

Intro to Artificial Intelligence

Lab1 : 8-Puzzle game

TEAM MEMBERS

Name	ID
Nayra Ibrahim Ahmed Mohamed	21011504
Ibrahem Mohamed El-Sayed	21010023
Mohamed Mohamed Mohamed Abd-ELmoneim	21011213

PROBLEM STATEMENT

You will need to implement the 8 puzzle search problem using the four search algorithms:

- BFS
- DFS
- Iterative DFS
- A*

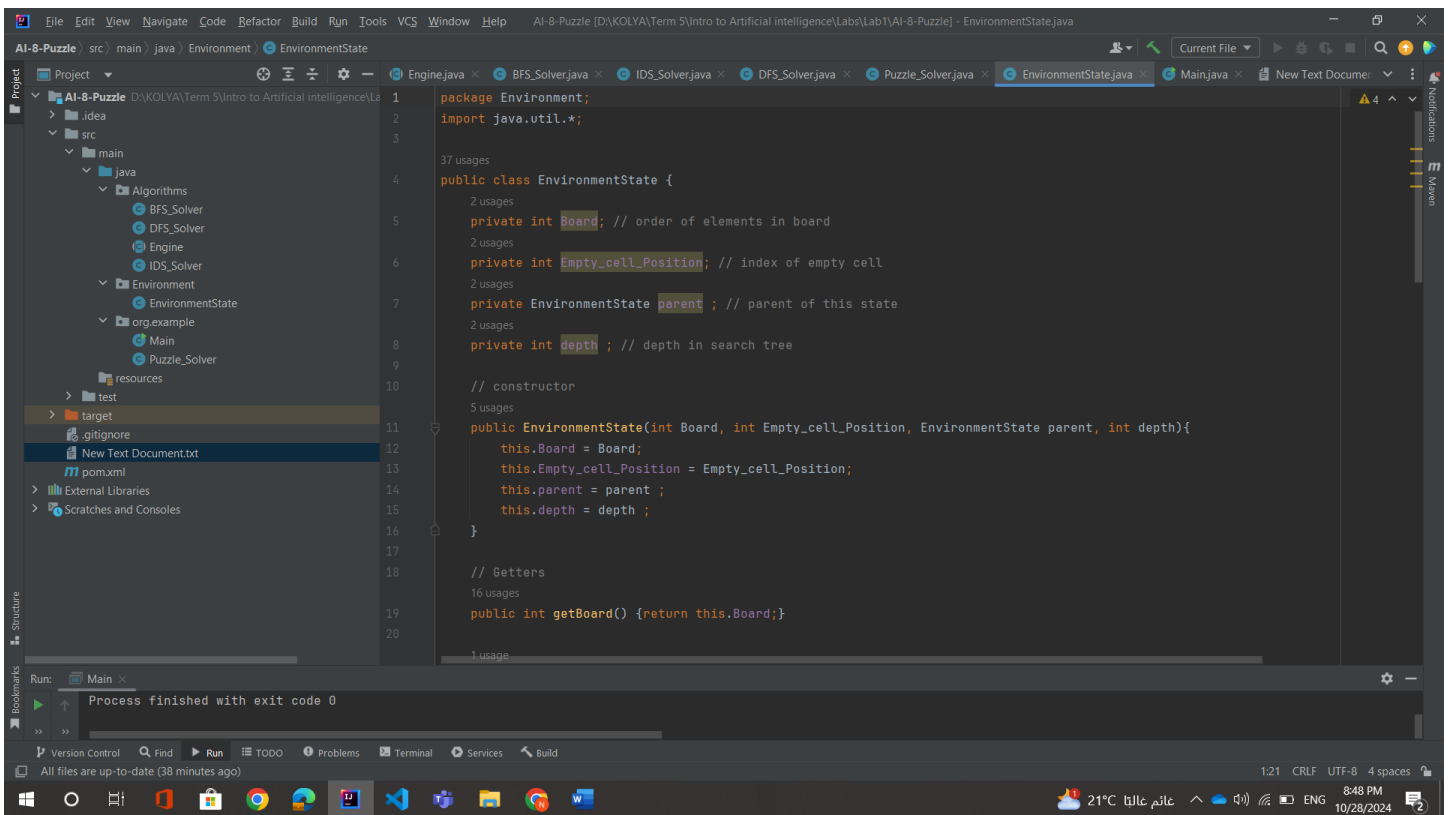
the report should contain the data structure used (if any) and the explanation for each algorithm, Assumptions and details you find them necessary to be clarified, Any extra work and Sample runs. You should show your algorithm and how it operates.

1. BFS

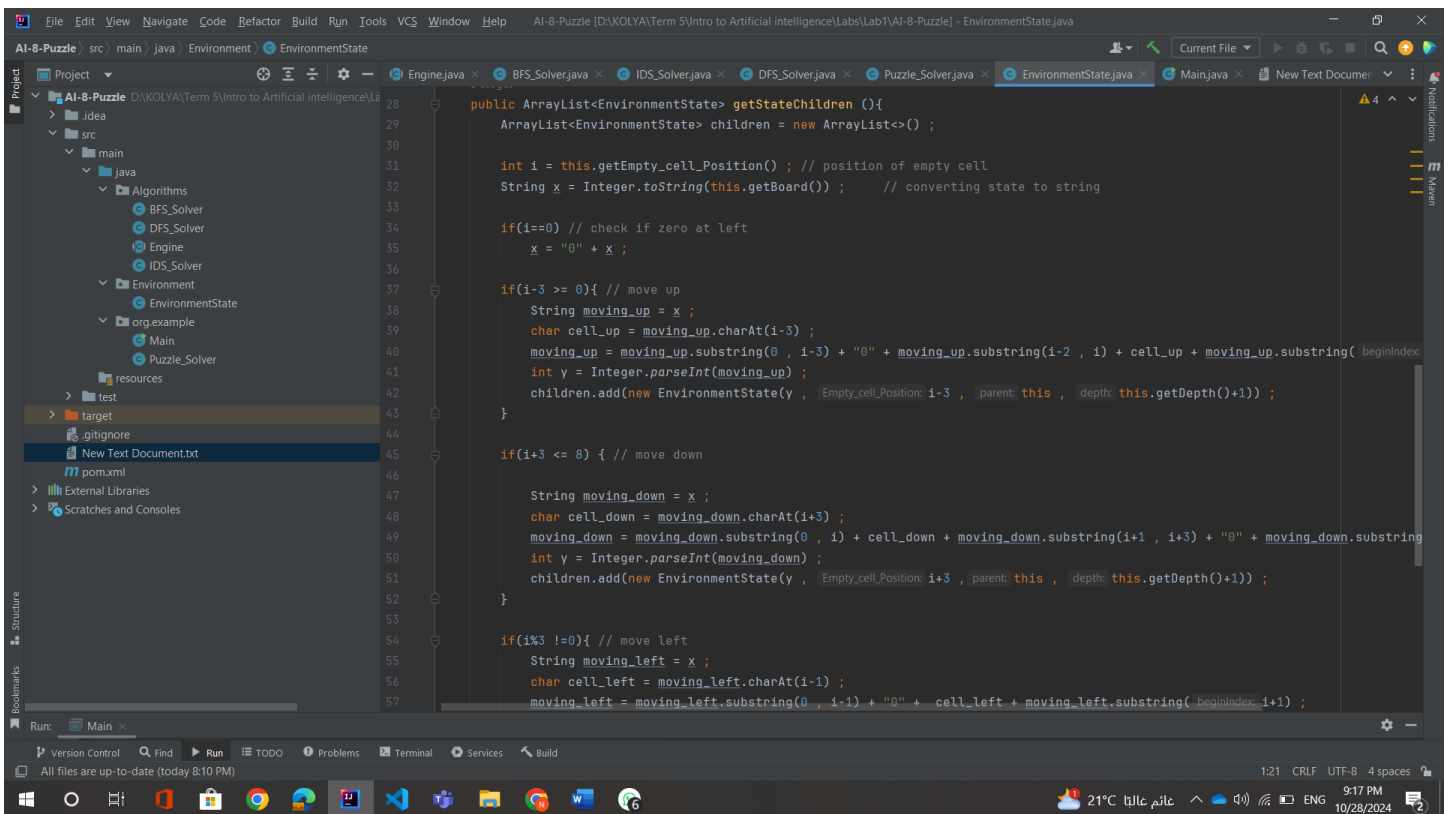
Data Structures :

1. We define a data Structure called EnvironmentState that contains a collection of some information about the puzzle at each step that will be used as node for the search tree for (BFS , DFS , IDS Algorithms).
It contains : Integer represent the board , integer represent the index of the empty cell instead of searching for empty cell every time we expand a node , depth of this node at the search tree , parent node of type EnvironmentState that links each node with the node created it to get the path of the game by backtracking.

This data structure return the all possible children of a node (up – down – left – right)
At this order.



```
1 package Environment;
2 import java.util.*;
3
37 usages
4 public class EnvironmentState {
5     2 usages
6     private int Board; // order of elements in board
7     2 usages
8     private int Empty_cell_Position; // index of empty cell
9     2 usages
10    private EnvironmentState parent; // parent of this state
11    2 usages
12    private int depth; // depth in search tree
13
14    // constructor
15    5 usages
16    public EnvironmentState(int Board, int Empty_cell_Position, EnvironmentState parent, int depth){
17        this.Board = Board;
18        this.Empty_cell_Position = Empty_cell_Position;
19        this.parent = parent;
20        this.depth = depth;
21    }
22
23    // Getters
24    16 usages
25    public int getBoard() {return this.Board;}
26
27    1 usage
```



```
28 public ArrayList<EnvironmentState> getStateChildren () {
29     ArrayList<EnvironmentState> children = new ArrayList<>();
30
31     int i = this.getEmpty_cell_Position(); // position of empty cell
32     String x = Integer.toString(this.getBoard()); // converting state to string
33
34     if(i==0) // check if zero at left
35         x = "0" + x;
36
37     if(i-3 >= 0) { // move up
38         String moving_up = x;
39         char cell_up = moving_up.charAt(i-3);
40         moving_up = moving_up.substring(0, i-3) + "0" + moving_up.substring(i-2, i) + cell_up + moving_up.substring(beginIndex
41         int y = Integer.parseInt(moving_up);
42         children.add(new EnvironmentState(y, Empty_cell_Position: i-3, parent: this, depth: this.getDepth()+1));
43     }
44
45     if(i+3 <= 8) { // move down
46
47         String moving_down = x;
48         char cell_down = moving_down.charAt(i+3);
49         moving_down = moving_down.substring(0, i) + cell_down + moving_down.substring(i+1, i+3) + "0" + moving_down.substring
50         int y = Integer.parseInt(moving_down);
51         children.add(new EnvironmentState(y, Empty_cell_Position: i+3, parent: this, depth: this.getDepth()+1));
52     }
53
54     if(i%3 !=0) { // move left
55         String moving_left = x;
56         char cell_left = moving_left.charAt(i-1);
57         moving_left = moving_left.substring(0, i-1) + "0" + cell_left + moving_left.substring(beginIndex: i+1);
```

2. We also used Queue for frontier list that makes us expand level by level.
3. We also used Hashset that contains all nodes that were visited or nodes that are in the frontier at the current time . Hashset gives O(1) operations (instead of linear search in visited and frontier queues) .we now don't want visited array.

Explanation of the algorithm:

Expand shallowest node

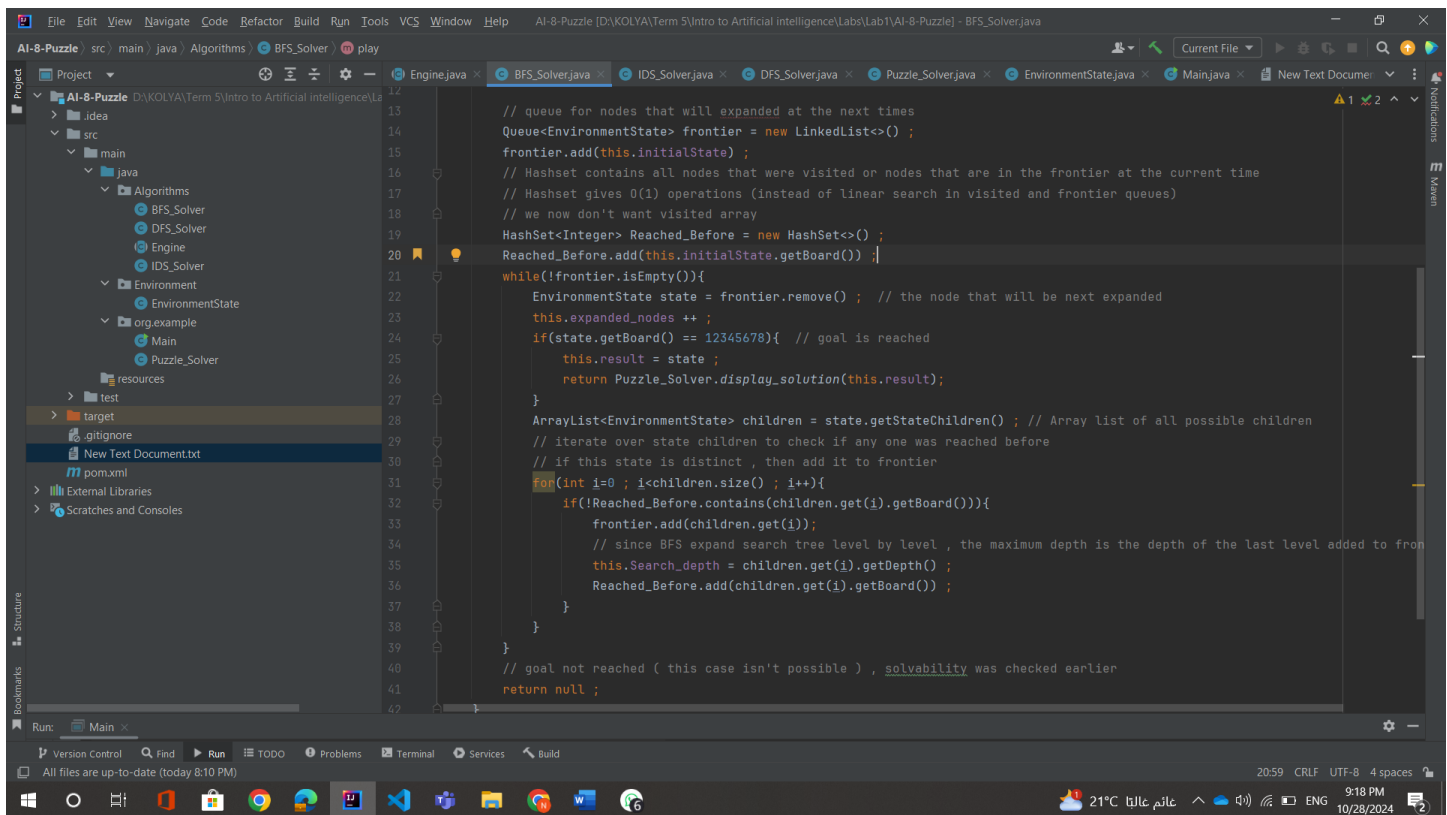
The algorithm takes the state of the puzzle , add it to the frontier queue.

Iterate over the frontier unless it is empty.

Remove the top of the frontier , check if it is the goal return it ,

If not the goal compute all possible children (up – down – left – right) ,

Check every child if it is expanded before then ignore it , if no add it to the frontier then repeat.



```
13 // queue for nodes that will be expanded at the next times
14 Queue<EnvironmentState> frontier = new LinkedList<>() ;
15 frontier.add(this.initialState) ;
16 // Hashset contains all nodes that were visited or nodes that are in the frontier at the current time
17 // Hashset gives O(1) operations (instead of linear search in visited and frontier queues)
18 // we now don't want visited array
19 HashSet<Integer> Reached_Before = new HashSet<>() ;
20 Reached_Before.add(this.initialState.getBoard()) ;
21 while(!frontier.isEmpty()){
22     EnvironmentState state = frontier.remove() ; // the node that will be next expanded
23     this.expanded_nodes ++ ;
24     if(state.getBoard() == 12345678){ // goal is reached
25         this.result = state ;
26         return Puzzle_Solver.display_solution(this.result);
27     }
28     ArrayList<EnvironmentState> children = state.getStateChildren() ; // Array list of all possible children
29     // iterate over state children to check if any one was reached before
30     // if this state is distinct , then add it to frontier
31     for(int i=0 ; i<children.size() ; i++){
32         if(!Reached_Before.contains(children.get(i).getBoard())){
33             frontier.add(children.get(i));
34             // since BFS expand search tree level by level , the maximum depth is the depth of the last level added to frontier
35             this.Search_depth = children.get(i).getDepth() ;
36             Reached_Before.add(children.get(i).getBoard()) ;
37         }
38     }
39 }
40 // goal not reached ( this case isn't possible ) , solvability was checked earlier
41 return null ;
42
```

Assumptions

The puzzle has solution at finite depth.

2. DFS

Data Structures :

1. Environment state as we mentioned at BFS algorithm.
2. We also used Stack for frontier list that makes us expand deepest first.
3. We also used Hashset that contains all nodes that were visited or nodes that are in the frontier at the current time . Hashset gives $O(1)$ operations (instead of linear search in visited and frontier queues) .we now don't want visited array.

Explanation of the algorithm:

Expand deepest first.

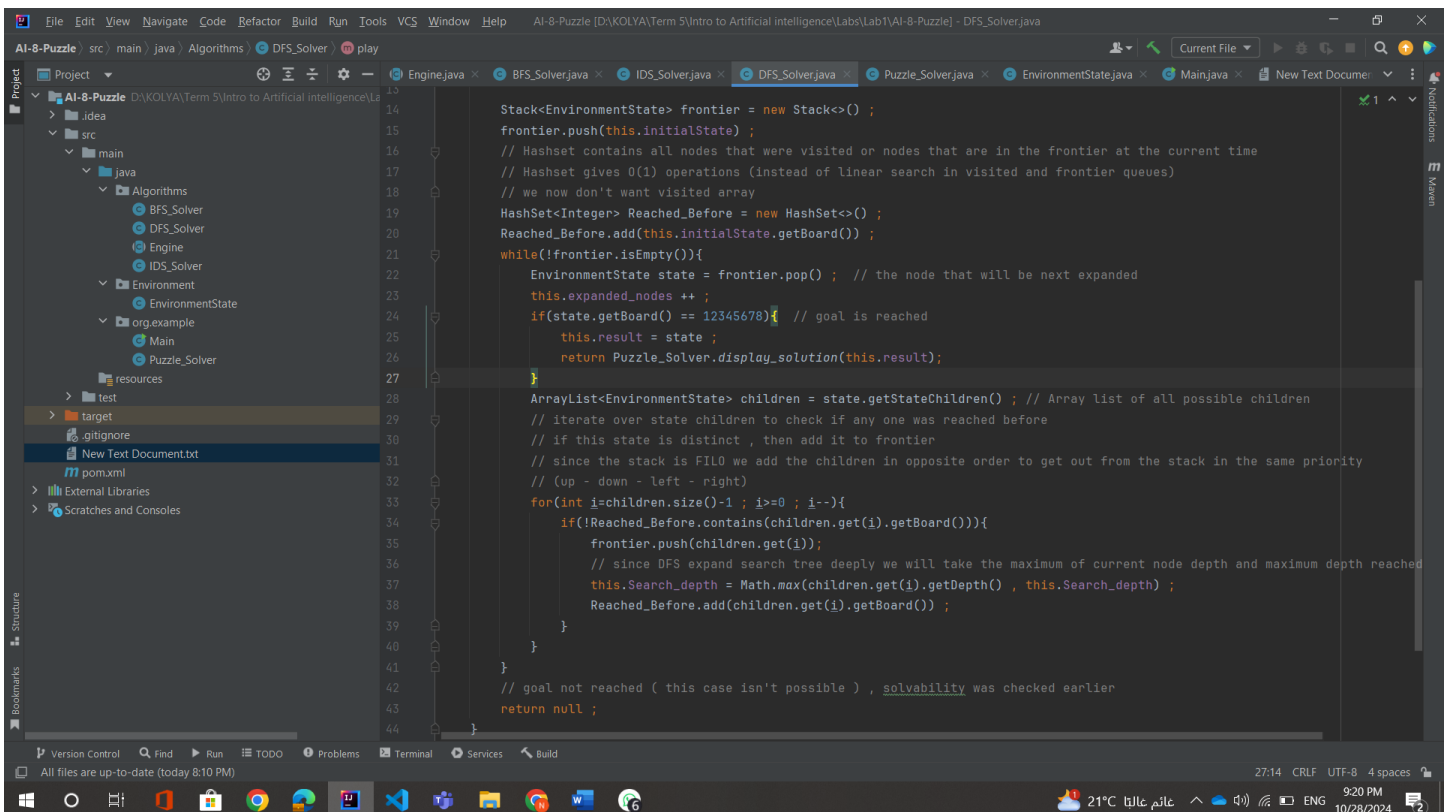
The algorithm takes the state of the puzzle , add it to the frontier stack.

Iterate over the frontier unless it is empty.

Remove the top of the Stack , check if it is the goal return it ,

If not the goal compute all possible children (up – down – left – right) ,

Check every child if it is expanded before then ignore it , if no add it to the frontier then repeat. Check deepest first if not the goal , back to lower depth then continue expansion.



```
14 Stack<EnvironmentState> frontier = new Stack<>() ;
15 frontier.push(this.initialState) ;
16 // Hashset contains all nodes that were visited or nodes that are in the frontier at the current time
17 // Hashset gives O(1) operations (instead of linear search in visited and frontier queues)
18 // we now don't want visited array
19 HashSet<Integer> Reached_Before = new HashSet<>() ;
20 Reached_Before.add(this.initialState.getBoard()) ;
21 while(!frontier.isEmpty()){
22     EnvironmentState state = frontier.pop() ; // the node that will be next expanded
23     this.expanded_nodes ++ ;
24     if(state.getBoard() == 12345678) { // goal is reached
25         this.result = state ;
26         return Puzzle_Solver.display_solution(this.result);
27     }
28     ArrayList<EnvironmentState> children = state.getStateChildren() ; // Array list of all possible children
29     // iterate over state children to check if any one was reached before
30     // if this state is distinct , then add it to frontier
31     // since the stack is FILO we add the children in opposite order to get out from the stack in the same priority
32     // (up - down - left - right)
33     for(int i=children.size()-1 ; i>=0 ; i--){
34         if(!Reached_Before.contains(children.get(i).getBoard())){
35             frontier.push(children.get(i));
36             // since DFS expand search tree deeply we will take the maximum of current node depth and maximum depth reached
37             this.Search_depth = Math.max(children.get(i).getDepth() , this.Search_depth) ;
38             Reached_Before.add(children.get(i).getBoard()) ;
39         }
40     }
41 }
42 // goal not reached ( this case isn't possible ) , solvability was checked earlier
43 return null ;
44 }
```

Assumptions

The puzzle has solution at finite depth.

3.IDS

Data Structures:

1. Environment state as we mentioned at BFS algorithm.
2. We also used Stack for frontier list that makes us expand deepest first.
3. We also used **HashMap** that contains all nodes that were visited or nodes that are in the frontier at the current time . HashMap gives $O(1)$ operations (instead of linear search in visited and frontier queues) .we now don't want visited array. HashMap contains the reached states and their depth.
HashMap <key , value> = HashMap<board , depth > because board is unique so it is the key.

Explanation of the algorithm:

Combines the benefits of BFS and DFS.

Idea: Iteratively increase the search limit until the depth of the shallowest solution d is reached.

Applies DLS with increasing limits.

The algorithm takes the state of the puzzle , add it to the frontier Stack.

Iterate over the frontier unless it is empty.

Remove the top of the frontier , if the node has depth equal to the limit of the search check if it is the goal (because we are sure that no goal exists at lower limit) if yes return it ,

If not the goal compute all possible children (up – down – left – right) ,

Check every child if it is expanded before check if current depth is less than its last depth then add it if no ignore it , if no compare child depth with the limit if the depth > limit ignore it , if no add it to the frontier then repeat.

```
14 this.limit = 0 ; // limit of searching
15
16 Stack<EnvironmentState> frontier = new Stack<>() ;
17 // HashMap contains the reached states and their depth
18 // HashMap <key , value> = HashMap<board , depth> because board is unique
19 HashMap<Integer , Integer> Reached_Before = new HashMap<>() ;
20
21 while (true) { // repeating dfs with increasing limit
22
23     frontier.push(this.initialState) ;
24     Reached_Before.put(this.initialState.getBoard() , 0) ;
25
26     while (!frontier.isEmpty()) {
27         EnvironmentState state = frontier.pop(); // the node that will be next expanded
28         this.expanded_nodes++;
29
30         if (state.getDepth() == this.limit) { // search for goal in the search depth equal to limit only
31             // because we are sure that no goal exists at lower limit
32             if (state.getBoard() == 12345678) { // goal is reached
33                 this.result = state;
34                 this.Search_depth = state.getDepth() ;
35                 return Puzzle_Solver.display_solution(this.result);
36             }
37         }
38
39         ArrayList<EnvironmentState> children = state.getStateChildren(); // Array list of all possible children
40         for (int i=children.size()-1 ; i>=0 ; i--) {
41             if (!Reached_Before.containsKey(children.get(i).getBoard())) {
42                 if (children.get(i).getDepth() <= limit) // if child isn't visited before and its depth <= limit , push it
43                     // if depth > limit ignore it
44                     frontier.push(children.get(i));
45                 Reached_Before.put(children.get(i).getBoard() , children.get(i).getDepth());
46             }
47         }
48         Reached_Before.clear();
49         this.limit ++ ; // increasing limit
50     }
51 }
52 // goal not reached ( this case isn't possible ) , solvability was checked earlier
53 }
```

```
35 return Puzzle_Solver.display_solution(this.result);
36 }
37
38 ArrayList<EnvironmentState> children = state.getStateChildren(); // Array list of all possible children
39 for (int i=children.size()-1 ; i>=0 ; i--) {
40     if (!Reached_Before.containsKey(children.get(i).getBoard())) {
41         if (children.get(i).getDepth() <= limit) // if child isn't visited before and its depth <= limit , push it
42             // if depth > limit ignore it
43             frontier.push(children.get(i));
44         Reached_Before.put(children.get(i).getBoard() , children.get(i).getDepth());
45     }
46     else { // if the child was visited before , if the child was explored at higher depth push it again
47         // if the child was visited at lower depth , there is no problem it is still in the stack and will be exp
48         if (Reached_Before.get(children.get(i).getBoard()) > children.get(i).getDepth()) {
49             frontier.push(children.get(i));
50             Reached_Before.replace(children.get(i).getBoard() , children.get(i).getDepth() );
51         }
52     }
53 }
54 Reached_Before.clear();
55 this.limit ++ ; // increasing limit
56 }
57 // goal not reached ( this case isn't possible ) , solvability was checked earlier
58 }
59 }
60 }
61 }
62 }
63 }
```

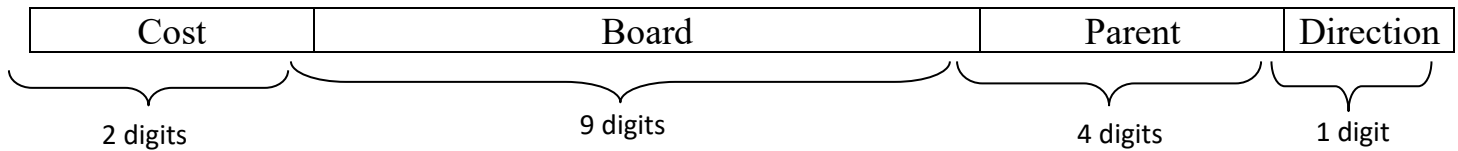
Assumptions

The puzzle has solution at finite depth.

4. A*

Way of storing states:

State: <Long> [stored in **64bits** in java]



○ Cost: $g(n) + h(n)$;

Worst case for $h(n)$: =

4	3	4
3	2	3
4	3	4

$4(4) + 4(3) + 2 = 30$ [according to Manhattan]

Worst case for $g(n) = \max(\text{depth}) = 31$ [proved by exhaustive search methods]

$\therefore g(n) = 6$; thus $f(n) = 30 + 31 < 61 \rightarrow$ **2 digits**.

Notice that: this estimation is impossible since as we get nearer to the goal $h(n)$ decreases, but this is just for the sake of number of digits.

○ Board: It has 9 slots; thus **9 digits**.

○ Parent: In case of Full tree (worst case) number of inner nodes (parents) $< N$;

Where N is the total number of nodes in the tree which is equal to number of permutations of

the empty slot:

2	3	2
3	4	3
2	3	2

$= 2^4 * 3^4 * 4 = 5184 = N$

\therefore Maximum number of parents $< 5184 \rightarrow$ needs **4 digits**.

So the hash map key of the state's parent that consists of 4 digits which is created randomly using `AStarSearch.createId()` function is stored in the 4 digits of the parent.

○ Direction: Is one of the following four: root (0) [came from no movement], right (1), left (2), up (3), down (4) \rightarrow needs **1 digit**.

Data structures used:

Visited List: Hash Map.

Frontier List: Priority Queue.

Path to goal is traced using: Stack.

Explanation of the algorithm:

New states expanded are popped from the frontier and added as visited, and its new children are put in the frontier in its correct order with respect to the cost of it, which is calculated as the addition of the depth of the current state and the estimation to the goal (heuristic $f(x)$) (with Manhattan or Euclidean) ($g(x) + h(x)$).

Comparison between the two heuristics:

Manhattan is **more admissible**, because it gets the sum of horizontal and vertical movements to the goal which is more accurate than Euclidean method, which is calculating the straight distance between the current state and the goal.

	Initial state	Manhattan	Euclidean
# of expanded nodes	5 1 2 3 4 0 6 7 8	32 nodes	59 nodes
Output path		[up, left, left, down, right, up, right, down, left, left, up]	[up, left, left, down, right, up, right, down, left, left, up]
Goal depth		11	11
Running time		14 msec	18 msec
# of expanded nodes	1 2 3 4 5 0 6 7 8	125 nodes	209 nodes
Output path		[left, left, up, right, right, down, left, up, left, down, right, right, up, left, left]	[left, left, up, right, right, down, left, up, left, down, right, right, up, left, left]
Goal depth		15	15
Running time		24 msec	33 msec
# of expanded nodes	8 6 7 2 5 4 3 0 1	2526 nodes	9684 nodes
Output path		[up, right, down, left, up, up, left, down, down, right, up, right, up, left, down, right, down, left, up, left, up]	[right, up, left, down, down, right, up, right, up, left, down, right, down, left, up, left, up, right, down, left, down, right, up, up, left]
Goal depth		27	25
Running time		389 msec	4176 msec

TEST CASES BY THE GUI:

8-Puzzle!

Chose Engin: **BFS**

0,8,1,7,4,5,2,3,6

Solve **Play**

Cost: 24

#Nodes: 117977

Depth: 25

Time: 152.7355 ms

Path:

- [0, 8, 1, 7, 4, 5, 2, 3, 6]
- [7, 8, 1, 0, 4, 5, 2, 3, 6]
- [7, 8, 1, 2, 4, 5, 0, 3, 6]
- [7, 8, 1, 2, 4, 5, 3, 0, 6]
- [7, 8, 1, 2, 0, 5, 3, 4, 6]
- [7, 0, 1, 2, 8, 5, 3, 4, 6]
- [7, 1, 0, 2, 8, 5, 3, 4, 6]
- [7, 1, 5, 2, 8, 0, 3, 4, 6]
- [7, 1, 5, 2, 0, 8, 3, 4, 6]
- [7, 1, 5, 2, 4, 8, 3, 0, 6]
- [7, 1, 5, 2, 4, 8, 3, 6, 0]

	8	1
7	4	5
2	3	6

8-Puzzle!

Chose Engin: **IDS**

4,0,2,5,8,7,3,1,6

Solve **Play**

Cost: 35

#Nodes: 509435

Depth: 35

Time: 284.1957 ms

Path:

- [4, 0, 2, 5, 8, 7, 3, 1, 6]
- [4, 8, 2, 5, 0, 7, 3, 1, 6]
- [4, 8, 2, 5, 1, 7, 3, 0, 6]
- [4, 8, 2, 5, 1, 7, 0, 3, 6]
- [4, 8, 2, 0, 1, 7, 5, 3, 6]
- [0, 8, 2, 4, 1, 7, 5, 3, 6]
- [8, 0, 2, 4, 1, 7, 5, 3, 6]
- [8, 1, 2, 4, 0, 7, 5, 3, 6]
- [8, 1, 2, 4, 3, 7, 5, 0, 6]
- [8, 1, 2, 4, 3, 7, 5, 6, 0]
- [8, 1, 2, 4, 3, 0, 5, 6, 7]

4		2
5	8	7
3	1	6

8-Puzzle!

Chose Engin

DFS

8,4,0,7,5,3,1,2,6

Solve Play

Cost : 53684

#Nodes :61939

Depth: 53684

Time: 105.5452 ms

Path:

[8, 4, 0, 7, 5, 3, 1, 2, 6]

[8, 4, 3, 7, 5, 0, 1, 2, 6]

[8, 4, 3, 7, 5, 6, 1, 2, 0]

[8, 4, 3, 7, 5, 6, 1, 0, 2]

[8, 4, 3, 7, 0, 6, 1, 5, 2]

[8, 0, 3, 7, 4, 6, 1, 5, 2]

[0, 8, 3, 7, 4, 6, 1, 5, 2]

[7, 8, 3, 0, 4, 6, 1, 5, 2]

[7, 8, 3, 1, 4, 6, 0, 5, 2]

[7, 8, 3, 1, 4, 6, 5, 0, 2]

[7, 8, 3, 1, 0, 6, 5, 4, 2]

8	4	
7	5	3
1	2	6

8-Puzzle!

Chose Engin

A* - Manhat.π

1,3,6,0,4,5,7,8,2

Solve Play

Cost : 23

#Nodes :1

Depth: 23

Time: 99.1937 ms

Path:

[1, 3, 6, 0, 4, 5, 7, 8, 2]

[0, 3, 6, 1, 4, 5, 7, 8, 2]

[3, 0, 6, 1, 4, 5, 7, 8, 2]

[3, 6, 0, 1, 4, 5, 7, 8, 2]

[3, 6, 5, 1, 4, 0, 7, 8, 2]

[3, 6, 5, 1, 0, 4, 7, 8, 2]

[3, 0, 5, 1, 6, 4, 7, 8, 2]

[0, 3, 5, 1, 6, 4, 7, 8, 2]

[1, 3, 5, 0, 6, 4, 7, 8, 2]

[1, 3, 5, 6, 0, 4, 7, 8, 2]

[1, 0, 5, 6, 3, 4, 7, 8, 2]

1	3	6
	4	5
7	8	2