Matchmaking Coding Challenge

Diego de Palacio Ruiz Cabañas

September 20th, 2020

Table of Content

Introduction

When I first read the challenge instructions, I was tempted to implement a system to find the best possible mathematical distribution from a big chunk of data (all the available players) in two small sets (teams) with the most similar and accurate combined rank between such sets.

The system could repeat the process, taking each time the best possible combination from the players left in the big chunk of data and that would be the matches.

However, we care about the player experience, satisfaction and fun and this is not possible with such a system as one scientific study [(1)], done with LoL players concludes (more about that and about the approach that I took in the next sections of this document).

[Joke] Probably will never be possible so have such fun and satisfaction for players playing against “Ann Owen” anyways XD. From the provided Json file, she had to spend almost 300 million years in 1 millisecond of duration games to achieve her results ;)

Content of what I’m delivering

The implementation of the proposed coding challenge is divided into:

- This document, with all the information about the code, the design and possible future improvements;
- 2 Unity’s projects one for the Server side related code and another containing a Client simulator in charge or sending multiple client data to the server, joining or dropping to the game lobby;
- Build scripts for generating the client and the server executables for macOS and Windows;
- Executables for macOS and Windows of both projects;
- A spreadsheet with all the data and the processing of the original player provided data to convert it to players with rating and category, based on statistical data (in-depth explanation about it in the section “About preparing the provided data of players” section of this document).

How the source code is organized

The implementation of the challenge is splitted in two different projects, one for the server containing about 75% of the code and one for a client simulator to simulate having clients joining to our new competitive online team-based game.

Most of the source code (around 90%) is inside of the “Assets/Scripts” folder on both projects, with the exception of code contained on the “Assets/CI” folder (build scripts) and on the “Assets/Tests” folder (unit test code).

The code inside of the Scripts folder is splitted in 4 folders, depending on the functionality in Core (Matchmaking related), Networking, Data Management and UI.

The main classes are inside of the “Assets/Scripts/Core/MatchManager.cs” script on the server project and inside of the “Assets/Scripts/Core/PlayersManager.cs” script for the client project.

Both projects with the source code and executables for Windows and MacOS are included.

To avoid possible conflicts, it's better to open the projects using Unity 2020.1.5f1.

Configuration options

Some "compilation-time" configuration options can be changed on the instances of Unity's Scriptable Objects, located inside of the "Assets/Config" folders of both projects, including rating, matchmaking and network options.

The next runtime configuration options can be changed directly on the UI of the server-side project, including the team size for matchmaking, the required similarity percentage for such matchmaking and if the matchmaking should be by rating or by category.

Modeling and Design

The matchmaking process on my implementation is done in a sequential way, following the next steps:

1. Preparing the data to be compatible with a Elo based rating;
2. Creating groups of players, candidates to become a good match for a game;
3. Qualifying such groups to decide if they're or not a good match;
4. Once found a good group, then distribute the players evenly in two teams;
5. Finally, updating the category and ratings of the players after the game.

Each of these 5 steps is explained in detail in the upcoming sections.

About preparing the provided data of players

In order to convert the "simple-data.json" data converted to Elo rating usable data, some assumptions has been made:

1. The players where the provided data (5 players highlighted on the spreadsheet attached) was corrupted, impossible or missing, was placed in the middle of the distribution by assigning manually 1 game won and 1 game lost;

2. Once this was done, the wins/losses ratio was used to calculate their percentile on the player distribution "p" with this formula

$$p = \frac{wins}{(wins + losses)}$$

3. Because for our players we're going to use the Elo system with categories, I choose to use the same ranking used by the USCF [(3)];

4. For approximate the current ranking and category of the players where the data was provided, I assume that the distribution of players will be similar than the USCF Distribution Chart [(10)];

5. For each player was found first where in the percentile was located to estimate his category and then I use a linear interpolation between the range of such category to estimate the rating with a little bit more of precision;

6. Finally, a new json file was created "sample-data-ranked.json" with the results, having as fields the name, the category and the rating.

About grouping players for matchmaking

After preparing the player's data to be used in an Elo based rating model, the next step is to figure out how to group players together in a way that keeps the game fun and fair.

A scientific study shows that "players generally enjoy games where their team is better or the game is even" [(1)]. The player enjoyment is related to both the game outcome and how well their team works together.

The same study mentions that "Players generally enjoyed the game more when they felt their team was better, and less when their team was worse." also is mentioning that "The most cited reason for losses was inferior player skill and lack of teamwork."

For that reason, the matchmaking system should avoid, as much as possible, grouping players with huge gaps in their personal ranking together in the same team. For that reason, my approach is taking the players with closer ranking between them at the moment of doing the matchmaking for the ones that are waiting in the lobby.

The implementation starts by trying to find a good match for the players with better rating first, because they're more likely to be more demanding on the fairness of the match and also because they're probably the ones that are more engaged with our game.

About qualifying the groups of players to decide if they are a good match

LoL used a formula based on the Elo rating [3] before the Season 3 [9], because it's simple and easy to use for matchmaking players in different games and sports, so I chose to use it in my implementation.

The current implementation includes the two “Skill-based” suggested approaches on the white paper “Matchmaking and Case-based Recommendations” [2]. One or the other can be chosen on the UI of the runtime configuration of the server-side project:

1. Average skill rating of the game

$$Sim(p, m) = 1 - \left| R_p - \frac{\sum_{i=1}^{|m|} R_i}{|m|} \right|$$

Where:
- “Sim(p, m)” is the similarity of the player “p” to join the match “m”;
- “|m|” is the number of players in the match;
- “R_i“ is the normalized ranking (in the range of 0..1) of the player “i”.

2. Weighting category differences

$$Sim(p, m) = \frac{\sum_{i=0}^{N} DistW(i, Cat(p)) \cdot numPlayers(m, i)}{|m|}$$

Where:
- “Sim(p, m)” is the similarity of the player “p” to join the match “m”;
- “i” represents the category;
- “DistW(i, Cat(p))” is a normalized weight (in the range of 0..1) that depends on the distance between the categories “i” and the category where the player “p” is. The longer distance between categories, the smaller the weight;
- “N” and “|m|” represents the total amount of players in the match;
- “numPlayers(m, i)” is the number on the match “m”, currently ranked on the category “i”.

The formula that I decided to implement for “DistW(i, Cat(p)) is a simple linear interpolation:

$$DistW(i,\ Cat(p)) \ = \ 1 \ - | \ Cat(p) \ - \ Cat(i) \ |$$

Where:
- “Cat(p)” is the normalized category where the player is;
- “Cat(i)” is the normalized “i” category.

However, for a more “relaxed” similarity (higher probability of having players of different categories with higher similarity), a quadratic formula can be used instead:

$$DistW(i,\ Cat(p))\ =\ 1\ -(\ Cat(p)\ -\ Cat(i)\ )^{2}$$

On the other hand, for a more “strict” similarity (disallowing players from different categories to be on the same match), a square root formula can be used:

$$DistW(i,\ Cat(p))\ =\ 1\ -(\ Cat(p)\ -\ Cat(i)\ )^{1/2}$$

Even having the square root as an “expensive” operation shouldn't be a problem, because the number of categories to be analyzed is small.

In my implementation, after calculating the similarity for each player to join to a specific match, I’m taking the lower similarity value of them to decide if a certain group of players are a good match.

About splitting the players between teams

Once the group of players is chosen, a new optimization challenge arises: how to split the players in two different teams with the fairest distribution.

For solving such a specific challenge, different existing algorithms were analyzed, due to the fact that it’s a type of a Partition problem [(4)].

One possible alternative is to use an specific implementation of Dynamic programming [(5)] to solve a Subset sum problem [(6)] by creating a table of booleans where the solution can be found as soon as the table is complete, as it’s explained on the best ranked answer of a stack overflow thread about a very similar challenge [(7)].

Nevertheless, if the difference in the ranks of the worst and best player is big, the table will also be very big, with the number of columns equal to the sum of all the ranks of the selected players (a considerable sum for a 5x5 match).

Because of that, from my point of view, a better approach for matches with teams of relative small size, is to remove the worst player from the set, decrease all the other rankings by subtracting such worst rank, and then search for the best combination of n/2 - 1 elements (the worst player will be part of this specific team) that can approximate better the average of the modified ranks.

That means finding the players “p” to be placed on the team “A”, together with the worst player, that minimize the next formula:

$$\sum_{i \varepsilon A} mp_i \; - \; \frac{1}{2} \sum_{j=1}^{n} mp_j$$

Where:

- “A” is the team in where the worst player will be;
- “mp_i“ is the modified rank of the player “i” (rank of the player minus the rank of the worst player in the match) and n is the total amount of players in the match;
- “n” is the total number of players on the match.

The performance of this approach is good to find the solution because the maximum number of sums that needs to be calculated is equal to:

$$\frac{(n-1)!}{(n/2)!}$$

For example in the case of a 3x3 match, the maximum number of sums will be 20, for a 4x4 match 210, for a 5x5 match 3024, for a 6x6 match 55440, etc.

Is a maximum number of sums, because some combinations can be automatically excluded during the process, because the search is done in order and we can stop searching for combinations that are worse than the best solution that we found until that moment in the algorithm.

For a bigger team size, the Greedy algorithm [(8)] can be used. This heuristic approach does not find a best solution, but it terminates with an acceptable result in a reasonable number of steps. Finding an optimal solution to such a complex problem for big teams, typically requires unreasonably many steps.

About updating the ratings of the players after the match

In my implementation, I’m following the recommendation on the white paper “Matchmaking and Case-based Recommendations” [(2)] to update the particular rating of an specific player by calculating the increase (if winning) or decrease (if losing) of rating with every other opponent of the other team (as if was a regular 1-1 match) using Elo rating and then add or decrease the average of those ratings.

Support resources of the projects

Some building scripts and unit tests are included as part of my implementation, and some information about them is explained in the upcoming sections.

About building scripts

The build scripts are located inside of the “BuildScripts” folder.

The builds are created on the ‘[Project]/Builds/[Platform]’ path, where ‘Project’ is “MatchmakingServer” or “MatchmakingClient”.

Warning: The previous build is deleted while executing one of the build scripts, so move the build to a different folder if you want to preserve an older version.

The build scripts were created to be executed on MacOS with the default installation path of Unity (using Unity Hub) of the 2020.1.5.f1. For executing them from a different OS or using a different installation path of Unity, the script's content needs to be updated accordingly.

To create a build, just execute from the command line, one of the build scripts, depending on the platform that you want to build. MacOS and Windows are currently included, but scripts for other platforms can be easily added.

The build scripts are calling Unity's code in charge of creating the builds. You can find such code in the scripts located in the respective “Assets/CI/” folder of both Server and Client projects.

Troubleshooting

- [Permission denied in MacOS] While trying to execute the build scripts in MacOS, if you have a “permission denied” message, execute ‘chmod +x [build script path]’ command first in order to set the executable permission on the build script, and allow it to be run.
- [Error building player because build target was unsupported] Make sure the module for the platform that you want to build is included on the Unity installation.

About unit tests

The unit tests can run through Unity's Test Runner window. There are implemented some EditMode and PlayMode tests.

The naming convention for the tests is: “[TypeOfTest]_[ActionTested]_[ResultExpected]”.

Currently there are only a couple of unit tests implemented on the Server side project, but the process to create more tests and to have some created on the Client side project is very similar.

Future improvements

There are million ways to improve this particular projects, but I want to mention some of the possibilities that came to my mind during the time that I spend on creating them:

Matchmaking

Currently adding a new player to the server is calculating all the similarities on the players in the lobby, including some that were previously calculated, so it's desirable to only calculate the similarities on the groups that involve the players that are added.

Add wins or loose streaks to the process of matchmaking. A good idea of how to implement this can be found on the "Introducing Momentum to the Elo rating System" document [(11)].

Add uncertainty factor on the current system, like Glicko [(13)] or TrueSkill [(12)] in order to improve the speed of placing new players to their real rank

Add time decay to the process of matchmaking, maybe increasing the uncertainty factor if the player didn't play for some time

Player Experience

To improve the player experience, if the players in the lobby does not have a good similarity, a timer can be added. After some time, a notification can be sent to the player offering the option of playing an unranked game or to keep waiting. Players waiting median times under 60 seconds were found for LoL games in the "The Importance of Matchmaking in League of Legends and its Effects on Users" [(1)] study. This fact can be used while implementing such a feature.

On the other hand, if there are not enough players, adding some AI managed players to participate in the game could be a good option.

Data

Saving the data to a proper database (or at least to the disk) every time a player rating is updated.

Currently, the player's name is used as the key for querying the data. Changing the key to be an integer based ID will be better.

Networking

Sending the messages in binary format and in bigger chunks when possible, to reduce the data and calls that need to be sent.

Encrypt the messages to improve the security instead of sending plain Json based strings;

Make the Network configuration available (port and address) to be changed in runtime

Change the networking code to receive and send messages to multiple clients, instead of using the "Client simulator"

Select time size and ranked type for individual player on the client side, instead of the server

Update the project to use a 3rd party networking library or the new Unity's native networking API as soon as it becomes ready.

Other

Add more relevant and extensive unit tests;

Implementation of multithreading for the algorithms. Currently all of them are being executed on the main thread.

Sorting the players by rating on the "Client Simulator". Currently they are only sorted at the beginning, but their positions are not changed after updating their ratings and categories.

Keep all the text in one place (instead of hard-coded) to make it easier changes or localization if needed.

Comparing the current algorithms with other implementations with some players of our target audience, taking into account their opinion about their experience to improve the system.

References

1. The Importance of Matchmaking in League of Legends and its Effects on Users. WPI Interdisciplinary Qualifying Project. Jonathan Decelle, Gabriel Hall, Lindsay O'Donnell (http://web.cs.wpi.edu/~claypool/iqp/lol-match/report.pdf). **Note**: The study was done with only 23 unique participants, so the results are limited, but I took it as good enough for the time limitations in this specific "coding challenge".
2. Matchmaking and Case-based Recommendations. Jorge Jiménez-Díaz, and Belén Díaz-Agudo. Dep. Ingeniería del Software e Inteligencia Artificial. Universidad Complutense de Madrid, Spain (http://sce.carleton.ca/~mfloyd/iccbr11games/papers/Jimenez-Rodriguez.pdf).
3. Elo rating system. Wikipedia (https://en.wikipedia.org/wiki/Elo_rating_system#:~:text=According%20to%20this%20algorithm%2C%20performance,the%20number%20of%20played%20games.)
4. Partition problem (https://en.wikipedia.org/wiki/Partition_problem).
5. Dynamic programming (https://en.wikipedia.org/wiki/Dynamic_programming).
6. Subset sum problem (https://en.wikipedia.org/wiki/Subset_sum_problem).
7. Algorithm for fair distribution of numbers into two sets (https://stackoverflow.com/questions/1507830/algorithm-for-fair-distribution-of-numbers-into-two-sets).
8. Greedy algorithm (https://en.wikipedia.org/wiki/Greedy_algorithm).
9. Elo rating system (https://leagueoflegends.fandom.com/wiki/Elo_rating_system).
10. USCF Regular Rating Distribution Chart. November 1, 2004 (https://web.archive.org/web/20080928012537/http://www.uschess.org:80/ratings/ratedist.php).
11. Introducing Momentum to the Elo rating System (https://www.ufs.ac.za/docs/librariesprovider22/mathematical-statistics-and-actuarial-science-documents/technical-reports-documents/teg418-2069-eng.pdf?sfvrsn=243cf921_0).
12. TrueSkill (https://en.wikipedia.org/wiki/TrueSkill).
13. Glicko rating system (https://en.wikipedia.org/wiki/Glicko_rating_system).