move is explored, that new board configuration will generate new neighbors which will be added to the back of the queue (FIFO).

**Public Node** is a constructor for the Node class as it initializes the board, the parent and calls **findZero()** method (below) to locate the empty tile in the initialized board. This is important as the code will be working with a randomly generated puzzle every time whether it has a solution or not. Because of this wildcard factor, the first thing that is needed to consider is finding the zero in the initialized puzzle.

**findZero** method looks for the “0” in a double for-look for each row and column and searches for an {row,column} coordinate that equals to 0. At this current state the numbers are represented as integers, however the puzzle, or the array will have to be converted into a string to print each step of the solution for the GUI.

**generateRandomPuzzle()** is to generate a random puzzle, which uses an ArrayList of numbers 0-8, and since this is done with Java, it is essential to import collections to shuffle and randomize the numbers for the random puzzle. These random numbers later get assigned to a 3x3 board in a double-for loop. As it will be returning the board, it will be in a public non-void method. (I \* 3 +j) is used to convert the 2D board indices (I,j) into the corresponding linear index for the 1D numbered list. Later on, Deeptostring is used to print string version of the deep copy of the multidimensional array.

**isGoal()** method simply checks if the current state of the puzzle matches the goal state. deepEquals is a function in the Java library that can the goal state was initialized earlier.

**getNeighbours()** method generates all possible next states by moving the empty tile in all possible directions. The initialized Node object will be used to store in the states that the in an ArrayList and use the initialized movements list to check if the empty tile can be moved in any direction. For each movement, calculate a new row and column. If a direction is possible, take a deep copy of the current board and swap the board to the new board with the new position for the empty tile. This creates a new node or a new board configuration which gets added to the neighbors list. Finally, it returns the list of all possible neighbor nodes.

Later it writes a **getBoard()** simple getter method for the last configuration for the current board.

Another method I had to add later one is a method that checks for duplicate puzzles in the node array. This was made possible by **Java’s Equals()** method from **Object()** which passes an object and compares this Node to passes object node using deepEquals. This essentially checks if two Node objects have the same board configuration or not. This method handles duplicates states by avoiding redundant operations and adding duplicate states to the collection. Since this is a custom object, this quality should be based on logical equivalence: hence by overriding this method, it will be needed to specify the two objects that should be considered “equal”.

And finally, in **Java’s** **hashCode()** method, return our stored node object in a hash-set to provide a numeric value that corresponds to the object. This means that if two objects are considered equal() then they have the same hashCode(), but same hashCode() does not necessarily have to be equal. The actual equality is determined by equals(). This method needs to be overridden to keep the equals() and hashcode() contract else the method can generate a hash code based on the object’s memory address.

1. Breadth-First Search

BFS is a uniformed search algorithm that will guarantee to find the shortest path to the goal state by exploring all states level by level.

The algorithm itself should work in this flow:

1. Start with initial node/puzzle
2. Explore all possible next moves/neighboring nodes and add them to the queue
3. Add initial node/puzzle to a queue that holds visited boards to store and print its path
4. Dequeue the first node/puzzle and explore its neighbors
5. Reiterate until goal configuration is found

One big disadvantage to this is that it uses too much memory as the array will be growing bigger in every level.

First initialize the data structures that are needed: a queue to store puzzles to be explored and a set to keep track of visited puzzles. A Hashset was implemented instead of array list for a faster computation in constant-time O(1) instead of O(N) for linear searches.

Later add the initial node or given puzzle to the queue and the hashset as visited.

The logic will work in this flow:

As long as the puzzles explored is not empty, retrieve the current node and pop it out of the queue and if the current node is the same as the goal, print the path to the solution via **reconstructPath**() later defined in this java file. Moreover, poll is used instead of pop to retrieve and remove the head of the queue every time.

For each node that **getNeighbours**() generate for the current, if they are not visited, add them to the queue and mark as visited, else return null. Due to its memory restraint, it was more plausible to use offer in a capacity restricted queue instead of add. If there is no solution, or neighbor, return null.

**reconstructPath()** is designed to show the paths the algorithms took in the process. For BFS, this will be a small number as it only prints the shortest path possible, for DFS however the steps to the solution will be much larger since the algorithm will stop and print once it find the goal, rather than finishing all possible paths and suggesting the shortest one.

This method initializes a list for the solution path, and starting from parent node/initial puzzle, it adds each node to the beginning of the list in a First In First Out order, This is ensured by the hard coded addition of the initial random puzzle to the path, as the method continues it will be adding the next possible in up,down,left,right combination as initialized in the Node class. The method will continue to store each node’s parent as states until it reaches the initial node where the parent is null. This means that BFS explores possible movements in the order they were added. Finally, a method to **printBoard** which takes in the 2D array and prints the array to string for each row.

1. Depth-Frist Search

Similar to BFS, is a unified search algorithm, however, for depth-search algorithm a stack was created as it will ensure to work with the Last-In First-Out order, meaning most recently added node will be explored. HashSet is used again to keep track of visited board combinations. Once again, start with creating the class and create the method which takes in the initial node/puzzle. The algorithm starts with pushing the first puzzle into the stack.

As long as the stack is not empty, take out the current node from the stack and assign it to current node to be explored. If the current node matches with the goal configuration, generate a path to the solution and return it with **reconstructPath()**.

If the visited nodes set does not contain the current node, add it to the set. For each neighbors returned from **getNeighbours()** for the current node, if the neighbor is not included in the in visitedNodes set, it gets added to the stack. Since the stack follows a LIFO structure it will give DFS its depth-first search ability. However, if there is no solution to the puzzle it will return none.

The **reconstructPath()** follows the same logic as the one in BFS which constructs a solution path from initial to the goal.

1. GUI

I have decided to work with Java to build a seamless GUI and worked with the JavaFX library as I had previous experience building calculators before. Since this is a Java project with JavaFX integration the flow of the project created 4 java files:

Node.JS for initializing the goal, movements, generate a random puzzle and keep deepcopies of the paths to print later. These methods will be used for both algorithms hence are in their own file to be more concise.

BFS and DFS files are separate because the goal of the GUI is to have the user interact and pick their desired algorithm to solve the problem, the GUI file includes a drop-down menu for this choice. Moreover, the GUI is also designed show the path of the algorithm despite the large number of steps to the solution it may take.

One design flaw I realized before making changes to it is that I wanted to print the solution as well however the solution will always be equal to the set goal so constructing the path was more important.

JavaFX requires importing essential libraries such as:

* application.application (which we override later on to create a main args where we can run the GUI)
* project.scene.layouts for Hbox(Horizontal) and Vbox(Vertical),
* Gridpane to create a grid of rows and columns, and setting the stage.

The class will extend Application to declare its JavaFX libraries.

Once again, create a currentNode that holds the current configuration of the puzzle, and a list of nodes for the solution path. For the application interface a grid is initialized and will arrange buttons in a 3x3 layout(defined later), and a button that will allow the user to see the next path to the solution.

A start method (which will be overridden by the main) includes the title of the application and sets the vertical layout that contains the UI elements and the spacing in between them. puzzleGrid is set as the newGridpane with added vertical and horizontal gaps for visual appeal and readability.

The menu is made up of two buttons: New Puzzle, and Solve button which triggers Next Step button: it is disabled until a solution is found.

The menu also includes comboBox which functions as a pull down menu designed to pick the algorithm choice and sets default value to BFS.

Later the initialized variables are assigned to the horizontal box named controlBox.

Initialize the random puzzle: Since this is the parent node, the status is at new puzzle so a new status label is created. The mainlayout, root, we add the created 3x3 grid, the controlBox(menu), and the statusLabel.

On the mainlayout, set the actions of the buttons to create new puzzle, solve the puzzle in BFS/DFS, and if and once solved, show each step.

Assign mainstage/root to the scene and run a new random puzzle with **generateNewPuzzle()** which is superimposed from Node class. Initialized puzzle, currentState, gets assigned a new node generated by generateRandomPuzzle() method which does not take in a parent node.

**Solve()** method solves the puzzle problem. It first checks if the puzzle is generated: If the currentstate is not assigned it will ask to generate a puzzle to run the generateNewPuzzle() method.

To chose between BFS and DFS a ternary operator is used: if useBFS is true it will use BFS.SolveInBFS(currentState) from the BFS.java file, else if its false it will use DFS.SolveInDFS(currentState).

If the steps to the solution is not 0, display the number of moves it took in a status label.

Since the 8 puzzle problem will have combinations with no solutions, there should be a status label for “no solution found.”

**showNextStep()** once again checks if a solution exists and if it does, update the puzzle display for every currentStep of the solution.

As long as the current step is smaller than the number of steps to the solution, then it will print a status label and set the Next Step button disabled else it will print the number of the current step out of the total steps to the solution.

Since the GUI has to be constantly updating to print the next step **updatePuzzleDisplay**() is used. It first clears the board, and displays the next state in double for loops for 3 rows and 3 columns and assign the current state of a coordinate of the board to an integer value. Later a new button tile is created which shows the value of the tile number unless it is the empty tile which is represented as “ “ and the buttons, equally sized, get added to the puzzleGrid.

And finally, a main args method to run the entire project.

**Notes:**

Breath First Search tends to use more memory than DFS due to its nature of checking all possible nodes level-by-level, which means it will keep all the current nodes in that current memory space. Since in a question like 8-puzzle problem the queue will grow exponentially at each level.

Depth First Search, however, goes through all possible paths until it finds the ideal one which is why we see a big difference in steps to solve between the two algorithms. DFS only visits the nodes by depth meaning it explores paths all the way to the deepest level before backtracking. This means that the list of nodes DFS stores is less than BFS which makes it more memory efficient because it holds less at the current level in memory.

BFS time complexity is

O(average number of neighbors of a node shortest path)

While for DFS it is:

O(average number of neighbors of a node maximum depth)

Another thing to consider is that BFS’s result of the steps to solution does not reflect the number of nodes the algorithm had to visit. BFS only reports the shortest path to desired goal even though it has traveled more nodes due to its level-by-level or side-by-side searching manner while DFS will search in a up-and-down manner rather than line-by-line which results in even more nodes visited but the reported number of steps to solution reflects the actual number of nodes DFS algorithm had to visit and is the path it found that is most likely not optional.