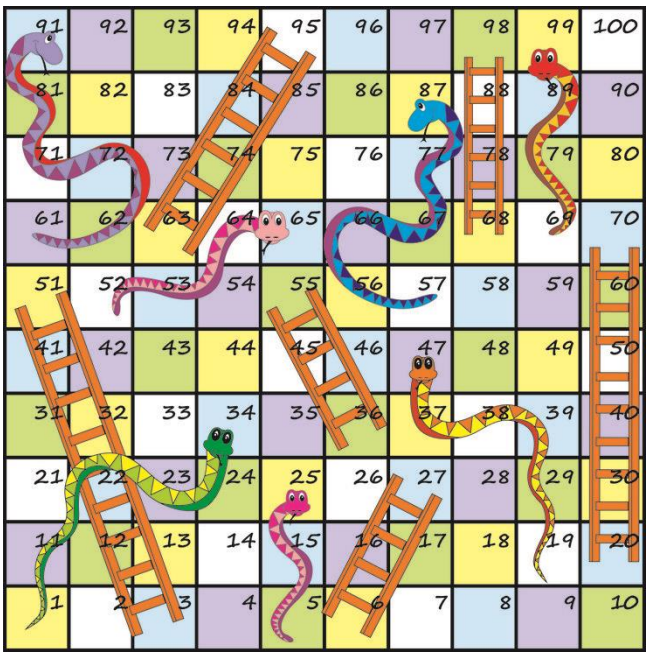
Snakes, Ladders and Data Structures

George Serafeim Vellios **Theologou Petros**

velliosg@ece.auth.gr **petrtheo@ece.auth.gr**



Problem description:

In this work we are asked to create the snake game with some variations (the stairs are destroyed after being used once and there are apples which add or subtract points from each player depending on their color). In order to achieve this we need to create the following classes:

1. *Snake*
2. *Ladder*
3. *Apple*
4. *Board*
5. *Player*
6. *Game*

The first three classes contain only their setters , getters and constructors , while the following classes are more complex (they contain additional methods than those mentioned) and use the previous classes as well. The class names are representative of their respective functions in the game.

Algorithm:

As mentioned above, the first three classes of the program are relatively simple. Nevertheless, it is considered correct to make a reference to the variables of each class.

- ***Snake Class :***

The objects of this class are the snakes on the board.

Variables :

- **int snakeId :** the serial number of the snake
- **int headId :** the number of the board tile where the snake's head is located
- **int tailId :** the number of the board tile where the snake's tail is located

- ***Ladder Class :***

The objects produced by this class are the ladders found on the board.

Variables:

- **int ladderId :** the sequence number of the ladder
- **int upStepId** : the number of the board tile where the base of the stairs is located
- **int downStepId :** the number of the board tile where the top of the stairs is located
- **boolean broken :** a boolean variable indicating whether the ladder is broken or not. If the ladder is broken, then it cannot be used again.

- ***Apple Class :***

The objects produced by this class are the apples on the board. Apples are divided into two categories: red, which increase the player's points, and black, which decrease the player's points.

Variables:

- **int appleId :** the serial number of the apple. Each color has its own serial number.
- **int appleTileId :** the number of the tile where the apple is located
- **String color :** the color of the apple. It can be red or black
- **int points :** The points each apple gives or takes away from the player

The next two classes are the main ones for the game operation and the last one is the game itself. More specifically:

- ***Board class :*** this class is the board of the game.

Variables:

- **int N, M:** the two dimensions of the board
- **int [][] tiles :** A 2-dimensional array containing the sequence number of each tile on the board
- **Snake [] snakes :** An array of snake objects that contains all the snakes on the board
- **Ladder [] ladders** : An array of objects of Type ladder that contains all the ladders present on the board
- **Apple [] apples** : An array of apple's type objects that contains all the apples in the array.

In addition to setters , getters and constructors , the board class also contains two more complex functions.

void createBoard (): this method initializes all values of the board elements.

> Initially variables headId , tailId , upStepId , downStepId , appleTileId are declared. These variables are later given random values which, after appropriate checking, will be placed in the corresponding class variables. Boolean type variables are also declared , which will be used later to check conditions. The numbers are then placed on the appropriate tiles on the board.

In order to do this it is not enough to simply initialize the array but special care must be taken. Thus, the outer for starts the iteration from i = N -1, where N is the number of lines, i.e. from the last line , and with each iteration it subtracts one unit until it reaches the first line. Inside for, the right variable is checked . This variable is Boolean and indicates whether the numbers on the line should be placed to the right or not. If so, then a for is executed that starts from the leftmost column and places the numbers in the appropriate positions. The number to be placed in each position is declared by the int variable num which starts at 1 and increases with each number placed. If the variable right is false , i.e. the numbers must be placed to the left, then a different for is executed starting from column M -1 and moving towards the smallest column.

```
boolean right=true;
int num=1;
for(int i=N-1; i>=0; i-- ) {
    if(right) {
        for(int j=0; j<M; j++) {
            tiles[i][j]=num;
            num++;
        }//endfor
        right=false;
    }//endif
    else {
        for(int j=M-1; j>=0; j--) {
            tiles[i][j]=num;
            num++;
        }//endfor
        right=true;
    }//endif
}//endforMegalo
```

After all the numbers are placed on the board tiles, the rest of the elements are created. Initially values are given in the table snakes . This happens inside a for that iterates as many times as the size of the array. Inside it is the do structure while which gives the variables headId , tailId random values with the maximum number num . Inside the do while also checks to see if the headId variable takes a value that matches an already existing snake head. If so, the boolean variable headTouch becomes true and will prevent do from exiting later while . To exit, the following conditions must not apply: headId - tailId)<=N || headTouch==true
i.e. headId - tailId >N so that the head and tail of the snake are not on the same line, and headTouch is false so that no two snake heads are on the same board tile. After exiting this structure, the values of the variables headId , tailId and snakeId are assigned to the corresponding snakes in the array. Then the sequence number of the snakeId is incremented by one .

```
do {
    headTouch=false;
    headId = (int) (Math.random()*num+1);
    tailId = (int) (Math.random()*num+1);
    for(int j=0; j<snakes.length; j++)
        if(headId==snakes[j].headId) {
            headTouch=true;
            break;
        }
}while( (headId-tailId)<=N || headTouch==true);
snakes[i].setHeadId(headId);
snakes[i].setTailId(tailId);
snakes[i].setSnakeId(snakeId);
snakeId++;
}
```

In a similar way, values are given to the variables of the other two elements of the board, i.e. the ladders and the apples, with the following differences:

- The exit condition of do while for stairs is !(upStepId - downStepId <=N || stepDownTouch || snakeTouch) , that is, as before, the start and end of the staircase must not be on the same line and also the boolean variables must stepDownTouch and snakeTouch , which indicate whether the ladder is touching its bottom with another ladder or a snake's head, to be false.

- The exit condition of do while for apples is touch==false where touch is a boolean variable indicating whether the apple is on top of a snake's head. Then with the boolean variable boolId it is selected whether the apple will be red or black and the corresponding values are given to the variables of the apples in the table.

public void createElementBoard ():

Create 3 tables containing the dashboard elements and print them

After creating the three arrays, we insert elements inside it via 2 for . First, we place the character "___" on all the elements of the three tables, which indicates that

there is nothing in the corresponding tile of the tableau. Then inside for loops that are repeated as many times as the size of each of the three arrays we are interested in each time, there are if conditions which, when true, place in the corresponding position of the array a series of characters, as requested by excercise.

```
for(int j=0;j<M;j++) {
    elementBoardSnakes[i][j]="  ___ ";
    elementBoardLadders[i][j]="  ___ ";
    elementBoardApples[i][j]="  ___ ";
    for(int k=0; k<snakes.length; k++) {
        if(snakes[k].getHeadId()==tiles[i][j])
            elementBoardSnakes[i][j]=" SH"+ snakes[k].getSnakeId();
        if(snakes[k].getTailId()==tiles[i][j])
            elementBoardSnakes[i][j]=" ST"+ snakes[k].getSnakeId();
    }
    for(int l=0; l< ladders.length; l++) {
        if(ladders[l].getDownStepId()==tiles[i][j] && ladders[l].getBroken()==false)
            elementBoardLadders[i][j]=" LD" + ladders[l].getLadderId();
        if(ladders[l].getUpStepId()==tiles[i][j] && ladders[l].getBroken()==false)
            elementBoardLadders[i][j]=" LU" + ladders[l].getLadderId();
    }
    for(int a=0; a<apples.length; a++) {
        if(apples[a].getAppleTileId()==tiles[i][j] && apples[a].getColor()=="red" && apples[a].getPoints() != 0)
            elementBoardApples[i][j]=" AR" + apples[a].getAppleId();
        if(apples[a].getAppleTileId()==tiles[i][j] && apples[a].getColor()=="black" && apples[a].getPoints() != 0)
            elementBoardApples[i][j]=" AB" + apples[a].getAppleId();

    }
}
```

Of particular interest are the conditions in ladders where, in addition to checking whether the corresponding box has a start or end of a ladder, it is also checked whether the ladder is broken and in apples where the color of the apple and its points are also checked as well as if it is zeros, that means it doesn't exist. Finally, all three tables are printed with the following combination of the two for iterations :

```
System.out.println("elementBoardSnakes");
for(int i=0; i<N; i++) {
    for(int j=0; j<M; j++) {
        System.out.print(elementBoardSnakes[i][j]);
    }System.out.println();
}
```

- ***Player Class :***

variables:

int playerId : the player code
String name : the name of the player
int score : the player's score which is given by the algebraic sum of the points of the apples the player eats
Board board : the board the player is playing on

Function int move(int id, int die):

It accepts as arguments the id of the player as well as the number he rolled.

First, an array of 5-position integers is defined in which the sizes to be returned will be placed. Then the variables that will be placed in this table are defined as well as an int variable mediumId indicating the player's position on the board after the die is rolled. Finally, a boolean variable found is defined that shows if any element was

found in the box where the player is. The main body of the function consists of a do structure while with output when found = false , i.e. when no other element is found in the box the player is in. This check allows each player to make more than one move if deemed necessary. Then, in a for iteration , performed for all the snakes on the board, it is checked whether the player's new box coincides with the head of a snake. If this is true, the player's position (mediumId) in the snake queue is changed, the corresponding message is printed, the total number of snakes is increased by 1, and the variable found becomes true .

```
for(int i=0;i<board.getSnakes().length;i++) {
    if(mediumId==board.getSnakes()[i].getHeadId()) {
        mediumId=board.getSnakes()[i].getTailId();
        System.out.println("Player got bit by a snake, Ouch!");
        snakesNum++;
        found=true;
    }

}
```

The check for ladders and apples is done in a similar way to the various ones that:

- In the check for the stairs it is also checked if the staircase is broken or not and, if it is not, and after the player's position changes, the broken variable of the staircase becomes true and the total number of stairs is increased.
- The check for apples also checks if the apple exists (board.getApples()[k].getPoints () != 0) . If there is, then the player's points change according to the color of the apple (red increases, black decreases), the points variable of each apple becomes zero so that it ceases to exist, and the number of red or black apples increases, depending on the color .

Finally, the appropriate values are given in the table that is returned and the meeting is exited.

- ***Class Game :***

  Variables:
  Int round : The current game round

The game is implemented in this class. Inside main there are some output commands inside comments. These print the positions of both players before, after the die is rolled, and after the move is made. Thus, a new game is created , a new board with dimensions 20x10, three ladders, three snakes and six apples and the createboard function is called to create the board. Then two players p1 , p2 are created with names player 1 and player 2 respectively, a table of 5 integers that will have the results of the move function and some variables for its operation. The game is implemented inside a do structure while where each player rolls their dice and moves according to the move function until one of the two players reaches the end of the board.

Finally, the total rounds played, the score of player 1 and player 2 as well as which player won are printed.

Part 2

In the second part of the project a Heuristic Player is created which can choose which number to roll trying to maximize its score.

HeuristicPlayer class :

Extends Player class . Its first variable (ArrayList < int > path) is the pchoices the player makes. An additional variable **int [] statistika was added to this class** of size 4, which will contain how many total snakes, ladders, red apples, and black apples the HeuristicPlayer encountered throughout the game.

Function public double evaluate(int currentPos , int dice):

First, an int variable newPosition sums the player's current position and the roll to find which position they choose to go to. Then, in order to find what will be the actual final position of the player (if he went down a snake or climbed a ladder) and how many points he got or lost from apples (if he encountered any), the body of the public function was copied into the function **int move (int id , int die)** of the class **Player** with some changes (there is no longer an array that returns how many items were encountered, and no print commands for them. But the most important one is that the commands that change the board, i.e. the ones that make ladders and apples disappear) have been removed. Then the weights of the player's movement are calculated from the formula

f (steps , gainPoints) = steps *0.7 + gainPoints *0.3

or with the corresponding command

double pontoi = (0.7 * (newPosition - currentPos)) + (0.3* (score - bPoints));

where bPoints the player's points before the move, reset the player's points, and return the evaluation of the player's move.

Function public int getNextMove (int currentPosition):

Its purpose is to return the roll the player chooses.

- Double max =-1000000; (the maximum value of the player's move evaluation. Initially set too small to be used later in finding the true maximum)
- int maxDie =-1; (the die that must be played in order to have the most points in the movement evaluation. It is also initially set to impossible for the same reason as max)
- **double** [ ][] pinakas = **new double** [6][2]; (a two-dimensional array containing in the first column the points of the player's roll rating and in the second the roll itself).

- int [] matrix1 = new int [5]; **And** int [] matrix2 = new int [8]; (will be explained in the following chapter)
- int bPoints = score ; (the player's score before the move)

First, the following code:

```
for(int i=0;i<6;) {
    pinakas[i][0]=++i;
    pinakas[i-1][1]=evaluate(currentPos,i);
    if(pinakas[i-1][1]>max) {
        max = pinakas[i-1][1];
        maxDie = i;
    }
}
```

Uses quicksort structure to find the die with the most rating points and those points are found. Then the move function is called for the player to move with the found die argument. The returned matrix of the function is stored in matrix 1. Matrix 2 is the one that will be added to the path variable of HeuristicPlayer and has a dimension of 7, while the content of each position is reported in the code with comments. Finally, it returns the player's new position.

public function void statistics ():

Its purpose is to find the player's statistics. The function consists of an **int variable round =1** which shows the number of the round whose information is printed and a **for (int [] tmp : path) loop** that is done until path ends and prints each time the die played and the elements it fell on, if it fell. Also, inside the for , the values of the statistika table are increased so that at the end of the repetitions it contains the total number of items encountered by the player. Finally, and outside of the iteration, this table is printed.

<u>Game Class :</u>

The class has one variable, int round , which shows which round of the game we are in. This class is extended to accommodate for the new Heuristic Player.

Class functions:

- constructors
- setters
- getters
- **Map< Integer,Integer > setTurns (ArrayList <Player> players):**

Int [ ] diceMatrix = new int [players . size ()] (an array of size the number of players that will have their dice

Map < Integer , Integer > unsortedMap = new HashMap < Integer , Integer >(); and **LinkedHashMap < Integer , Integer > sortedMap = new LinkedHashMap <**

Integer , Integer >(); (Two maps containing in the key positions the Ids of the players and in the value positions their dice, one unsorted and the other sorted.

With the following code:

```
do {
    flag=false;
    for(int i=0; i<players.size(); i++)
        diceMatrix[i] = (int)(Math.random()*6+1);
    for(int i=0; i<players.size(); i++) {
        for(int j=0; j<players.size(); j++) {
            if(diceMatrix[i]==diceMatrix[j] && i != j)
                flag=true;
        }
    }
}while(flag==true);
```

Values are given to the dice roll so that no value is the same as any other. Then the players' Ids are placed on the unsorted map with the dice that are in the diceMatrix table with the code:

```
for(int i=0; i<players.size(); i++)
     unsortedMap.put(players.get(i).getId(), diceMatrix[i]);
```

And the map is sorted with the following command:

```
unsortedMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEachOrdered(x-> sortedMap.put(x.getKey(), x.getValue()));
```

Finally, the sorted map is returned.

main function :

First, one is created **Game game = new Game (1)** (starts from round 1), one **Board board** with arguments the values given to us in the declaration, two players **p 1 and p 2** , one normal and the other Heuristic , a **matrix** table where the return of the move function will be stored , an **ArrayList < Player > players** , a **map map** and **3 int** variables for the function operation. Then the **board is created** , printed, the players are placed in the **ArrayList players** and is created **map** . The game takes place in a **do structure while** which terminates when the round reaches 100 or one of the two players terminates. Inside it, with a **for (int pId : map . keySet ())** players always play in the order set on the map and with an **if else** the appropriate commands for each player are executed. The round also increases. Once the game is over, the winning player is printed (how one wins is defined by the utterance), with the code:

```
double score1,score2;
score1 = 0.7*pos1 + 0.3*p1.getScore();
score2 = 0.7*pos2 + 0.3*p2.getScore();
if(score1>score2)
    System.out.println("Player 1 won");
else if(score2>score1)
    System.out.println("Player 2 won");
else {
    if(pos1>pos2)
        System.out.println("Player 1 won");
    else
        System.out.println("Player 2 won");
}
```

Finally the score of the two players and the total number of rounds played are printed, as well as the **HeuristicPlayer statistics** .

Part 3

***Problem description*:**

In part 3 are asked to create a MinMaxPlayer which chooses the best move taking into account the opponent's counter move and uses **Nodes** to achieve this.

Node Class :

The functions implemented in the class are the setters and getters for all variables, the constructors (with and without arguments) and an additional **static function void printTree (Node root)** which prints a tree of depth 2 rooted at Node root . (The function does not print all the variables of the class but only those that were needed to test the correct operation of the program).

Class MinMaxPlayer :

Player class . Its first variable (ArrayList < int > path) is the path the player will follow. An additional variable **int [] statistika was added to this class** with size 4 (set in constructors), which will contain how many total snakes, ladders, red apples, and black apples MinMaxPlayer encountered throughout the game.

Function public double evaluate(int currentPos , int opponentCurrentPos , int dice, Board board):

We chose to create a new function that will also take into account the position of the opponent. It was observed that the only case that needs to be checked is when our player is in front of the opposing player but not at a distance greater than 6. Then, if the opponent reaches an element that benefits him positively (eg a red apple), the player will prefer to eat this element than to make a move with a higher rating, so as to damage the opponent.

First, the following variables are declared:

- newPosition : the position of the player after the move
- int [ ] temp = new int [5] : an array set equal to the return of move (…)
- int ofa=0,ofl=0; Two variables showing how many red apples and stairs respectively are in front of the opponent and less than 6 spaces away. They are initially set to 0.
- double pontoi=0.0 : The rating points returned
- boolean reach = false : A variable that indicates if our player is in front of the opponent and at a distance less than 6

  First a value is given to the reach variable . If true, then both ofa , ofl variables are given values with two iterations. Then, with the following code:

```
for(int i=2; i<7; i++) {
    pinakas[i-2][0]=i;

    Board newBoard= new Board(board);
    int bPoints=score;
    temp=move(opponentCurrentPos, i, newBoard,false);
    newPosition=temp[0];
    pinakas[i-2][1] = (0.7 * (newPosition-currentPos)) + (0.3* (score - bPoints));
    score=bPoints;

    if(pinakas[i-2][1]>max) {
        max = pinakas[i-2][1];
        maxDie = i;
    }
}
opponentBestPosition= opponentCurrentPos + maxDie;
```

  We find out what is the best move for the opponent. The roll starts from 2, since the closest position our player can be in front of the opponent is the one right next to him. So we care about moves for dice greater than 2. The opponentBestPosition variable stores the position the opponent would have if he made the move with the highest rating. With the following code:

```
if((opponentBestPosition == (currentPos + dice)) && (ofa!= 0 || ofl != 0)) {
    pontoi=200;
    return pontoi;
}
else {
    reach=false;
}
```

  It is checked whether our player can actually make the given move (since the position with the most points for the opponent may be further ahead than his current position) and whether with this move he eats a red apple or climbs a ladder. If both are true, then the evaluation points are set equal to

200 and returned. Otherwise reach is set to false to use the evaluation function created from the second exercise with a few changes (not explained because we think the code is pretty self-explanatory). Finally, if with his move the player terminates, the evaluation score becomes equal to infinity.

Function int chooseMinMaxMove (Node root) :

This function selects the best move for the player by implementing the MinMax algorithm . This function works for a tree with depth 2. From depth 2 it selects from the 36 opponent moves (6 for each move of our player) the 6 with the lowest rating (the rating at depth 2 is equal to the rating of our move minus the opponent's move rating), depth is set to 1 and of those 6 the algorithm chooses the highest and returns the die corresponding to it. So, in a for iteration that is done for each of the 6 children of Node root , another iteration takes place which, from these 6 children, selects the one with the lowest rating and sets this rating equal to the rating of the father (depth 1) `root .getChildren().get( i ).setNodeEvaluation( evalMinOp );`

Then, it chooses the largest of these 6 and returns its index incremented by 1.