Test Report

Project Title: River Crossing Game

(with customizable puzzle and solver)

Table on Contents

1. Test Report 4

2. Methodology 5

3. Coverage Analysis 6

4. Testing details (RuleSet.java) 7

4.1. Testing function isValidMove() 8

4.1.1. Unit testing – getNextState() 8

4.1.2. Unit testing – isTravelable() 10

4.1.3. Integration Test – isValidMove() 11

4.2. Testing function isWin() 14

4.2.1. Unit testing – isIdentical() 15

4.2.2. Integration Test – isWin() 17

5. Testing details (Solver.java) 18

5.1. Testing function getLegalMove() 18

5.1.1. Unit testing – getTravelable() 19

5.1.2. Integration Test – getLegalMoves() 22

5.2. Integration test of solve() 24

6. Testing Player Mode Undo/Redo Command 25

7. System Test 27

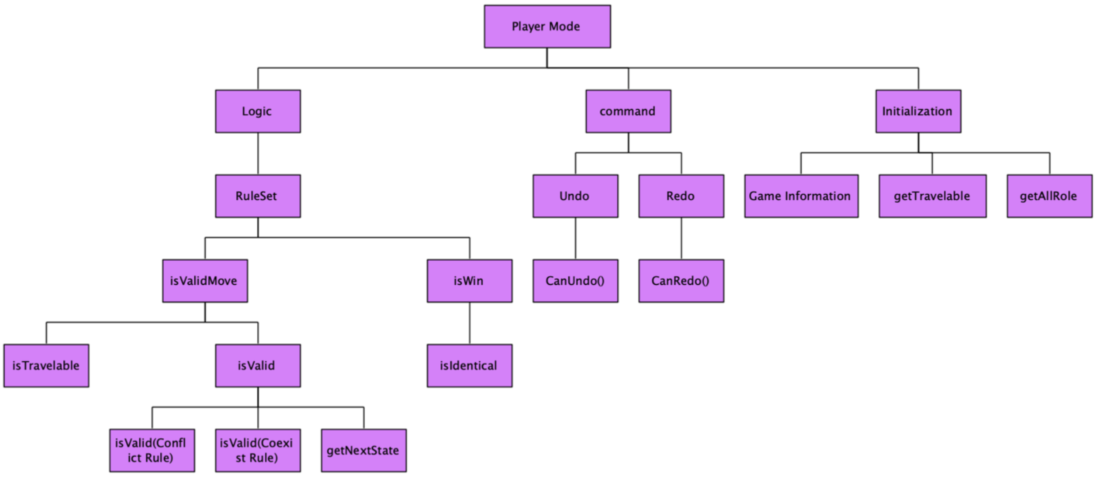
7.1. Solver Mode 27

7.2. Player Mode 29

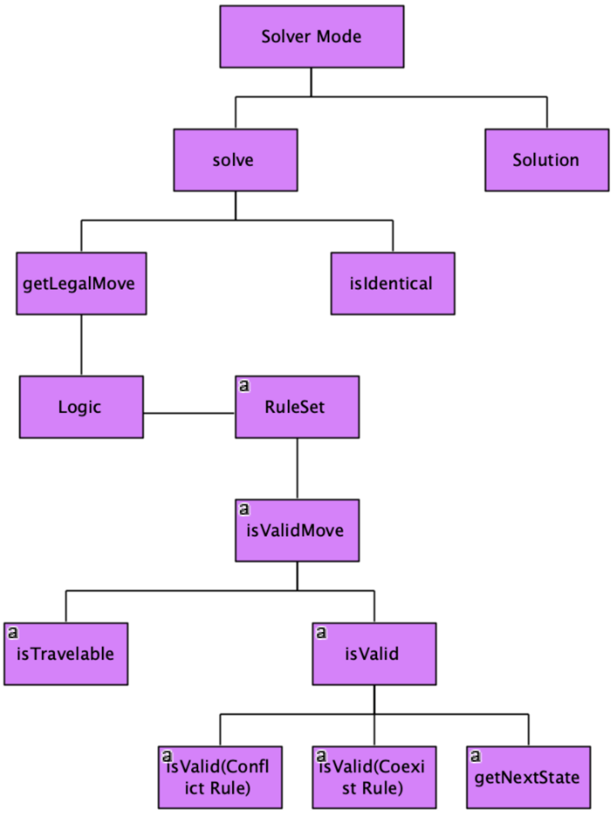
8. User acceptance testing (UAT) 30

1. Test Report

Game Mode A (Player Mode) Hierarchy Diagram



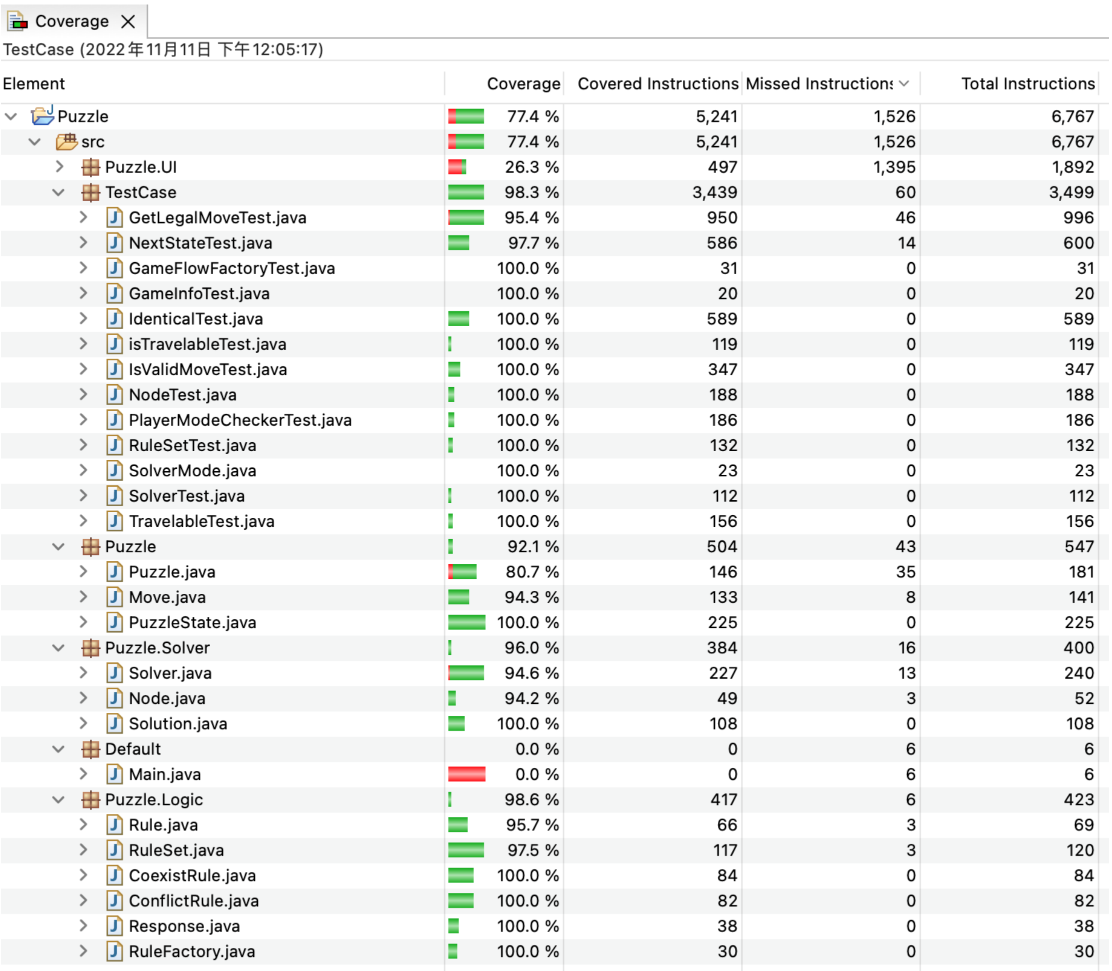
Game Mode B (Solver Mode) Hierarchy Diagram



2. Methodology

Our group has chosen Bottom-up testing strategy as our testing method. This is a kind of integration testing where low-level modules are tested first, then followed by high level module. It involves taking integrated code and test those code together, before testing a whole system. The reason why we chosen this testing method is that it is easy to develop test conditions. Besides, Disjoint subsystems can be also tested at the same time. Therefore, it can ensure all the modules inside the system are tested as a single unit. In our design code, our function output depends on different sub-function, especially the logical part in our design. To test every function thoroughly, Bottom-up testing strategy is chosen by our group.

3. Coverage Analysis



Coverage

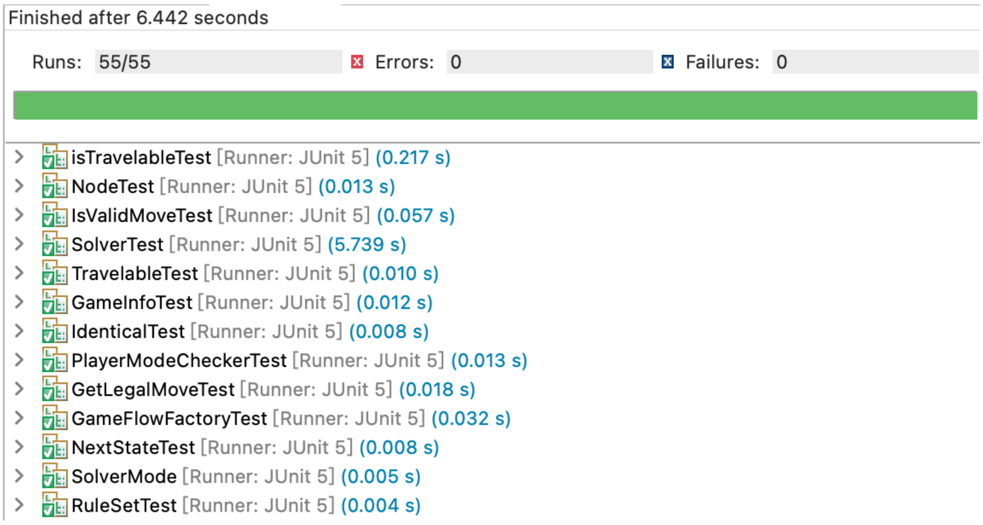
TestCase (2022年11月11日 下午12:05:17)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Puzzle	77.4 %	5,241	1,526	6,767
src	77.4 %	5,241	1,526	6,767
Puzzle.UI	26.3 %	497	1,395	1,892
TestCase	98.3 %	3,439	60	3,499
GetLegalMoveTest.java	95.4 %	950	46	996
NextStateTest.java	97.7 %	586	14	600
GameFlowFactoryTest.java	100.0 %	31	0	31
GameInfoTest.java	100.0 %	20	0	20
IdenticalTest.java	100.0 %	589	0	589
isTravelableTest.java	100.0 %	119	0	119
IsValidMoveTest.java	100.0 %	347	0	347
NodeTest.java	100.0 %	188	0	188
PlayerModeCheckerTest.java	100.0 %	186	0	186
RuleSetTest.java	100.0 %	132	0	132
SolverMode.java	100.0 %	23	0	23
SolverTest.java	100.0 %	112	0	112
TravelableTest.java	100.0 %	156	0	156
Puzzle	92.1 %	504	43	547
Puzzle.java	80.7 %	146	35	181
Move.java	94.3 %	133	8	141
PuzzleState.java	100.0 %	225	0	225
Puzzle.Solver	96.0 %	384	16	400
Solver.java	94.6 %	227	13	240
Node.java	94.2 %	49	3	52
Solution.java	100.0 %	108	0	108
Default	0.0 %	0	6	6
Main.java	0.0 %	0	6	6
Puzzle.Logic	98.6 %	417	6	423
Rule.java	95.7 %	66	3	69
RuleSet.java	97.5 %	117	3	120
CoexistRule.java	100.0 %	84	0	84
ConflictRule.java	100.0 %	82	0	82
Response.java	100.0 %	38	0	38
RuleFactory.java	100.0 %	30	0	30

The overall coverage in our testing is 77.4% (**95% if ignoring the UI**). The main reason why the coverage is under 90% is that the Graphic User interface is hard to test since it mainly depends on user's input and print out some colorful graphic such as the boat, river, and role. For example, in DrawGraphic.java, The function is only print out some symbol. It does not relate to the logical part and algorithm in our design. The purpose of implementing UI is only for better visual effect. Without considering the user

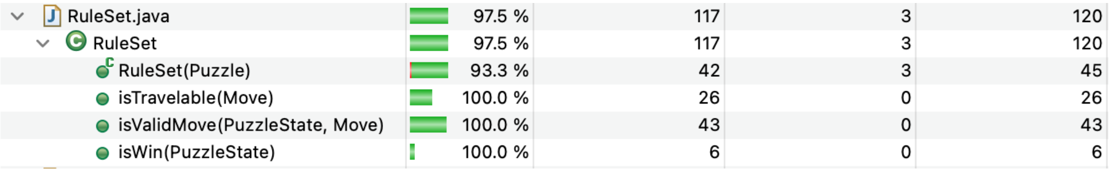
Interface part, the overall coverage is above **95%** if we focus on the logic and algorithm part in our design.

Total number of test Case: 55



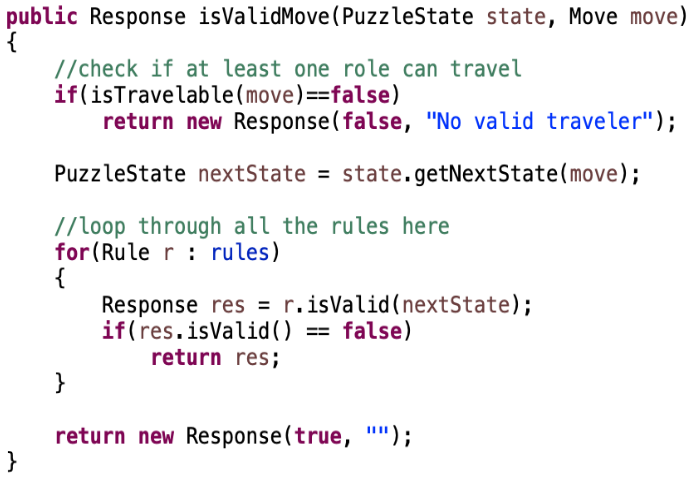
```
Finished after 6.442 seconds

Runs:  55/55        Errors:  0        Failures:  0

> isTravelableTest [Runner: JUnit 5] (0.217 s)
> NodeTest [Runner: JUnit 5] (0.013 s)
> IsValidMoveTest [Runner: JUnit 5] (0.057 s)
> SolverTest [Runner: JUnit 5] (5.739 s)
> TravelableTest [Runner: JUnit 5] (0.010 s)
> GameInfoTest [Runner: JUnit 5] (0.012 s)
> IdenticalTest [Runner: JUnit 5] (0.008 s)
> PlayerModeCheckerTest [Runner: JUnit 5] (0.013 s)
> GetLegalMoveTest [Runner: JUnit 5] (0.018 s)
> GameFlowFactoryTest [Runner: JUnit 5] (0.032 s)
> NextStateTest [Runner: JUnit 5] (0.008 s)
> SolverMode [Runner: JUnit 5] (0.005 s)
> RuleSetTest [Runner: JUnit 5] (0.004 s)
```

4. Testing details (RuleSet.java)



RuleSet.java	97.5 %	117	3	120
RuleSet	97.5 %	117	3	120
RuleSet(Puzzle)	93.3 %	42	3	45
isTravelable(Move)	100.0 %	26	0	26
isValidMove(PuzzleState, Move)	100.0 %	43	0	43
isWin(PuzzleState)	100.0 %	6	0	6

4.1. Testing function isValidMove()



```
public Response isValidMove(PuzzleState state, Move move)
{
    //check if at least one role can travel
    if(isTravelable(move)==false)
        return new Response(false, "No valid traveler");

    PuzzleState nextState = state.getNextState(move);

    //loop through all the rules here
    for(Rule r : rules)
    {
        Response res = r.isValid(nextState);
        if(res.isValid() == false)
            return res;
    }

    return new Response(true, "");
}
```

The main function in RuleSet.java is function - isValidMove(PuzzleState, Move). This function returns a response whether the move is valid or not. The return value depends on different components and functions. They are isTravelable(), isValid() and getNextState(). Following the principle of Bottom-up testing, we should first test these three functions separately first, then perform a integration testing on function isValidMove().

4.1.1. Unit testing – getNextState()

This function accept a move from user and act as a parameter for function isValid().

The testing method for this function is testing all the combination.

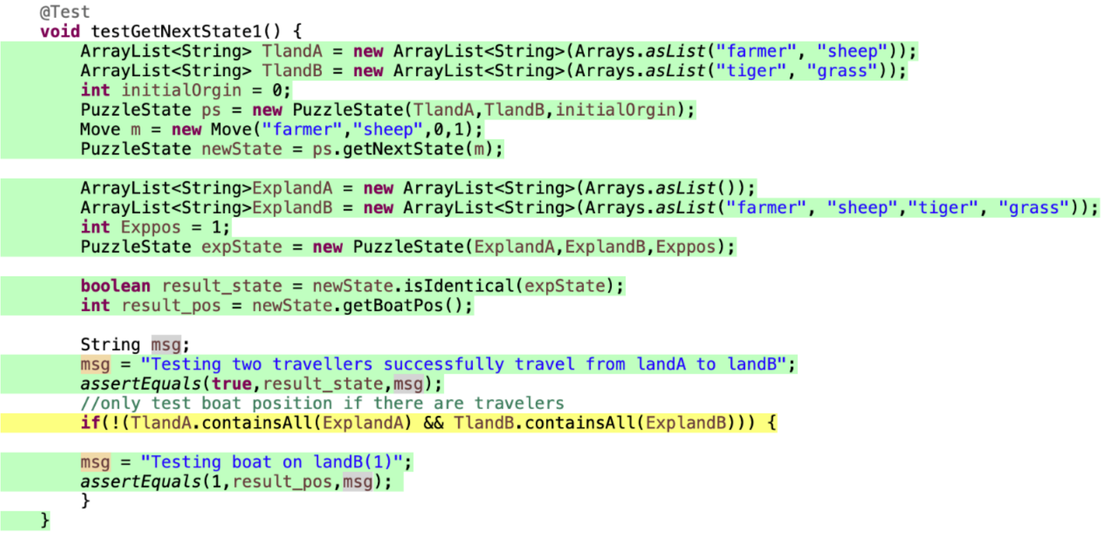
The combination is as following:

1. Two roles cross the river from land A to land B.
2. Only one role crosses the river from land A to land B.
3. No role crosses the river
4. Two roles cross the river from land B to land A.

5. Only one role crosses the river from land A to land B.

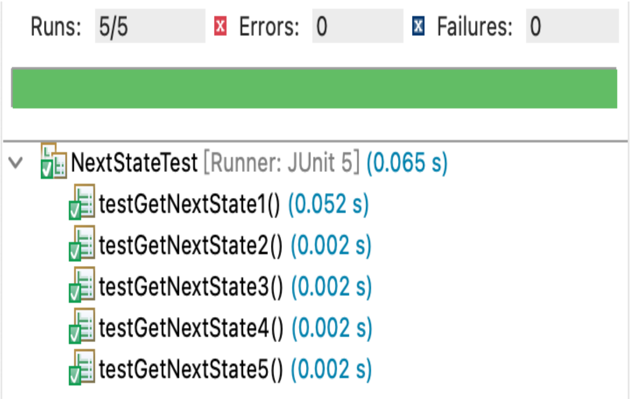
Example:

This is a test case simulating two roles “farmer” and “sheep” cross the river from land A and land B successfully.



```
@Test
void testGetNextState1() {
    ArrayList<String> TlandA = new ArrayList<String>(Arrays.asList("farmer", "sheep"));
    ArrayList<String> TlandB = new ArrayList<String>(Arrays.asList("tiger", "grass"));
    int initialOrgin = 0;
    PuzzleState ps = new PuzzleState(TlandA,TlandB,initialOrgin);
    Move m = new Move("farmer","sheep",0,1);
    PuzzleState newState = ps.getNextState(m);

    ArrayList<String>ExplandA = new ArrayList<String>(Arrays.asList());
    ArrayList<String>ExplandB = new ArrayList<String>(Arrays.asList("farmer", "sheep","tiger", "grass"));
    int Exppos = 1;
    PuzzleState expState = new PuzzleState(ExplandA,ExplandB,Exppos);

    boolean result_state = newState.isIdentical(expState);
    int result_pos = newState.getBoatPos();

    String msg;
    msg = "Testing two travellers successfully travel from landA to landB";
    assertEquals(true,result_state,msg);
    //only test boat position if there are travelers
    if(!(TlandA.containsAll(ExplandA) && TlandB.containsAll(ExplandB))) {

    msg = "Testing boat on landB(1)";
    assertEquals(1,result_pos,msg);
    }
}
```

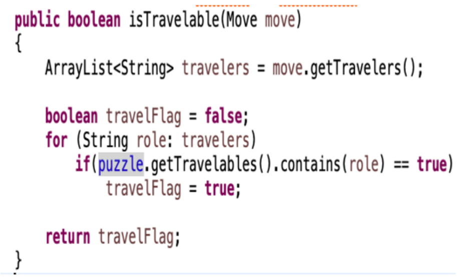
Coverage:



```
Runs: 5/5    Errors: 0    Failures: 0

NextStateTest [Runner: JUnit 5] (0.065 s)
    testGetNextState1() (0.052 s)
    testGetNextState2() (0.002 s)
    testGetNextState3() (0.002 s)
    testGetNextState4() (0.002 s)
    testGetNextState5() (0.002 s)
```

4.1.2. Unit testing – isTravelable()

This function accepts a move from user and check whether roles in this move is travelable or not. The testing method for this function is testing all the combination.



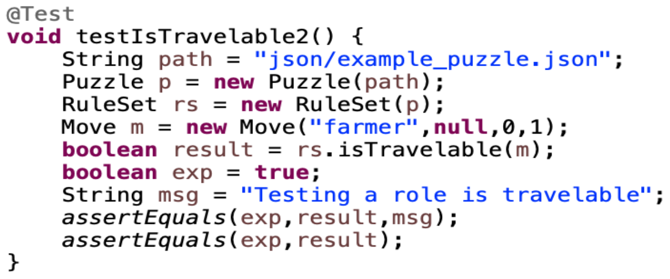
```
public boolean isTravelable(Move move)
{
    ArrayList<String> travelers = move.getTravelers();

    boolean travelFlag = false;
    for (String role: travelers)
        if(puzzle.getTravelables().contains(role) == true)
            travelFlag = true;

    return travelFlag;
}
```

The combination is as following:

1. The move involves at least one traveler
2. The move involves no traveler
3. The move is not valid

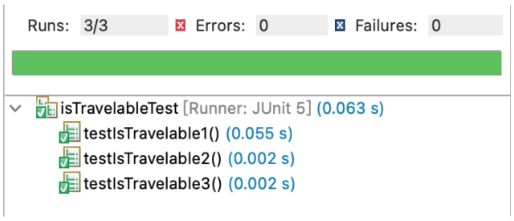
Example:

This is a test case simulating at least one role “farmer” is on the travelable list, so the function returns true.



```
@Test
void testIsTravelable2() {
    String path = "json/example_puzzle.json";
    Puzzle p = new Puzzle(path);
    RuleSet rs = new RuleSet(p);
    Move m = new Move("farmer",null,0,1);
    boolean result = rs.isTravelable(m);
    boolean exp = true;
    String msg = "Testing a role is travelable";
    assertEquals(exp,result,msg);
    assertEquals(exp,result);
}
```

Coverage:



Runs: 3/3 Errors: 0 Failures: 0

isTravelableTest [Runner: JUnit 5] (0.063 s)
testIsTravelable1() (0.055 s)
testIsTravelable2() (0.002 s)
testIsTravelable3() (0.002 s)

4.1.3. Integration Test – isValidMove()

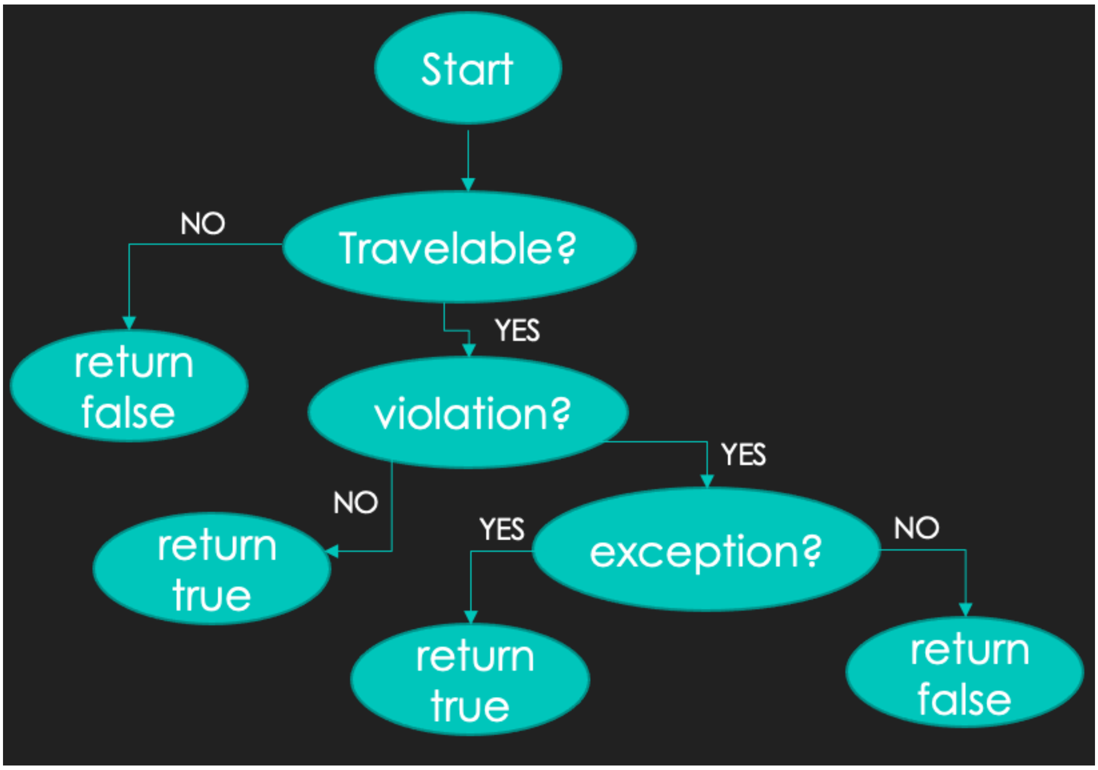
In isValidMove() function, there are total of four path. Our test case has covered all the paths.

Path 1: Start -> Travelable? -> False

Path 2: Start -> Travelable? -> Violation? -> True

Path 3: Start -> Travelable? -> Violation? -> Exception? -> True

Path 3: Start -> Travelable? -> Violation? -> Exception? -> False



Since we have tested isTravelable() and getNextState(), the last step we test is isValid(). isValid() test whether the state violate the conflict or coexist rule. The game rule we decided is as following: If the move violate conflict or coexist rule with exception, then it will return true. Otherwise, it returns false. Therefore, we can design the test case with the truth table below.

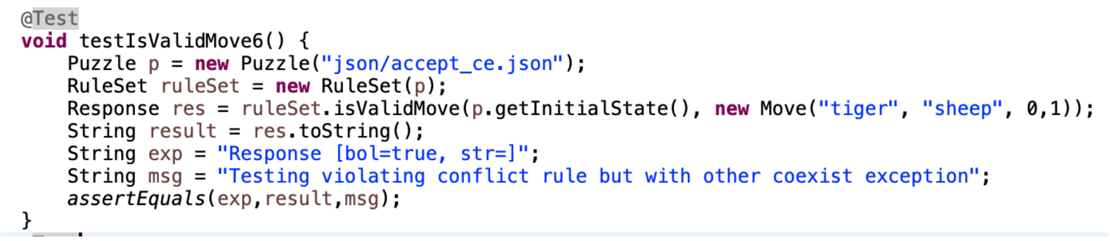
Predicate testing:

Test Case	Travelable?	Valid State?	Exception?	Result
1	True	False	False	False

		(Violate conflict rule)	(Conflict rule exception)	
2	True	False (Violate conflict rule)	True (Conflict rule exception)	True
3	True	False (Violate coexist rule)	False (Conflict rule exception)	False
4	True	False (Violate coexist rule)	True (Conflict rule exception)	True
5	True	False (Violate conflict rule)	False (Coexist rule exception)	False
6	True	False (Violate conflict rule)	True (Coexist rule exception)	True
7	True	False (Violate coexist rule)	False (Coexist rule exception)	False
8	True	False (Violate coexist rule)	True (Coexist rule exception)	True
9	False	-	-	False
10	True	True (No rule violation)	-	True

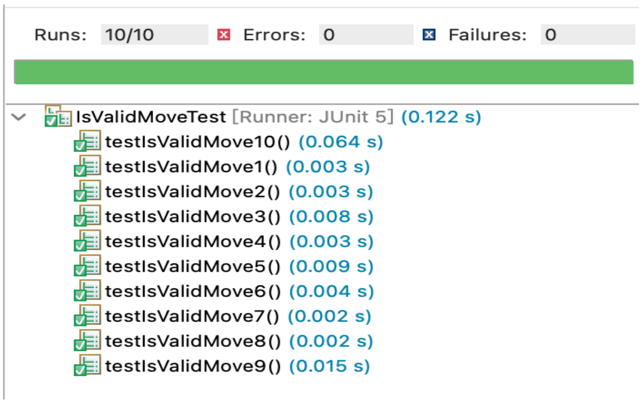
Example:

This is a test case simulating there is violation of conflict rule but with exception of coexist rule.



```
@Test
void testIsValidMove6() {
    Puzzle p = new Puzzle("json/accept_ce.json");
    RuleSet ruleSet = new RuleSet(p);
    Response res = ruleSet.isValidMove(p.getInitialState(), new Move("tiger", "sheep", 0,1));
    String result = res.toString();
    String exp = "Response [bol=true, str=]";
    String msg = "Testing violating conflict rule but with other coexist exception";
    assertEquals(exp,result,msg);
}
```

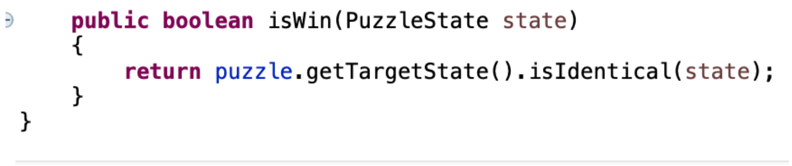
Integration test coverage



```
Runs: 10/10   ☒ Errors: 0   ☒ Failures: 0

IsValidMoveTest [Runner: JUnit 5] (0.122 s)
    testIsValidMove10() (0.064 s)
    testIsValidMove1() (0.003 s)
    testIsValidMove2() (0.003 s)
    testIsValidMove3() (0.008 s)
    testIsValidMove4() (0.003 s)
    testIsValidMove5() (0.009 s)
    testIsValidMove6() (0.004 s)
    testIsValidMove7() (0.002 s)
    testIsValidMove8() (0.002 s)
    testIsValidMove9() (0.015 s)
```

4.2. Testing function isWin()

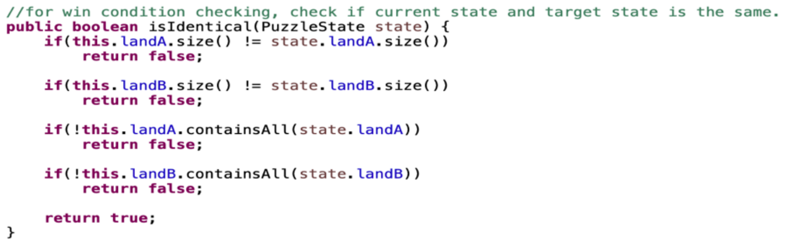
The Win() function is depend on the return value of getTargetState() function inside other class (PuzzleState.java). Therefore, we will first test the function isIdentical() inside PuzzleState.java.



```
    public boolean isWin(PuzzleState state)
    {
        return puzzle.getTargetState().isIdentical(state);
    }
}
```

4.2.1. Unit testing – isIdentical()

The function accepts a parameter-PuzzleState and return true if current land A is equal to final land A and current land B is equal to final land B. Otherwise, it returns false. In our test case, we will consider every if statement inside the isIdentical() function.



```
//for win condition checking, check if current state and target state is the same.
public boolean isIdentical(PuzzleState state) {
    if(this.landA.size() != state.landA.size())
        return false;

    if(this.landB.size() != state.landB.size())
        return false;

    if(!this.landA.containsAll(state.landA))
        return false;

    if(!this.landB.containsAll(state.landB))
        return false;

    return true;
}
```

There are total of 6 combinations. They are as following:

1. Final land B is equal to current land B but Final land A is not equal to current land A

2. Final land A is equal to current land A but Final land B is not equal to current land B

3. Final land A is not equal to current land A and Final land B is not equal to current land B

4. Final land B size is not equal to current land B

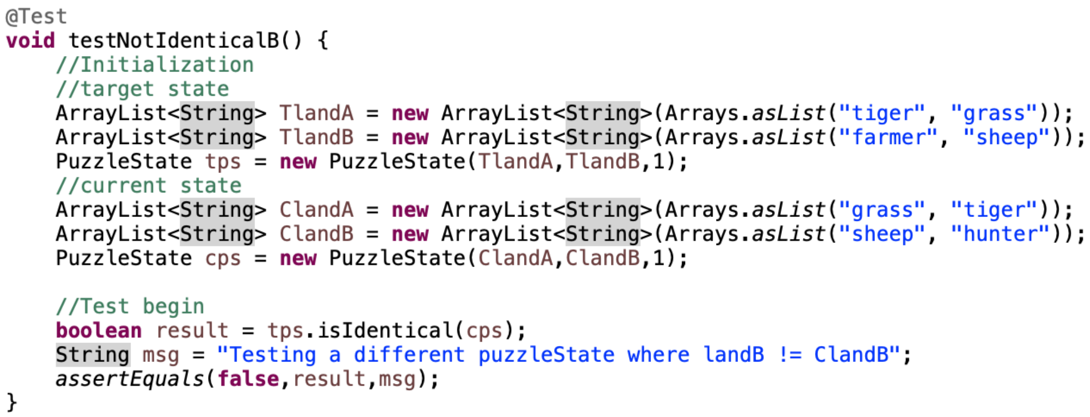
5. Final land A size is not equal to current land A

6. Final land A and land B is equal to current land A and land B

Besides, we have also considered the case of empty puzzleState which mean there is no roles exist in land A and land B respectively.

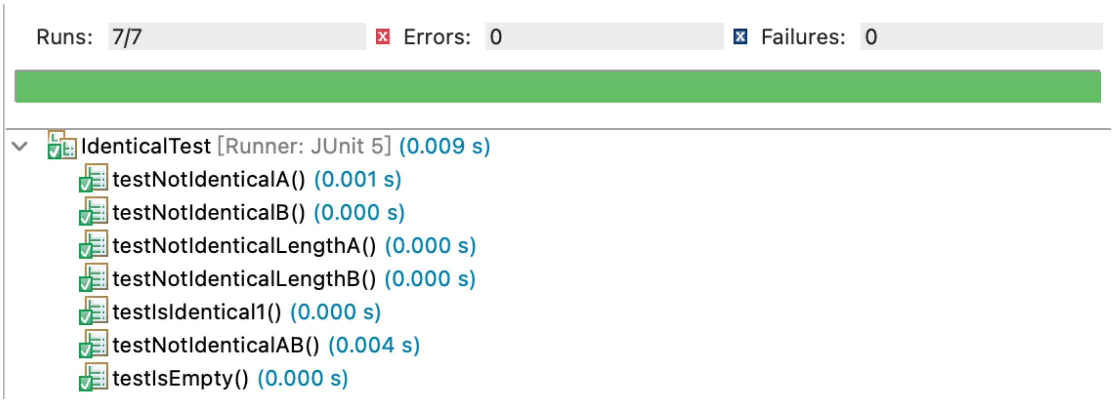
Example:

This is a test case simulating final land B is not identical to current land B so the return value is false.



```
@Test
void testNotIdenticalB() {
    //Initialization
    //target state
    ArrayList<String> TlandA = new ArrayList<String>(Arrays.asList("tiger", "grass"));
    ArrayList<String> TlandB = new ArrayList<String>(Arrays.asList("farmer", "sheep"));
    PuzzleState tps = new PuzzleState(TlandA,TlandB,1);
    //current state
    ArrayList<String> ClandA = new ArrayList<String>(Arrays.asList("grass", "tiger"));
    ArrayList<String> ClandB = new ArrayList<String>(Arrays.asList("sheep", "hunter"));
    PuzzleState cps = new PuzzleState(ClandA,ClandB,1);

    //Test begin
    boolean result = tps.isIdentical(cps);
    String msg = "Testing a different puzzleState where landB != ClandB";
    assertEquals(false,result,msg);
}
```

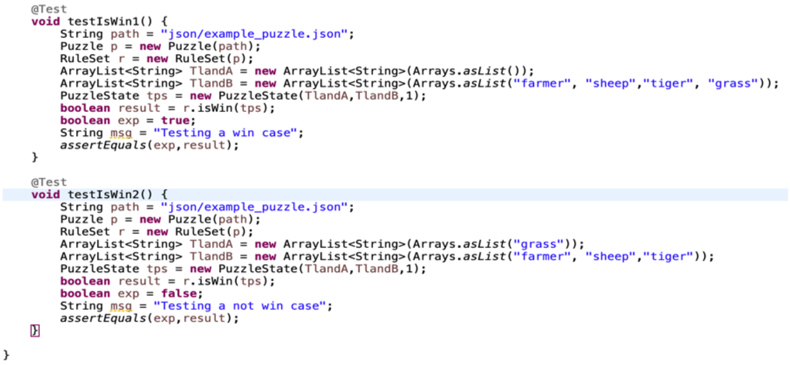
Coverage:



```
Runs:  7/7          ☒ Errors:  0          ☒ Failures:  0

IdenticalTest [Runner: JUnit 5] (0.009 s)
    testNotIdenticalA() (0.001 s)
    testNotIdenticalB() (0.000 s)
    testNotIdenticalLengthA() (0.000 s)
    testNotIdenticalLengthB() (0.000 s)
    testIsIdentical1() (0.000 s)
    testNotIdenticalAB() (0.004 s)
    testIsEmpty() (0.000 s)
```

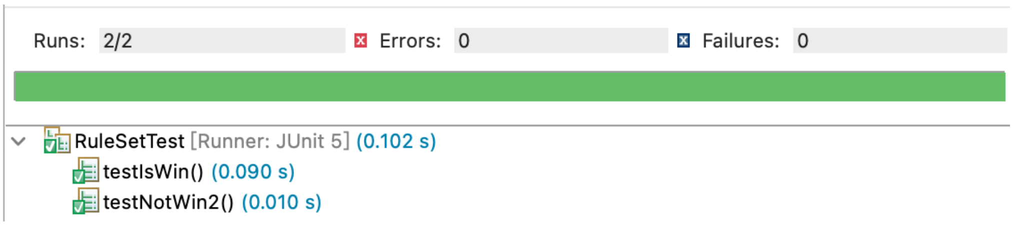
4.2.2. Integration Test – isWin()

Since the isWin() function is only depend on isIdentical() so we need only two test cases (is win case / not win case) to test the function.



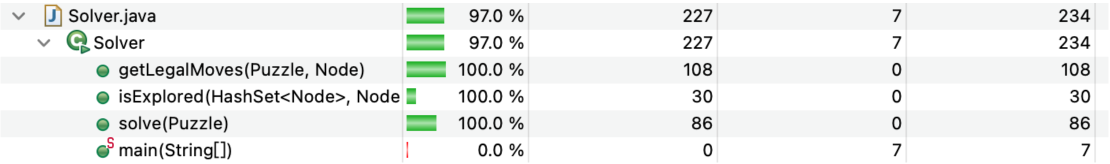
```
    @Test
    void testIsWin1() {
        String path = "json/example_puzzle.json";
        Puzzle p = new Puzzle(path);
        RuleSet r = new RuleSet(p);
        ArrayList<String> TlandA = new ArrayList<String>(Arrays.asList());
        ArrayList<String> TlandB = new ArrayList<String>(Arrays.asList("farmer", "sheep","tiger", "grass"));
        PuzzleState tps = new PuzzleState(TlandA,TlandB,1);
        boolean result = r.isWin(tps);
        boolean exp = true;
        String msg = "Testing a win case";
        assertEquals(exp,result);
    }

    @Test
    void testIsWin2() {
        String path = "json/example_puzzle.json";
        Puzzle p = new Puzzle(path);
        RuleSet r = new RuleSet(p);
        ArrayList<String> TlandA = new ArrayList<String>(Arrays.asList("grass"));
        ArrayList<String> TlandB = new ArrayList<String>(Arrays.asList("farmer", "sheep","tiger"));
        PuzzleState tps = new PuzzleState(TlandA,TlandB,1);
        boolean result = r.isWin(tps);
        boolean exp = false;
        String msg = "Testing a not win case";
        assertEquals(exp,result);
    }

}
```

Coverage:



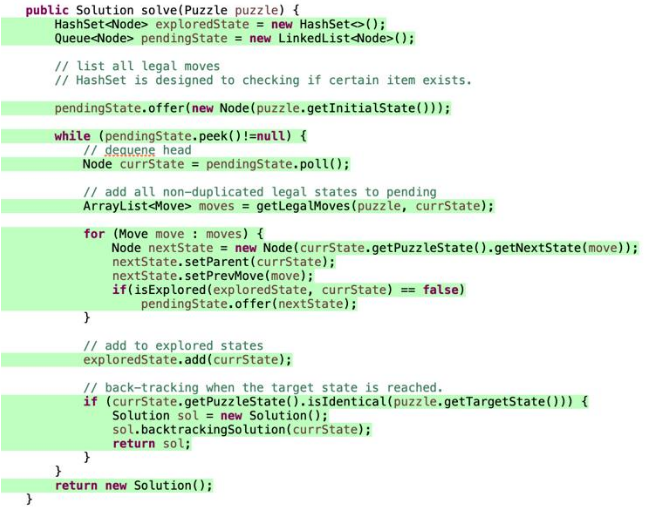
```
Runs:  2/2          ☒ Errors:  0          ☒ Failures:  0

RuleSetTest [Runner: JUnit 5] (0.102 s)
    testIsWin() (0.090 s)
    testNotWin2() (0.010 s)
```

5. Testing details (Solver.java)



Solver.java	97.0 %	227	7	234
Solver	97.0 %	227	7	234
getLegalMoves(Puzzle, Node)	100.0 %	108	0	108
isExplored(HashSet<Node>, Node	100.0 %	30	0	30
solve(Puzzle)	100.0 %	86	0	86
main(String[])	0.0 %	0	7	7

The main function inside Solver.java is solve () function which returns a solution of river crossing game with input customized role and rule. The solution is mainly depend on the getLegalMove() function since it records every moves occur in a solution.



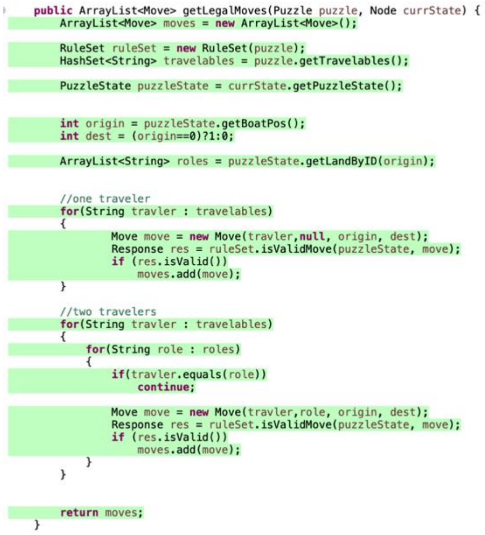
```
public Solution solve(Puzzle puzzle) {
    HashSet<Node> exploredState = new HashSet<>();
    Queue<Node> pendingState = new LinkedList<Node>();

    // list all legal moves
    // HashSet is designed to checking if certain item exists.

    pendingState.offer(new Node(puzzle.getInitialState()));

    while (pendingState.peek()!=null) {
        // dequene head
        Node currState = pendingState.poll();

        // add all non-duplicated legal states to pending
        ArrayList<Move> moves = getLegalMoves(puzzle, currState);

        for (Move move : moves) {
            Node nextState = new Node(currState.getPuzzleState().getNextState(move));
            nextState.setParent(currState);
            nextState.setPrevMove(move);
            if(isExplored(exploredState, currState) == false)
                pendingState.offer(nextState);
        }

        // add to explored states
        exploredState.add(currState);

        // back-tracking when the target state is reached.
        if (currState.getPuzzleState().isIdentical(puzzle.getTargetState())) {
            Solution sol = new Solution();
            sol.backtrackingSolution(currState);
            return sol;
        }
    }
    return new Solution();
}
```

5.1. Testing function getLegalMove()

Inside the function, it mainly depends on two functions which are getTravelable() and isValidMove(). Since isValidMove() is tested in RuleSet.java so we only need to perform unit testing on getTravelable() function and then perform integration testing of getLegalMoves().

5.1.1. Unit testing – getTravelable()

Since getTravelable() is only to get the travelable role in the json file. It means it highly depend on the input by players. We don't know what players will input inside the json file. Therefore, we have tested all possible combination and description of every test case is as following:

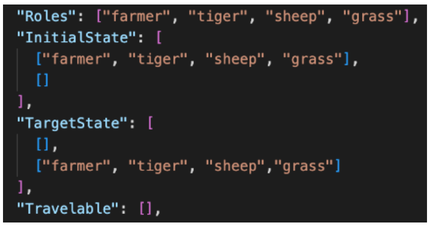


```
    public ArrayList<Move> getLegalMoves(Puzzle puzzle, Node currState) {
        ArrayList<Move> moves = new ArrayList<Move>();

        RuleSet ruleSet = new RuleSet(puzzle);
        HashSet<String> travelables = puzzle.getTravelables();

        PuzzleState puzzleState = currState.getPuzzleState();


        int origin = puzzleState.getBoatPos();
        int dest = (origin==0)?1:0;

        ArrayList<String> roles = puzzleState.getLandByID(origin);


        //one traveler
        for(String travler : travelables)
        {
                Move move = new Move(travler,null, origin, dest);
                Response res = ruleSet.isValidMove(puzzleState, move);
                if (res.isValid())
                    moves.add(move);
        }

        //two travelers
        for(String travler : travelables)
        {
            for(String role : roles)
            {
                if(travler.equals(role))
                    continue;

                Move move = new Move(travler,role, origin, dest);
                Response res = ruleSet.isValidMove(puzzleState, move);
                if (res.isValid())
                    moves.add(move);
            }
        }


        return moves;
    }
```

Test case 1:

Testing there is only one travelable character.

Json testing file:

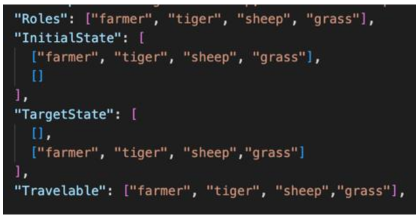


```
"Roles": ["farmer", "tiger", "sheep", "grass"],
"InitialState": [
  ["farmer", "tiger", "sheep", "grass"],
  []
],
"TargetState": [
  [],
  ["farmer", "tiger", "sheep","grass"]
],
"Travelable": [],
```

Test case 2:

Testing no travelable character.

Json testing file:

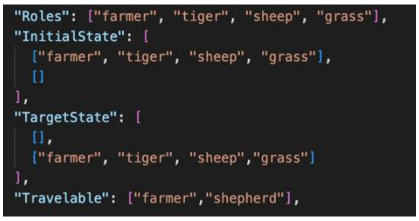


```
"Roles": ["farmer", "tiger", "sheep", "grass"],
"InitialState": [
  ["farmer", "tiger", "sheep", "grass"],
  []
],
"TargetState": [
  [],
  ["farmer", "tiger", "sheep","grass"]
],
"Travelable": ["farmer", "tiger", "sheep","grass"],
```

Test case 3:

Testing all character is travelable character.

Json testing file:

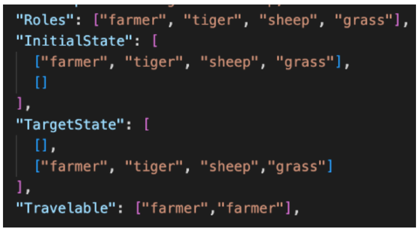


```
"Roles": ["farmer", "tiger", "sheep", "grass"],
"InitialState": [
  ["farmer", "tiger", "sheep", "grass"],
  []
],
"TargetState": [
  [],
  ["farmer", "tiger", "sheep","grass"]
],
"Travelable": ["farmer","shepherd"],
```

Test case 4:

Testing a non-exist travelable character.

Json testing file:

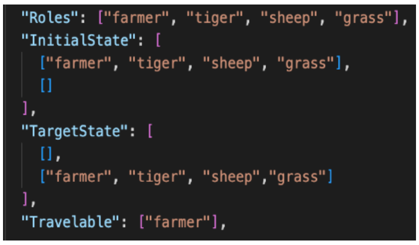


```
"Roles": ["farmer", "tiger", "sheep", "grass"],
"InitialState": [
  ["farmer", "tiger", "sheep", "grass"],
  []
],
"TargetState": [
  [],
  ["farmer", "tiger", "sheep","grass"]
],
"Travelable": ["farmer","farmer"],
```

Test case 5:

Testing a duplicated travelable character.

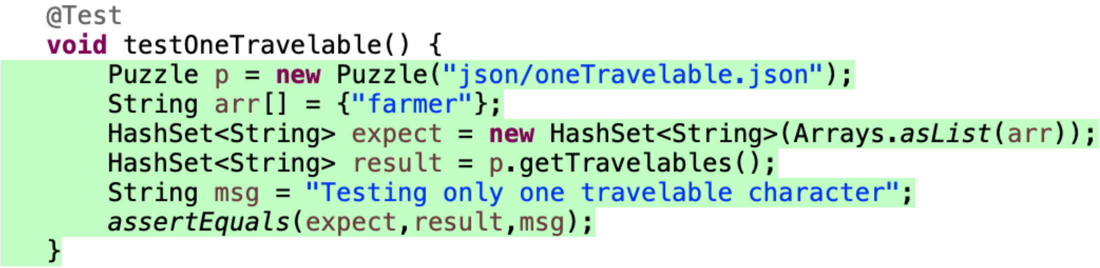
Json testing file:



```
"Roles": ["farmer", "tiger", "sheep", "grass"],
"InitialState": [
  ["farmer", "tiger", "sheep", "grass"],
  []
],
"TargetState": [
  [],
  ["farmer", "tiger", "sheep","grass"]
],
"Travelable": ["farmer"],
```

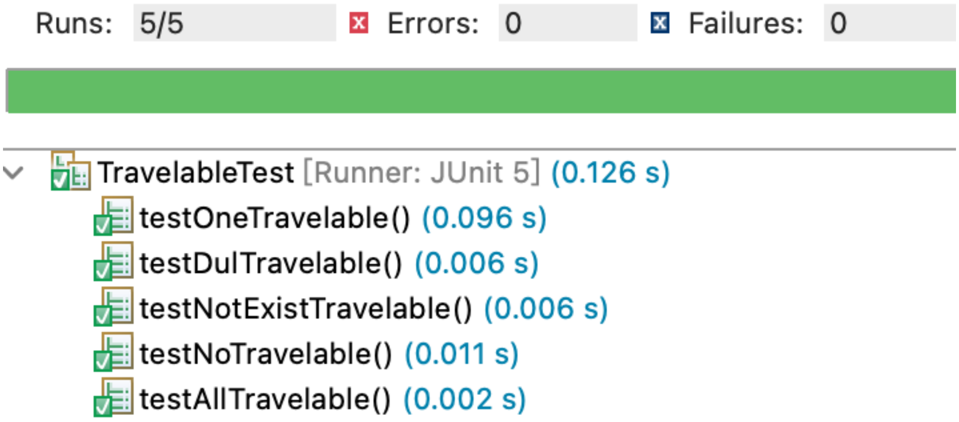
Example:

This is a test case simulating there are only one travelable character.



```
@Test
void testOneTravelable() {
    Puzzle p = new Puzzle("json/oneTravelable.json");
    String arr[] = {"farmer"};
    HashSet<String> expect = new HashSet<String>(Arrays.asList(arr));
    HashSet<String> result = p.getTravelables();
    String msg = "Testing only one travelable character";
    assertEquals(expect,result,msg);
}
```

Coverage:

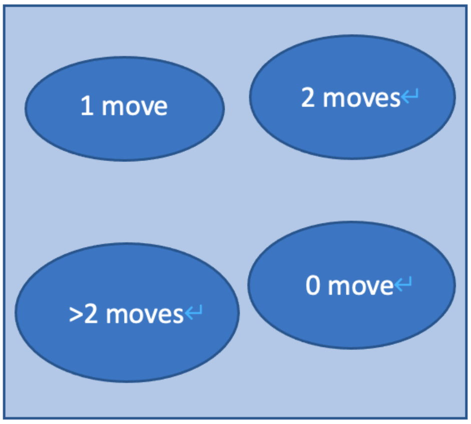


Runs: 5/5 ☒ Errors: 0 ☒ Failures: 0

```
TravelableTest [Runner: JUnit 5] (0.126 s)
    testOneTravelable() (0.096 s)
    testDulTravelable() (0.006 s)
    testNotExistTravelable() (0.006 s)
    testNoTravelable() (0.011 s)
    testAllTravelable() (0.002 s)
```

5.1.2. Integration Test – getLegalMoves()

The limitation of this test case is that we might not cover all the possible case. For example, there could be 10 legal moves, 100 legal moves and 1000 legal moves. In our case, we adopt partition testing approach to test every representative from subdomain. Therefore, we just assume that >2 legal moves are inside the testing domain. In other words, “>2 legal moves” is the representative of from the subdomain (e.g. 10, 100, 1000……).



Our test cases cover 8 situations

1. There is only one legal move from land A to land B

2. There is only one legal move from land B to land A

3. There is two legal moves from land B to land A

4. There is two legal moves from land A to land B

5. There is >2 legal moves from land A to land B

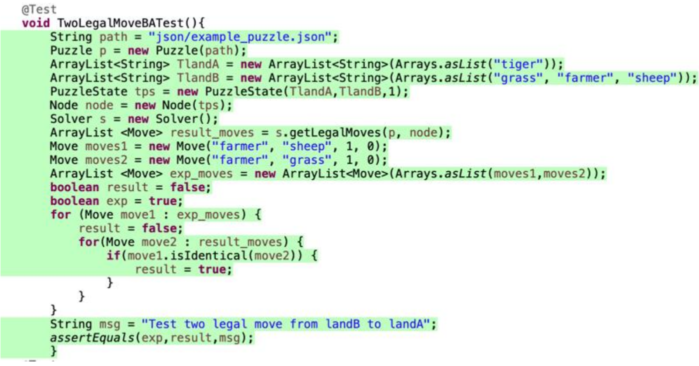
6. There is >2 legal moves from land B to land A

7. There is no legal move from land A to land B

8. There is no legal move from land B to land A

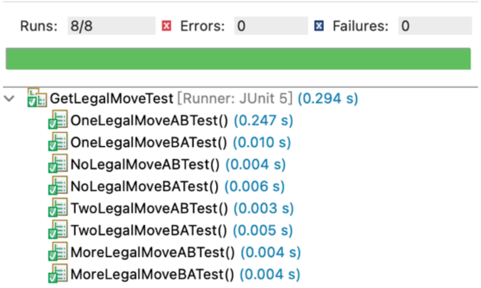
Example:

This is a test case simulating there are >2 legal move from land B to land A.



```
@Test
void TwoLegalMoveBATest(){
    String path = "json/example_puzzle.json";
    Puzzle p = new Puzzle(path);
    ArrayList<String> TlandA = new ArrayList<String>(Arrays.asList("tiger"));
    ArrayList<String> TlandB = new ArrayList<String>(Arrays.asList("grass", "farmer", "sheep"));
    PuzzleState tps = new PuzzleState(TlandA,TlandB,1);
    Node node = new Node(tps);
    Solver s = new Solver();
    ArrayList <Move> result_moves = s.getLegalMoves(p, node);
    Move moves1 = new Move("farmer", "sheep", 1, 0);
    Move moves2 = new Move("farmer", "grass", 1, 0);
    ArrayList <Move> exp_moves = new ArrayList<Move>(Arrays.asList(moves1,moves2));
    boolean result = false;
    boolean exp = true;
    for (Move move1 : exp_moves) {
        result = false;
        for(Move move2 : result_moves) {
            if(move1.isIdentical(move2)) {
                result = true;
            }
        }
    }
    String msg = "Test two legal move from landB to landA";
    assertEquals(exp,result,msg);
    }
```

Coverage:



```
Runs: 8/8    Errors: 0    Failures: 0

GetLegalMoveTest [Runner: JUnit 5] (0.294 s)
    OneLegalMoveABTest() (0.247 s)
    OneLegalMoveBATest() (0.010 s)
    NoLegalMoveABTest() (0.004 s)
    NoLegalMoveBATest() (0.006 s)
    TwoLegalMoveABTest() (0.003 s)
    TwoLegalMoveBATest() (0.005 s)
    MoreLegalMoveABTest() (0.004 s)
    MoreLegalMoveBATest() (0.004 s)
```

5.2. Integration test of solve()

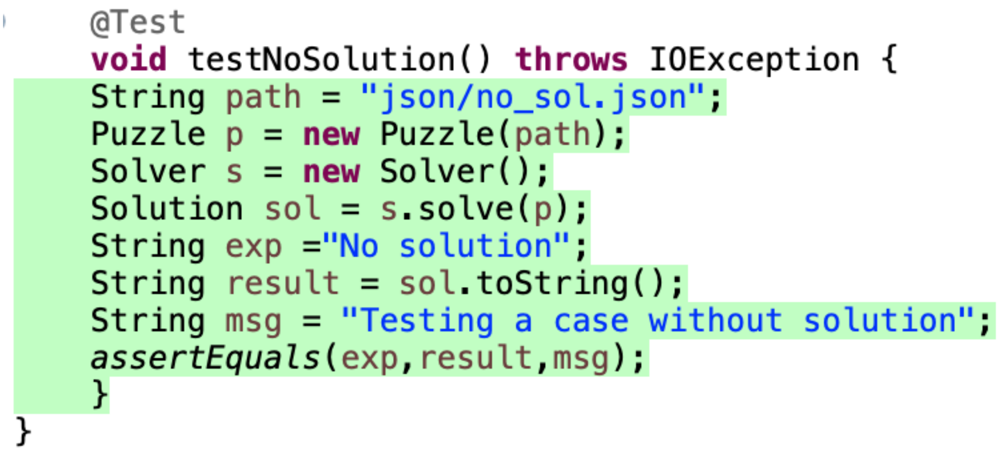
Since we have performed integration testing of the main function inside solve() which is getLegalMove(). Then, we can try to test the function to test the solution provided by this function.

There will be two combinations:

1. Return a valid solution
2. Return no solution

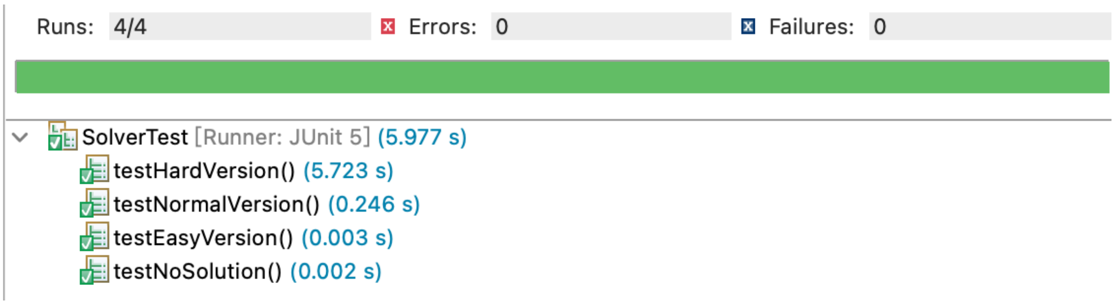
Example:

This is a test case simulating there is no solution.



```
    @Test
    void testNoSolution() throws IOException {
        String path = "json/no_sol.json";
        Puzzle p = new Puzzle(path);
        Solver s = new Solver();
        Solution sol = s.solve(p);
        String exp ="No solution";
        String result = sol.toString();
        String msg = "Testing a case without solution";
        assertEquals(exp,result,msg);
    }
}
```

Coverage:



```
Runs: 4/4        ☒ Errors: 0        ☒ Failures: 0

SolverTest [Runner: JUnit 5] (5.977 s)
    testHardVersion() (5.723 s)
    testNormalVersion() (0.246 s)
    testEasyVersion() (0.003 s)
    testNoSolution() (0.002 s)
```

(* We have also test the different difficulty to see if it function well)

6. Testing Player Mode Undo/Redo Command

In player mode, player can perform undo & redo command respectively. For testing the functionality of those commands, we will consider the following test case:

1. Testing an allowed Undo command followed by a move

2. Testing an allowed Undo command followed by a redo command

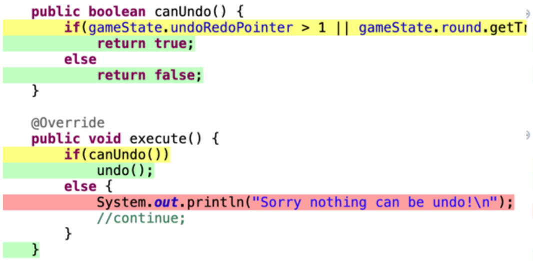
3. Testing a disallowed Undo command without a move or a redo command

4. Testing an allowed Redo command followed by an undo command

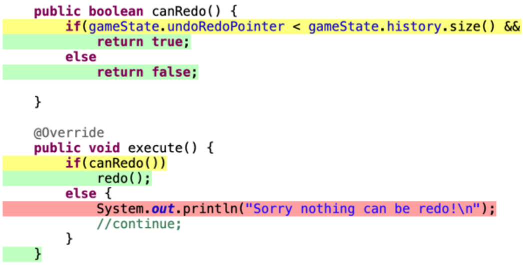
5. Testing a disallowed Redo command without previous undo command

Idea behind:

1. Undo can only be performed if there is previous move and redo.
2. Redo can only be performed if there is previous undo.



```java
public boolean canUndo() {
    if(gameState.undoRedoPointer > 1 || gameState.round.getT
        return true;
    else
        return false;
}

@Override
public void execute() {
    if(canUndo())
        undo();
    else {
        System.out.println("Sorry nothing can be undo!\n");
        //continue;
    }
}
```

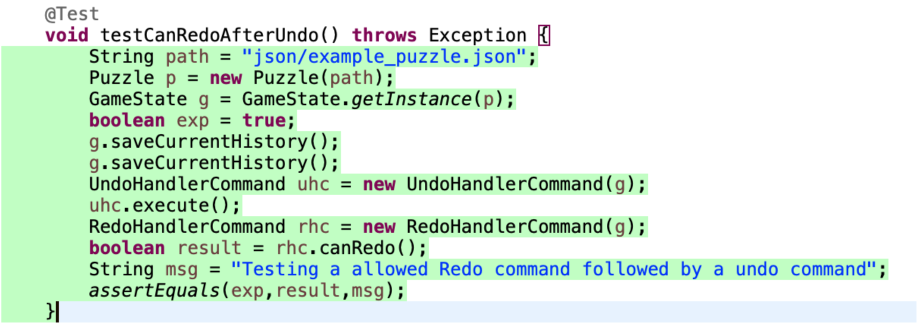


```java
public boolean canRedo() {
    if(gameState.undoRedoPointer < gameState.history.size() &&
        return true;
    else
        return false;

}

@Override
public void execute() {
    if(canRedo())
        redo();
    else {
        System.out.println("Sorry nothing can be redo!\n");
        //continue;
    }
}
```

Since undo () and redo() function is depend of the return value of canUndo() and canRedo() respectively. Therefore, we will test the value returned by those functions.

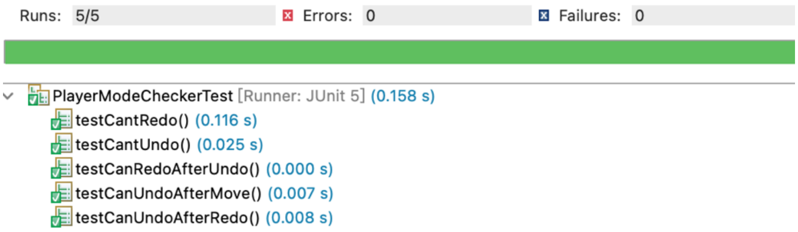
Example:

This is a test case simulating a Redo command is allowed followed by an undo command.



```
@Test
void testCanRedoAfterUndo() throws Exception {
    String path = "json/example_puzzle.json";
    Puzzle p = new Puzzle(path);
    GameState g = GameState.getInstance(p);
    boolean exp = true;
    g.saveCurrentHistory();
    g.saveCurrentHistory();
    UndoHandlerCommand uhc = new UndoHandlerCommand(g);
    uhc.execute();
    RedoHandlerCommand rhc = new RedoHandlerCommand(g);
    boolean result = rhc.canRedo();
    String msg = "Testing a allowed Redo command followed by a undo command";
    assertEquals(exp,result,msg);
}
```

Coverage:

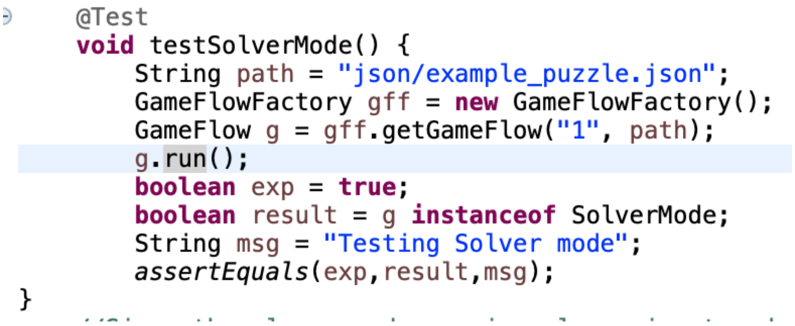


```
Runs: 5/5        Errors: 0        Failures: 0

PlayerModeCheckerTest [Runner: JUnit 5] (0.158 s)
    testCantRedo() (0.116 s)
    testCantUndo() (0.025 s)
    testCanRedoAfterUndo() (0.000 s)
    testCanUndoAfterMove() (0.007 s)
    testCanUndoAfterRedo() (0.008 s)
```

7. System Test

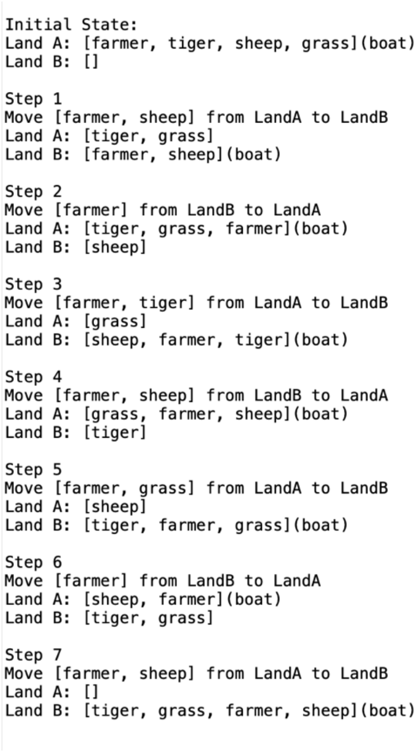
7.1. Solver Mode

Test Case for Solver Mode System Testing:



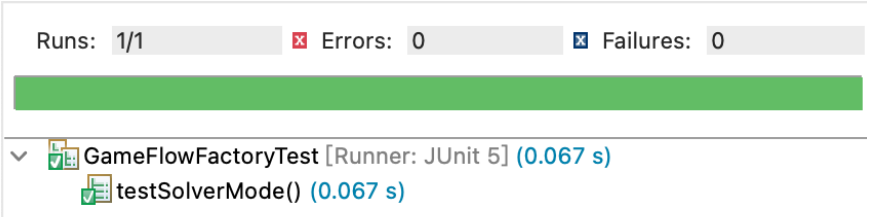
```
@Test
void testSolverMode() {
    String path = "json/example_puzzle.json";
    GameFlowFactory gff = new GameFlowFactory();
    GameFlow g = gff.getGameFlow("1", path);
    g.run();
    boolean exp = true;
    boolean result = g instanceof SolverMode;
    String msg = "Testing Solver mode";
    assertEquals(exp,result,msg);
}
```

Actual output in console:



```
Initial State:
Land A: [farmer, tiger, sheep, grass](boat)
Land B: []

Step 1
Move [farmer, sheep] from LandA to LandB
Land A: [tiger, grass]
Land B: [farmer, sheep](boat)

Step 2
Move [farmer] from LandB to LandA
Land A: [tiger, grass, farmer](boat)
Land B: [sheep]

Step 3
Move [farmer, tiger] from LandA to LandB
Land A: [grass]
Land B: [sheep, farmer, tiger](boat)

Step 4
Move [farmer, sheep] from LandB to LandA
Land A: [grass, farmer, sheep](boat)
Land B: [tiger]

Step 5
Move [farmer, grass] from LandA to LandB
Land A: [sheep]
Land B: [tiger, farmer, grass](boat)

Step 6
Move [farmer] from LandB to LandA
Land A: [sheep, farmer](boat)
Land B: [tiger, grass]

Step 7
Move [farmer, sheep] from LandA to LandB
Land A: []
Land B: [tiger, grass, farmer, sheep](boat)
```

Coverage:



Runs: 1/1 ☒ Errors: 0 ☒ Failures: 0

GameFlowFactoryTest [Runner: JUnit 5] (0.067 s)

testSolverMode() (0.067 s)

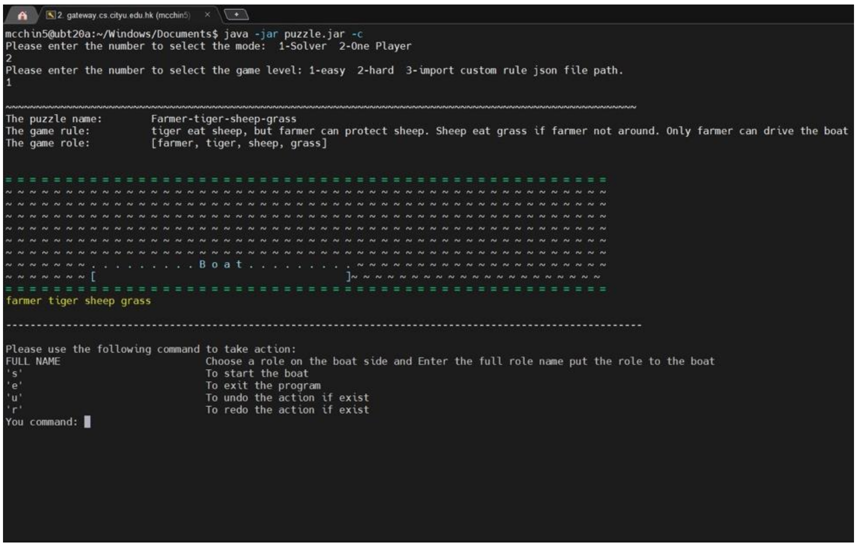
7.2. Player Mode

Since player mode is difficult to perform system testing since it must keep accepting player’s input and command in the process. However, the logic behind is same as solver mode. The only difference is that solver mode generate solution automatically, while player mode is manually generated solution by user. As we have tested the logical part by performing unit testing and integration testing separately above. Therefore, we assume solver mode testing is enough to be our system testing.

8. User acceptance testing (UAT)

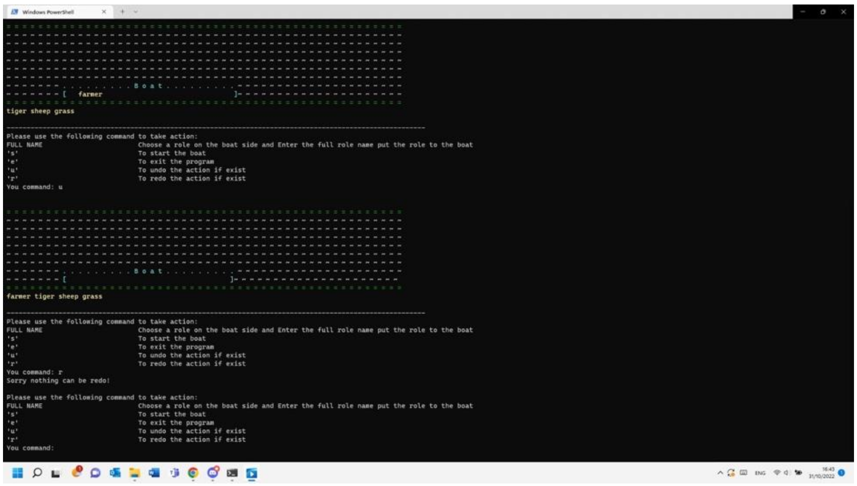
In UAT, we tested our program on different operation system. The result are as follows:

Linux platform:



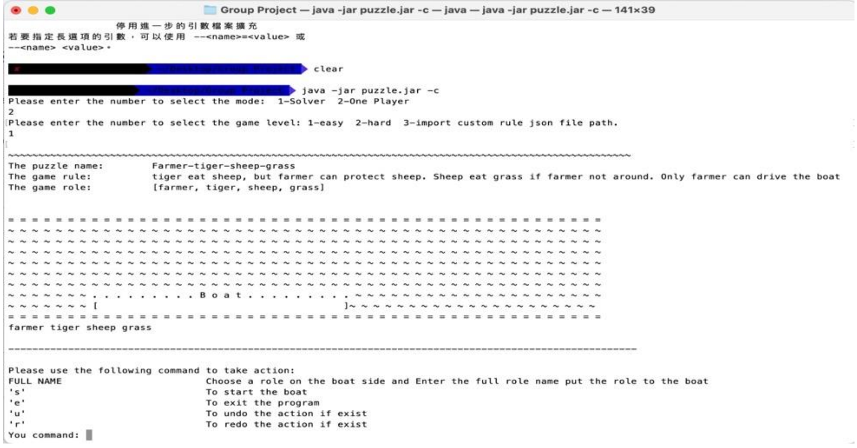
```
mcchin5@ubt20a:~/Windows/Documents$ java -jar puzzle.jar -c
Please enter the number to select the mode:  1-Solver  2-One Player
2
Please enter the number to select the game level: 1-easy  2-hard  3-import custom rule json file path.
1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The puzzle name:        Farmer-tiger-sheep-grass
The game rule:          tiger eat sheep, but farmer can protect sheep. Sheep eat grass if farmer not around. Only farmer can drive the boat
The game role:          [farmer, tiger, sheep, grass]

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ . . . . . . . . . B o a t . . . . . . . . . . ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ [                                         ]~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
farmer tiger sheep grass

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

Please use the following command to take action:
FULL NAME                        Choose a role on the boat side and Enter the full role name put the role to the boat
's'                              To start the boat
'e'                              To exit the program
'u'                              To undo the action if exist
'r'                              To redo the action if exist
You command: 
```

Window platform:



```
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ . . . . . . . . . B o a t . . . . . . . . . . ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ [   farmer                                  ]~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
tiger sheep grass

---------------------------------------------------------------------------------------------------------
Please use the following command to take action:
FULL NAME                        Choose a role on the boat side and Enter the full role name put the role to the boat
's'                              To start the boat
'e'                              To exit the program
'u'                              To undo the action if exist
'r'                              To redo the action if exist
You command: u


= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ . . . . . . . . . B o a t . . . . . . . . . . ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ [                                         ]~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
farmer tiger sheep grass

---------------------------------------------------------------------------------------------------------
Please use the following command to take action:
FULL NAME                        Choose a role on the boat side and Enter the full role name put the role to the boat
's'                              To start the boat
'e'                              To exit the program
'u'                              To undo the action if exist
'r'                              To redo the action if exist
You command: r
Sorry nothing can be redo!

Please use the following command to take action:
FULL NAME                        Choose a role on the boat side and Enter the full role name put the role to the boat
's'                              To start the boat
'e'                              To exit the program
'u'                              To undo the action if exist
'r'                              To redo the action if exist
You command:
```

Ios platform:



Group Project — java -jar puzzle.jar -c — java — java -jar puzzle.jar -c — 141×39

```
                 停用進一步的引數檔案擴充
若要指定長選項的引數，可以使用 --<name>=<value> 或
--<name> <value>。

[redacted] clear

[redacted] java -jar puzzle.jar -c
Please enter the number to select the mode:  1-Solver  2-One Player
2
Please enter the number to select the game level: 1-easy  2-hard  3-import custom rule json file path.
1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The puzzle name:        Farmer-tiger-sheep-grass
The game rule:          tiger eat sheep, but farmer can protect sheep. Sheep eat grass if farmer not around. Only farmer can drive the boat
The game role:          [farmer, tiger, sheep, grass]


= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ . . . . . . . . . B o a t . . . . . . . . . ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
~ ~ ~ ~ ~ ~ ~ [                                         ]~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
farmer tiger sheep grass

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

Please use the following command to take action:
FULL NAME                       Choose a role on the boat side and Enter the full role name put the role to the boat
's'                             To start the boat
'e'                             To exit the program
'u'                             To undo the action if exist
'r'                             To redo the action if exist
You command: █
```