1. iterative Deeping Tree Search (IDS)

1.1 Main Function

The main function takes an integer input from console and feeds the test case associated with the input value to the function IDS. The input value can be 1,2 and 3, represent the first test case, the second test case and the third test case respectively.Main function is in the class IDS.

1.2 Class included

1) Node

Class members:

1. an array named “configuration”, which represents the positions of the tiles.
2. an instance of Node named “parent”, which represents its’ parent node.
3. an string list named “step”, which records the sequence of actions which represents the path from root to this node.
4. an bool array named “actions”, which records which actions the node has already taken. Like “Up”, “Right” and so on.

Class function:

1.Node(String s): this is a constructor which return an object of Node, when given a string. Like “cutoff”, “failure” and so on. Cutoff means solution can’t be found ,when given a level restriction.Failure means the solution can’t be found within 100000 nodes.

2. Node(int[][] array, Node parent, List<String> step): this is a constructor which return an object of Node, when given a array, a parent node, the path from root to its’ parent node.
3. Node(): this is a constructor which receives nothing but returns an object of Node.

2) IDS

Class members:

1. Node : cutoff, failure.
2. int: count, number.

Class functions:

1. check(Node state): this function verifies whether the given state is the final state.
2. canMove(Node state): this function verifies whether the given node can still move with consideration of the actions it has already taken.
3. childNode(Node state): this function returns a child node of the given node.
4. getBlockPosition(Node state): this function return the position of “*” in a given matrix.
5. moveLeft(Node state): this function return a child node of the given node and the tie between is “move left”.
6. moveRight(Node state): this function return a child node of the given node and the tie between is “move right”.

7.moveUp(Node state): this function return a child node of the given node and

the tie between is “move up”.

8.moveDown(Node state) :this function return a child node of the given node and the tie between is “move down”.

9.eightPuzzle(Node state, int level): this function return a node, when give a initial node and the restriction level. The node returned can either be a valid solution, a invalid cutoff or failure.The detail of this function is below:

```
Function eightPuzzle(node, limit) return a solution or cutoff/failure
    If(nodeExpanded > 100000) then return failure
    If(check(node)) then return node
    else if(limit == 0) return cutoff
    else {
    for i from 1 to 4{
    If(canMove(node)) {
    child = childNode(state)
    Result = eightPuzzle(child,limit-1)
    If(result != cutoff) return result
          }
       }
   Return cutoff
    }
```

10. IDSSearch(node state): this function calls eightPuzzle many times until eightPuzzle return an invalid value or return failure when the number of node expanded is larger than 100000. Each time eightPuzzle is called, the parameter level feed to it is one more larger than the older level.

1.3 Execution Results

Suppose the input is

```
. 1 3
4 2 5
7 8 6
```

States of the first 5 nodes in the order they would be expanded are shown below:

```
---the 1th Node been expanded is :
           * 1 3
           4 2 5
           7 8 6
---the 2th Node been expanded is :
           1 * 3
           4 2 5
           7 8 6
---the 3th Node been expanded is :
           4 1 3
           * 2 5
           7 8 6
---the 4th Node been expanded is :
           * 1 3
           4 2 5
           7 8 6
---the 5th Node been expanded is :
           1 * 3
           4 2 5
           7 8 6
```

Solution found is

```
------initial state: ------
        * 1 3
        4 2 5
        7 8 6

---1th step: move Right----
        1 * 3
        4 2 5
        7 8 6

---2th step: move Down-----
        1 2 3
        4 * 5
        7 8 6

---3th step: move Right----
        1 2 3
        4 5 *
        7 8 6

---4th step: move Down-----
        1 2 3
        4 5 6
        7 8 *
```

# of moves from initial state to goal state is 4, and # of nodes expanded is 61. time used is 1ms.

Suppose the input is

```
. 5 2
1 8 3
4 7 6
```

States of the first 5 nodes in the order they would be expanded are shown below:

```
---the 1th Node been expanded is :-
        * 5 2
        1 8 3
        4 7 6
---the 2th Node been expanded is :-
        1 5 2
        * 8 3
        4 7 6
---the 3th Node been expanded is :-
        5 * 2
        1 8 3
        4 7 6
---the 4th Node been expanded is :-
        * 5 2
        1 8 3
        4 7 6
---the 5th Node been expanded is :-
        1 5 2
        * 8 3
        4 7 6
```

Solution found is :

```
-----initial state: -----
       * 5 2
       1 8 3
       4 7 6

--1th step: move Down----
       1 5 2
       * 8 3
       4 7 6

--2th step: move Down----
       1 5 2
       4 8 3
       * 7 6

--3th step: move Right---
       1 5 2
       4 8 3
       7 * 6

--4th step: move Up------
       1 5 2
       4 * 3
       7 8 6

--5th step: move Up------
       1 * 2
       4 5 3
       7 8 6


---6th step: move Right------
        1 2 *
        4 5 3
        7 8 6

---7th step: move Down-------
        1 2 3
        4 5 *
        7 8 6

---8th step: move Down-------
        1 2 3
        4 5 6
        7 8 *
```

# of moves from initial state to goal state is 8, and # of nodes expanded is 3642. time used is 3ms.

Suppose the input is

```
8 6 7
2 5 4
3 . 1
```

States of the first 5 nodes in the order they would be expanded are shown below:

```
---the 1th Node been expanded is :----
            8 6 7
            2 5 4
            3 * 1
---the 2th Node been expanded is :----
            8 6 7
            2 5 4
            3 1 *
---the 3th Node been expanded is :----
            8 6 7
            2 5 4
            * 3 1
---the 4th Node been expanded is :----
            8 6 7
            2 * 4
            3 5 1
---the 5th Node been expanded is :----
            8 6 7
            2 5 4
            3 * 1
```

The number of expanded node is more than 100000, when given this input. Solution can't be found

1.4 Findings

1) From the chart below, we can see the time used is positive correlated to the number of node expanded in the test cases.

Test case #	# of moves to reach the goal	# of nodes expanded	Time elapsed (ms)
1	4	61	1
2	8	3642	3
3	No solution	> 100000	No solution

2. depth-first graph search (DFGS)

2.1 Main Function

The main function takes correct input from console and trigger the depth-first graph search and return the solution as well as the time used by this process. If the input is incorrect, the program will terminate and some warnings will be printed on the console.

2.2 Classes included

NODE members:

```
String state;
NODE parent;
// make the actions a stack rather than a set to ensure the first //
generated child node will be popped out first
LinkedList<ACTION> actions = new LinkedList<ACTION>();
int path_cost;
```

where *state* records the sequence of numbers in the 8-puzzle, like ".12345678", *parent* is a pointer to the parent node, *actions* records the movement this node could

have, like UP, DOWN, LEFT, and RIGHT, which are represented in a enumerate structure. *path_cost* records the cost to move from the initial state to the current state.

Constructor of the class NODE takes input into an instance of the class NODE, and set actions it may have in the tiling number ascending sequence.

```
// constructor
NODE(String state, NODE parent, int p)
{
    this.state = state;
    this.parent = parent;
    this.path_cost = p;
    setAction();
}
```

Set and get methods helps to set and retrieve attributes of each node in node generating and path searching. We will talk about `void setAction()` which is a rather important one.

```
HashMap<Character, ACTION> possibleActions = new HashMap<Character,
ACTION>();
```

In `void setAction()`, first a variable called `possibleActions` is declared as a HashMap, whose key is a character like '1', '4', '6', and value is an enumeration type representing moving direction like `UP`, and `DOWN`. Once all the possible actions of current node is found, we need to sort these actions in the sequence of tiling number ascending and put them in the member `actions`. This code is realized in the following snippet.

```
while(possibleActions.isEmpty() == false)
{
    Iterator<Character> itr = possibleActions.keySet().iterator();
    Character tempChar1 = itr.next();
    while(itr.hasNext())
    {
        Character tempChar2 = itr.next();
        if(Character.getNumericValue(tempChar1) <
Character.getNumericValue(tempChar2))
        {
            tempChar1 = tempChar2;
        }
    }
    actions.addLast(possibleActions.remove(tempChar1));
}
```

DFGS members:

```
Stack<NODE> fringe = new Stack<NODE>();
HashSet<String> closed = new HashSet<String>();
```

```
Stack<String> PATHtoGoal = new Stack<String>();
// to count # of node expanded
private static int expandCnt = 0;
```

where *fringe* is a LIFO queue, or stack, to store the nodes in the fringe. *closed* stores the states that have been explored. *PATHtoGoal* records the path from the initial state to the goal state if the goal state is found. *expandCnt* helps to record the number of expanded nodes, display the first 5 nodes that is expanded, and helps to terminate the program if more than 100,000 nodes is expanded even though the goal state has not been found yet.

The function `void depthFirstGraphSearch(String inputStr)` follows the pseudocode, initializing the graph searching and executing the loop.
The initialization includes:

```
fringe.clear();
closed.clear();
NODE n = new NODE(inputStr, null, 0);
fringe.push(n);
```

And in the loop, our code exactly follows the pseudocode:

- Check if *fringe* is empty. If yes, return failure, else continue.
- Pop out one element on top of the stack *fringe* and check if this element is the goal. If yes, print out some useful information and done; else continue.
- If this element is in the pool *closed*, do nothing; else, expand it and insert its child nodes.

The function `void expandAndInsert(NODE p)` will expand the parent node according to different actions it could perform, generate the children nodes and insert the children nodes into the fringe structure.

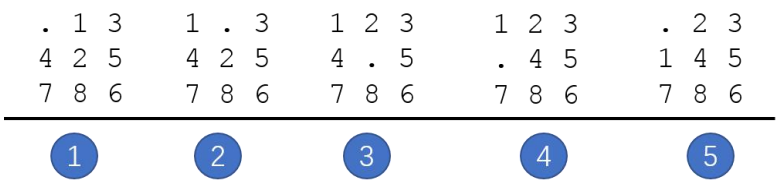
The function `static String move(final String s, ACTION act)` takes the parent node and its possible action as input, and return the mode for its children node.

2.3 Execution Results

Suppose the input is

```
. 1 3
4 2 5
7 8 6
```

States of the first 5 nodes in the order they would be expanded are shown below:



Solution found is

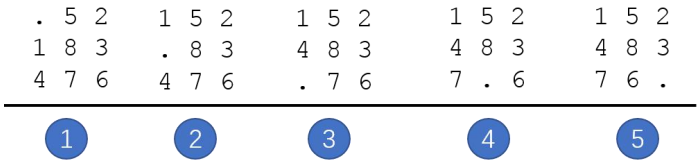
```
1 2 3
4 5 6
7 8 .
```

The solution, that is the sequence of tile ID from the initial state to goal state is too long that I cannot put it here. Number of moves from initial state to goal state is 43436, and number of nodes expanded is 46272. time used is 337 ms.

Suppose the input is

```
. 5 2
1 8 3
4 7 6
```

States of the first 5 nodes in the order they would be expanded are shown below:

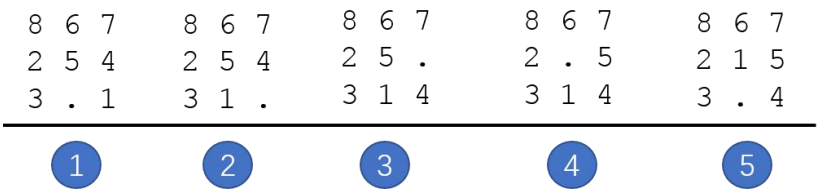


The solution, that is the sequence of tile ID from the initial state to goal state is too long that I cannot put it here. Number of moves from initial state to goal state is 50278, and # of nodes expanded is 53789. time used is 386 ms.

Suppose the input is

```
8 6 7
2 5 4
3 . 1
```

States of the first 5 nodes in the order they would be expanded are shown below:



The solution, that is the sequence of tile ID from the initial state to goal state is too long that I cannot put it here. Number of moves from initial state to goal state is 24837, and # of nodes expanded is 25901. time used is 267 ms.

2.4 Findings

2) From the chart below, we can see the time used is positive correlated to the number of node expanded in the test cases.

Test case #	# of node expanded	Time elapsed (ms)
1	46272	337
2	53789	386
3	25901	267

3) To make the algorithm a little “smarter”, DOWN and RIGHT actions of the “.” node could be first considered before the UP and LEFT actions. This might decrease the #

of moves from initial state to goal state.

3. A* search with total Manhattan distance heuristic

3.1 Main Function

The main function takes correct input from console and trigger the A* search with total Manhattan distance heuristic as well as time the searching.

3.2 Useful Classes

Next go into the details in the classes: NODE and AStarSearch.

NODE members:

```
String state;
NODE parent;
// make the actions a stack rather than a set to ensure the first //
generated child node will be popped out first
LinkedList<ACTION> actions = new LinkedList<ACTION>();
private int gn;
private int fn;
```

where `state` records the sequence of numbers in the 8-puzzle, like ".12345678", `parent` is a pointer to the parent node, `actions` records the movement this node could have, like `UP, DOWN, LEFT`, and `RIGHT`, which are represented in a enumerate structure. `gn` records the cost to move from the initial state to the current state, and `fn` is the estimated cost from the initial state to the goal state.

The **constructor** goes like this:

```
// constructor
NODE(String state, NODE parent, int gn)
{
    this.state = state;
    this.parent = parent;
    this.gn = gn;
    this.fn = gn + manhattan(state);
    this.setAction();
}
```

The method `int` manhattan(String `state`) takes the state of current node as input and outputs the Manhattan distance between the current state and the goal state.

Set and get methods helps to set and retrieve attributes of each node in node generating and path searching. Please refer to the code for details.

AStarSearch members:

```
LinkedList <NODE> fringe = new LinkedList <NODE>();
```

```
Stack<String> PATHtoGoal = new Stack<String>();
// to count # of node expanded
private static int expandCnt = 0;
```

where *fringe* is a linked list, to store the nodes in the fringe. Since we need to traverse all the nodes and then grab the node with the least *fn* to expand, using a linked list structure will help accelerate the speed. *PATHtoGoal* stores the path from the initial node to the goal node once the goal is reached. *expandCnt* helps to record the number of expanded nodes, display the first 5 nodes that is expanded, and helps to terminate the program is more than 100,000 nodes is expanded even though the goal state has not been found yet.

The function **void** `AStar_Search (String inputStr)` follows the pseudocode for A* search. In each iteration, the node with the least *fn* will be popped out of *fringe*.

The function **void** `expandAndInsert(NODE p)` will expand the parent node according to different actions it could perform, generate the children nodes and insert the children nodes into the fringe structure.

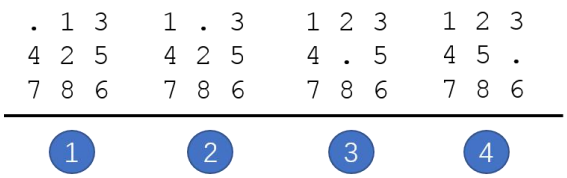
The function **static** `String move(final String s, ACTION act)` takes the parent node and its possible action as input, and returns the mode for its children node.

3.3 Execution Results

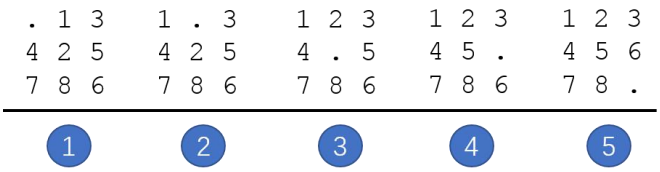
Suppose the input is

```
. 1 3
4 2 5
7 8 6
```

States of the first 4 nodes in the order they would be expanded are shown below:



Solution found is



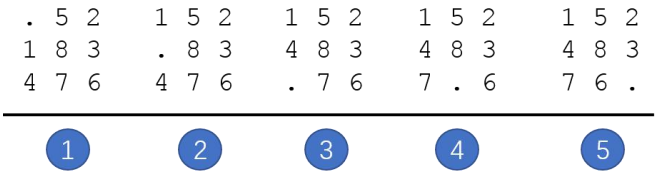
# of moves from initial state to goal state is 4, and # of nodes expanded is 4. time used is 4 ms.

Suppose the input is

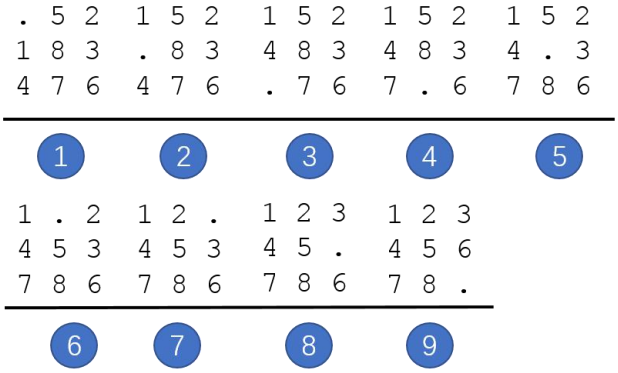
```
. 5 2
1 8 3
```

```
4 7 6
```

States of the first 5 nodes in the order they would be expanded are shown below:



Solution found is

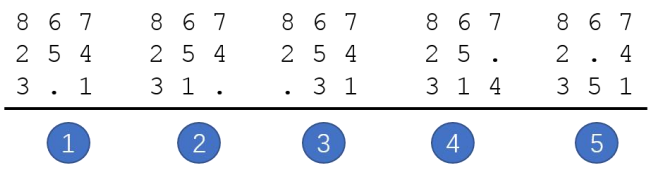


# of moves from initial state to goal state is 8, and # of nodes expanded is 11. time used is 5 ms.

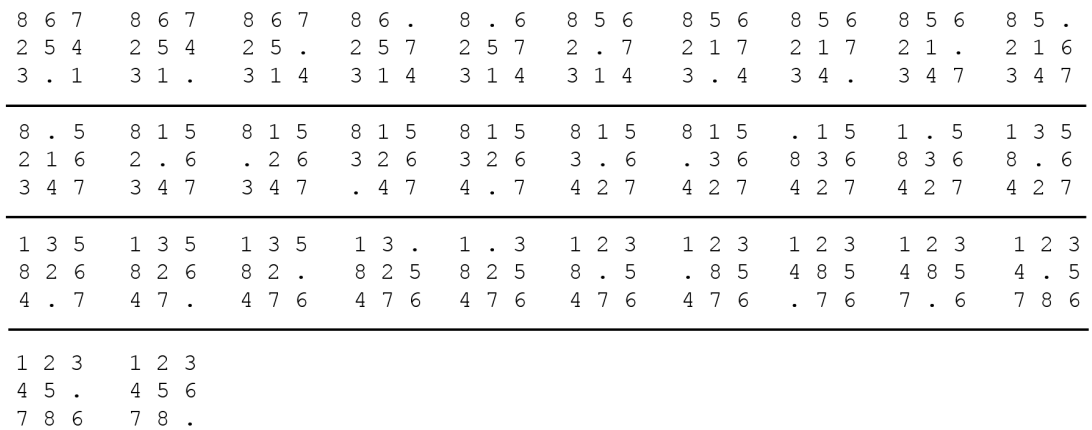
Suppose the input is

```
8 6 7
2 5 4
3 . 1
```

States of the first 5 nodes in the order they would be expanded are shown below:



Solution found is



# of moves from initial state to goal state is 31, and # of nodes expanded is 45789. time

used is 23742 ms.

3.4 Findings

1) From the chart below, we can see the time used is positive correlated to the number of node expanded in the test cases.

Test case #	# of moves to reach the goal	Time elapsed (ms)
1	4	4
2	11	5
3	45789	23742

2) The method of A* with total Manhattan distance heuristic performs better than depth-first graph search in these test cases for the 8-puzzle problem. Sometimes the CPU time is larger than that used in depth-first graph search, perhaps mainly due to the traverse in `fringe` when looking for the element with the least `fn` to pop out and continue.