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 21, 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 algorithms 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:

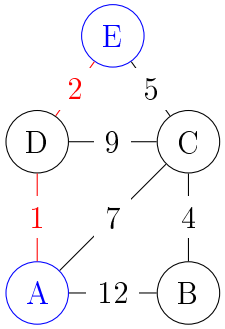
1. If it leaves the board;
2. If it hits its own body;
3. If it hits the other snake's body;
4. If it takes more than one second to make a decision (i.e. which direction to take).

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 definitely 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 behavior that our snake goes for apples. But that is 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 that is 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

this is my test to-do

7.1 Application Structure

The Snake AI project consists of one Interface, one Enum and eight classes. They are following,

- Bot(Interface)
- Direction(Enum)
- BotLoader
- Coordinate
- Snake
- SnakeGame
- SnakeCanvas
- SnakesRunner
- SnakeUIMain
- SnakesWindow

7.2 GUI Details

In this section all the classes will be discussed in details.

7.2.1 SnakeUIMain

This class is the Entry point of the UI and implements tournament of the Snake game with several rounds. The main method of the class gets two classes(snake & opponent) implementing the Bot interface as arguments. The class is also responsible to launch several rounds of snake game between bots. In this class the basic I/O actions are performed to record scores of opponent, apple eaten by both the snakes and time taken in each rounds which will be later used to make further statistical analysis.

7.2.2 BotLoader

The class BoatLoader fetches the class given the name of the class and package. The class is taken from the classpath and could be dynamically added after the game of the Bot class. This class inharites the abstrack Java class ClassLoader.
A class loader is an object that is responsible for loading classes. The class ClassLoader is an abstract class. Given the binary name of a class, a class loader should attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name into a file name and then read a "class file" of that name from a file system.(to-do code snippet)
The only method of BotLoader class gets classBinName as argument and returns an instance of the Bot class.

7.2.3 SnakesWindow

This class is responsible for the game's GUI window. The class creates and set ups the window and runs the UI. At the end it closes the frame. This class implements Runnable.
The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.
This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.
In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.(to-do ref)

7.2.4 SnakesRunner

This class also implements Runnable interface and is used for running bots in a separate threads. The Constructor of the class SnakesRunner recieves running bot, snake that is controlled by the current bot, opponent's snake, size of the board and coordinate of the current apple as arguments. In addition

the chooseDirection method of the current bot is executed and saved(current direction) by this class.

7.2.5 SnakesGame

The class SnakesGame implements the main games flow and runs game for two bots. The Override toString() method converts the game to string representation and return game state as a string. randomNonOccupiedCell() method selects random non-occupied cell of maze which is used to create new location of a apple. If a Snake takes more than one seconds to choose its next direction the runOneStep() methods stopps that rounds.

7.2.6 SnakeCanvas

The purpose of this class is to design the UI and give the UI a better look. This class helps to fill the snake Body wicht color and make it visible on the UI. After filling the body of snake and opponent and apple the render() method renders the game. After every movement of the snake, opponent and apple the previous occupied positions are recolord and marked the free for next movement. This class inharites the Java swing class JPanel which is used to create different generic lightweight container with one or more single component.

7.2.7 Snake

This class implements snake body(not brain) that determines the place of head, body, and length of body on the game board. For implementing the body of the snake two logical and efficient data structure from Java Collections framework are used. They are,

- Dequeue
- HashSet

Deque(double ended queue) data structure helps to add or delete elements from both side(head and tail) within $\mathcal{O}(1)$ constant time coplexity. On the other hand HashSet data structure ensures basics search operation in $\mathcal{O}(1)$ time complexity.
This class has a important feature of cloneability which allows to copy one object to another object without using new operator.
A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class. Invoking Object's clone method on an instance that

does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.(to-do ref)

7.2.8 Coordinate

Coordinates class implements position of a cell on the game board. It helps to grow up the snake by adding new coordinate at the beginning of the snake body. inBounds() methods helps to keep the snake in side the game board. This methods implements Comparable interface which is useful while comparing two Coordinate.

7.2.9 Bot

This Interface provides functions that should be implemented to create smart and intelligent snake bot for the game. chooseDirection() methds returns an appropriate next moves which plays a vital role to continue the game.

7.3 UML Diagram

Figure 1: UML diagram of the GUI

7.4 Used Libraries

For the Developement of the Snake AI Game Java Swing Framework has been used. Some Components from AWT framework have also been used. Swing is a Java Foundation Classes [JFC] library and an extension of the Abstract Window Toolkit [AWT]. Swing offers much-improved functionality over AWT, new components, expanded components features, and excellent event handling with drag-and-drop support.(to-do ref)

8 Our Bots with different implementations

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. 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 Bot2

8.3 Bot3

This bot is using theorem of Pythagoras, a simplified shortest path algorithm and a kind of reversed Dijkstra which uses the longest path.
The bot needs to handle the following problems:

- avoid boundaries
- avoid enemy snakes body
- avoid snakes head
- avoid himself
- take shortest path to apple

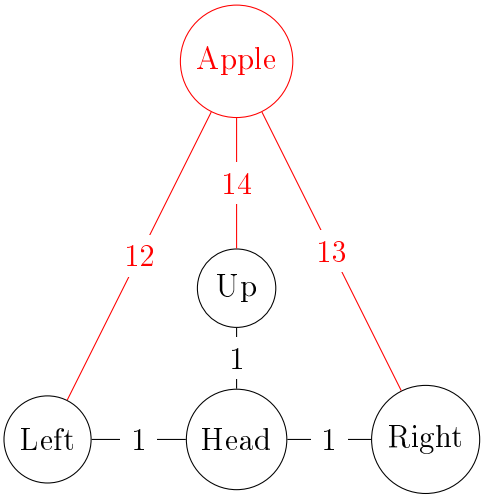
To explain how we solved these problems let us look into the source code. As usual we first implement the static array of directions and with that we also create a array of direction which filters with lambda functions the the valid moves.

```
private static final Direction[] DIRECTIONS = new Direction[]
{ Direction.RIGHT, Direction.DOWN, Direction.UP, Direction.LEFT};
.
.
.
Coordinate head = snake.getHead();
        Coordinate oppHead = opponent.getHead();
        Coordinate afterHead = null;

        if(snake.body.size() >= 2) {
            Iterator<Coordinate> it = snake.body.iterator();
            it.next(); // first element
            afterHead = it.next(); // second element
        }

        final Coordinate afterHeadPos = afterHead;
        // avoids going backwards
        Direction[] validMoves = Arrays.stream(DIRECTIONS)
                .filter(d -> !head.moveTo(d).equals(afterHeadPos))
                .sorted()
                .toArray(Direction[]::new);
```

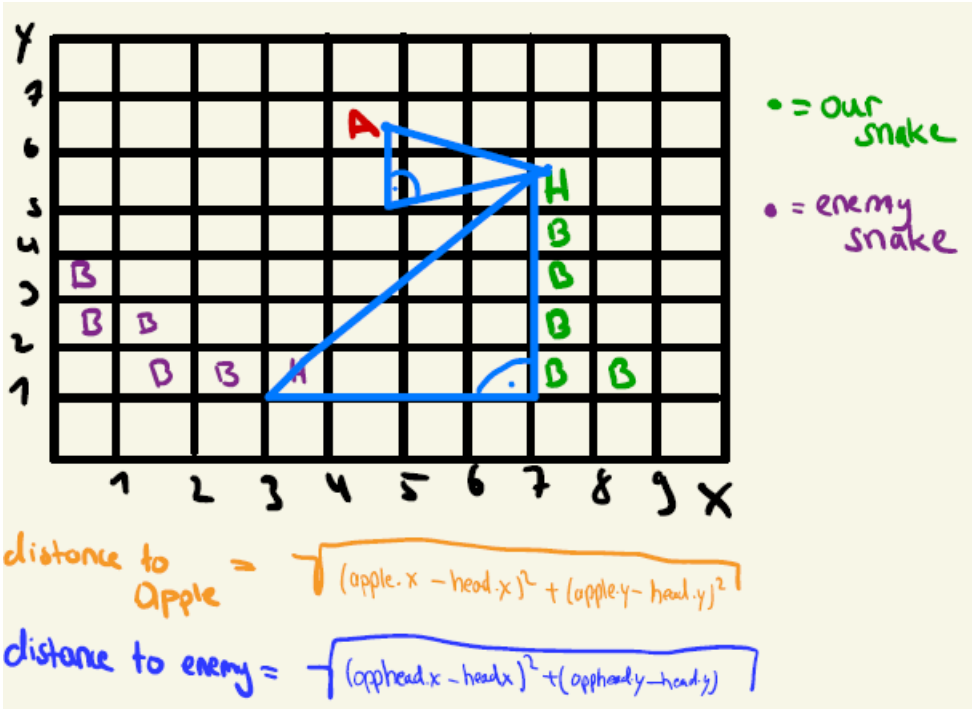
So far we avoid going backwards. Now we need to handle the maze boundaries, enemies body and our snake's body. To do so we used lambda functions to filter the maze boundaries with ".filter(d -> head.moveTo(d).inBounds(mazeSize))", filter the opponents body with ".filter(d -> !opponent.elements.contains(head.moveTo(d)))" and filter our body with ".filter(d -> !snake.elements.contains(head.moveTo(d)))".

```
  // avoids hitting the mazebounds, opponent body, and yourself
        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)))
                .sorted()
                .toArray(Direction[]::new);
```

However now we get to the more interesting part. The part where our snake needs to know how to decide in which direction it should go. We need to consider that our snake should take the shortest path to the apple. In theory this is really simple.



So first we need to know the costs of each direction to the apple. To do that we are calculating the distance by using the theorem of Pythagoras. After applying Dijkstra's algorithm to this graph, we get as result that our bot has to go to *left*. The following picture will describe how we calculate the distances. We have to calculate two distances before every decision, the distance to the apple and the distance to the opponent's head.



The code would look like this:

```
//calculate the distance to other snake
double distanceToOtherSnake = 0;
   distanceToOtherSnake =
   Math.sqrt(Math.pow((head.x - oppHead.x),2+Math.pow((head.yoppHead.y),2));
   System.out.println("Distance to other snake: " + distanceToOtherSnake);
```

We need the distance to the other snakes head to decide whether our snake should dodge the opponents head or not. Because it only makes sense to dodge the opponent's head if it is one pixel away from our snakes head.
To handle the distances for each direction we used array lists. We basically calculate for each direction our snake possibly could take the distance to the apple and to the other snakes head. You probably wonder why we should consider the distance to the enemy in this case. This is reasonable doubt but it is necessary for dodging the opponent's head. Because we take the directions which is the furthest distance to our opponent's head, to dodge his head.

```
ArrayList<Double> distancesToApple = new ArrayList<Double>();
ArrayList<Double> distancesToOpp = new ArrayList<Double>();
// mapping each direction of notLosing with the distance to the apple
for (Direction dir: notLosing) {
        //calculate distances and put it in the arraylist
        int newHeadCoordinateX = snake.getHead().x + dir.dx;
        int newHeadCoordinateY = snake.getHead().y + dir.dy;
        double x = apple.x - newHeadCoordinateX;
        double y = apple.y - newHeadCoordinateY;
        double distance =Math.sqrt(Math.pow(x,2)+Math.pow(y,2));
        distancesToApple.add(distance);
        x = oppHead.x - newHeadCoordinateX;
        y = oppHead.y- newHeadCoordinateY;
        distance = Math.sqrt(Math.pow(x,2)+Math.pow(y,2));
        distancesToOpp.add((distance));
}
```

Now after knowing the distances our bot can look for the best directions. We are looping through our notLosing array and looking for min, max indexes of distances. So which direction is the one with the closest distance to the apple and the direction which is the furthest distance to the head of opponent.

```
double max;
double min;
int indexOfMinDistance = 0;
int indexOfMaxDistanceToOpp = 0;
min = distancesToApple.get(0);
max = distancesToOpp.get(0);
 // choose the index of smallest distance
 // choose the index of furthest distance to opponents head
for(int i = 1; i < notLosing.length; i++){
   if(distancesToApple.get(i) < min) {
       min = distancesToApple.get(i);
       indexOfMinDistance = i;
       // System.out.println(min);
   }
   if(distancesToOpp.get(i) > max){
       max = distancesToOpp.get(i);
       indexOfMaxDistanceToOpp = i;
   }
}
```

As last step our bot needs to decide which direction it should take based on the distance to the other snake and the possible directions of notLosing. If there is a possible direction which leads us not to die and where the distance to opponent snake's head is higher then 2 the bot takes the shortest path to the apple. If the distance to the opponents head is less or equal to 2, the bot dodges the head of the opponent. If none of these conditions met the bot just takes a valid move.

```
if(distancesToApple.size() > 0 && distanceToOtherSnake > 2){
    System.out.println("using shortest path");
    return notLosing[indexOfMinDistance];
} else if (distanceToOtherSnake <= 2) {
    System.out.println("dodging opponents head");
   return  notLosing[indexOfMaxDistanceToOpp];
} else{
    System.out.println("using valid moves");
    return validMoves[0];
}
```

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

hello world