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

January 10, 2023

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

Our main objective is to make two efficiently working Snake Bots. To create a working bot, all we have to do is to implement a class that implements the Bot interface. This interface has one method which is the *chooseDirection()* method. We can say that the method simulates the brain of the bot. The return value of this method is the direction in which the snake should move to. For directions we have *up, down, left or right*. For example we could implement a first simple bot that just turns right one time. But this type of bot is really uninteresting. For our project we are aiming to implement a bot that uses alogrithms from the fields of Artificial Intelligence. But for now let us look into the game rules.

The rules are described as follows:
Rule 1: A bot controls only the direction (going either north, south, east, or west) to be taken by its own snake.
Rule 2: Snakes always move simultaneously and forward. Their size increases by one position (i.e. pixel) after taking an apple.
Rule 3: A Snake loses in any of these conditions:

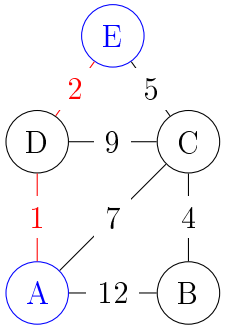
1. If it leaves the board;
2. If it hits its own body;
3. If it hits the other snake's body;
4. If it takes more than one second to make a decision (i.e. which direction to take).

Rule 4: If snakes collide head to head, the longest snake wins the game.
Rule 5: Apples appear randomly at an unoccupied position of the board,

and there is only one apple available at any time.
Rule 6: An apple will disappear if it is not eaten by either snake in 10 seconds and reappear somewhere else on the map.
Rule 7: At the end of the tournament, players are ranked according to the number of victories; then the number of draws; then the result of

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

Furthermore, since our bot should be smart then it definetly should use intelligent algorithms. What if we could implement a simple bot that just goes into the direction of the apple and gets closer and closer after every decision. But would this be enough to call it a smart bot? Of course not. Well it is a good behaviour that our snake goes for apples. But thats not enough. It should also take the shortest path to the apple. For this we want to use algorithm of dijkstra. Dijkstra algorithm is a graph based algorithm which calculates the shortest path from one node to every other node. Let us get a better understanding of this by looking into an example.



We want to know the shortest path from node A to node B. We as human can easily look into that graph and see that taking the route from A-D with cost of 1 is much lesser than rather taking A-C with cost of 7 or A-B with cost of 12. As next we would also see that we have a cost of 2 from D-E. So in total we see that the shortest path from A to D is A-D-E. But telling this to a computer which only knows 0's and 1's is nonsense. Thats why we have algorithms like Dijkstra, which calculates and erases paths with higher costs and returns us the shortest path. Anyway there are many daily life problems where we are using algorithms to compute the shortest path. For example when you are using a navigation tool to drive from one city to an other city. We all would have serious problems if the navigation would take

any random path to our destination rather than taking the shortest path or the path with less travel time. Or even when you are crossing over a street where cars are driving. You surely want to take the shortest and safest walkway to the other side.

Well thats not the only use case of dijkstra in real life applications. When we talk about computer networks we will see a use case of dijkstra too. Considering IP routing to find the shortest path between source and destination router. Dijkstra algorithm is common ly used in routing protocols for updating the forwarding tables.

Even many dating or social media applications are using dijkstra to recommend you a person which you may know or be interested in. The users can be seen as nodes. The distance could be defined on different aspects like common friends, same interest or even living in the same city.

A really interesting real life application would be that we could use pathing algorithms in case of emergency calls. If there is a fire anywhere and a emergency call was send out. The algorithm could calculate based on the location which firefighter department is the closest one. Based on this they could send out firefighters. The same could be used for police department and hospitals.

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 Max

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. This concept was also used in the provided example of a bot on the following website: https://github.com/BeLuckyDaf/snakes-game-tutorial The only thing that this bot was concerned with was not killing itself.

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];
}
```

```
}
```

The biggest drawback of this bot was the frequent entrapment with itself or the opponent's snake. Even though basic evasive movements were often enough to avoid losing the game, more complex dangers like a U-shape of the opponent's snake or the bot's snake could not be recognized by this bot. This becomes more apparent the more elements the bodies of the snakes have.

8.2 Rana

The idea behind my Bot is the famous Dijkstra Algorithem which is most widely used to find the shortest path. In my Programm before I give my Snake a Direction first of all I always choose four adjucent cells from the current Head position of the Snake. Secondly, I have to eliminate one of the cell from them which is already located in the Snake body (next Cordinate after the Head). Thirdly, I have been checking whether the given move leads the Snake out of the Maze. Then with the help of Pythagoras formula I calculate the distance between the next postion of the Snake and the apple. After that I calculate the minimum between all the distances. Finally I give the Snake the next Direction.

But in this Algorithem I was not considering the position of the opponent Snake. As a result there were many collisions between my Snake and opponent Snake. To get rid of this I have also figured out all the possible next move of the Opponent Snake and saved them in a HashSet. While giving the next move to my Snake I check whether this next move is also the next move of the opponent Snake or not. This has solved the Head to Head collision. But still there were some collision with Body of the opponent Snake. That's why I have also justified whether the next move is one of the element of the opponent Body or not.

In term of Time and Space Complexity this Algorithem is quite efficient. The Algorithem has Time Complexity of O(1) and Space Complexity is approximately O(n). Wher n is the number of elements of Opponent Snake body.

Inspite of all the Consideration there is still some Collisions in critical Situation. Anothe problem is Self Entrapping.

```
public class MyBot implements Bot {
     // Arry to contains all the moves
    private static final Direction[] DIRECTIONS = new Direction[]{Direction.RIGH
            Direction.DOWN, Direction.UP, Direction.LEFT};

    // functions to evaluate a valid move
    boolean isValidMove(Snake snake,Coordinate mazeSize,Direction d,HashSet<Coor
        if(snake.getHead().moveTo(d).inBounds(mazeSize) && !snake.elements.conta
        !opponentPos.contains(snake.getHead().moveTo(d)) && !opponent.elements.c
            //&& !opponent.elements.contains(snake.getHead().moveTo(d)
            return true;
        }
        return  false;

    }
    // Function to give next direction
    @Override
    public Direction chooseDirection(Snake snake, Snake opponent, Coordinate maz
        Coordinate head = snake.getHead();
        Coordinate head2 = opponent.getHead();

        //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;
        // save all the possible next move of the opponent snake validMovesOppon
        Direction[] validMovesOpponent = Arrays.stream(DIRECTIONS)
                .filter(d -> !head2.moveTo(d).equals(afterHead2))
                .sorted()
                .toArray(Direction[]::new);
        // save all possible next move of the opponent in a HashSet. beacuse Has
        Set<Coordinate> possiblePositionsOpponent = new HashSet<>();
        for(int i=0; i< validMovesOpponent.length; i++){
            possiblePositionsOpponent.add(head2.moveTo(validMovesOpponent[i]));
        }
```

```
//  variable to find all the distance from next move to apple
 int disFromLeft = Integer.MAX_VALUE, disFromRight=Integer.MAX_VALUE,disF

 if (isValidMove(snake, mazeSize, Direction.UP, (HashSet<Coordinate>) pos

     Coordinate toUp = snake.getHead().moveTo(Direction.UP);
     disFromUp =(int) Math.sqrt(Math.pow(toUp.x- apple.x,2)+Math.pow(toUp

 }
 if (isValidMove(snake, mazeSize, Direction.LEFT,(HashSet<Coordinate>) po

     Coordinate toLeft = snake.getHead().moveTo(Direction.LEFT);
     //disFromLeft = Math.abs((toLeft.x- apple.x)+(toLeft.y- apple.y));
     disFromLeft =(int) Math.sqrt(Math.pow(toLeft.x- apple.x,2)+Math.pow(

 }
 if (isValidMove(snake, mazeSize, Direction.DOWN,(HashSet<Coordinate>) po

     Coordinate toDown = snake.getHead().moveTo(Direction.DOWN);
     disFromDown = Math.abs((toDown.x- apple.x)+(toDown.y- apple.y));
     disFromDown =(int) Math.sqrt(Math.pow(toDown.x- apple.x,2)+Math.pow(

 }
 if (isValidMove(snake, mazeSize, Direction.RIGHT,(HashSet<Coordinate>) p

     Coordinate toRight = snake.getHead().moveTo(Direction.RIGHT);
     //disFromRight = Math.abs((toRight.x- apple.x)+(toRight.y- apple.y))
     disFromRight =(int) Math.sqrt(Math.pow(toRight.x- apple.x,2)+Math.po
 }
// find minimum from all the possible paths
 int minDis = Math.min(Math.min(disFromRight,disFromDown),Math.min(disFro

     //give direction to the snake

 if(minDis==disFromRight){
     return Direction.RIGHT;
 }
 else if(minDis==disFromDown){

     return Direction.DOWN;
```

```
        }
        else if(minDis==disFromLeft){

            return  Direction.LEFT;
        }
        else if(minDis==disFromUp) {
            return Direction.UP;
        }
        else{
          // to avoid worst cases.
            Random rn = new Random();
            int pos = rn.nextInt(3);
            switch (pos){

                case 0:
                    return Direction.RIGHT;
                case 1:
                    return Direction.LEFT;
                case 2:
                    return Direction.UP;
                default:
                    return Direction.DOWN;
            }
        }

    }
}
```

8.3 Ibrahim

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