Rated Multiplayer Competitions

Samuel White
swhite@gyassa.com
June 22, 2011

1 Overview

This document is an explanation of a Java program that I wrote that simulates randomly grouped multiplayer competitions and implements a rating system that gives individual ratings to the participants. I also attempt to explain why I was motivated to write this program and what I hoped to achieve. At the start of the process I was more concerned with creating an Elo rating system for multiplayer teams, but by the end of the process I started exploring issues that were more relevant to traditional usages of rating systems such as chess. In particular, I became intrigued by certain types of failures in the Elo rating system and the lack of any clear explanation for the cause. This document will show this evolution.

2 Multiplayer Games

The sports golf, bowling, and running all have one thing in common. They all have a mechanism for rating a player and having some type of score that defines how good a particular player is at the sport. They also all have a handicapping system to allow for unequal players to have a legitimate contest. I believe that this ability to quantify the abilities of the participants is why these sports have a large number of participants, who at best can be described as “minimally competent”.

In other sports, the non-elites quickly learn that participating in the sport, except with known friends, is a painful thing. The more serious elite players dominate the game and make playing the game a fruitless exercise for anybody else. Sports of this type, like soccer, that have a large participation from children through third grade tend to have only dedicated elite fanatics participating by the end of high school.

In chess, which some say is a sport for the mind, a carefully constructed rating system is used, called the Elo rating system. It does a good job determining how to assign matches between players so that the matches are guaranteed to be competitive. The Elo system does have some deficiencies, but it is widely used and the best proposed alternatives are usually tweaks that provide slightly better outcomes for particular types of player participation and improvement rates. A quick survey with google on Elo ratings and chess will bring up articles that point to the Elo rating system as the primary reason for the sustained increase in popularity of chess in the U.S. starting in the middle of the twentieth century.

With the advent of online gaming, the need for an effective rating system has become even more apparent. Some rating systems do exist, but usually the focus on the rating system is on single person versus person competitions or rating a team of players that maintain a fairly permanent and unchanging roster. The rating systems are largely modeled on variants of the Elo rating algorithm and do a fairly effective job given their limited focus.

Recently, a new type of online team game has begun to gain prominence. The teams are randomly assembled and compete against each other. One example is the random battlegrounds in World of

Warcraft (WOW). But these competitions do not have an effective rating system and suffer from the same problems as encountered in many real world team sports. They also have problems that are unique to the online nature of the games. The winner of a competition is usually determined by which team has the most elite players on its team and players who are sufficiently weak have strong incentives to not even make a show of effort. And some players react by making hostile and non-constructive comments degrading the experience for everyone.

I believe that this does not have to be the case. In fact, I believe that online multiplayer team competitions offer a unique new opportunity to create a new type of stimulating gaming experience that is a step above the experience found in any other form of competitive game. I believe that this new type of game must have the following characteristics.

1. It must have a quick resolution. It should not require more than thirty minutes of time to sign up to be recruited for team play, get selected for a random team, and finish a competition. It would probably be better if that time were under twenty or even fifteen minutes. This allows casual gamers to participate when they have a bit of unpredictable idle time. It also means that if something else suddenly demands their time, they know their forced absence will not doom a game that has already required great investment of time from the participants.
2. The strength and abilities that are granted by the game to the player are not static but grow with continued participation by the player in more games. For example, if the player has an avatar character that character gets more powerful as the player participates in more games. There is a general comment to be made here. The most successful computer games seem to share a common characteristic. They allow the player to feel they are growing from a simpler, less capable state to a more powerful complex state over a long duration of time. It makes the participation feel more like an investment than just a simple random game. For example, WOW has refined this slow improvement model to a fine art. The larger the range of improvement, the better it tends to work. Evolving from lowly nonentity to demigod can provide powerful incentive for continued participation.
3. Because the strength and abilities given to the player by the game software is changing, it is possible to give rewards to players that improve the abilities granted to them by the game. More specifically, it is possible for these rewards to be given for accomplishing certain goals – such as winning a game. Rewards can also be given for achieving other objectives and reaching certain milestones. This capacity for providing rewards makes the games meaningful in a way that can not be achieved in other more static gaming environments.
4. Different players on a team should have different, crucial roles. It is better to allow players to choose from a list of possible types after they have joined the team rather than assembling teams that guarantee minimum numbers of each role. Allowing players to have different roles encourages a sense of team spirit and emphasizes cooperation skills. This works best if the players can switch around their abilities before the game starts to optimize the team configuration. If you don't allow this, it is possible to guarantee an appropriate mix of player types during the matching process but the simulation that I created shows that this can create significant distortions and delays in the queueing process for matches.
5. There should be larger agendas for the players to achieve. One typical approach is to make the players be part of a larger alliance and have some of the rewards go to the alliance. When creating teams for a match, it is better if the players all come from the same alliance to foster a sense of achievement for the greater community. Alliances can cause queueing problems, if some alliances are much smaller or less active than others. Alliances may need to be reconfigured or alliances merged to optimize the scheduling queue. At times it may be

necessary for alliance members to scrimmage against each other if the other alliances have too few players participating. At higher levels of play, this feature may have to be abandoned because the quantity of highly rated players may be limited. In that case, teams may have members from multiple alliances on them.

3 Battle Bots

In order not to talk about generalities, I will give a model for a specific game and describe its rules and mechanics. This will allow me to talk in specific and less confusing terms when describing the rating system and other aspects of the simulation.

Battle Bots is a theoretical online game with 10 randomly selected players on each team. Each player has a set (or "stable") of battle bots that they maintain. At the time the team is assembled, there is a two minute preparation period before the game commences. During that time players can choose which battle bot they will play and discuss strategy with their teammates.

Battle Bots fight on a large hex map about 200 hexes on a side. The game is turn based with all players entering their intended action simultaneously. Each side has a 15 minute clock and at the start of each turn each side is charged 5 seconds from its clock, and all the players have five seconds to move. If a player takes more than five seconds to move that additional time is subtracted from the player's team clock. If a player takes more than fifteen seconds to move, the player loses his action and the player's action is entered as "non action". If a side loses all of its time, then it loses the game. After all the players have entered their actions, the turn ends and the consequences of the actions are computed. For a five second period, the results of the actions are described in rolling text to both sides and visual markers on the playing field are manipulated accordingly. After this five second results display period, the next round starts.

If a bot is killed on the field of play, the bot will regenerate at a designated area on the map in 2-4 rounds of play. It is expected that bots will die and regenerate many times during a combat. The regeneration is quick enough to guarantee that one side cannot eliminate more than a few players from the other side at any time.

When a team wins, the winners are given rewards that allow them to make improvements to all or some of the Battle Bots in their stable. The improvements generally improve some fundamental characteristic about 0.2% or so. At times, larger rewards may be given for more important matches or the first match played by that player that day.

The Battle Bots have five fundamental attribute values.

Movement – How many hexes the bot can move per turn.
Damage – How much damage its main weapon can do.
Hit Points – How much damage the bot can take before it is destroyed.
Range – The number of hexes away an opponent can be and still be hit by the main weapon.
Initiative – If two players are contesting for an item or the results of a round are different if one player goes first, then initiative decides who acts first.

Generally, an average bot of the same power as another will take 10 shots to kill another bot. In other words the hit points will tend to be about 10 times as large as the damage value.

The attribute values are computed from a characteristic of the same name. Each characteristic is measured in 10s of thousands of points to allow small increments in improvement possible. The cost of improving an attribute value is carefully calculated to create a balanced game. For example, initiative may be easier to improve quickly than range.

The amount of characteristic improvement that can be made will degrade by a factor of 8 after a certain number of improvements are made on a bot. So a .8% improvement in a characteristic may now only improve the characteristic by .1%, once the bot is sufficiently powerful. Another degradation of 8 will occur once the bots reach an even higher level of power. Degradation will also occur naturally, because you get less of a percentage improvement when adding to a characteristic that is much larger than it was at the start, which may allow a less severe explicit degradation to create the desired outcome.

The simplest game is a capture the flag game. A flag is placed in the center of the field and any bot in an adjacent hex gets an action choice of “pick up flag”. If a bot successfully carries the flag back to an area marked as the home base of the bot's team, the team wins a point. After each point, the flag is reset and the first side to get 3 points wins. If two bots try to pick up the flag at the same time, the initiatives are compared after adding a small randomization factor and the bot with the higher resulting initiative gets the flag. If a bot carrying the flag dies, the flag is dropped into the hex in which the bot died and can be picked up by either team as opportunity dictates. A bot not carrying a flag can choose to move at double speed, but then it must spend one round taking one non-offensive action or move at regular speed for a round before they can attack an opponent. A bot can also choose to not move or attack, in which case it is assumed to be a “defensive” posture and all damage against it is halved. If the gaming mechanics for the design of the bots is controlled correctly, flags will not be carried successfully to the victory area until after the death of a few flag carriers. If two bots on the same team are next to the flag, it is possible to perform deception where it will not be clear for one round, which bot intends to pick up the flag.

During the starting period of two minutes before the game actually starts, players make a choice of a bot from their stable of bots. Some may choose bots that are high in hit points and speed to act as flag carriers. Others may choose high damage and range to act as artillery. Some others will choose high damage and hit points to act as tanks. And finally, some may choose high damage and speed to act as chasers.

There are still quite a few details that I will leave undefined. For example, can two bots occupy the same hex? What happens when two try to occupy the same hex? Initiative can determine who occupies the hex but what happens to a multi-hex move that failed when it tried to enter the last hex of its movement? Also, the characteristics are measured in 10s of thousands, but attribute values such as hex movement are measured in small numbers. What happens to improvements to a characteristic that are insufficient to bump up the attribute value by a perceivable amount? One solution is to give a percentage chance that the attribute value will be bumped up for a given round by computing what percentage of an improvement has been bought. Also, just how much does an attribute value cost and how much does it cost to increase the value given its current value? Does the cost for improving an attribute value go up if an attribute is improved past a certain point? One of the standard problems that all games of this type face is that there are usually certain choices of attributes, skills, and play pattern that are over powering compared to others. This is such a common problem that dealing with this issue as it occurs is called “nerfing” and for bots, the most likely way to nerf is to make certain attribute values cost more at certain value points than they otherwise might cost. Other patterns of play may also

be less successful than desired and they may need to be “buffed”, which is also a common correction made as designers of a game watch the actual outcomes of the implementation.

This game allows us to define two critical measurements that will be used when we start trying to apply a rating system to this game. The first measurement is “intrinsic skill”. The intrinsic skill of a player is not just in how well he plays the game, but also in how he chooses to design the bots that he plays, and which bot he chooses to use in a particular game given his expectations about his teammates and opponents. The intrinsic skill is also a measure of how well he cooperates and communicates with his teammates. The second measure is the player's equipment score. This is a measure of the total amounts of characteristic that were bought for the bot and the resulting attributes that were given to the bot.

We define the “skill” of the player to be the combination of the intrinsic skill and the equipment score. The “skill” of a player is an attempted measure of total outcome and it is by this evaluation that it will be judged. A true calculation of either intrinsic skill or equipment score for battle bots may be difficult, but their conceptual value is great and they will be referenced, when it comes to judging the effectiveness of a rating system.

4 Multiplayer Rating System

The main problem in a multiplayer game is that it is hard to determine which player actually contributed to the win or loss. There may be a high rated player playing in a game with lower rated players and lose, because the player's teammates are weak and not because the rating of the high rated player was higher than the player's skill and equipment. Likewise a low rated player could have an undeserved win by playing with very strong teammates. Also the randomness in outcomes increases with the number of players. In a perfectly modeled rating system, a 200 point difference in rating means that one player will win 75% of the time and the other 25% of the time. But what happens when you combine 10 such variances? Clearly, there have to be some simplifying assumptions made and a calculation done for combining random variances from each of the players.

There is one advantage that a typical multiplayer game has over other typical single player games. There is a measure of how well the player played in the game that is independent of win or loss. In the Battle Bot game, a player can get a “skill productivity” statistic by measuring things such as the number of bots that were damaged by this player that eventually died, how many bots were killed by a shot from this player, now much total damage the player generated against opponents, how many times the player delivered the flag to home base, and how few times the player was killed. Clearly some of these values have to be weighted more than others and constant experimentation is required to figure out the best way to calculate this statistic. But nonetheless, a viable statistic should be available to measure each player's productivity in the game.

Given a player's rating, the average rating of his team and the average rating of the opponents, it is theoretically possible to calculate the “expected skill productivity”. By comparing the actual “skill productivity” against the expected productivity, it should be possible to figure out whether the player was playing below or above his or her rating. Given the expected randomness and error in these values, only fairly severe differences between expectation and reality should cause much of an adjustment to the rating calculation after a win or loss.

5 Advantages of Ratings

The most common cited reason for using ratings is to create more equal games. But it has other uses.

Rating points can be spent or deducted. They can also be boosted by concerns independent of direct win or loss. Doing this allows rating points to be used as motivation to get proper behavior or to correct certain team play problems. For example, if one of the team members appears to no longer be playing the game, the other players on the team can mark this player as "away from keyboard". If two players on the team mark a player this way, the game can be stopped until a new player is brought in to replace the current player. However, this game play mechanic can be used to either try to halt play for abusive reasons or to boot out a badly equipped player. If players have to spend rating points (about four points is sufficiently costly) in order to boot out a teammate, then it is more likely that a player will be evicted from the game only if he truly is non performing.

Another idea is to put some rating points after a win into a small pool (about $1/4^{th}$ of a typical rating award given to a single player) and players can vote on who should get the extra rating points. If more than one player is voted for, the points are split in proportion to the votes. Of course, players cannot vote for themselves.

Also, if the statistics of game play are used to compute skill productivity and that is used to adjust how ratings adjustments are applied, players will have a strong motivation to put in a good effort even in clearly losing games.

One of the big things that ratings allow us to do is to give larger awards, if all the players competing are higher than a minimum threshold. This gives players a strong incentive to play well to get higher ratings, because it will allow them to upgrade their equipment faster. It will also add prestige to combats among higher rated players. Since equipment is such an important factor in the game, players will naturally progress up the rating ladder as their equipment improves and it will make getting the more expensive upgrades possible. This assumes that equipment gets more costly. In a prior section, I remarked that as a player progresses up past a certain level, equipment costs should become eight times what they were before. This leveling of equipment costs can happen more than once. If such a leveling system for equipment is used, then a player cannot get the best equipment in a reasonable span of time without boosting his rating substantially.

One way to handle how ratings boost awards is as follows. Compute the minimum rating across both teams. Then take the difference between this rating and a "experienced player rating floor" (a good value for this is 800). If the result is negative, there is no boost to the award. However, if it is positive, divide the difference by 200 and then take two to that power and use that as an award multiplier. So if 800 is the floor and the minimum rating is 1633, then (1633-800)/200 = 4 and two to the fourth is 16. This means a match with that minimum rating would get 16 times as much equipment rewards as a standard game. This can be clearly indicated as the game progresses and important sounding nomenclature can be used for each 200 point level such as experienced, skilled, expert level A, expert level B, master level A, master level B, grandmaster level A and so on.

6 Injecting Rating Inflation

One of the problems that rating systems suffer from is a constant rating deflation. Generally, the higher rated players quit and lower rated players enter the system. Each time this happens, the average rating decreases. To counter this, it is necessary to artificially inflate ratings. This is especially problematic with games that use equipment to slowly boost the total skill of a player over an extended period of time. In this case, the player is guaranteed to start out at a low rated state and eventually transition into a much higher state.

One way to account for this is to give a boost to the rating for every win by the player for an extended period. For example, for the first 800 wins the rating could be boosted by an extra rating point. Also, if the first win of the day gives an extra bonus equipment improvement, the rating can likewise get a bigger bump. This is precisely the approach that is used for this simulation and it does appear to make a significant improvement in the accuracy of the ratings.

There is one perk to this approach. Every new player can start with a rating of zero and climb up to much higher ratings over time. The rating inflation will move them quickly to their actual rating. Since equipment is such a decisive factor in outcomes and new players start at the bottom end of equipment, starting with a zero rating will not be far from their true rating. Taking this approach gives the player a bigger sense of accomplishment than if he started at some provisional intermediate rating and only moved slowly from there.

7 Limitations in the Elo Rating System

I will not go into the technical details of the Elo rating system. An excellent source for the algorithm and the source used for this simulation can be found by searching Wikipedia. But there are two driving facts. A player that is 200 rating points higher should have about a 75% (actually the more accurate figure is 75.97%) chance of winning. A player about 400 points higher should have a 90% (the more accurate figure is 90.91%) chance of winning if doing single player matches. Unfortunately, these results are hard to achieve even for players with fairly static ratings who play a lot of games. But the Elo system does do a reasonably good job of determining relative standing among players. But there are other problems. One of the main issues is that the rating lags behind the true skill of a player. For example, if a new player is improving rapidly, the rating will not keep up with the actual current skill of the player. Some players only play a few rated games a year and again the rating may not keep up with current skill.

To correct for ratings lagging behind skill, the K-factor in the Elo rating system can be increased to improve the rate at which the rating converges to the player's true rating. As a quick reminder, the K-factor is twice the amount a rating of a player would increase, if the player were to win against an opponent with an identical rating. For larger K-factors (greater than 24), the ratings tend to spread further apart from each other more quickly and the ratings converge more quickly to their expected value. But larger K-factors make the ratings significantly less predictive of actual game outcomes, if the skills of the players are relatively stable. By examining the simulations, a general fact can be deduced. A K-factor above 16 tends to make a noticeable degradation in the outcome prediction, but a K-factor of 16 or smaller is so small that the rating convergence tends to be too slow. Some variants of the Elo system take this into account by increasing the K-factor for players who have played only a few (or no games) recently or if the recent game outcomes demonstrate unusual variance from expected results.

In our multiplayer simulation, there is an extra piece of information that we can use to improve our ratings computation. The simulation emulates the collection of productivity statistics on all the players, and from these statistics, it is possible to determine whether a win was a marginal win, a solid win, or a blowout win. If the winner's team rating total is not too much larger than the loser's team rating total, then the margin of victory can be used to change the K-factor. By trying out various parameters, it was determined that doubling the K-factor for a solid win, and quadrupling it for a blowout win produced the best results. I should note that this technique for collecting statistics can be used for many single player game and it appears that using this extra information reduces the need to adjust the K-factor

based on other criteria such as duration since last game played. This allows us to use a base K-factor of 16 with good effect even if the players are new or their ratings are changing quickly.

8 Simple Game Model

In order to simulate Battle Bot multiplayer game and to determine the adequacy of a rating system, we have to simplify our gaming model. One simplification is to reduce a player to two numbers, intrinsic skill and equipment score. The sum of these two numbers is the skill of the player. To get the skill of a team we add all the skills of the team members. If one team has a higher sum, then it is a better team. Whether linear addition is the correct way to combine skills of various members of a team or whether equipment and intrinsic skill add linearly is beyond the scope of this investigation. But it does not much matter exactly how the team skill is computed as long as it creates results at least somewhat consistent with expectations. In theory, if a rating system works it does not much matter how players and teams arrive at a particular effective skill.

In the simulation program, I use integer arithmetic as much as possible. The computations tend to be a bit faster and there is more explicit control of the expected precision in the answers. But this does cause some tweaks in the number ranges for various values. An example would be skill.

Since skill is incremented in thousands of tiny increments with equipment improvements, the simulation cannot use the 400 point range that is typical of a rating system to differentiate between two players that are substantially different in ability. Instead, the simulation multiplies that value by 10 and says that two players that are 4000 skill apart are on different levels of skill with the better player having about a 91% chance of winning a match.

I then use a standard fact from statistics and probability that if you add N bell curve distributions with deviation X, the deviation of the sum of the bell curves is square root of N times X. So on a 10 player team, the variation in results is expected to be the square root of 10 times the variation of one player. In particular, if two teams have total skills that are about 12,650 apart, then the stronger team has about a 91% chance of beating the weaker team. As an example, if three team members are 4220 skill points better than their opponents and the rest of the team have identical skills when compared to their opponents, then this team has about a 91% chance of winning a match.

The next issue is coming up with a reasonable range for expected intrinsic skill and equipment score. In the simulations, I use a fairly large range for intrinsic skill. This puts more stress on the rating system than a smaller range. The current test default has a range 8000-16,000 for initial skill and a player can improve by an additional 8000-16,000 with practice. If such as skill range were applied to chess it would account for rating that went from 800 (novice) to 2800 (senior master) which is probably a significantly larger range than the skill range for a simple game such as Battle Bots.

The other issue is what randomization function to use when selecting values from the interval 8000-16,000. The easiest approach is to distribute the skills by a simple linear random function so that getting a top 5% skill is just as easy as getting a middle 5%. Though this does not match reality, it is simple to implement and it helps when it comes to making judgements during data analysis. It also creates a tougher test for the rating system and there is no evidence that a more bell curve like distribution would be a better test for the rating system, though it might produce more believable results. There is evidence that a bell curve distribution of player skill values would produce misleadingly good results without truly indicating whether the rating system works, because too many

matches would be between reasonably equal teams by chance alone.

I have also made the assumption that skill improves linearly with practice. This again is probably fairly far from reality. My belief is that this does not matter much since a rating system is validated by how well it handles a variation in skills and not in how these skills are achieved or whether the skills are distributed among the players in a particular way. For the simulation, the variation of skills was made purposively larger and flatter than would likely occur in any actual online game.

The equipment score range is expected to be fairly large, but it is also supposed to have more predictable qualities. In a prior section, I mentioned that rating inflation can be used to anticipate rating improvements from equipment rewards. This makes the equipment score a somewhat unusual element in the system. For my simulations, I used a range of 0 to 24,000.

When the equipment score is combined with the intrinsic skill range, the lowest possible skill is 8000 and the highest possible skill is 56,000 (assuming a player has practiced sufficiently). This should create an expected rating range of 0 to 4,800 (every 4000 skill equals 400 rating based on the Elo model and our simulation rules for randomization of competition outcomes). In practice, we do not get this full range because of issues in how skills spread out over the various rating ranges. This issue becomes the focus of a later part of this document and it is one of the fundamental unsolved mysteries of the Elo rating system. But there are factors in this simulation model that do not occur in more standard games, which contribute to this issue.

As an example, one of the problems is that higher ratings abet higher ratings since the higher ratings cause players to increase the rate at which their equipment score increases. This creates a feedback system into the ratings, which distorts expected outcomes. In particular, the middle of the ratings tend to have fewer players in it than one would expect in a more normal distribution of player skills. If you are below a threshold of skill, you tend never to get that much better, but if you get past a certain point then you tend to increase to a much higher level. The middle of the skill range tends to be unstable and players usually do not stay there for long.

Another problem is controlling the rate at which rewards are given out. In the simulation, I have used 2000 games as a typical period of time for a player to max out their potential. This is assuming that a reasonably large percentage of players will need at least 1000 games to perfect their intrinsic skill. In the simulation, the reward rate was specifically designed so that a fairly good player (top 25%) would reach the maximum equipment score after about 2000 games. A great player may achieve maximum equipment score in as few as 500 games. Because higher ratings cause a higher rate of rewards, the rewards have to be tuned after the rating spread is computed. But there are solutions if this is not done correctly at first. You can add yet another level of equipment upgrades that is 8 times as expensive as the prior equipment upgrades and keep doing this until you have created as high a bar for achievement as you like.

There is one thought that I have on the pace at which you give out rewards. It is desirable that your top 2-5% of players reach maximum equipment score after a thousand games or so. Elite players tend to be more interested in evaluating and judging intrinsic skill and are less motivated by the rewards of equipment upgrades. For Battle Bots, this may not be so true, but this is true for some of the more complex online games that are now available such as Blizzard's StarCraft. One nice extra would be capturing of the details of game play between elite players so that other players can learn from them. Games between elite players could also be watched live by other players giving a “spectator”

dimension to the multiplayer game.

Another issue is prolonging the interest by regular players in the game. One trick that works is to give bonus rewards and rating boosts to the first win that is achieved every day and reduce the awards for wins after the first. If a particular alliance of players is not playing as much as needed to create matches between alliances in a reasonable amount of time, then the bonus reward could be stretched out to multiple wins encouraging the players to play more games per day. Using a bonus reward for the first win (or first few wins) means that a player who plays only a game or two a day does not feel that he is falling that much behind a player who dedicates four hours a day to playing the game. It also stretches out the period of time that the player maintains interest, since they are playing fewer games per day than they otherwise might. The last benefit is that players will tend to quit playing for the day after a win with a big reward bonus, which has its own psychological positive effects.

The last issue is player retirement and the constant influx of new players. Generally, if a player is not doing that well the player will quit sooner. If the player is progressing up the equipment improvements reasonably rapidly and seeing his rating climb to good heights, he is likely to continue playing. New players show up all the time and create a constant influx of low rated, but quickly improving players. A rating system needs to account for these issues and be adjusted with these assumptions in mind.

In the simulations that I perform, all these above factors are taken into account, because each of them is judged to have a significant effect on the success of the rating system. A lot of factors are randomized, such as how long a player is likely to play until he retires permanently from the game or how likely he is to quit until the next day after the first win of the current day. Even the time a player will take between games is randomized. Because of this the simulation has a lot of controllable inputs and many of these inputs can be varied to create somewhat different outcomes.

9 Normalized Ratings and Skills

A player has both his true skill value and the realized skill that occurred in a match. The assumption is that the realized skill is a randomized variation from the true skill value and that randomization has a computable statistical deviation. If you have N players playing, then the increase in deviation of the sum of their realized skills goes up as the square root of N. To account for this, both the rating total for the team and skill total for the team are divided by square root of N before they are used for deciding match outcomes or creating reports on rating accuracy. This is called *normalizing* the team attribute totals and it is assumed in the remainder of this document that any rating and skill totals for teams are *normalized*.

This allows straightforward comparisons between simulations with different numbers of players. In particular, it allows simple numeric comparison with the single player (on a team) model.

10 Formulas

The simulation program uses various mathematical facts and formulas in its implementation. In what follows, we use Java like syntax and declarations to describe these formulas instead of the more traditional mathematical symbols.

10.1 Elo Rating Adjustments

The first of these formulas is the Elo rating adjustment formula. The Elo algorithm is built on the

principle that the lower rated player's chance to win falls exponentially as a negative power of 10 as the difference in the ratings increases. The formula is defined as follows. Let `winnersRating` be the rating of the winner of a match. Let `losersRating` be the rating of the loser. Then define `eloPower`, the power of ten to use in the formula, as follows.

```
float eloPower = (winnersRating – losersRating) / 400;
```

Remember that the function `Math.pow(double a, double b)` takes `a` to the power `b`. The rating factor is the following.

```
float ratingFactor = 1 / (1 + Math.pow(10, eloPower));
```

One of the important aspects of this definition is that if the loser had won the match and the ratings were otherwise left the same, then the `ratingFactor` computed after the flipped result match plus the `ratingFactor` of the current match would add up to 1. This is essentially the identity

```
1/(1 + Math.pow(10, eloPower)) + 1/(1 + Math.pow(10, -eloPower)) = 1
```

It is this identity that underlies the justification for the usage of the Elo approach to determining rating adjustments. The `ratingFactor` can also be seen as the expected loss rate. If a player plays the same identical match again and again with the same ratings and loses a `ratingFactor` percentage of the time, then his rating will essentially stay unchanged over time.

If `kFactor` is the current K-factor that was determined for this match then the rating adjustment is defined as follows.

```
int ratingAdjustment = (int)(ratingFactor * kFactor + 0.5);
```

Adding the 0.5 is the typical programming trick for rounding to the nearest integer. The rating adjustment is added to the winner and taken away from the loser.

10.2 Randomization of Team Skill Totals

The next topic is the randomization functions used in the simulation program. Define the linear randomization function `rndOfRange(int a, int b)` as the function that returns a random integer in the interval range `[a, b]`. Most of the randomized values assigned in the simulation use this function. However, when determining which team wins, *both* teams have a function of the form `rndOfRange(-someConstant, someConstant)` added to their *normalized* skills before they are compared to see who wins. This means that the `rndOfRange(-someConstant,someConstant)` function is subtracted from itself (which is the same as adding to itself), when determining the chance of success or failure. The essential question becomes, what should be the actual chance of winning given the current difference in skills and what should be the value of `someConstant`. This motivates the following.

Define `doubleRnd(int a)` as follows.

```
int doubleRnd(int a)
{
      return rndOfRange(-a, a) + rndOfRange(-a, a);
}
```

Note that each `rndOfRange` function call is evaluated independently and usually returns different

values with no dependency on each other. Define the cumulated distribution function `cdfDoubleRnd(float x, int a)` as the estimated probability that `doubleRnd(a)` is greater or equal to `x`. Then by some simple mathematics which I will leave to the reader as an exercise, a reasonably close estimate when `a` is fairly large (`a > 100`) is the following.

```
float cdfDoubleRndEstimate(float x, int a)
{
     float xnormalized = x / (2 * a);
     return (xnormalized > 0) ? fallOffFunction(xnormalized) :
          1 - falloffFunction(-xnormalized);
}
```

The function `falloffFunction` is defined as follows.

```
float falloffFunction(float x)
{
     if  (x >= 1)
     {
          return 0;
     }
     float linearFalloff = 1 - x;
     float quadraticFalloff = linearFalloff * linearFalloff;
     return quadraticFalloff / 2;
}
```

If we are to build our Elo rating system around a 4000 difference in skill being equivalent to a 400 point change in rating, then the Elo rating system (see above computation for `ratingFactor`) predicts about a 25% (actually closer to 24%) win rate against an opponent 2000 points better in skill and a 10% (actually closer to 9%) win rate against an opponent that is 4000 points better. So the question becomes the following: what value for `a` should we use that causes the function `cdfDoubleRndEstimate(2000,a)` to be close to 25% and `cdfDoubleRndEstimate(4000,a)` to be close to 10%? Some quick computations show that using `a` with a value of `5/6 * 4000 = 3333` gives a reasonable approximation, and that is what is used in the simulation. To summarize, each team's normalized skill total is modified by the function

```
rndOfRange(-3333,3333)
```

and then the modified totals are compared to see who wins. The chance that a team with a normalized skill of 2000 worse than their opponent could win the match would be `cdfDoubleRndEstimate(2000,3333),` which is approximately 24%. The chance that a team with a normalized skill of 4000 worse than their opponent winning the match would be `cdfDoubleRndEstimate(4000,3333),` which is approximately 8%. It is not perfect, but most likely any real world probability distribution for wins and losses would be even further off.

10.3 Deviation from Linear Approximation

Let the Java class Point be defined as follows.

```
class Point
{
	float x;
	float y;
}
```

Let

```
	Point[] setOfPoints = { … Allocation of some point values … }
```

Assume that the `setOfPoints` is supposed to be approximated by a line of slope M. In other words, the points will be approximated by the line `y = M * x + C` where `C` is a yet to be determined coefficient of the line. By the theory of linear regression, the best least squares fit for the coefficient can be computed by the following function.

```
float computeCoefficient(Point[] points, float slope)
{
	float accumulatedDiffs = 0;
	for (Point p : points)
	{
		accumulatedDiffs += p.y – slope * p.x;
	}
	return accumulateDiffs / points.length;
}
```

So the computation for the coefficient C is the following.

```
	float C = computeCoefficient(setOfPoints, M);
```

Once you have an approximating line, you can compute the deviation from the line with the following function.

```
float calculateDeviationFromLine(Point[] points, float slope,
	float coefficient)
{
	float sumOfSquares = 0;
	for (Point p : points)
	{
		float diff = p.y – (slope * p.x + coefficient);
		sumOfSquares += diff * diff;
	}
	return Math.sqrt(sumOfSquares / points.length);
}
```

So the computation of `DEV`, the deviation, is the following.

```
	float DEV = calculateDeviationFromLine(setOfPoints, M, C);
```

In the particular case of ratings and skill, the simulation is built around computations that expect for every delta change in the rating, a similar 10 fold delta change in the skill will occur. This means that

for our simulation, `M = 10` and `setOfPoints` is actually a set of players with the x-coordinates being ratings and the y-coordinates being skill values.

10.4 Alpha

Given `setOfPoints` as in the last section and an expected slope M, the actual slope of the best fitting line might be something a bit different. This slope divided by the expected slope M produces a ratio that we call *alpha*. The term *alpha* appears in the paper http://www.glicko.net/research/chance.pdf, where the authors compute alpha using the deviation of actual winning percentages at various rating differences compared to the expected.

The calculation of the slope of the best least squares fitting line requires some preliminary methods. The first of these is the dot product and it looks like this.

```
float dotProduct(float[] x, float[] y)
{
      float sum = 0;
      for (int i = 0; i < x.length; i++)
      {
            sum += x[i] * y[i];
      }
      return sum;
}
```

The next is a function that computes the average of an array.

```
float computeAverage(float[] x)
{
      float sum = 0;
      for (float v : x)
      {
            sum += v;
      }
      return sum / x.length;
}
```

The next function adds a scalar constant to a clone of an array.

```
float[] cloneAndAddScalar(float[] x, float a)
{
      float[] retVal = new float[x.length];
      for (int i = 0; i < x.length; i++)
      {
            retVal[i] = x[i] + a;
      }
      return retVal;
}
```

The next function returns an array made up of the x-coordinates of an array of `Point` objects.

```
float[] xValues(Point[] points)
{
      float[] x = new float[points.length];
      for (int i = 0; i < points.length; i++)
      {
            x[i] = point[i].x;
      }
      return x;
}
```

The next function returns an array made up of the y-coordinates of an array of `Point` objects.

```
float[] yValues(Point[] points)
{
      float[] y = new float[points.length];
      for (int i = 0; i < points.length; i++)
      {
            y[i] = point[i].y;
      }
      return y;
}
```

Then the definition of the function that computes the slope for the best fitting line for an array of points is the following.

```
float computeBestFittingSlope(Point[] points)
{
      float[] x = xValues(points);
      float[] y = yValues(points);
      float xAvg = computeAverage(x);
      float yAvg = computeAverage(y);
      // Create versions of arrays whose average is zero but any best
      // fitting line will still have the same slope. This eliminates
      // the coefficient as a factor in the computation.
      float[] xBalanced = cloneAndAddScalar(x, -xAvg);
      float[] yBalanced = cloneAndAddScalar(y, -yAvg);
      // Best fitting slope is norm of yBalanced after projection onto the
      // xBalanced vector divided by the norm of the xBalanced vector.
      // By a standard theorem from linear algebra, that is
      // equal to the following.
      return dotProduct(xBalanced, yBalanced) / dotProduct(xBalanced,
            xBalanced);
}
```

That means that *alpha* for our `setOfPoints` is the following.

```
float alpha = computeBestFittingSlope(setOfPoints) / M;
```

In our simulation `M` is 10 and the set of points comes from the rating and skill of all the players in a particular rating range.

11 Measures of Success

One of the issues in the simulation is trying to judge whether the rating system is being effective. One of the primary tests is to see how well the rating difference between teams predicts the skill difference. Generally, if the difference in ratings predicts the skill difference within 2000 skill (after ratings and skill are normalized), then that is considered to be a successfully predicted match. Given the generally wide range of skills and ratings in the simulation, coming that close indicates that the ratings are doing their job. In single player matches, over 97% of the matches between somewhat experienced players tend to be well predicted. In multiplayer matches, it tends to be more like 80%. This is a significant fall off, but it is still much better than if matches were not created using the rating system.

Another test of both the algorithm for setting up matches and the rating system is how many matches were won when the team with lower skill won the match because of luck. The more often that occurs, the more successful the matching process. Oddly enough, the multiplayer simulation does not suffer when compared to the single player simulations. Over 20% of the multiplayer simulated matches are decided by luck beating out skill. The percentage is only a bit higher for single player matches and only when simulations are done with large numbers of players. There are two potential reasons for this. The number of random inputs for multiplayer teams has increased creating a larger variation due to luck compared to the total possible range of player skill levels. The second is that the variations in players tends to be balanced out by random chance as more players are put on each team.

Another judge of success is to fix a particular rating difference (200 is the chosen value for the simulation) and pick out a small subset of matches whose rating difference is within 20 rating points of this fixed difference. In other words, the total rating of the first team minus the total rating of the second team divided by the square root of the number of players should be in the range 180-220. Then a histogram of the normalized skill differences in this small subset of matches can be captured. If the histogram shows a reasonable distribution of values and has a deviation of not more than 2000 skill difference, then again the ratings are working effectively. In the actual simulation, the value is about 1600 deviation for multiplayer and 1000 for single player.

But there is a simpler standard for success. If I am a player that is 400 points in rating higher than another player, then I should feel that this reliably indicates that my actual skill is much higher. The simulation shows that this is very much the case when examining a histogram of ratings versus skills for all the players. Since the ratings typically go up to 4000 in the simulation, players involved in this simulated online game will perceive players at the top to be vastly superior to the average player below them. Using ratings for multiplayer games may not be as accurate and precise as for single player games, but ratings still bring most of the same tremendous value to multiplayer games as they do for single player games.

12 Simulation Code Overview

The simulation code has two Java directory roots for Java packages inside the zip file.

- **rdd/source/RuntimeDifferencing/src** – This is the source for the Runtime Data Differencing library which includes quite a few generically useful utility methods. It is the result of another

project that I have been working on.

- **GroupRatings/src** – This is the source for this simulation. The "main(...)" method can be found in the class GroupRatingsMain in the *groupratings* package.

The best way to build this code is to use Eclipse and point it at the top level directory and let it figure out where the two Java directory roots are located. The packages have no other dependencies outside of the base Oracle distribution of Java, so compiling and execution should be relatively straightforward. You can use startup command line parameters such as -1 or -5 to execute the single player or five man team models as well.

For those reading source code, there are three classes with attributes which control most of the processing.

- **Players** – This class has all the attributes that govern the behavior of the simulation. This includes instances for each alliance player group, a *MatchCreationParameters* instance, and a *MatchProcessParameters* instance. It also has attributes that accumulate various results of tests that are performed during the simulation to determine how well the rating system is working. The class also comes with global control parameters relevant to all parts of the simulation. In theory, two simulations can be run simultaneously, if each simulation has a different instance of a Players class.
- **MatchCreationParameters** – This has all the constants used when queuing up players and putting them into matches. It also controls starting values for the Player objects that are built.
- **MatchProcessParameters** – These parameters control how matches are evaluated and how the attributes of the players are adjusted as a result of the matches. It also has parameters that control the success condition for various tests that are done in the simulation.

The *SimulationProfile* class is a class designed to set some of the parameters in the above classes that were deemed to be the most interesting for testing purposes. Details on the constructor for the class can be found in GroupRatingsMain, where a few of the most common test scenarios have been constructed for convenience.

There are two classes that implement the logic of the rating simulation.

- **MatchCreationUtils** – This class creates new Player objects, manages the waiting players queue, and creates the matches from the currently available players. Players may not be available for two reasons. The first is that they are involved in a match. The second is that they are taking a random duration break from competition. If they have finished their "first win" quota, then they will tend to take a much longer break. There are randomized attributes in the Player object that control just how long a particular player tends to spend taking a break. In general, the queueing logic is fairly complex given the need to create even matches between teams from each side from a pool of active players, which usually have less than a 100 players on each side that are ready to be put into matches. Optimizing the queueing is itself a separate research topic that was only minimally examined in this particular simulation.
- **MatchProcessingUtils** – This class processes the matches and determines how to adjust a player's current intrinsic skill, the player's equipment score, and the player's rating given the outcome of the match. It also calculates many *boolean* tests to see if this match satisfied particular "success" criteria. Examples of such are whether the differences in rating predicted the skill well, whether the lower skill team won because of luck, and so on.

To get a good idea of how this code works, you can turn on tracing in the *TraceUtils* class and watch the tracing output and run critical code in the debugger.

13 Interpreting the Report Data for the Simulation

In the standard simulation, 100,000 rounds are run with each round simulating 30 seconds of time. The simulation presumes that it is only looking at a small portion of the active user base and only simulating a small portion of each day. The slice of "day" lasts 200 rounds or 100 minutes and if a player has gone more than 200 rounds since their last "first win", then the next win is considered to be a "first win" for the next "day".

As stated in a previous section, both team ratings and skill ratings are normalized by dividing them by the expected increase in variance due to the randomizing contribution of all the players on the team. The rating and skill values in the report are the values after this normalization has occurred. As said before, the dividing factor used for the normalization is usually set to be about the square root of the number of players, but it is an easily modifiable value that has been varied for some simulations to see if changing the value improves the rating system.

Every 10,000 rounds, a report is dumped to the standard output, which tries to create a summary of the most relevant information about the simulation. The first set of data is specific to a particular alliance or side. The report data about a particular side is mostly table histograms of various types. The first of these is a rating versus skill histogram table for retired players (if there are any retired players), and then a rating versus skill histogram for active players. Here is a line in the rating versus skill histogram table from an actual simulation.

```
3600 [count=139,coeff=50468,dev=731,alpha=82%] (48000:72[%wins=53,avg games
1834,avg equip=24000], 52000:67[%wins=54,avg games 1746,avg equip=24000])
```

The first number is the starting point of the rating interval and the end point of the range is 400 more than the starting point. So for the above example, the rest of the line describes summary data about linear regression computations and a breakdown of all the different skills of players that have a rating between 3600 and 4000. The first entry in square brackets is a report on a computation of a linear regression (and its deviation from the actual data) about players with ratings in this range. Here is a copy of that portion.

```
[count=139,coeff=50468,dev=731,alpha=82%]
```

The first value is "count" and it is simply the number of players in this range. The next value is a little more complicated. The general assumption by the simulation program is that skill values are generally ten times as large as ratings and much of the logic is built around this assumption. In order to gauge how well this is working for this particular rating range, a linear regression is performed against the ratings versus skills for players in this range, but with the assumption that the slope of the line is 10. In other words, the code computes the best fit line of the form *10 X (DeltaRating) + Coefficient,* where the DeltaRating is the amount the rating is above the start point of the rating range. In this particular case, it is how much a particular rating value is above 3600. The *Coefficient* is the best fit coefficient for the line and gives a general sense of the "floor" of the skills of the players in this rating range. The "coeff" value in the report expression above is this coefficient for this group of players. So for example, if a player has a rating of 3700, then the predicted value for the skill would be

```
10 X (3700 – 3600) + 50468 = 51468.
```

Once you have a linear approximation for a set of points, it is possible to compute the deviation of

those points from the line. This is the *dev* value in the report expression above. So by general probability theory, the chance that a particular player will have a skill that deviates more than 731 from the linear equation's predicted skill is 31%. The chance that a player will have a skill that varies more than twice this is 4.5%. This, of course, presumes that the data follows general bell curve randomness.

The last value is called alpha and is based on a paper called "Rating the Chess Rating System" which describes a general issue with the chess rating system. The problem is that players who are 360 points better than another player are supposed to win 89% of the time, but in reality the win rate is significantly less. The paper calculates a value called "alpha" which is the ratio of the rating difference necessary to produce the actual win rate over the rating difference computed from the ratings currently assigned to the players. This simulation program estimates alpha by doing the following. It computes the slope of the line that would be the best linear regression fit for the map from ratings to skill in a particular rating interval. It takes that value and divides by the expected slope of 10 and turns it into a percent. So the alpha value of 82% given in our particular example says that the estimated skill range size (the amount of "linear spread" from bottom to top in the skills for all the players in this rating interval) for the 400 point rating interval 3600 to 4000 is 82% the size of what was expected (expected value is 4000). Alphas below 100% imply the ratings need to be collapsed closer together to be more accurate and alphas above 100% imply the ratings need to be spread out more from each other in order to be more accurate.

Following the linear regression data for the rating range of 3600 to 4000 is the skill data. The skill data is in comma separated reports for each skill range. If a skill range is not mentioned then no players with a rating in the range of 3600 and 4000 have skill values in that range. For example, take the portion

```
48000:72[%wins=53,avg games 1834,avg equip=24000]
```

The first number is the start point for the skill range. The end point is 4000 higher so the range of skill covered is 48,000-52,000. The number that follows the colon is the number of players in that range. In square brackets following the number of players in the range is computed statistics about the players in this range. The first entry is `%wins=53` which indicates that the average percentage of wins by players in the skill range 48,000-52,000 with a rating between 3600 and 4000 is 53%. The percentage is for the entire lifetime of the player and it is quite possible that many of the wins or losses were tallied when the player had a significantly different rating or skill. The second statistic is the average number of games for all the players in this subset of players. In the example, for players whose rating is between 3600 and 4000 and whose skill is between 48,000 and 52,000, the average number of games that they played is 1834. The last statistic is the average equipment score of the players. In the example, this same set of players has an average equipment score of 24,000, which is the maximum possible equipment score for this particular simulation. If you subtract the equipment score from the skill of the player, you will get the intrinsic skill of the player. So a typical player covered by this particular bracket has an intrinsic skill of about 50000 – 24000 = 26000, where 50,000 is used because it is midway between 48,000 and 52,000.

For the entire line from our example, we can say the following. For players whose rating is between 3600 and 4000, the following facts are true. The total number of players in this range is 139 and the linear equation estimation for the skill of a player in this range is 10 times the amount the rating is over 36,000 plus 50,468. This estimate of skill has a probabilistic deviation of 731. The estimated skill range size is about 82% of 4000, which is smaller than desired. The number of players with a skill between 48,000 and 52,000 is 72 and these players won about 53% of their games, played an average 1834 games, and maxed out their equipment score. The number of players with a skill between 52,000 and 54,000 is 67 and these players won about 54% of their games, played about 1746 games, and all maxed

out their equipment score as well.

At the end of the histogram for ratings versus skill is an overall deviation computed from the deviations of each range weighted by the number of players in each range. The computation follows the usual approach of squaring each deviation, multiplying by the number of players, adding it to a cumulative total, dividing by the total number of players, and then taking the square root. This one number is surprisingly effective at judging the overall health of the rating system and many of the later statistics that are computed can be estimated effectively using this deviation value. If the deviation is less than 1000 (equivalent to 100 rating points), then this is as good as any single player rating system that is out there given the results of the simulations and anecdotal evidence. If the deviation is less than 2000 (this is equivalent to 200 rating points), then the rating system is adequate and will provide real value. If the deviation gets above 4000 (or 400 rating points), then the rating system starts losing much of its utility. For the simulations that are done by this program, ten man team competitions typically have less than 1500 (150 rating points) deviation, which is well within the margin of utility.

The next report for a particular side is the number of games won. Generally, it is better if the two sides have won about the same number of games. However, it is quite possible for these numbers to become quite unequal. This is because a small edge in wins will cause one side to improve their equipment score faster causing the small edge to be turned into a large edge. The code that creates matches tries to work against this by creating matches that favor the side with fewer wins, but that code only ameliorates the problem. It does not solve it completely.

The next histogram table for this particular side is a report that shows how the difference in team ratings of the two competing teams influenced the chances of wins or losses. As was mentioned earlier, the ratings in the report are normalized by dividing by a value that accounts for the increased variance produced by the contributions of multiple ratings into the sum (usually the square root of the number of players). A typical element of this report looks like the following.

```
[-200,-160](win=3459[28%],lose=9027[72%])
```

The numbers in brackets specify the range of normalized rating differences. This particular element covers all matches, where the difference of team one's normalized cumulative rating minus team two's normalized cumulative rating is in the range of -200 to -160. This element says that for teams on side one whose normalized rating difference for a match was in the range -200 to -160 won 28% or their matches and lost 72% of their matches. More specifically, the teams on side one won 3459 matches and lost 9027 matches. This percentage of wins is about what is expected for the Elo rating algorithm. Not all the rating differences are reported in this table, just ones in the range -400 to 400. The *InReportRange* boolean counter indicates the number of matches that were reported in the table and the number that did not. The histogram report concludes with the overall average rating regardless of win or loss, the expected overall average rating, and the statistical deviation for all the ratings from their expected average (the expected average is zero in this case). Generally, if a rating difference of 360 does not guarantee the winning of 80% of the games by the team with the higher rating, then the overall *alpha* for the rating system is low enough to noticeably effect accuracy. The one big perk of this particular histogram is that it can be done without knowing the actual underlying skills of the players. It can be done on any competition that uses ratings. If the actual skills of players involved were not known, then the data values summarized in this report can be used as an alternative way to estimate *alpha* for various rating intervals. But this computation is significantly less accurate than the one that uses the player's true skill values.

The histogram table that follows gives an enhanced report on a slice of data from the prior histogram. In particular, for normalized rating differences that were in the range of 180 to 220, a histogram table is

created that reports actual normalized skill differences versus the percentage of wins or losses. A typical element looks like the following.

```
[2000,2400](win=905[79%],lose=236[21%])
```

The numbers in brackets specify a normalized skill differences range for matches. It is important to note that the only matches that are used to create this element are those where the normalized rating differences were in the range of 180 to 220. This particular element focuses on the much smaller subset of these matches which are in the range 2000 to 2400 of normalized skill differences. As expected, when a team is that far ahead it gets a lot of wins and that is shown in the win and loss percentages and values. Not all skill differences are reported in the table, just the ones in the range -2000 to 6000. The *InReportRange* boolean report specifies the count of differences that were inside this range and showed up in this table and the count outside the range that were not in this table. It also gives the numbers as percentages. The histogram report concludes with the overall average, the expected average (which is 2000) and the deviation from the expected overall average. Generally, the average and expected average should be fairly close and the deviation should not be over 2000. Changing the inflation parameters can change the nature of the data in this histogram. This means that the average reported by this histogram can be used to tune inflation parameters. If the average is too high, then more inflation is needed. If the average is too low, then less inflation is needed.

The next report is global to both sides and focuses on the results of boolean tests that were made during the match creation and resolution process. Each named test has a report that specifies how many times the test was passed and how many times it failed and the same numbers as percentages. If some matches had values that made the test invalid, then a third counter tracks the number of "Not Applicable" or "N/A" results.

Here is a typical "counter report".

```
ExpectedProductivity(yes=130066[57%],no=99161[43%])
```

The test performed was "ExpectedProductivity", which is a test to see if the productivity computations showed that the losing team was performing up to the expectations of its ratings. The number of matches with expected productivity was 130,066 and that was 57% of the matches. The number of matches where the losing team did not live up to expectations was 99,161 and that was 43% of the matches. About half of matches that have reasonably close ratings should live up to expectations,

For some of the counter report booleans, there is a third value called "N/A" which is incremented when the conditions required to do the test were not satisfied. Here is the full list of the current boolean counters that are tracked.

- **RandomBeatSkill** – Reports a "yes" result, if the team skill totals indicated the result of the match should go one way, but the match went the other way because of luck. A percentage over 20% is very good and is the primary criteria for success of a queuing and rating system. Since the team with the lower skill rating total has less than a 50% percent chance of winning, a 20% value for "yes" actually indicates that at least 40% of the matches were close enough that luck was the primary deciding factor. Getting such a good result is harder, if there is a large range of skills, which is why the simulation artificially increases the range of skills in order to truly test the rating solution.
- **LongQueueWaitSide1** – This test monitors the effectiveness of queueing for side one. The simulation has requirements for a minimum number of players of particular types being on each team. Also, the number of players simulated is kept as small as possible to minimize the

execution time. Because of both of these issues, some players will be waiting to get into a match, but have to wait for a number of rounds before they are selected to be in a match. A "yes" value is reported, if a player waits for 8 or more rounds (or four minutes or more) before getting into a match. A high percentage here means that a lot of players are waiting quite a long time to play a match. If this were a live system, this would be a cause of frustration and impatience among the player base. The simulation can be manipulated (by removing player type requirements or increasing the size of the player base) to eliminate this delay, but that would reflect the current online games that are out there.

- **LongQueueWaitSide2** – The same as LongQueueWaitSide1, but for side two. If the values for side one and side two are far apart, then that shows a severe queueing issue potentially caused by a logical glitch in the queueing algorithm. It can also be a sign that one side is winning far more games than the other causing that side to participate less often since a "first" win increases the duration of time the player will wait until entering the queue again.
- **ExpectedProductivity** – The test looks at the productivity statistics for the losing players. If the productivity statistics of the losing team is greater or equal to the expected statistics, then the test is passed. Otherwise, it will fail. The productivity calculation is some what ad hoc and subject to change and has a large randomization factor to emphasize its unreliability. Because of this, the current threshold for expected productivity is somewhat speculative and may also be adjusted in the future. Nonetheless, the simulation does give the result of this test serious weight. If a match shows under-performance by the criteria of this test, then the rating adjustment to both winner and loser is multiplied by two. If the normalized productivity falls short by over a 1000 skill per player from the expected productivity, then the rating adjustment is multiplied by another factor of two. Experimentation shows that even this somewhat unreliable mechanism is so powerful that without it, the rating solution would be significantly less accurate.
- **ExpertMatch** – Passing this test says that the minimum rating of all players on both teams for a match was at least 1600. A player with a rating of at least 1600 is called an "expert". Usually a minimally competent player, who has played at least 400 games, should be an expert because of equipment score increases. Generally, the high rate of failure of this test says more about how matches are created than about the general strength of the field. If the number of players on each side is dramatically increased for the simulation, the percentage of success for this test increases without increasing the percentage of expert players.
- **GoodSkillMatch** – This test is passed if the team with the lower skill total has at least a 25% chance of beating the team with the higher skill due to randomization factors (in other words "luck"). Getting a result over 65% for this boolean counter is very good and is one of the primary tests used to judge the effectiveness of the queueing and rating system.
- **ProperlyCreatedMatch** – This is a test done at the time matches are created. The match creation algorithm attempts to create opposing teams that have reasonably close rating totals. When the algorithm knows that it has succeeded, then this test is passed. In some cases, the match creation algorithm will determine that the number of remaining players in the queue is too small to allow a ratings based team assignment. In that case, teams will be created on a first come first served basis and such matches will be deemed to have failed this test. However, it is possible that such matches may still be reasonably close matches by pure chance. This test is an evaluation of the queueing algorithm and says nothing about the predictive value of the actual ratings.

- **GoodRatedMatch** – The actual differences in rating totals of the team predict that the team with the lower rating has at least a 25% chance of winning. The percentage of tests that pass should be reasonably close to the percentage that pass the ProperlyCreatedMatch test, but variations are definitely possible because randomly created matches may still pass this test. Like ProperlyCreatedMatch, this boolean test is a judgement of the queueing solution and says nothing about the accuracy of the ratings.
- **RatingsPredictedSkillWell** – The test is passed if the differences in normalized rating totals for the two teams is multiplied by 10 and compared against the differences in normalized skill totals for the teams and the absolute difference is less than 2000. Matches that are predicted to be blowouts by both ratings and skill are considered invalid for this test. More specifically, if the normalized team rating difference is below -400 or above 400 and the skill difference is less than -4000 for a rating below -400 or greater than 4000 for rating a difference greater than 400, then the match is considered invalid for the test and the "N/A" counter is incremented instead. Generally, if a match passes this test, then the ratings will do a reasonable job of predicting the outcome of the match. For example, if the ratings predict a completely even match, then neither team had a better than 75% chance of actually winning.

  This statistic is examined more than any other because it gives a simple reliable numerical judgement about the accuracy of the rating solution. A minimum passing rate of 50% seems to be required in order for the other tests to even have a chance of giving good results. If the passing percentage goes below 50%, then the utility of the rating solution is very much in question. The goal is 70%, which indicates a highly useful system. It is this test that is used to validate various aspects of the simulation. For example, if skill productivity calculations are not used when doing rating adjustments, then it can take a lot of rounds before this value goes significantly above 70% and only if the rate of player turnover is reduced or eliminated. Likewise, if inflation adjustments are not used effectively, then this value may never go much above 70%. It is this statistic where single player team games give much better results. Generally, it is unusual for the success rate of this test to be below 92% for single player teams and it is not unusual for it to reach 98%, if the roster of players is fairly stable. Even if skill productivity computations and inflation are turned off, the single player team result is still above 80% making it a quite usable rating system. But for multiplayer team matches, it is important to effectively use all the information that is available in order to insure a good rating system.
- **ExpertRatingsPredictedSkillWell** – If a match had a minimum rating above 1600 and it passed the RatingsPredictedSkillWell test, then it passes this test. If the minimum rating is below 1600, then the result is marked as "N/A" or not applicable. Since all new players start with the same rating, they start with a rating that is known to be highly unreliable. Generally, it takes 100 games in multiplayer team games for a rating to converge to a reasonable close approximate of its true value (in single player team games, the convergence is much faster). This accounts for a lot of the failures for the RatingsPredictedSkillWell test. To eliminate "new player" noise from the RatingsPredictedSkillWell test, a variation of this test was created that focuses only on matches where the minimum rating was above 1600 (or expert level). This test shows the true core rate of prediction success for the rating system.
- **RatingsPredictedSkillNotBadly** – This is the same as the RatingsPredictedSkillWell test, but the threshold difference is 4000. Generally, the success rate for this should always be over 90%. If it is less than that, then there are some serious issues.

At the very end of the test, there is also a report on the computation time spent in various activities.

14 Queueing for Matches in the Simulation

For each side, there are two pools of players that are currently not involved in matches. The first pool is called the “waiting players” queue and it contains the players actively waiting to be assigned to a match. The second pool are players who are taking a break from the game and will not be put back into the main “waiting players” queue until the duration of their break has ended. In some cases, the break can be quite long, if the the player will not get “first win” bonuses for any matches that the player might play and win. An unfinished “first win” bonus is considered to be an incentive to come back more quickly to the “waiting player” queue and the code in the simulation reflects this.

The two waiting players queue are compared, and if one queue is much larger, then it is important to take action to fix this. If the queue lengths get particularly severe in their inequality, then players at the back of the longer queue will start waiting a long time and are likely to fail the “LongQueueWaitSide<N>” test. A long queue is actually a good thing. It is the short queue that is the problem. The fix for the shorter queue is to stretch the “first win” equipment bonus over the first four wins instead of delivering it all in the first win for the players on the side with the shorter queue. The normal “first win” bonus is an 8 times multiplier on the usual equipment score upgrade. When stretched over four matches, the bonus is only a two times multiplier on the usual equipment score. Once a player has won enough games to get all his “first win” bonuses, he will have to wait until he is at the round that is 200 rounds later than the round where he got his first “first win” bonus. The idea is that players tend to play until they have won all their bonuses for the day and the code in the simulation mimics this.

The specific implementation details are as follows. If there are no more bonuses to be won on the next win, then the player will multiply by four his normal waiting period which is then further multiplied by a factor particular to that player. Each player is given a fixed random attribute that varies the amount of time he spends outside of the “waiting players” queue after he has finished a match and there are no more bonuses to be earned with the next win. This attribute is called the “casualness level” and it is the additional multiplier on the “length of break” time. Players with a low “casualness level” will tend to come back from their break before 200 rounds (the simulated “slice of a day duration”) have passed since their last bonus win and are considered the “intense” players of the group.

The match creation process uses the following algorithm. The player's rating is divided by 200 to determine his “rating level index” and collated with other players of the same index. When choosing the next players for a match from both sides, the indices on each side are added to the cumulative totals of the indices computed for both sides and the matching algorithm tries to keep the differences in the sums of the two sides reasonably close. The algorithm has two modes. It can match the strongest players against each other first and work down the rating indices or it can match the weakest players against each other first and work up the rating indices. What may not be immediately apparent is that this approach will create some matches that unfairly favor one side or the other, especially if the queues are unequal. Generally, matching from top down favors the longer queue and matching from bottom up favors the shorter queue. The reason for this is essentially the same for either matching direction so we focus on the top down direction. Typically, the longer queue will have more highly rated players in its queue than the shorter queue. When you go top down on the queue all the highly rated players of the shorter queue will be used up, but there will still be strong players left in the longer queue. Since the matching is going top down, there is a bias to pick higher rated players from both queues, which means higher rated players from the longer queue will start showing up with weaker players at the tail end of

the shorter queue. This creates matches that will tend to be won by the side with the longer queue.

This bias could be fixed by paying closer attention to the rating indices of the longer queue or switching to bottom up from top down in the middle of the matching process. Instead this bias is actively used to try to fix a different issue. As a general rule, once one side starts winning more games than the other side that uneven win rate will worsen, because the side that wins more games will get better equipment. Over 100,000 rounds a huge discrepancy in the number of wins can appear. For example, if this issue were not corrected, it would not be unusual for one side to be winning 70% of the games at the end.

To fix this, the queue lengths are compared and the choice of matching direction is chosen to favor the side that currently has fewer wins. Another issue that arises from this is that some players in the longer queue can suffer queue starvation, because they do not have the correct rating level index (either too low or too high depending on the direction chosen for matching) to be chosen into the matches. To minimize this, only so many players from the longer queue are considered for matching with the shorter queue and those players are always taken from the front of the longer queue. Also, towards the end of the matching process for a round when there are no longer enough players to create at least two teams, the matching process reverts to first come is first served which tends to flush out the players that have been waiting a long time.

15 Skill Productivity

In multiplayer matches that I have participated in, there have always been statistics given to the players about how each player is doing in the match. By examining those statistics, it was always quite clear which players were dominating the game. When creating this simulation, it became clear that the rating solution was not going to provide quick convergence of a player's rating to his true ability unless this information was taken into account. The other typical solution to this problem is to increase the Elo K-factor, which works reasonably well in single player team games. But in multiplayer games, if you increase the Elo K-factor to a point where the player's rating is converging sufficiently fast, you introduce random walk inaccuracies that are so severe that the rating system starts to lose its useful predictive powers. For this reason, this simulation is very much dependent on the usage of productivity computations. But this has its own issues.

One of big issues is that if the game has complex goals, then a player may be torn between a choice to boost his productivity statistics or actions that increase the overall chance of team success. For example, if a resource needs to be guarded and there are many such resources, defenders may not interact with opponents during much of the match, if the main conflict centers on other resources. Players that volunteer to be a defender of resources generally sacrifice their productivity statistics to help the team goals. Also, some player types or even player playing styles tend to produce more or less productivity statistics, and this needs to be taken into account as well. In the statistic computations there are ways to compensate. One solution is to increase productivity statistics gained when close to a resource or in general come up with a way to detect “defender behavior” and reward it.

There is an interesting fact about this. This issue tends to apply to the winning team much more than the losing team. The losing team knows it has an insufficient number of resources to win at the current moment. Because of this, defense loses much of its value, since capturing new resources is the only action that has a potential for gaining a win. In other words, losing players do not have as valid an excuse for why they did not show their true productivity. For this reason, the productivity statistics are only computed for players on the losing team. There are other reasons for doing this. Winning teams tend to boost all the statistics of its players and the differences in ability are less apparent. On a losing

team, it is not unusual to have only one or two players who have anything in the way of productivity at all. The better players stick out much more clearly on losing teams.

There is one last good reason for computing productivity for losing teams only. One of the problems in multiplayer games is the presence of paid players or even computer controlled players ("bots"). When such players are on a losing team, they usually give up and move their focus to another match that is being played on a different user account (usually simultaneously with other accounts). These players tend to give up even if there is still a real chance for success by the losing team. These players also tend to play poorly anyway and usually bank on the fact that random selection will put them on a team so strong that it will win no matter what they do. But using productivity statistics can make it impossible for these players to thrive.

It works as follows. When a team wins, the players share equally in the rating boost. But when a team loses, the members share unequally in the loss. The players that showed productivity that reflected their rating or better get less of a share of the rating deduction. Players whose productivity fell below the current expectations for their current rating will get a much larger share of the rating deduction. This makes it quite difficult for players that play poorly in losing matches to get good ratings. Since equipment costs more as you reach certain equipment levels, higher ratings are required in order to get the better equipment. Paid players and bots will not get these higher ratings and will not be able to "mine" the game for valuable equipment and currency that they can sell to others. The other advantage is that the members of the losing team will have an incentive to put up a good fight making the game more enjoyable for the winning team.

This turns out to be so effective that it actually helps to exaggerate this effect. This is done as follows. If the losing team's productivity does not live up to the productivity estimates given by the rating differences between the teams, the rating adjustment is multiplied by two. If the losing team falls short by more than 1000 normalized skill per player, then the current ratings are considered to be badly indicative of current skill and the rating adjustment is multiplied by four. The rating adjustment is handed out evenly to the winners, but it is handed out quite unevenly to the losers as explained before. The few players on the losing team that showed any productivity will be spared much of the severe loss in their ratings. See the description of the *ExpectedProductivity* test for more on this.

Using productivity in rating calculations turns out to be so successful that it becomes necessary to deliberately sabotage it to reflect a more real world scenario. Two things have been done to reduce the utility of productivity. The first is to make half of a player's productivity completely random. So if a player could potentially produce 100, then he will actually produce a random number in the range of 50-100. The other is to assign a random attribute value to each player which is a player style reduction coefficient for productivity. It is called the stat correlation percentage.This random value is in the range of 50% to 100% and once a value is chosen for a player it does not change for that particular player. For example, a player with a 50% stat correlation percentage who would ordinarily show a productivity of 100, would have that reduced to 50 because of the stat correlation percentage and it might go down to 25 because of the deliberately introduced randomness. Another player that has a stat correlation of 100% and got the full 100 value in the random range would show 100 productivity, which is four times as much as the first player, but they were both equally skillful. Players with low stat correlation percentages will tend to have lower ratings. It is unfair, but there is not much that can be done about it. One of the continuing long term research projects for any multiplayer game that adopted this strategy for ratings is to determine all the factors in game play that contribute to winning the game and correctly weighting them for the productivity computation. As a small note, the skill productivity of a player in the simulation is doubled after it is calculated in order to compensate for the average expected degradation from randomness and stat correlation.

Even in the very simple model in the simulation, the productivity calculation has its complications. For example, how much should a really good player's productivity be reduced, if the good player is surrounded by poor players. In particular, how much better does the good team have to be than the bad team for the good player on the bad team to have his or her productivity significantly diminished. The calculations in the simulation assume a linear degradation based on the differences in normalized skill of the two teams, but this is just a guess. Arguments can be made that the calculation be done in other ways. This is another reason why this simulation randomizes productivity output so much. It is to reflect the doubt in the current algorithm for its calculation.

Even with this randomization, the effect of using productivity on rating calculations is still large and is the significant factor in the success of the multiplayer rating system.

16 Equipment Score

The simulation uses three levels of equipment score. A player with an equipment score of less than 8000, can improve his equipment at normal cost. A player that is between 8000 and 16,000 equipment score needs to pay eight times as much as the base cost. A player that is between 16,000 and the maximum of 24,000 needs to pay eight times more again, or 64 times the base cost for equipment improvements.

When two teams play, the minimum rating of the two teams is computed. As mentioned in an earlier section, for every two hundred points this rating is above 800, the equipment score bonus for a win is multiplied by two. For example, a match with a minimum team rating of 2436 has an exponent of (2436 – 800)/200 = 8, so the multiplier on the equipment score given out is 2 to the eighth or 256.

In the standard simulation, the base increment for equipment score improvement after a win is 10 and it is multiplied by eight for a “first win”, unless the player needs four “first win”s to use up his bonus in which case the first four wins have the equipment score improvement multiplied by two. Since a typical player plays for about four wins per each 200 round period (200 rounds is how long it takes before the next “first win” interval starts), this averages to about 20-30 points per win. For simplicity, I will use the figure of 20 points per win. I should note that these parameters are considered configuration values and many different values have been tried, but a base increment of 10 seems to be about right if you want an elite player (skill in the top 5% of players) to be able to achieve maximum equipment score in about 500 games.

Using the the figure of 20 equipment score improvement per win, a player has to win 400 + 400 * 8 + 400 * 64 = 29,200 games before he would maximize his equipment score, assuming his rating never gets above 1000. At 10-30 minutes a game, that is not a reasonably achievable goal for even very dedicated players.

However, if you played games with a minimum rating above 2400, then you have to win about 228 games at that level to max out your equipment score. Generally, elite players in the simulation reach ratings of 2800-3600, so this is quite achievable. There is one small note to made here. If the queueing algorithm for creating matches creates matches from the top down, then elite players will tend to get into matches with high minimum ratings. But instead if the matches are created from the bottom down, the elite players will tend to find themselves grouped with weaker players. The choice of how to create matches can have a significant impact on the equipment improvements for the entire group of players.

17 Player Retirement

The single largest disturbance to a rating system is when a highly rated player who has played thousands of games and whose skill has not substantially changed for many of the most recent games, is replaced by a new player of indeterminate skill, whose skill is likely to be changing quickly in the near future.

There are other disturbances as well. A player could take a tutorial and get a big jump in skill. A player could quit for a long time and come back and play at a substantially different skill level. Players can deliberately sabotage their rating for various reasons (in chess, a player can enter at a lower section in a chess tournament and have a greater chance to win the prizes). Players might experiment with different playing styles that may temporarily change their skill levels dramatically. In the Battle Bot example, a player may master one type of role and then decide to master a different type of role. As a specific example, a player that is good at being a “tank” might decide to try becoming a good “flag carrier”.

The “new player” disturbance is so dramatically larger that when it comes to simulating disturbances, I decided to implement only this one disturbance. In its current implementation, the simulation tries to bring in a number of new players equal to the starting number of players during the full run of 100,000 rounds. To keep things simpler, new players are brought in only when a current player “retires” or quits permanently from the game. The base number of games until retirement for a player is a random number between 100 and 1500. This range is subject to change in the future and has been altered constantly during various simulations. The retirement period is extended, if the player does well. This simulates a player whose enthusiasm increases, if they feel they are succeeding. The player's current rating times ten plus his current equipment score is added together. For every 32,000 this total increases, his retirement game count doubles. So for example, a player whose base number of retirement period is 500 games, will not quit until playing 1000 games, if the player's rating is 2000 and he has an equipment score of at least 12,000.

When adding new players to the simulation after the simulation has started and players are retiring, the new players are given a larger range of initial skill by granting them a random percentage of the maximum amount of training that they can have. The potential range of percentage grows as the number of rounds passed so the new players have larger and larger intrinsic skill variations as the simulation runs. This means that the disturbance to the simulation gets larger and that can be seen in the reports made after every 10,000 rounds. Usually the best numbers are achieved at about round 20,000 and then tend to degrade. If retirement is turned off, then the numbers continue to improve to the point that they no longer represent potential reality.

There is an interesting note to be made here. The disturbances of the new players are not felt equally throughout the rating ranges. Players at the top of the rating ranges tend to be more stable and players at the lower ratings tend to have their ratings move so fast that they achieve their true value fairly quickly. The players with the largest persistent disturbance are the players at about 1600 to 2400 rating points. The new players usually quickly move up to this level and then stay at this level for a period of time. In their upward movement, they push down the ratings of the players that they beat as they push themselves quickly up the ratings. This has the effect of pushing down the rating range of the players from 0 to 1600 creating a significant skill gap between players at 1600 and players at 2400. This can cause alpha to rise significantly over 100%. Strangely enough, the gap is usually larger in one player per team simulations, because new players with skill significantly above their current rating are more guaranteed to win their matches and move up the ratings more decisively and more quickly. In

multiplayer simulations, the deviation of skill from ratings is higher for longer at the lower rating levels than it is in single player simulations. The single player simulation essentially migrates that disturbance quickly upwards to the higher ratings and creates a larger skill gap.

The general problem is the "hollowing of the middle". The middle skill range tends to be empty of players, because either the players retire because they do not get the multipliers that would keep them playing, or because they improve out of the middle skill range, because of equipment improvements. In order to sustain an accurate rating system for a particular skill range, you need a comparatively sizable body of players for the weaker and stronger players to compete against. The fact that rating increases contributes to faster equipment improvements, which then feedbacks into rating improvements creates middle skill ranges that have fewer players than would be typical in games that do not use ratings as a feedback into equipment improvements.

One consequence of this is that players that reach a 2000 rating may feel trapped there. Whenever they play players in a higher rating bracket, they get stomped on because there are just not that many players around that are slightly more skilled. It may take some time before their equipment improves enough for them to feel like their rating is advancing up to its true value.

18 Understanding Alpha

In an earlier section, I described an attribute called "alpha" that was used to describe the ratio of the calculated slope of the linear regression fit for a 400 point rating interval to its expected slope. This value has been the focus of a lot of experimentation in an effort to determine what types of conditions change this value. As a reminder, please see http://www.glicko.net/research/chance.pdf for an introduction to the concept of alpha. In intuitive terms, alpha is an estimated computation of the true size of the skill range for a 400 rating interval divided by the desired skill range size of 4000. Using the slope for the linear regression as a proxy for this gives a reasonable estimate of the underlying spread in skills that is not due to simple randomness coming from the general inaccuracy in the ratings. As a general note, the alpha computation is somewhat suspect for the lowest and highest rating ranges, because those ranges tend to not be well populated and the skill min or max value is likely reached in those rating ranges, causing the rating variances to be significantly larger than the possible skill variances (the skill values being capped at one end). This usually creates artificially lower alphas that are essentially meaningless.

The goal is to try to bring alpha as close to 100% as possible and make the deviation for ratings accuracy as small as possible. These two goals are not always compatible. By varying the parameters for the simulations, it was possible to create good alpha values, but get unacceptable deviations, or do the reverse and get good deviation values, but get alpha values that varied significantly from 100%. The two primary parameters that are manipulated are the K-factor, and speed and length of rating inflation. Increasing the K-factor significantly above 16 has a fairly simple effect. It decreases alpha, which can be useful if alpha is too large. This can happen if skill values are changing rapidly, especially if they are changing due to "rating feedback". However, increasing the K-factor increases the overall deviation value, so it is not clear that this is that useful a solution. Decreasing the K-factor below 16 increases alpha somewhat. In most of the simulations, the higher rating brackets have an alpha significantly below 100% so using a lower K-factor for matches against players with higher ratings can have a positive effect. This is not always the case and varying various parameters can cause unpredictable results in the alpha values. The big issue with a low K-factor is that it can also cause increased deviation because the ratings fail to keep up with changes in skill. However, if the players in

the higher rating brackets tend to have stable skill then reducing the K-factor to 8 (or lower!) can help to alleviate problems with the alpha being too small for such rating brackets.

Changing the inflation rate by either changing the number of wins duration or the amount of inflation also effects alpha values, but not always in ways that are predictable. Generally, inflation causes an overall reduction in alpha (good for alpha over 100%), but sometimes in some rating intervals it does the reverse. Other parameters also have an unpredictable effect.

There is one interesting fact. The single player games have a much larger range of alpha variations than the multiplayer games, but nonetheless still have lower deviation values compared to the deviation values of multiplayer games. When performing these experiments on altering alpha, I concentrated mainly on the single player case.

One test that was done was to turn off the equipment improvement. When that was done, then alpha dropped significantly. This is most likely because the lower rated players are no longer having their skill values constantly increase at a faster and faster pace as the ratings go up.

Another scenario I created was designed to mimic the chess game scenario. I started ratings at 2200 and eliminated equipment score as a factor. When I eliminated the adjustments for “productivity calculations” as well, it dramatically reduced the alpha values in the middle rating ranges, but the alpha values towards the end were larger and were surprisingly consistent with the alpha pattern computed by analyzing historical games and ratings for chess players. However, if the “productivity calculations” were put back in then a different result occurred. In that case, the middle rating ranges had slightly higher alpha values, the reverse from the simulations that did not use “productivity calculations”.

Here is some data for a single player game that did not use productivity calculations to multiply the rating adjustments (as a side note, the intervals 1600-2000 and 4000-4400 have so few players that their alpha values should be ignored).

```
1600 [count=1,coeff=7611,dev=0,alpha=0%] ...
2000 [count=29,coeff=12625,dev=865,alpha=112%] ...
2400 [count=56,coeff=17232,dev=635,alpha=90%] ...
2800 [count=43,coeff=20615,dev=639,alpha=73%] ...
3200 [count=46,coeff=23326,dev=615,alpha=58%] ...
3600 [count=22,coeff=26231,dev=415,alpha=89%] ...
4000 [count=3,coeff=29869,dev=605,alpha=6%] ...
```

Here is the data for a single player game that did use productivity calculations to alter rating adjustments.

```
1600 [count=18,coeff=11140,dev=464,alpha=69%] ...
2000 [count=34,coeff=14456,dev=425,alpha=75%] ...
2400 [count=39,coeff=17514,dev=764,alpha=93%] ...
2800 [count=29,coeff=20526,dev=493,alpha=72%] ...
3200 [count=41,coeff=22818,dev=593,alpha=59%] ...
3600 [count=32,coeff=25522,dev=575,alpha=69%] ...
4000 [count=7,coeff=28459,dev=418,alpha=77%] ...
```

As a reminder, if the losing player does not live up to expectations based on their productivity calculations the K-factor is doubled (and sometimes quadrupled if the losing player fell far short of expectations). Increasing the K-factor does reduce alpha and rating intervals with more players tending to have more matches with players that are close in rating. It is for those matches that the productivity calculations have their most dramatic effect. This may explain why the “productivity calculations” seem to reduce the alpha somewhat for well populated rating intervals.

In general, alpha appears to be an unpredictable artifact of the rating system. There is a big open issue about exactly what causes particular values of alpha for various rating ranges given various configurations for the matches. There is one possible bit of enlightenment I can try to provide. If you are a player in the middle of a well populated rating range, you will tend to play an equal number of opponents that are better than you and an equal number who are worse, and the general rating differences will be the same for higher or lower ratings. This means that if the player's rating accurately reflects his “relative ranking” based on comparison to other players and their ratings, his rating won't change much from its current value even if this player loses more than 20% of his matches against players 350 points below him and wins more than 20% of his matches against players 350 points above him. This applies equally to all other players in the same rating interval, which means once a particular value of alpha begins to take hold, it won't be fixed by the rating system. Since the rating system puts little pressure on ratings to change the slope of the line relating rating to skill, alpha is free to float to values that accommodate other pressures that are put onto the ratings. One of the biggest of these is the random walk pressure built into the rating system, which pushes ratings apart as the cumulative rating differences are applied. The higher the K-factor, the bigger this pressure to push ratings apart which is what appears to happen if the K-factor is increased. However, the other factors seem to have less of a clear result. For example, the speed with which a player's skill changes effects the alpha values significantly. Faster skill changes in low populated rating intervals does appear to increase alpha significantly, but why it does is not completely clear.

19 Summary

This project was started because I was convinced of two things. The first is that the current multiplayer games (Random Battleground in World of Warcraft being my only example at this point) were being ruined because there was no credible system for creating matches that eliminated from the matches two classes of players that I thought wrecked the gaming experience. The first class of players are players who do not make a legitimate attempt to win the game. These could just be bad players, but many times they are “bots” or paid players, and they are there just to grind through the equipment improvements. The second class of players are the well equipped elites. Their presence on an opponent's team is devastating and if you have them on your side they make you feel irrelevant. Their influence on the outcome is so strong that they actually help perpetuate the first class of players by allowing players to win matches a reasonable amount of the time just by sitting around and doing nothing. And if you were losing a match because of these elite players, usually the most productive use of your time would be to run away and hide and then do other things besides playing the game (like browsing some websites) until the game is done.

My second conviction is that a multiplayer game with randomly assembled teams could be accurately rated. In the multiplayer games I played in, I saw productivity statistics that clearly outlined which players were performing much better than others. The typical problem one might have in a multiplayer rating system is the low correlation between your current skill and the match outcome. But if you use

the productivity statistics, then you can determine who actually causes the loss and adjust his rating appropriately. At that point it seemed obvious to me that an accurate rating system could be created.

To prove this, I wrote a program that simulated players and matches in a simulated online multiplayer team gaming universe and put into the simulation all the relevant complications that occurred to me. During this effort, I searched for others who might have done the same and I was surprised to find almost nothing. What was even more of a surprise was that nobody seemed to have even simulated single player matches equivalent to chess and then try out various rating solutions to judge the adequacy of the simulation and the rating system. There seemed to be a big hole here.

Now, it is quite possible that there others who have done something like what I have done and my hope is that somebody who reads this will alert me to these other efforts, but for now I will assume that some of the deductions I make are new and have interest to others. Some of these deductions apply to single player (on a team) games as much to any other games. My conclusions are possible because I have access not just to game outcomes and the ratings of the players, but to the actual underlying skill of the player as well.

One example of this is the calculation of alpha. Since the actual skill values were available it was possible to provide answers to some questions about alpha that otherwise could not be answered. For example, http://www.glicko.net/research/chance.pdf speculates that the lower win rates were caused by the errors in the ratings themselves implying that alpha was not properly estimated. Using the simulations, it is quite clear that the skill compression (alpha less than 100%) in a rating interval is real and not an artifact of rating error. Other hypothesis can also be eliminated. For example, it was hypothesized that large numbers of matches between players with relatively stable ratings would not have this effect. In simulations that I ran where player retirement was turned off and players were given fixed unchanging ratings, most of the rating intervals had alpha below 70% even after thousands of games. In fact, the alpha tended to get lower the more games the players played.

Because of this project, I have become convinced of two things. The rating system for multiplayer games can be made to work and would be highly useful. The second is that rating systems seem to have a general problem with large variations in alpha and understanding the cause and effect is a worthy goal for the future.

20 Final Thoughts

This project started out as an attempt to prove that multiplayer rating systems could work. I feel that the proof of that is fairly conclusive. But on the way, I discovered what others have discovered, rating systems do a good job of determining relative ranking, but seem to do a poor job of determining absolute ranking. The variance in alpha through the rating levels was a bit of a surprise and there appears to be some potential for new research exploring how to get better alpha values.

This document did not address one issue that occurs in other games. If a competition between two teams is clearly going to have a highly unequal outcome, because of a large rating disparity, it is possible to handicap one side (or “buff” the other) so that the team with the lower rating has a reasonable chance of winning. It is certainly possible to simulate that, but it is a distraction from the main focus of this document, which is to show that randomly grouped multiplayer competitions can work and analyze various artifacts in the rating system itself. It does make sense that in a real world implementation, there should be some type of handicapping (or “buffing”), which would then be accounted for as an adjustment in the rating of the team being altered.

This project also had an ulterior goal. Besides competitions of teams of players vs. players (PVP), there are also tests of ability of a team against a fixed set of computer controlled enemies or hazards. This is called player vs. environment (PVE) and it suffers from many of the same issues as PVP, maybe even more so. The rating system espoused in this document can also be used to give ratings to players for how well they survive PVE tests. As the total ratings of the players go up, they would be allowed to tackle tougher PVE tests. A PVE test would not have to be a simple fail or not fail test, but have various goals that could be accomplished and would increase the rating boost for finishing those goals. For example, a timer could be put on the test and the time remaining on the timer could be used to give additional rating points. Generally, players would “invest” a certain number of rating points at the start of the test and lose them if they did not perform at least somewhat adequately on the test.

There are many interesting things you can do in PVE tests that you cannot do in normal PVP. You can tell a story, and as players go up higher in ratings and get to the tougher PVE, the story can advance as well. The PVE tests can be used to pose clever puzzles to amuse or frustrate. Also, PVE scenarios can be used to specifically train players in certain abilities that the player has in the game so that they learn how to more effectively play the game. This can be done in some cases to train players to become better in PVP battles.

That is all that I have for now on this topic. There are more things that can be said, but I will leave those for a later time. For those who have read this document to the end, I hope you come away with the same conviction that I have which is that there is an exciting opportunity in rating randomly grouped multiplayer games.