AI Lab 1

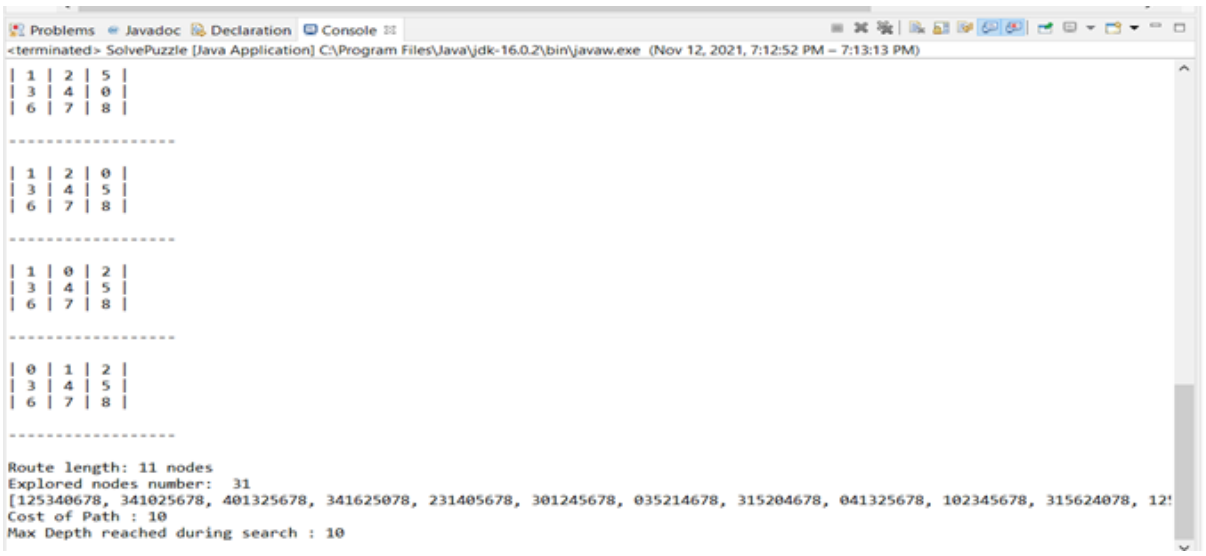
- Marwan Mahrous Selim 18011732
- Youssef Hussein Mohamed 18012118
- Eman Mohamed Abdo 18010431
- Fares Mohamed Fouad 18011223

Required Comparison between using Manhattan heuristic and Euclidean heuristic

we noticed that on using Manhattan distance it is nearer to the fact or the real solution and it also takes less or equal time to find the path this means that Number of nodes explored by Manhattan is smaller than or equal to number of nodes explored on using Euclidean heuristic . But Both of them reach the same path at the end .

For example we can see this example with initial state 310245678

On Using Manhattan :



On using Euclidean :

We can say that A* Using Manhattan is more admissible as we are sure the result got from it is always <= real cost and it is calculated by the way we move the actual sum of vertical and horizontal distances .

The Euclidean distance is always smaller than the Manhattan distance as it represents the length of hypotenuse which is the last side of the triangle sum of 2 sides of triangle is always greater than the third side but we don't actually move on diagonal in this game so it is further from real cost . it is admissible as it is < Manhattan and Manhattan is admissible . but since Manhattan is nearer to the true cost , Manhattan is more admissible .

1. **Path to Goal:**

DFS:

The path to goal is consisting of the root state(input) and one of the children then one of its children and so on until an invalid state is found then we take it's sibling and in the case of no valid siblings we remove the parent and replace it with it's brother and continue (the invalid state is a state that is either in the route or is stored)

BFS:

The path to the goal is based on the level so it's of the root and then all of its children then all of the grand children and so on

A*:

The path consists of the the root(input) then the child with the least cost then we

add its children to priority queue then we study their cost compared to the older nodes and the estimated remaining distance to the goal whether by manhattan heuristic or by Euclidean heuristic to find the least cost node and visit it next time and so on .

2. **Cost of The path:**
3. **Nodes expanded:**

   DFS:

   The nodes that we study here are a single line of children and we only switch child if it's invalid so we study in a single line which is why we reach the lowest depth before some of the upper level nodes, but all the siblings of the nodes that we study are stored in case the node that we study is invalid, so we expand in the form of subtree and the root of the others sub tree

BFS:

The nodes that we study are the next level of nodes after the current level is fully visited so we study the root (input) then all of its children then all of the grandchildren and so on, so we expand in the form of levels

A*:

The nodes that we study are the nodes with the least cost at the current point so this may cost us to expand only one path to its depth like depth first search or we may expand some more paths but the only path we continue is the true path with minimum cost as this algorithm can expect minimum cost path by trying to minimize sum of heuristic function (expected cost from current node to end)

and real cost till reaching this current node from initial state .

4. **Search Depth:**

DFS:

The search depth here will be the deepest path of the subtrees we look in until we find the goal

BFS:

The search depth here is the level that the goal is at

A*:

The depth here isn't predictable because we keep changing the path according to the price but the worst case is the maximum depth of the tree .

5. **Running Time:**

DFS:

O(B^d)

BFS:

O(B^m)

A*:

It depends on the heuristic but in the worst case it is exponential as we can say the number of nodes expanded is exponential in depth of solution O(b ^d) as nearly nodes expanded will be at a path like in depth first search . so time we need is time to expand these nodes . This assumes that a goal state exists , and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

Data Structure:

Hash set

Stack

Queue

Priority queue

Array lists

6. Algorithms:

BFS:

```
BFS(State initial){
Queue frontier queue
Hash set frontier
Hash set explored
Frontier queue .add(initial)
Fronter. add(initial value)
While frontier queue isn't empty{
    S = frontier queue.remove
    Remove s value from frontier and add
    it to explored
    If s is the goal{
        Stack route = new stack
        Add s to stack
        While s has parents
            Add parents to route
            Make s its parent
         Print route

  }
```

```
    Arraylist neighbours = get s
    Neighbours
    For neighbours
        Neighbour = get current neighbour
        From neighbours{
         If frontier and explored doesn't {
              contain neighbour
              Make a state for neighbour
              With the parent set to s
               Add it to frontier queue and
               Frontier
               }
         }
}
          Print failed
}
```

<u>DFS:</u>

```
DFS(State initial){
```

```
Stack frontier stack
Hash set frontier
Hash set explored
Frontier stack .add(initial)
Fronter. add(initial value)
While frontier stack isn't empty{
    S = frontier stack.remove
    Remove s value from frontier and add
    it to explored
    If s is the goal{
        Stack route = new stack
        Add s to stack
        While s has parents
            Add parents to route
            Make s its parent
        Print route

  }
   Arraylist neighbours = get s
```

```
    Neighbours
    For neighbours
        Neighbour = get current neighbour
        From neighbours{
         If frontier and explored doesn't {
              contain neighbour
              Make a state for neighbour
              With the parent set to s
               Add it to frontier stack and
               Frontier
               }
         }
}
          Print failed
}
```

A*:

```
AStar_UseManhatenDist(State initial){
```

HashMap<String,State_cost>parentTable
PriorityQueue<State>frontier
HashSet<String>explored
HashSet<String>frontierSet
HashMap <String,Integer>manhatenDis

initial.setCostTillNow(0)
Int
m=getManhatenDistances(initial.getValue())
initial.setManhatenDist(m)
manhatenDist.put(initial.getValue(),m)
parentTable.put(initial.getValue(), new State_cost(null,0))
Add initial state to frontier and its string value to frontier set
while(frontier isn’t empty) {
State s= peek state of the frontier

```
s.setCostTillNow(frontier.peek().getCost
TillNow())

s.setManhatenDist(frontier.peek().getMa
nhatenDist())

s.setTotalCost(frontier.peek().getTotalCo
st())
Remove peek state s from frontier
Add state s to the explored
                if(state s is the goal) {
                    display("Goal is reached
successfully")
                    //get path and print it
LinkedList<State>path
                    State st=s;
                    while(st!=null) {
                        Add st to the path
```

```
                    st=st.getFather()
                }
                int i=path.size()-1
                while(i>=0) {
                    path.get(i).printState()
                    i--
                }
                display("Route length: " +
path.size()+" nodes");
                display("Explored nodes
number:  " + explored.size());
        display(explored.toString());
        display("Cost of Path :
        "+s.getCostTillNow());
        display("Max Depth reached
        during search :
        "+getMaxDepth(parentTable,explo
        red));
                End of function so return  ;
```

}else if it isn’t goal {

ArrayList<String>a=s.getNeighbours()
int i=0
while(i is less than a.size()) {
if(a.get(i) isn’t in explored and not in the frontier) {
State newSt=new State(a.get(i), s)

newSt.setCostTillNow(s.getCostTillNow()+1)
newSt.setManhatenDist(getManhatenDistances(a.get(i)))
int x=newSt.getManhatenDist()

newSt.setTotalCost(x+newSt.getCostTillNow())

```
        manhatenDist.put(a.get(i),x)
        State_cost st=new State_cost()

        st.setCostG(newSt.getCostTillNow())
        st.setParent(s)
        parentTable.put(a.get(i),st)
        frontier.add(newSt)
    }else if(frontierSet.contains(a.get(i))) {
        //this means look at it in the parent
        table and get its cost till now
        // if g+h was < its current total cost
        from node so decrease  key (change
        total cost in priority queue)
        and change parent in parent table

        decreaseKey(frontier,a.get(i),manhate
        nDist.get(a.get(i)),s,parentTable);
        }
```

```
    i++;
    }
    }
    }
display("Failed! This puzzle is
unsolvable!"); //if he didn't return in the
loop
        //so he didn't reach a solution
    }
-------------------------

AStar_UseEucledianDist(State initial){

HashMap<String,State_cost>parentTable
PriorityQueue<State>frontier
HashSet<String>explored
HashSet<String>frontierSet
HashMap <String,Double>eucledianDist
initial.setCostTillNow(0)
```

Double m = getEucledeanDistances(initial.getValue())
initial.setEucledeanDist(m)

parentTable.put(initial.getValue(),new State_cost(null,0))

eucledianDist.put(initial.getValue(),m)
Put initial state in frontier and its string value in frontier set

while(frontier.isn't empty()) {
State s=peek state of frontier

s.setCostTillNow(frontier.peek().getCostTillNow())

s.setEucledeanDist(frontier.peek().getEucledeanDist())

```
s.setTotalCost(frontier.peek().getT
otalCost())
Remove s from frontier
 And its string value from frontier
                    set
explored.add(s.getValue())


    if(state s is the goal) {
        display("Goal is reached
successfully")
   LinkedList<State>path
    State st=s
        while(st!=null) {
            path.add(st)
            st=st.getFather()
        }
        int i=path.size()-1
```

```
                    while(i>=0) {
                        path.get(i).printState()
                        i--
                    }
                     display("Route length: " +
                path.size()+" nodes")
                    display("Explored nodes
number:  " + explored.size())
                    display("Cost of Path :
"+s.getCostTillNow())

          display(explored.toString());
              display("Max Depth reached
                    during search :
 "+getMaxDepth(parentTable,explored))
                    End of function so return
}else {
    ArrayList<String>
a=s.getNeighbours()
```

```
int i=0
while(i<a.size()) {
if( ! (frontierSet.contains(a.get(i)) ||
(explored.contains(a.get(i)) ) ) ) {
State newSt=new State(a.get(i), s)

newSt.setCostTillNow(s.getCostTillNow
()+1)

newSt.setEucledeanDist(getEucledeanDi
stances(a.get(i)))
double x=newSt.getEucledeanDist()

newSt.setTotalCost(x+newSt.getCostTill
Now())

eucledianDist.put(a.get(i),x)
State_cost st=new State_cost()
```

st.setCostG(newSt.getCostTillNow())
st.setParent(s)

parentTable.put(a.get(i),st)
frontier.add(newSt)
}else if(frontierSet.contains(a.get(i))) {
//this means look at it in the parent table and get its cost till now + Euclidean
// if new total cost was < its current cost from node f so decrease key (change its total cost in priority queue)
//and change parent in parent table

decreaseKey(frontier,a.get(i),eucledianDist.get(a.get(i)),s,parentTable)
}
i++;
}

```
}
}
        display("Failed! This puzzle is
unsolvable!");
        //if he didn't return in the loop
        //so he didn't reach a solution
    }
```

7. Assumption and Clarification:

- We made each state point to its parent so that we track the route from the goal
- We made every state create its neighbours as string to check if we already have them or not
- We store the visited and to be visited states as strings to be able to store

them in hash sets to minimize the searching time and then assumed the time we take to look them up is constant

- We assumed that the only way for a case to be unsolvable is that our visited and to be visited is empty so we checked all reachable states and couldn't reach the goal

---

8. Sample Runs :

Unsolvable cases :

DFS

```
456         //Now you have a valid puzzle to search for a solution for
457
458         if(choise.equals("1")) {
459             DFS(new State(puzzle, null));
460         }else if(choise.equals("2")) {
461             BFS(new State(puzzle, null));
462         }else {
463             // To be replaced when A* is implemented
```

Problems Javadoc Declaration Console

```
<terminated> SolvePuzzle [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe  (Nov 12, 2021, 9:43:59 PM – 9:45:05 PM)
Choose a method to use:
1) DFS
2) BFS
3) A*
Your choise: 1
Please enter the puzzle initial state, example: 125340678 where the zero is the empty cell.
Your puzzle: 432650781
Failed! This puzzle is unsolvable!
```

BFS

```
458         if(choise.equals("1")) {
459             DFS(new State(puzzle, null));
460         }else if(choise.equals("2")) {
461             BFS(new State(puzzle, null));
462         }else {
463             // To be replaced when A* is implemented
```

Problems Javadoc Declaration Console

```
<terminated> SolvePuzzle [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe  (Nov 12, 2021, 9:46:20 PM – 9:4
Choose a method to use:
1) DFS
2) BFS
3) A*
Your choise: 2
Please enter the puzzle initial state, example: 125340678 where the zero is the empty cell.
Your puzzle: 432650781
Failed! This puzzle is unsolvable!
```

A*,Manhattan

Problems Javadoc Declaration Console

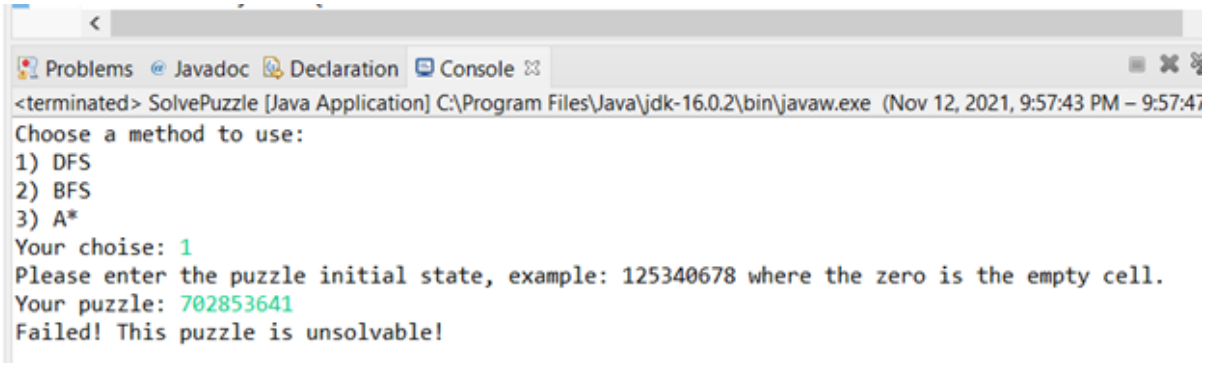
```
<terminated> SolvePuzzle [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe  (Nov 12, 2021, 9:47:01 PM – 9:47:30 PM)
Choose a method to use:
1) DFS
2) BFS
3) A*
Your choise: 3
Please enter the puzzle initial state, example: 125340678 where the zero is the empty cell.
Your puzzle: 432650781
For :
A* with manhaten distance ----> 1
A* with Eucledian distance ---> 2
1
Failed! This puzzle is unsolvable!
```

A*,Euclidean

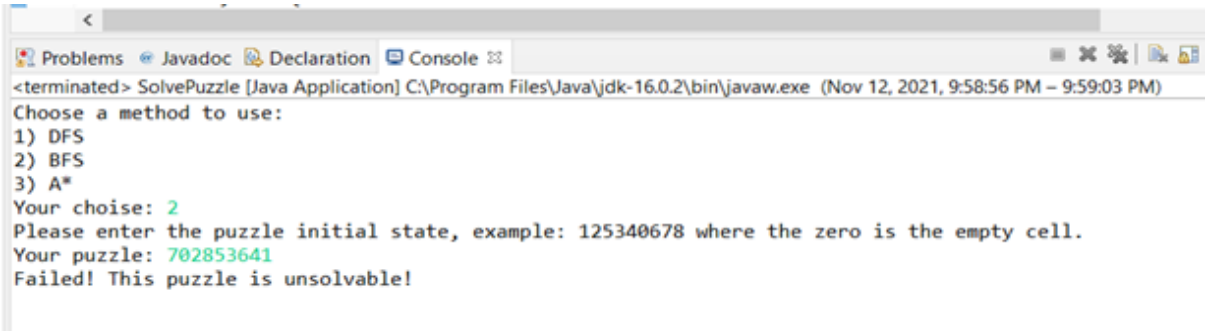
Problems Javadoc Declaration Console

```
<terminated> SolvePuzzle [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe  (Nov 12, 2021, 9:48:30 PM – 9:48:37 PM)
Choose a method to use:
1) DFS
2) BFS
3) A*
Your choise: 3
Please enter the puzzle initial state, example: 125340678 where the zero is the empty cell.
Your puzzle: 432650781
For :
A* with manhaten distance ----> 1
A* with Eucledian distance ---> 2
2
Failed! This puzzle is unsolvable!
```

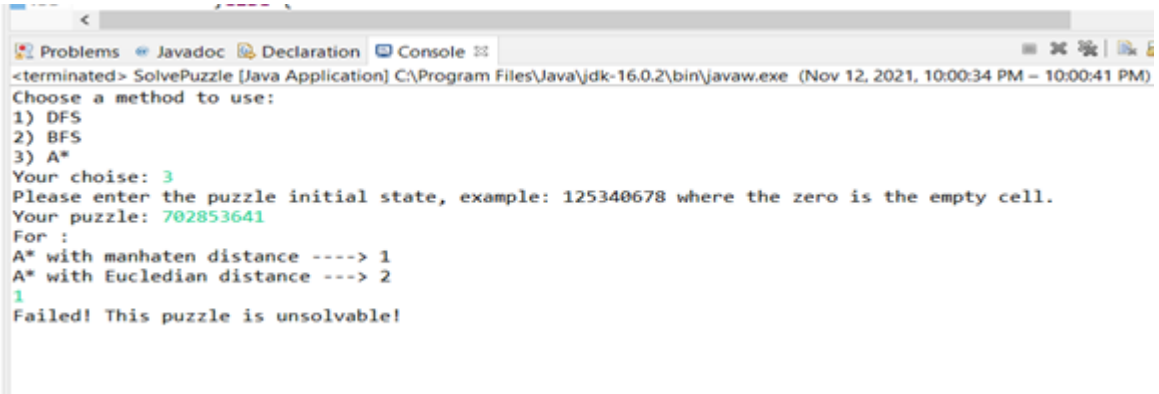
DFS



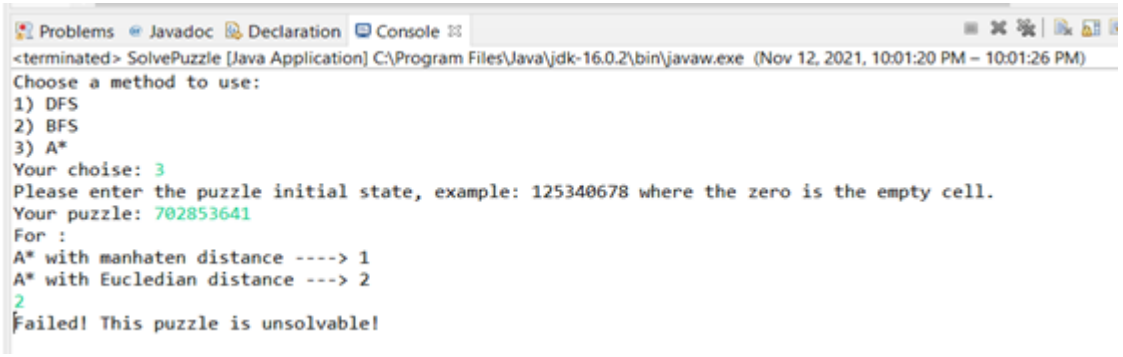
BFS



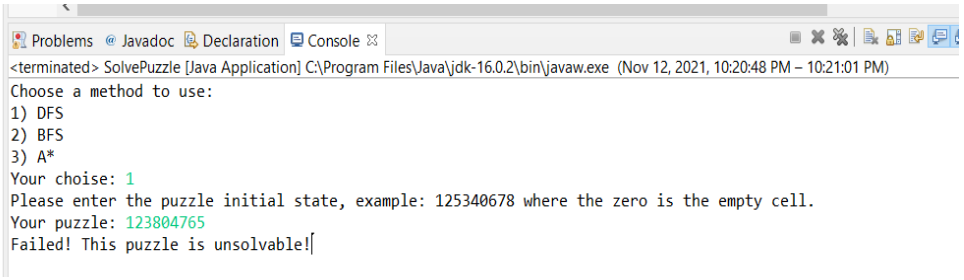
A*,Manhattan



A*,Euclidean

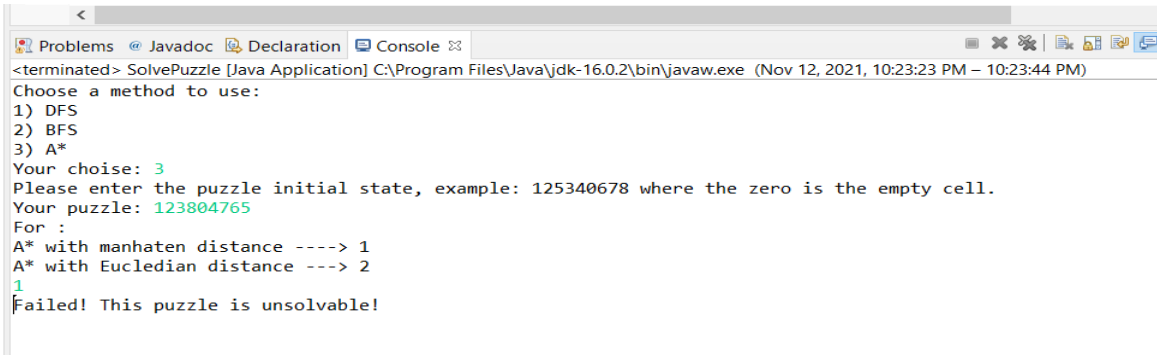


DFS

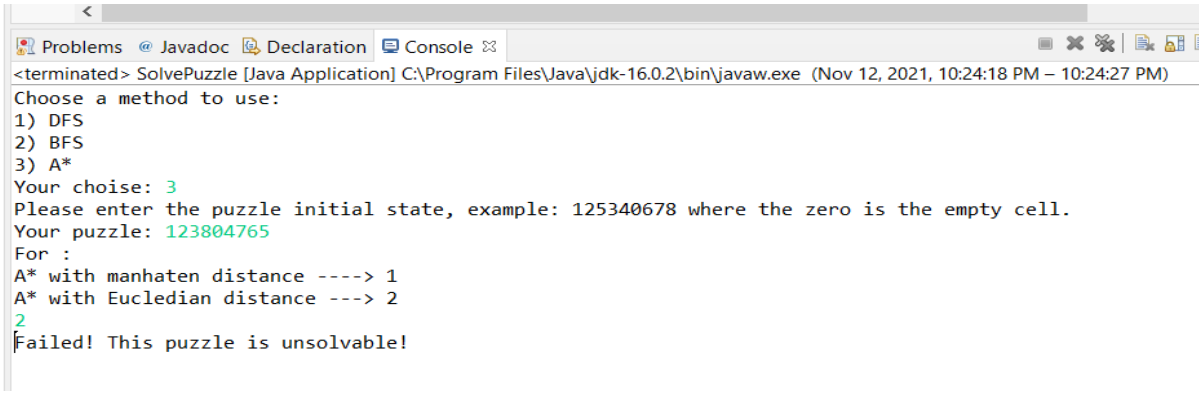


BFS

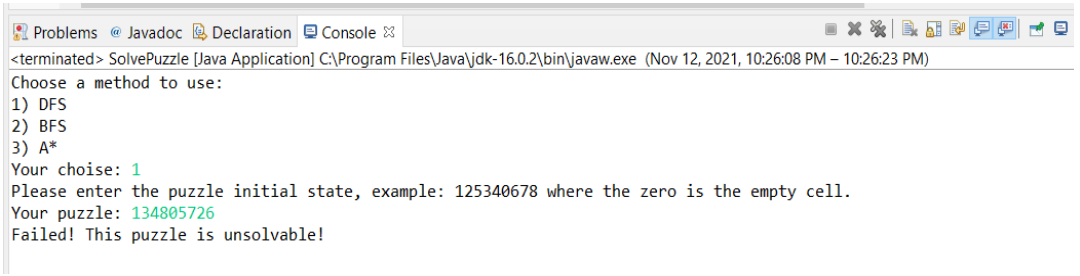
A*,Manhattan



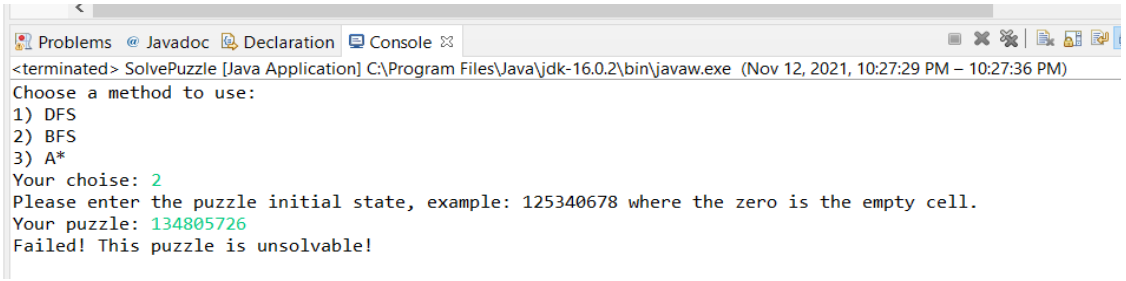
A*,Euclidean



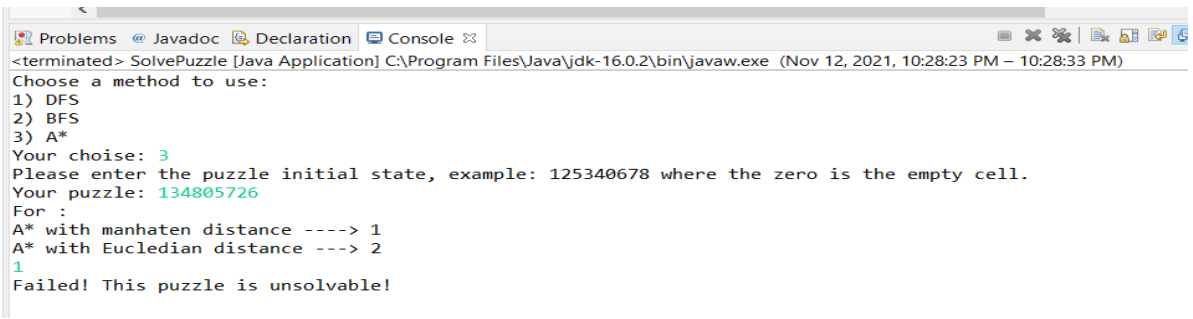
DFS



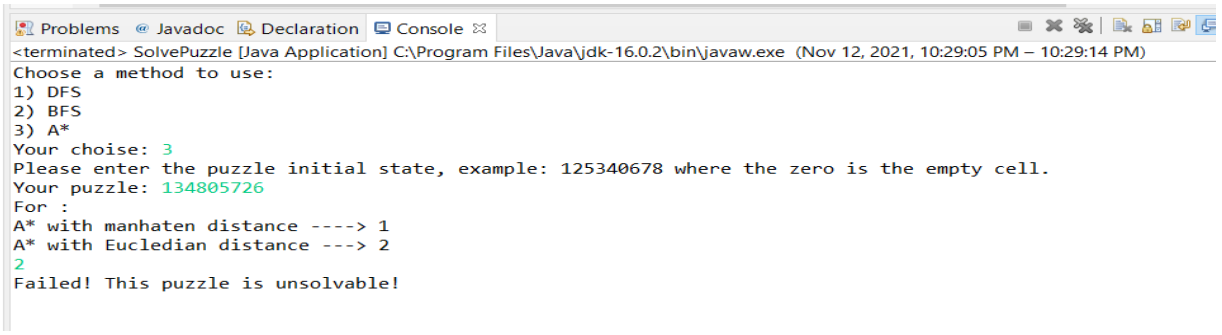
BFS



A*,Manhattan

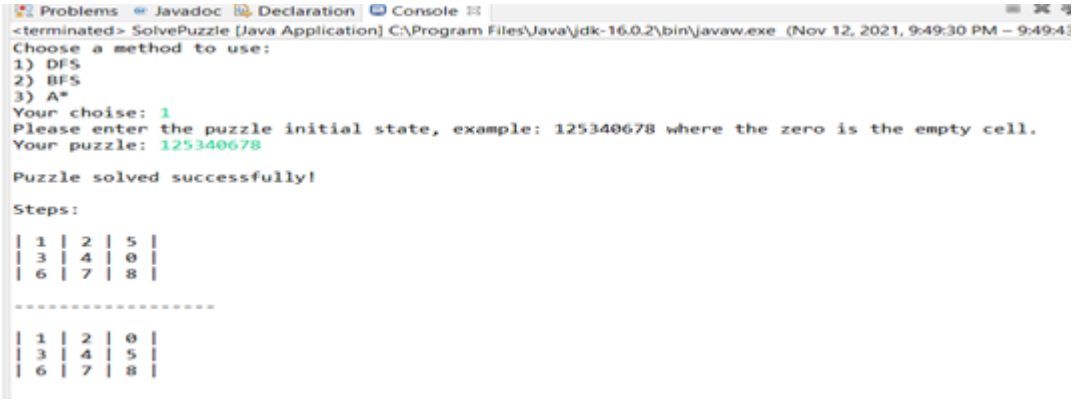


A*,Euclidean

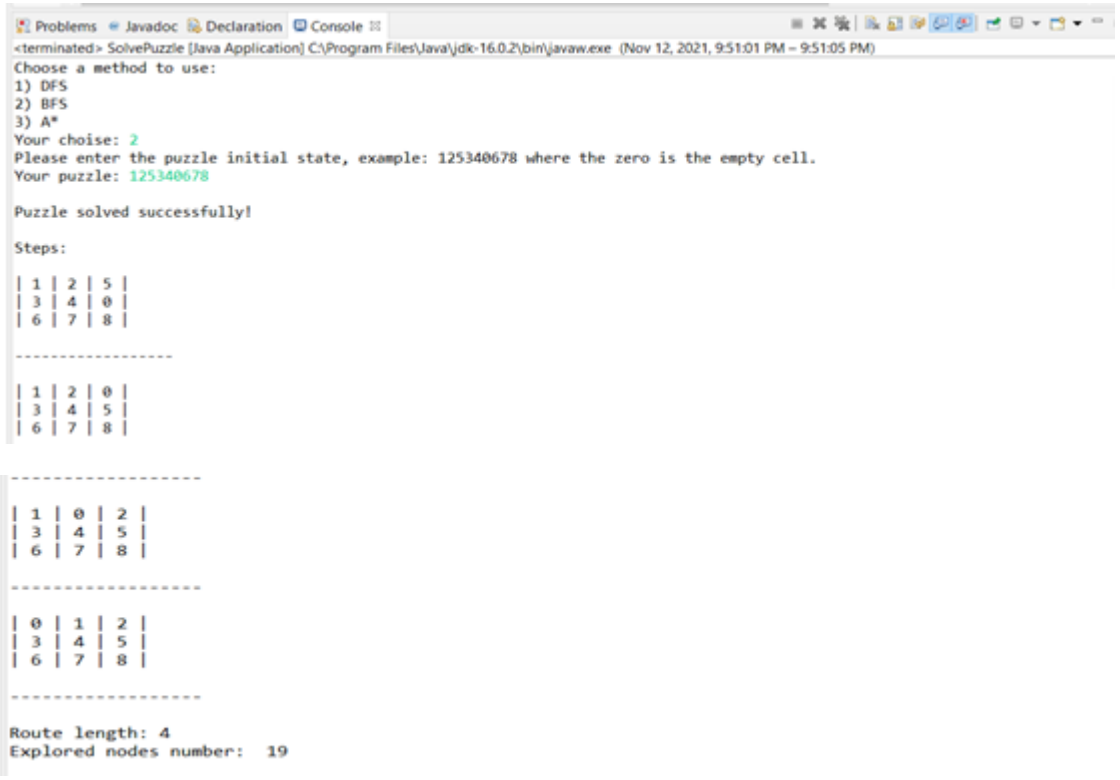


Solvable test cases

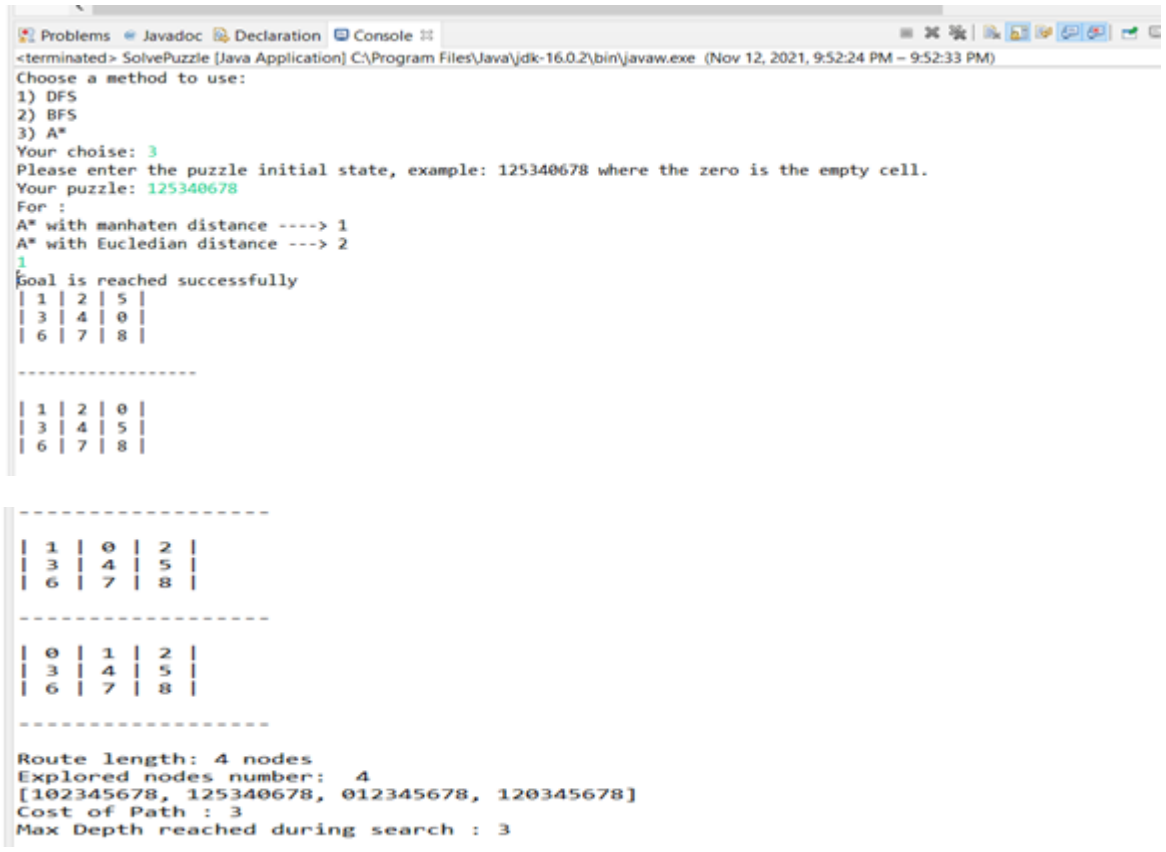
DFS



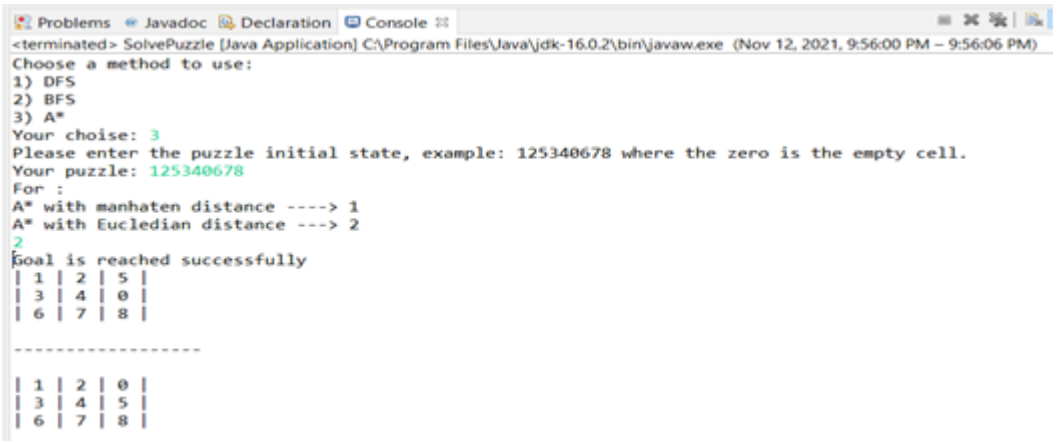
BFS

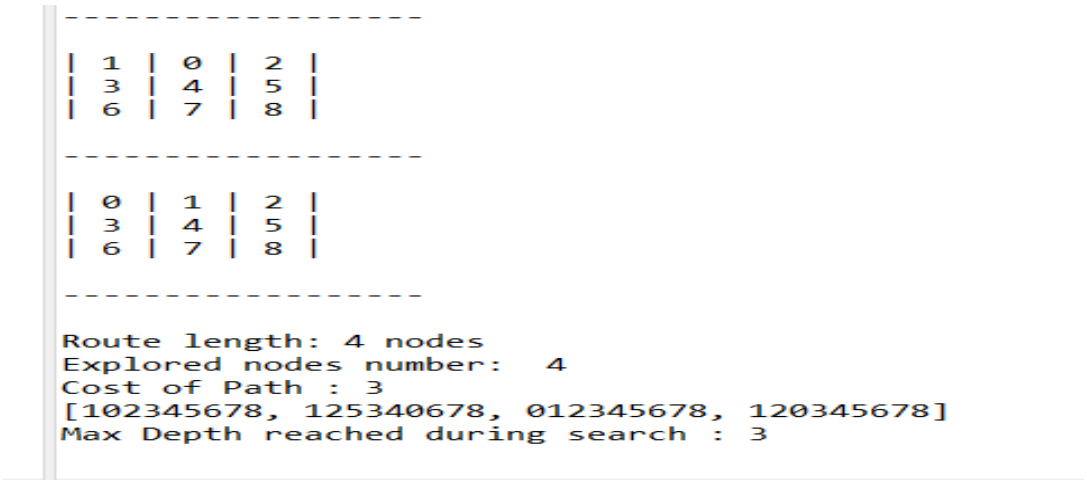

Problems Javadoc Declaration Console
<terminated> SolvePuzzle [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe (Nov 12, 2021, 9:51:01 PM – 9:51:05 PM)
Choose a method to use:
1) DFS
2) BFS
3) A*
Your choise: 2
Please enter the puzzle initial state, example: 125340678 where the zero is the empty cell.
Your puzzle: 125340678
Puzzle solved successfully!
Steps:
1	2	5
3	4	0
6	7	8
1	2	0
3	4	5
6	7	8
1	0	2
3	4	5
6	7	8
0	1	2
3	4	5
6	7	8
Route length: 4
Explored nodes number: 19

A*,Manhattan

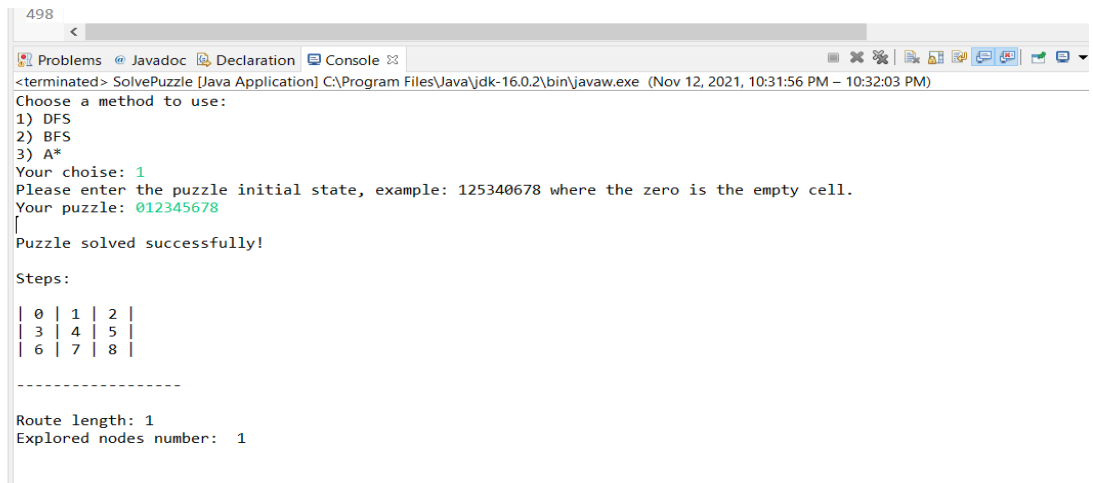

Problems Javadoc Declaration Console
<terminated> SolvePuzzle [Java Application] C:\Program Files\Java\jdk-16.0.2\bin\javaw.exe (Nov 12, 2021, 9:52:24 PM – 9:52:33 PM)
Choose a method to use:
1) DFS
2) BFS
3) A*
Your choise: 3
Please enter the puzzle initial state, example: 125340678 where the zero is the empty cell.
Your puzzle: 125340678
For :
A* with manhaten distance ----> 1
A* with Eucledian distance ---> 2
1
Goal is reached successfully
1	2	5
3	4	0
6	7	8
1	2	0
3	4	5
6	7	8
1	0	2
3	4	5
6	7	8
0	1	2
3	4	5
6	7	8
Route length: 4 nodes
Explored nodes number: 4
[102345678, 125340678, 012345678, 120345678]
Cost of Path : 3
Max Depth reached during search : 3

A*,Euclidean

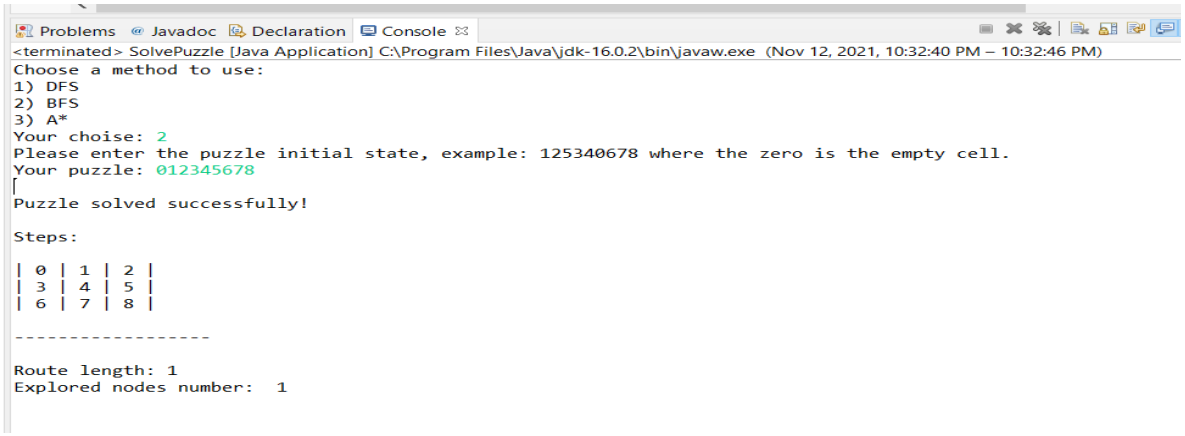




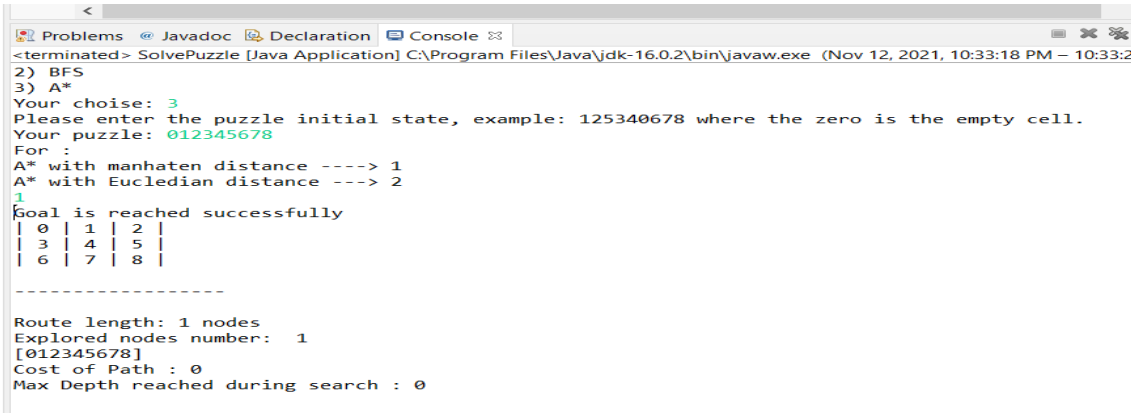
DFS



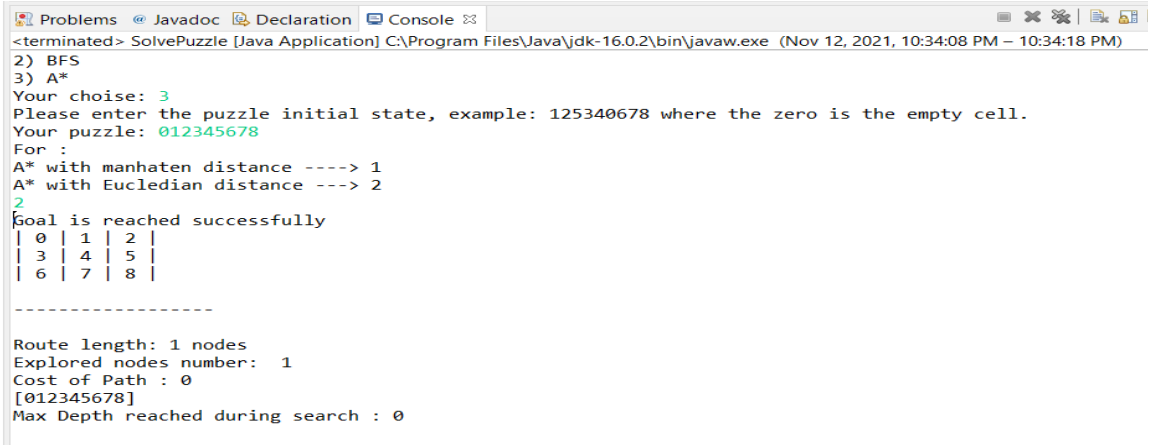
BFS



A*,Manhattan



A*,Euclidean



Case of input which needs to explore a lot of nodes to reach goal

At initial state : 647850321

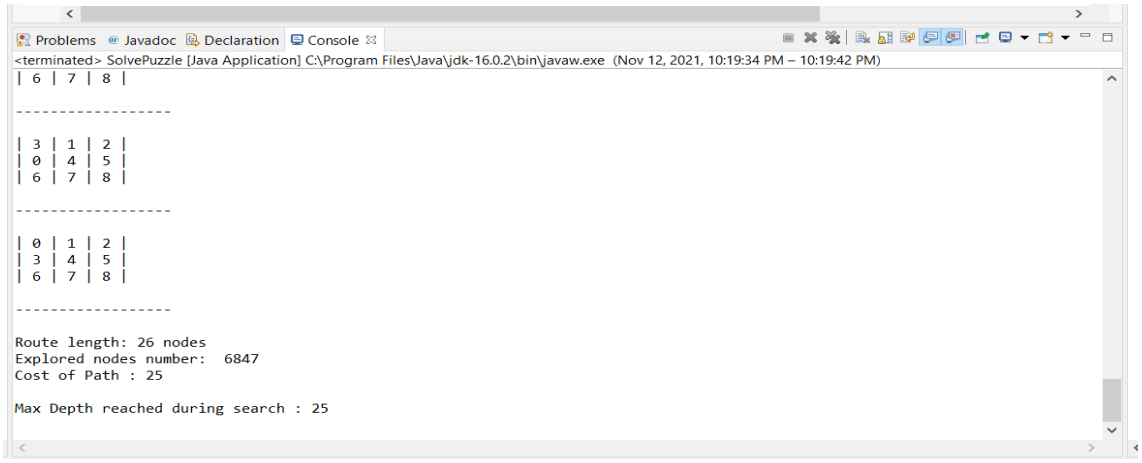
DFS

BFS

A*,Manhattan

```
| 6 | 7 | 8 |

-----------------

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

-----------------

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

-----------------

Route length: 26 nodes
Explored nodes number:  3228

Cost of Path : 25
Max Depth reached during search : 25
```

A*,Euclidean



At initial state 867254301

DFS

BFS

A*,Manhattan

A*,Euclidean