13 Sep 2015

MODELLING ROLE-PLAYING CHARACTER CLASS PERFORMANCES USING RANDOM WALKS

FYTN03

Johan Book

Department of Astronomy and Theoretical Physics, Lund University

Project done together with Ola Olsson

Contents

1 Introduction

The performance of each class in role-playing games such as DnD and Pathfinder is widely discussed. The aim of this study is the investigate said performance for individual characters of different classes picking fights at random. It will build upon a random-walk model where combatants must fight to until death. The model will serve measure of solo performance.

2 Theory

2.1 General role-playing and Pathfinder statistics

A common role-playing game is Pathfinder (which is a variant of DnD). The player gets to create a character of a certain class to fight enemies. Each character has some hit-points which corresponds to a score determining how healthy or hurt the character is. In order to kill an enemy the character must inflict damage on the enemy reducing the enemy's hit-points. After the character has slain an amount of enemies it levels up gaining more hit-points, a higher attack damage and is allowed to use more advanced abilities.

The character has a set of ability scores which correspond to e.g. how strong, intelligent or charismatic the character is. This does in turn affect how well a character can perform any kind of action such as damaging an enemy, climb a tree or casting a spell.

Table 1: Generalized ability scores for some classes.

Class	Strength	Dexterity	Constitution	Wisdom	Intelligence	Charisma
Paladin	18	11	13	8	8	14
Fighter	18	14	13	8	11	8
Rogue	13	18	11	8	14	8
Ranger	14	18	11	13	8	8
Wizard	8	14	13	8	18	11
Cleric	13	8	14	18	8	11
Skeleton	14	14	-	8	-	10
Orc	16	14	13	8	11	8

Further explanation of the statistics can be found in the Pathfinder core rule book [2].

Each class has some fixed attributes (or which here is considered fixed), such as hit-points gained when levelling. The classes will have some armour which is presented with an armour class score.

Table 2: Statistics for different Pathfinder classes.

Class	Speed [feet]	Base hit-points	Hit-points per level	Armour class
Paladin	20	12	5	23
Fighter	30	12	5	20
Rogue	25	8	4	17
Ranger	35	10	5	20
Wizard	30	6	3	16
Cleric	30	10	4	23
Skeleton	30	6	3	16
Orc	20	12	5	18

Most of the presented classes does also have a special attribute. Below some of them are displayed, somewhat modified and simplified. First strike does in this context designate the first strike a character does on another, initiating a combat.

Table 3: Some bonuses and whether classes is ranged or not.

Class	Is ranged	Bonus
Paladin	False	Can heal during combat. Extra damage against skeleton and orc.
Fighter	False	
Rogue	False*	Damage bonus if first strike (does not affect skeleton).
Ranger	True	
Wizard	True	Can cast spells.
Cleric	False	Extra damage against skeleton. Regain full hp after combat. Can cast spells.
Skeleton	False	
Orc	False	

* Can be considered to be ranged.

3 Methods

3.1 Basic idea

Characters of different classes will be positioned in some space V. They are then let to wander randomly until two meet and initiate a fight until death. This is meant to correspond to how often these characters meet and fight in some fictional world. In most game-plays characters will form teams. However this will be neglected for the sake of simplicity.

3.2 Initial character placement

Let the number of characters to be placed from class i be C_i. Define M as the number of discrete possible positions in V. For each allowed position do the following:

1. Draw a random number X. If $X \leq C_i/M$ place a character of class i at the probed position and decrease C_i by 1. Do only place one character at the same position.
2. Decrease M by 1.

3.3 Movement and basic combat structure

Combats have been greatly simplified. Movement and combat was done by an algorithm presented below. Damage was calculated according to the rules of combat specified in the Pathfinder core rule book [2].

1. Pick one alive character P at random.
2. Move P using a random step with a maximum step length of the speed of the character depicted in table 2.
3. Pick any other character O in a range less than the character speed. If none can be found skip remaining steps.
4. Let P inflict damage on O.
5. If O is not alive increase the level of P by 1 and skip remaining. If P is ranged and stroke a first strike go to 4 else let O attack P.
6. If P is alive go to step 4 else increase the level of O by 1.

One step consist of the process above being run N times where N is the current number of alive characters. After one step is completed all characters are healed and their spells are reset.

3.4 A continuous model

Instead of the discrete model proposed in [1] a continuous one was used. The reason for this choice is that for the studied types of reactions do a continuous model seem more realistic. One major drawback is that one must at each time step check a particles distance from every other particle - which is a resource-intensive process.

3.5 Comments

The speeds presented in table 2 was reduced by 1/7 in order to slow down the simulation.

4 Results

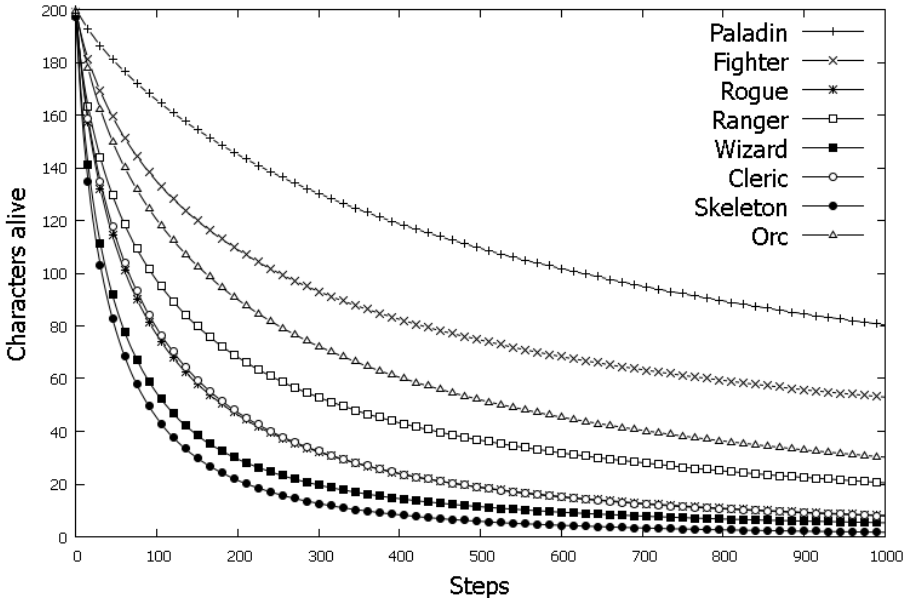

Figure 1: Evolution of the system.

The average number of characters in each class was studied during 1000 simulations. Note that in the graph only a few data points are shown by points in order to increase readability.

Table 4: The parameters used to generate the data.

Parameter	Value
Simulations	1000
Steps	1000
System size	1000^2
Characters of each class	200

Table 5: The percetage surviors of each class.

Class	Percentage survivors
Paladin	38.8 %
Fighter	25.6 %
Rogue	3.9 %
Ranger	9.9 %
Wizard	2.6 %
Cleric	3.8 %
Skeleton	0.8 %
Orc	14.6 %

5 Analysis

In fig. 1 does the paladin character class dominate - which agrees with the general consensus. The fighters and orcs did perform well which is not very surprising considering they are specialized in close combat. The only close combat class which did not perform well was skeletons. This is due to that this class is naturally weak.

The result seem realistic considering the model set-up. It implies that solo combats favours classes specialized in close combat. The remaining classes are as most effective when working as a team.

One can draw the conclusion that when constructing a team of characters at least one need to be specialized in close combat. From this simulation can the paladin be considered the optimal choice.

A Code: Main

```
package src;

import gui.GUI;
import java.util.ArrayList;
import java.util.Map;
import character.Character;
import character.Class;
import react.Collision;
import settings.Settings;
import util.Pair;
import util.Printer;
import util.Random;
import util.Util;

public class Main {
   private final ArrayList<Character> characters;
   private final Map<Class, Double> densities;
   private final Settings settings;

   public Main() {
      settings = new Settings();
      Initializer placer = new Initializer(settings);
      characters = placer.getInitalCharacters();

      // Assumes equal amount of each class
      densities = Util.initiateMap(settings.numbPALADIN);
   }

   // Simulates all time-steps. If supplied with a printer writes data to file
   // Stores densities of each step in map
   public void simulate(Printer printer, Map<Class, Double>[] map) {
      // Print file header
      if (printer != null)
         printer.write("#" + Class.classesToString());

      // Take each step
      for (int step = 0; step < Settings.steps; step++) {
         int num = characters.size();
         for (int j = 0; j < num; j++)
            simulate();

         // Heal characters
         healAllCharacters();
```

```
      if (printer != null)
         printer.write(step + "\t" + densitiesToString(false));

      if (map != null)
         map[step] = Util.addMaps(map[step], Util.copyMap(densities));
   }
   if (printer != null)
      printer.close();
}

// Simulates time-steps together with a GUI
public GUI createGUI() {
   return new GUI(characters, settings.getDimensions());
}

// Put all densities into a string
public String densitiesToString(boolean normalize) {
   String string = "";
   int norm = normalize ? densities.size() : 1;
   for (Map.Entry<?, Double> entry : densities.entrySet())
      string += entry.getValue() / norm + "\t";
   return string;
}

// Simulates one time-step
private void simulate() {
   // Create lists to store Characters to remove
   ArrayList<Character> remove_list = new ArrayList<Character>();

   // Pick and move one random Character
   Character selected_Character = characters.get(Random.nextInt(characters
         .size()));
   selected_Character.move();

   // check for reactions/fights
   for (int i = 0; i < characters.size(); i++) {
      if (characters.get(i) == selected_Character)
         continue;

      // Check for reaction
      Pair<Boolean, Character> reaction = Collision.check(
            selected_Character, characters.get(i));

      // if there is no reaction continue
```

```
            if (!reaction.first)
                continue;

            // If there is a fight remove the one who loses
            if (reaction.second != null) {
                remove_list.add(reaction.second);
                densities.compute(reaction.second.getType(), (k, v) -> v - 1);
            }
            break;
        }

        // Remove and add Character
        for (Character Character : remove_list)
            characters.remove(Character);
    }

    private void healAllCharacters() {
        for (Character character : characters)
            character.heal();
    }

    // Main [int: number of simulations][boolean: display GUI]
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        int numbSim = 1; // number of simulations to run
        boolean useGUI = true; // whether to display GUI
        if (args.length >= 1) {
            numbSim = Integer.parseInt(args[0]);
            useGUI = false;
        }
        if (args.length >= 2)
            useGUI = true;

        // Initiate a printer to write the final result
        // from each simulation and a map to record the average.
        Printer printer_summary = new Printer("summary");
        printer_summary.write("#Sim\t" + Class.classesToString());

        // Setup map for recording final values and average for each step
        Map<Class, Double>[] step_average = new Map[Settings.steps];
        for (int i = 0; i < Settings.steps; i++)
            step_average[i] = Util.initiateMap(0);

        for (int i = 0; i < numbSim; i++) {
            Main simulation = new Main();
```

```
            // Initiate GUI if used
            GUI gui = null;
            if (useGUI)
               gui = simulation.createGUI();

            // Run a simulation and print data to file
            simulation.simulate(new Printer("result_sim_" + (i + 1)),
                  step_average);
            printer_summary
                  .write(i + "\t" + simulation.densitiesToString(true));

            System.out.println("Simulation " + (i + 1) + "/" + (numbSim)
                  + " completed");

            // Close GUI
            if (useGUI)
               gui.dispose();
         }
         printer_summary.close();

         // Calculate average for each step and print
         Printer average_each_step = new Printer("step_average");
         average_each_step.write("Step\t" + Class.classesToString());
         for (int step = 0; step < Settings.steps; step++) {
            String[] string = Util.mapToString(step_average[step], numbSim,
                  false);
            average_each_step.write(step + "\t" + string[1]);
         }
         average_each_step.close();

         // Calculate final average and put into string
         String[] results = util.Util.mapToString(
               step_average[Settings.steps - 1], numbSim, true);

         // Print the final average from all simulations
         printer_summary = new Printer("summary_average");
         printer_summary.write("#" + results[0]);
         printer_summary.write(results[1]);
         printer_summary.close();

         System.out.println("All simulations complete");
         System.exit(0);
      }
   }
```

B Code: Initializer

```
package src;

import java.util.ArrayList;
import java.util.Random;

import character.Character;
import character.Class;
import settings.Settings;

public class Initializer {
   private Random random = new Random();

   private int numbPALADIN;
   private int numbFIGHTER;
   private int numbROGUE;
   private int numbRANGER;
   private int numbWIZARD;
   private int numbCLERIC;
   private int numbSKELETON;
   private int numbORC;

   private double M2;

   private final Settings settings;

   public Initializer(Settings settings) {
      this.settings = settings;
      numbPALADIN = settings.numbPALADIN;
      numbFIGHTER = settings.numbFIGHTER;
      numbROGUE = settings.numbROGUE;
      numbRANGER = settings.numbRANGER;
      numbWIZARD = settings.numbWIZARD;
      numbCLERIC = settings.numbCLERIC;
      numbSKELETON = settings.numbSKELETON;
      numbORC = settings.numbORC;

      M2 = Math.pow(settings.M, settings.n);
   }

   public ArrayList<Character> getInitalCharacters() {
      ArrayList<Character> molecules = new ArrayList<Character>();

      // Place molecules evenly
```

```
    for (int x = 0; x < settings.M; x++)
        for (int y = 0; y < settings.M; y++) {
            double[] position = new double[settings.n];
            position[0] = (int) settings.pixel / 2 + x * settings.pixel;
            position[1] = (int) settings.pixel / 2 + y * settings.pixel;
            chanceOfCharacterPlacement_Evenly(molecules, position);
        }
    return molecules;
}

// For evenly placed molecules
private void chanceOfCharacterPlacement_Evenly(
        ArrayList<Character> molecules, double[] position) {

    // Create molecule if random nr [0,1] < nr small_cubes/nrMolecules

    if (random.nextDouble() <= numbPALADIN / M2 && numbPALADIN > 0) {
        molecules.add(new Character(Class.PALADIN, position));
        numbPALADIN--;
    }

    else if (random.nextDouble() <= numbFIGHTER / M2 && numbFIGHTER > 0) {
        molecules.add(new Character(Class.FIGHTER, position));
        numbFIGHTER--;
    }

    else if (random.nextDouble() <= numbROGUE / M2 && numbROGUE > 0) {
        molecules.add(new Character(Class.ROGUE, position));
        numbROGUE--;
    }

    else if (random.nextDouble() <= numbRANGER / M2 && numbRANGER > 0) {
        molecules.add(new Character(Class.RANGER, position));
        numbRANGER--;
    }

    else if (random.nextDouble() <= numbWIZARD / M2 && numbWIZARD > 0) {
        molecules.add(new Character(Class.WIZARD, position));
        numbWIZARD--;
    }

    else if (random.nextDouble() <= numbCLERIC / M2 && numbCLERIC > 0) {
        molecules.add(new Character(Class.CLERIC, position));
        numbCLERIC--;
    }
```

```
      else if (random.nextDouble() <= numbSKELETON / M2 && numbSKELETON > 0) {
         molecules.add(new Character(Class.SKELETON, position));
         numbSKELETON--;
      }

      else if (random.nextDouble() <= numbORC / M2 && numbORC > 0) {
         molecules.add(new Character(Class.ORC, position));
         numbORC--;
      }

      M2--;
   }
}
```

References

[1] Lund University
SP, Lecture 2, FYTN03 Computational Physics.

[2] Jason Bulmahn, Gary Gygax, Dave Arneson - Paizo
Pathfinder roleplaying game: core rulebook.
Paizo Publishing 2009.