2018

C++ RISK Simulation to Generate AI

David Sky Nunez
Parkland College

David Sky Nuñez

CSC 123 Honors Project

Instructor: Gerald Kolb

Fall 2018

C++ RISK Simulation to generate AI

In the fall semester of 2018 I took an introductory C++ class. For my honors project for this class I set out to create a way to run thousands of simulations of situations in RISK the board game, and use that information to help make strategic decisions when playing the game.

Firstly, let's dig into RISK the board game according to the rules published on Hasbro.com. At its core it is a game of attrition where 2-6 players take control of nations and try to take over the world via warfare. This takes place on a world map with mostly abstracted regions and countries which act as "spaces" or "territories" that each player may occupy with tokens representing their forces there. Then, as each player takes their turn, players will replenish their forces, adding more tokens, and then proceed to make attacks against adjacent territories occupied by another player. The core element of my simulation program begins by focusing on the chance of success in these attack actions.

Let me explain in detail how an attack action works. First, the player whose turn it is currently will declare an attack to a particular location. Then, the player defending this attack will get a number of dice equal to the number of tokens on their space that is getting attacked, to a maximum of three dice. The attacking player can also only have a maximum of three dice in the attack, but must always leave one token behind in the space they are

attacking from. This means if an attacker has three tokens, they can only attack with two, but if they have four tokens in their territory they can attack with three. Now to rolling the actual dice. Both players will roll the dice they have available. Then, both players will rank their dice highest to lowest. Then comparing the highest dice of the defender, and highest dice of the attacker, if the attacker dice value is greater than the defender dice value, the defender loses a token. If the defender dice value is equal to or greater than the attacker dice value, the attacking player loses a token. Do this comparison for each next highest pairing of attack and defense dice. When the attacker or defender get different amounts of dice, only the paired comparisons do anything, and extra lone die are ignored ("Risk The World Conquest Game").

The first simulation: Finding percent chance to succeed with continuous attacking given the number of attackers and defenders as inputs:

My first job was to find a way to simulate this attacking action in C++, inputting how many attacker tokens and defense tokens there are, and getting an output for the results. First I created a random number generator **sub-function** called ***RollDice*** using "rand()%6+ 1" to simulate a dice with 6 sides, that would return a random value between 1and 6. Then I built the ***attackRun*** function that would take a number of dice for defender and attacker, roll them using ***RollDice,*** and rank them highest to lowest for each. Still within the ***attackRun*** function, comparisons were then made to determine who won of each dice pair, and thus who would lose tokens.

Second, I needed to account for the fact that a player who attacks can always immediately attack again on the same turn. I chose to use a **loop** where the simulation

would continue until either player could either not defend, or not attack any longer, and recorded that result. I called this a ***"Sim Loop"*** **or** ***attackSim.*** I created a **while loop,** that depended on the total number of tokens of defender and attacker, which would run ***attackRun*** after ***attackRun.*** Essentially, this told the program to keep attacking with whatever tokens are left until either the attacker wins, or can no longer attack. In the ***howManyDice*** function, the program determines how many dice to use by both the defender and attacker in the next attack action, which was helpful to **increment** the number of dice as either attacker or defender ran out of tokens and began rolling less dice in each subsequent attack run. All this together got me to the point I could determine a result of continuous attacking, with either the attacker winning, the attacker failing, or in some strange cases, both players becoming simultaneously unable to attack or defend. This was now a complete **simulation** of a continuous attacking action between two territories.

```
while (simFidelity >0)
{
int simAttacker = attackers;
int simDefender = defenders;
///Running Attack Sim Once to Completion
while ((simAttacker > 0)  && (simDefender  > 0))
{
howManyDice(simAttacker, simDefender, atkDice, dfnDice);
attackRun(simAttacker,   simDefender,  atkDice,  dfnDice);
}
victoryCheck(simAttacker,simDefender,rareCase,attackFail,victory);
simFidelity=simFidelity  - 1;
}
```

This was not enough however, as the true power of a computer program running simulations is being able to run millions of them and quickly derive trends in the data in a way that is simply impossible by hand. This led me to the final step of the simulation portion of the program, running the ***Sim Loop*** thousands of times for a particular given

value of attacker and defender tokens. I used another **while loop,** which would run over and over until the number of simulations described by the **variable *SimFidelity*** reached zero. This **loop** would effectively take the number of simulations requested and count down to zero, and tally up the results from each simulation using the ***victoryCheck*** function after each individual simulation. Overall, the user would input the number of simulations, the number of attackers, and the number of defenders, to get results of percent chances to win, lose, or tie from numerous simulations as the attacker. This result of course would be for a specific single scenario that was input, such as finding that in a 10,000 simulations with 4 attackers and 3 defenders, the attacker would win overall 60% of the time. This program file was called SimpleRiskAnalysis.cpp.

The Second Layer; creating a table of data for percent chance to succeed given number of attackers and defenders:

Now I had something functional, but I wanted to make it more usable. Having to input the number of attackers and defenders each time was tedious, and left players to decide what percent chance to win was worth it when considering an attack. So, I had an idea, which was to generate all of the simulation results beforehand for each combination of attackers and defenders. This meant I needed to run a series of simulations for 2 attackers and 2 defenders, 3 attackers and 3 defenders, and so forth, and organize the percent chance to win from each case into a readable chart. Then players could simply reference this chart quickly.

To do this, I created an even larger **loop** in a separate function, which would send the number of attackers and defenders to the ***attackSim*** function by **calling** it and **return** a

percent chance to win. This large **loop** would **increment** itself, counting up the numbers of attackers, and then the number of defenders, creating a two dimensional table of data. This program file was called RiskTable.cpp.

```
cout << "#DFN  | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 ||| " <<endl;
cout <<  ----------------------------------------------------------------"<<endl;
int y = 2;
while (y < 11)
    {
    attackers = y;
    if (attackers <  10)
    cout << "ATK " << attackers << " |";
    if (attackers > 9)
    cout << "ATK " << attackers << " |";
    int x = 1;
    while (x < 12)
        {
        defenders = x;
        attackSim( simFidelity, attackers, defenders, atkDice, dfnDice);
        x = x + 1;
        }
    cout << "||" <<endl;
    y = y + 1; }
```

Then, instead of printing this out onto the screen for a user to see, I placed this data into a **two dimensional array. A two dimensional array** is effectively a coordinate system in the form of data, in this case with an X and a Y. The X would represent number of attackers, and Y would represent number of defenders, and at the position in the data from these coordinates, I stored the final percent chance to win returned from the simulations. At any time, I could ask the computer for the percent chance for a given number of attackers and defenders, or print out the whole chart. The only downside to this, was every time I started up the program, it would have to do all of these thousands upon thousands of simulations at the beginning, making it slow at first, but then completely fast after initializing.

Building a better assistant program that recommends when to attack or ally with other players:

Over the next couple weeks, I reflected on where to go with the program next. It was by no means an Artificial Intelligence, but more of an automated assistant, to help inform players before making decisions. I wanted to expand on this assistant idea, so I decided to build a framework around the existing simulation data foundation.

In the game of RISK, players can at any time ally with other players. This leads to a lot of discussion among players to form alliances and later on betray each other and return to regular warfare. There is no in game restriction on when to attack or ally, so it is solely up to players. Now while my program in its current state could not understand the physical map, it could try to analyze who is a valid threat, and who is not. Using this idea, I tried to build a function that would take in how many total tokens each player has on the entire board each turn, and compare that to the player's own amount, and make a recommendation about who to ally with and who to attack ("Risk The World Conquest Game.").

There were a few different steps of logical analysis going on here in the data. Firstly, if an opponent is exceptionally small, they are not much of a threat, and a player's tokens would be better allocated at more significant threats. On the opposite side of things, if a particular opponent is so massive they are an immense threat, it may be beneficial for self-preservation to ally with this giant threat temporarily. In this way, there is a certain mathematical region where a player should look to attack opponents, and if they are too large or too small, a player should focus efforts on something else.

There was one more step to this though. I currently had a huge chart of percent

chance to win, but how would the program know when to attack and when not? 60%, 70%, 80% chance to win? As a stopgap measure, users will set the **variable *"aggression level"*** of the Risk Assistant at the beginning of the program. This value helps decide at which threshold the program will recommend to attack, or not. When a player chooses a lower aggression level, the program will only recommend attacking actions when it is highly likely to win, while a higher aggression level will lead the program to take more risk, and even recommend attacking at percentages nearing 50%. The aggression level also played a role in how the program decided on which player to ally with, or betray and attack.

Account for gaining continents:

In the Game of RISK, players gain tokens at the beginning of each of their respective turns based on how many territories they hold at the beginning of that turn. Essentially, players gain more tokens for every multiple of three territories they own, and also larger quantities of tokens if they own entire continents. This makes owning a continent especially valuable.

As my final step in this project, I wanted to account for this increased relative value. Any time a player wanted to make a query for whether or not they should attack, the program would ask for whether winning this battle would grant the player an entire continent or not. This would affect the logic in comparing percent chance to win and aggression data, making the program tend to be more aggressive when there is more value at stake.

The current iteration of RISK Assistant framework:

Initial Setup
-Asks for simulation fidelity, and aggression level.
-Runs simulations and generates percent win values
Turn Loop (Occurs Each Turn)
-Asks for opponent total token data
-Returns advice about who to ally with or attack
Attack Query Loop
-Asks for number of attacker and defenders
-Returns recommendations to attack or not
*Proceed back to **Turn Loop** again, until game ends.

The current state of the program and how it operates is demonstrated above.

Upon starting the program, it takes an aggression level and generates percent chances to win in each case by running background simulations. Once this is done it proceeds to taking data for the first turn. After deciding who to attack or ally with, it will allow users to query whether a certain attack is recommended or not any number of times for many different cases. Once users end this attack querying, it will proceed to the next turn, and repeat this process over and over until the game ends. This program file was called RiskAl.cpp.

Works Cited

Brothers, Parker. "Risk The World Conquest Game." *Www.hasbro.com,* 1959, https://www.hasbro.com/common/instruct/risk.pdf

Savitch, Walter J., and Kenrick Mock. *Problem Solving with* C++. Pearson, 2018.