Snake-AI
Project Documentation

Ibrahim Enes Hayber, Rana MD Jewel, Maximilian Lüttich
Frankfurt University of Applied Sciences
Faculty 2, Computer Science (B. Sc.)
Object-oriented Programming in Java by Prof. Dr. Doina Logofatu

December 20, 2022

Contents

1 How we started

This chapter describes how we started with the project.

As we all know the hardest part of a journey is the start. But once started it is a easy on going. It is like the impact on the first domino stone, which brings the whole project in rolling.
As usual in any task of life we need to understand what the problem is and which requirements are necessary to solve the problem. To do so, we read the description from the organizers carefully. In addition to that we watched a few videos about our topic to get a better understanding.
After gaining knowledge about the problem we felt ready to start working with the Snake-AI framework, provided by the organizers on a GitHub repository. The framework was written in the Java programming language. To get familiar we looked into the UML class diagram. Besides that we followed a YouTube Tutorial which was also provided by the organizers, where we implemented our first working version. A detailed introduction and our results will now be presented.

2 Introduction

Many people know the popular game "Snake" which appeared on a Nokia device back in 1997. This project focuses on reestablishing Snake but in a different way. The most special is that instead moving one single snake manually, we will have two intelligent Snake bots competing against each other. The aim of a bot is to be the last standing snake or have the highest score by eating apples in 3 minutes. *Survival of the fittest* is a great saying. This chapter focuses on the general overview of this project and comparison to similar problems in practice.

2.1 General Topic

2.2 Similar problems in practice (References every time, look for actual ones)

to-do

3 Team Work

to-do

3.1 Work Participation

to-do

3.2 work structure (communication, decisions, bug tracking, repository, engineering, ...)

to-do

3.3 ideas (brainstorming)

to-do

4 Problem Description

to-do

4.1 Formal description: definitions, examples, ...

to-do

5 Related Work

to-do

5.1 Related Algorithms, Applications

to-do

!for example: Fractals, Data Generation, Games, Evolutionary Algorithms (Genetic Algortihms; Collective Intelligence like Particle Swarm Optimization - PSO, Ant Colony Optimizaton - ACO; Evolutionary Multiobjective Optimization, ...)

6 Proposed Approaches

to-do

6.1 Input/Output Format, Benchmarks (Generation, Examples)

to-do

6.2 Algorithms in Pseudocode

to-do

7 Implementation Details

to-do

7.1 Application Structure

to-do

7.2 GUI Details

to-do

7.3 UML Diagram

to-do

7.4 Used Libraries

to-do

7.5 Code Snippets

to-do

8 Evolution of Bots

to-do

8.1 Bot1

Bot 1 (IntBot1.java):

After getting to know the framework of our task, one of our first objectives was implementing a basic bot by ourselves that does the following:

- Calculate all valid moves
- Exhibit a basic avoidance behavior regarding the opponent
- Go for the apple

This basic bot also served as an opponent for more advanced bots later on.

To keep things organized, we decided to create a package for each bot. The packages were created in the „src“-folder of the given project. Each bot consists of a public class that implements the Interface „Bot“, which as previously described, consists of only one method: „chooseDirection“. Properly overriding this method turned this class into a bot that could be used in the game.

The basic idea for creating this bot was that a static array that contains all possible directions (Direction.RIGHT, Direction.DOWN, Direction.UP, Direction.LEFT) is created outside the method „chooseDirection“ and inside the method each element of this array is checked for its worth concerning the

stated objectives.

Creating the static array:

```
public class IntBot1 implements Bot {
private static final Direction[] DIRECTIONS = new Direction[]{Direction.RIGHT,
Direction.DOWN, Direction.UP, Direction.LEFT};
@Override
public Direction chooseDirection(Snake snake, Snake opponent, Coordinate
mazeSize, Coordinate apple) {
```

The first thing we did was to create an array of possible directions of the opponent's head and a set that contains the resulting coordinates, so that at the end of the method „chooseDirection" the coordinates could be removed from the pool of possible directions of our bot's head. To create such an array, it was necessary to determine the coordinate of the second element of the opponent snake. We used an iterator to achieve this in the following way:

```
//Position of second Element of opponent
Coordinate afterHeadNotFinal2 = null;
if(opponent.body.size()>=2){
Iterator<Coordinate> it = opponent.body.iterator();
it.next();
afterHeadNotFinal2 = it.next();
}
final Coordinate afterHead2 = afterHeadNotFinal2;
//second element of opponent snake
```

Note that the class „Snake" featured a deque of elements of the „Coordinate"-class called „body".

After that, we used a stream method to get a sequential stream of the elements from the array „DIRECTIONS" to create a new array of directions called „validMovesOpponent". The direction whose result would equal the second element of the opponent snake was filtered out using the "equals"-method on coordinate objects. Finally, a Set of coordinates called "possiblePositionsOpponent" was created by simply using the "moveTo" method on the head of the opponent snake and having all elements of "validMovesOpponent" as parameters.

```
Direction[] validMovesOpponent = Arrays.stream(DIRECTIONS)
.filter(d -> !head2.moveTo(d).equals(afterHead2))
```

```
.sorted()
.toArray(Direction[]::new);
Set<Coordinate> possiblePositionsOpponent = new HashSet<>();
for(int i=0; i< validMovesOpponent.length; i++){
possiblePositionsOpponent.add(head2.moveTo(validMovesOpponent[i]));
}
```

Since no other restrictions were made, the set „possiblePositionsOpponent“ always contained three elements, even if they were to be outside the maze.

Analogous to the array „validMovesOpponent“ we created an array of directions called „validMoves“ for our bot's snake that excluded the possibility of backwards movement of the snake. This array „validMoves“ was subsequently used as a parameter of the stream method to create a new array of directions called „notLosing“, that with the use of the filter method excluded some unfavorable outcomes.

```
Direction[] notLosing = Arrays.stream(validMoves)
.filter(d -> head.moveTo(d).inBounds(mazeSize))
.filter(d -> !opponent.elements.contains(head.moveTo(d)))
.filter(d -> !snake.elements.contains(head.moveTo(d)))
.filter(d -> !possiblePositionsOpponent.contains(head.moveTo(d)))
.sorted()
.toArray(Direction[]::new);
```

Using the filter method, we checked the following: Is the resulting coordinate outside the maze? Is the resulting coordinate part of the opponent's body? Is the resulting coordinate part of our snake's body? And finally, does the set „possiblePositionsOpponent“ contain the resulting coordinate? If the answer to each of these question was „no“ the streamed element was added to the array.

Finally, we used the bubble sort algorithm to sort the elements of „notLosing“ based upon the distance to the apple. „chooseDirection“ returns the first element of „notLosing“ provided that this array is not empty. If this array happens to be empty, the first element of „validMoves“ is returned which, is never empty.

```
//Bubble Sort; Sorting Elements of "NotLosing"
double distance1, distance2;
int a1, a2, h11, h12, h21, h22, d1, d2;
```

```
Coordinate test1, test2;
Direction temp;
if (notLosing.length > 0){
a1=apple.x;
a2=apple.y;
for(int i=0; i< notLosing.length; i++){
for(int a= i+1; a< notLosing.length; a++){
test1 = head.moveTo(notLosing[i]);
test2 = head.moveTo(notLosing[a]);
h11= test1.x;
h12 = test1.y;
d1 = (h11-a1)*(h11-a1)+(h12-a2)*(h12-a2);
distance1=Math.sqrt((double)d1);
h21= test2.x;
h22 = test2.y;
d2 = (h21-a1)*(h21-a1)+(h22-a2)*(h22-a2);
distance2=Math.sqrt((double)d2);
if(distance2<distance1){
temp=notLosing[i];
notLosing[i]=notLosing[a];
notLosing[a]=temp;
}
}
}
return notLosing[0];
}
else{
return validMoves[0];
}
}
```

8.2 Bot2

8.3 Bot3

9 Experimental Results and Statistical Tests

to-do

9.1 Simulations (Play with the Parameters!)

to-do

9.2 Use Benchmarks

to-do

9.3 Tables: input data details, results of different algorithms

to-do

9.4 Charts

to-do

9.5 Evaluations

to-do

10 Conclusions and Future Work

to-do

10.1 How the team work went

to-do

10.2 What we have learned

to-do

10.3 Ideas for the future development of your application, new algorithms

to-do