Adan Rivas & Jonah Sternthal

ENGS 65: Engineering Software Design

16W

March 14, 2016

Monte Carlo Simulation of 2015-16 NBA Regular Season

**Introduction**

Every fall the National Basketball Association (NBA) restarts its regular season, which culminates into a four round playoff series to determine the league’s seasonal champion. What we are implementing is a low level monte carlo simulation to simulate a full regular season in the NBA. Our program aims to determine the playoff contenders from the NBA’s two conferences: eastern and western conference.

*More about the NBA*

The NBA is the marquee men’s professional basketball league in world. It is affiliated with 30 member franchised clubs (29 in the US and 1 in Canada) divided into two conferences comprised of three divisions. And each division consists of five teams. During the regular season each team plays 82 games, 41 home and 41 away. Every team faces opponents in its own division four times a year (16 games). That same team plays six of the ten teams from the other two divisions in its conference four times (24 games), and the remaining four teams three times (12 games). Finally, each team plays all the teams in the other conference twice apiece (30 games). Thus, each team hosts and visits every other team in the league at least once every season.

The NBA Playoffs begin in late April, with eight teams in each conference competing for the Championship. The three division winners, along with the team with the next best record from the conference are given the top four seeds. The next four teams in terms of winning records are given the lower four seeds.

The playoffs follow a tournament format. Each team plays an opponent in a best-of-seven series, with the first team to win four games advancing into the next round, while the other team is eliminated from the playoffs. In the next round, the successful team plays against another advancing team of the same conference. All but one team in each conference are eliminated from the playoffs.

*About the game and traditional positions*

Basketball is a sport played by two teams of five players on the court. The objective is simply to outscore your opponent. NBA games are played in four quarters of 12 minutes. Overtime periods are five minutes in length. The time allotted is actual playing time; the clock is stopped while the play is not active. Therefore, games generally take much longer to complete than the actual game time.

Each player on the court is typically assigned a position, separating big men from agile and ball handling guards. The following is a brief outline of the five conventional basketball positions[[1]](#footnote-1)

* [Point guard](https://en.wikipedia.org/wiki/Point_guard) (often called the "1") : usually the fastest player on the team, manages the team's offense by controlling the ball and facilitating plays for other players.
* [Shooting guard](https://en.wikipedia.org/wiki/Shooting_guard) (the "2") : creates a high volume of shots on offense, mainly long-ranged; and guards the opponent's best perimeter player on defense.
* [Small forward](https://en.wikipedia.org/wiki/Small_forward) (the "3") : often primarily responsible for scoring points via cuts to the basket and dribble penetration; on defense seeks rebounds and steals, but sometimes plays more actively.
* [Power forward](https://en.wikipedia.org/wiki/Power_forward_%28basketball%29) (the "4"): plays offensively often with their back to the basket; on defense, plays under the basket (in a zone defense) or against the opposing power forward (in [man-to-man defense](https://en.wikipedia.org/wiki/Man-to-man_defense)).
* [Center](https://en.wikipedia.org/wiki/Center_%28basketball%29) (the "5"): uses height and size to score (on offense), to protect the basket closely (on defense), or to rebound.

For most teams today, the shooting guard and small forward have very similar responsibilities and are often called the wings, as do the power forward and center, who are often called post players. Thus, it is not uncommon to not a have multiple players of a given position on the floor at a given time.

**Goal and Problem Statement**

The primary purpose of our program was to simulate a full NBA regular season. This requires using probability theory to quantify the likelihood of a team winning based on fundamental regular season team statistics. We simulate each game of the 2015/16 season and kept track of each team’s win and loss record over its 82 game schedule. After this initial goal was accomplished, we then strove to simulate several seasons and average each teams wins in order to predict the playoff contending teams for each conference. We accomplish these goals by producing a stochastic simulation and kept track of player and team properties during the course of a season. As a result we had several interactions in our model that fluctuated from game to game; thus we implemented techniques of object oriented programming to store this information and to encapsulate these random variable dynamics.

**Markov Matrix**

In order to run a stochastic simulation it was necessary to create a Markov matrix, also known as a transition or probability matrix, quantifying the likelihood that the home team would beat the visiting team. Given that the NBA consists of 30 teams, our probability matrix was 30 x 30, with home teams aligning with the row indices and away teams aligning with the column indices. Our simulation is not a true Markov chain, as it takes into account a team’s recent history, including winning and losing streaks and player injuries, to update team probabilities on a game by game basis. However, we still modeled our simulation as one, using a static probability matrix which we updated accordingly if the game scenario ever deviated from the norm, which we assumed to be no winning or losing streak bias and the rosters of each team to be injury free.

To quantify the transition probabilities, we used regular season data from the 2014-2015 season, and we filtered for regular season matchups between two teams. We only considered at home wins and losses, and scoring differentials between those teams. More advanced statistics could have been considered in the generation of our probability matrix, however, the focus of our project was a regular simulation with class interactions, so implementing basketball analytics at a high level was not our primary project goal

**Data Collection/Processing**

Our regular season data was based off the most recent completed regular season, 2014 -2015 and taken from Basketabll Reference.com (http://www.basketball-reference.com/), an open source data base for regular and postseaon statistics, and team rosters used the most current depth charts from ESPN.[[2]](#footnote-2)

*Probability Calculations*

As stated previously, we used regular season data to produce a transition matrix. Using R, we were able to scrape data from the internet and read into into a dataframe. With this data structured, we selected columns most relevant to our analysis: competing teams and their scores. Next, for every team in the league, we reduced our general data frame to cases where one teams was playing at home, and then we filtered that even further to games where it was playing every other team in the league. Depending on the visiting team there were either 2 or 4 games worth of data, a small sample size for statistical purposes, but adequate for our simulation. With the reduced data, we added two columns: point differential (between the home and away, so a positive value was in favor of the home team) and a dummy variable for home team wins. If the scoring margin was positive, then we converted it into a decimal and squared it and added that the product of the at home winning percentage and one half. Otherwise, with a negative scoring margin, we subtracted the margin converted to a decimal (i.e divided by 100) from one half, as an indicator for how much the probability has deviated from an even match up. The decision to use certains powers and have 0.5 as a scaling factor provided a reasonable range of probabilities. Negative values or probabilities greater than one does not make mathematical sense. With the described transformation, the lowest probability of the team winning being 12% and the highest one was 78%.

It is important to note that, the resultant probability matrix, assumes no bias such as player injuries and teams winning or losing streaks. Our simulation algorithm, however, does alter these probabilities to reflect the game conditions. Thus, our transition matrix serves as a baseline when both teams are at full health and there is no reason to believe recent team performances should increase the home team’s likelihood of winning.

*Team Rosters*

Our program consists of three classes, with the largest being our probability matrix that contains a list of all the teams in the NBA. Each NBA franchise is encapsulated in a team class, with contains a roster, among other data objects. Each team roster is an array of player classes. Our player class contains information related to a basketball player’s profile including age, position, and a basketball overall rating. Player ratings are based off NBA 2k16’s grades for each player.[[3]](#footnote-3) Think about adding a footnote explaining that NBA 2k16). These ratings are an overall assessment of basketball’s player accounting for athleticism, basketball IQ, historical performance/statistics. To generate a roster for each team, we collected depth charts for each NBA team from ESPN. This resource was particularly useful because it ranked each player at any given position for each team. Thus every player ranked at 1 for his respective position is assumed to be the starter. However, when these two lists, player ratings and depth charts, were merged, not every player had a rating, which was the case for marginal players that don’t play regular minutes. And that was a limitation of our sources for NBA 2k ratings. Thus some teams only had 8 people on their rosters while others had over 12. We assume that excluded players consumed minutes that starters and role players didn’t play, which varies based off the size of a team’s main rotation. We felt it was necessary to create a player class for these “bench warmers” because they don’t have a significant impact on the game’s dynamics, and if they were ever to get injured it would have a nominal influence on the outcome of a game, giving their limited role.

Our primary objective is to simulate a full NBA season, specifically the 2015-16 regular season. We aim to simulate thousands of times and using the 2015-2016 season schedule to determine the order of our match ups for our simulation. We will keep track of each team's record for every season trial and average the results.

**Heart of algorithm**

We created a single 30 by 30 probability matrix for the entire league. Teams along the row indices are assumed to be home. So the index of value [row, column] returns the probability of the row (or home) team winning. The complement of said probability is the chance the away team wins on the road. Our probabilities will be contingent on the health of players and the team (5 starters, and on average 5 role players), the minutes played by a player, and lastly team hot and cold streaks. We will keep track of minutes played by each player, as that has implication on a player’s health.

A typical basketball game is 48 minutes per game played by 5 players per team. Therefore, each team has to distribute 48\*5 or 240 player minutes a game. For simplicity, we assumed that all starters averaged 30 minutes a game, and role players consumed 12-18 minutes per game, depending on a team’s depth chart. The remaining player minutes would be played out by a team’s “bench warmers”, marginal players that eat up marginal minutes to give starters and role players rest. During the course of the season, a player’s minutes will accumulate and after a certain threshold (2000 minutes for a season), there likelihood of getting injured/sitting out a game increase. After every game we run the probability and determine if any of the players have accrued injuries and we also determine the severity of the injury.

**The Utilization of Object Orientation and Classification**

**System architecture and organization**

The object oriented properties of C++ undoubtedly allowed for the relative ease in the coding of this simulation. With data structures and classes, it was easy to package functions, pieces of data, as well as other objects into one larger structure. Once the base classes and their functions were prepared, object orientation and abstraction made it easy to forget about their inner workings and only perform mutations on those classes using the well named and accessible functions associated with them.

Object orientation and data abstraction also facilitated the design of the simulation as code for the multiple classes that interact and depend on each other could be written simultaneously. After a brief discussion where class data members, function headers and outputs, and basic functionality was discussed and decided upon, the members of the team could design each class independently of one another and still effectively create a working algorithm.

For the this simulation, three classes were created: a player class, a team class, and a probability matrix class. Though the inner workings of each class are discussed below, a brief introduction of class and their functionality is necessary here.

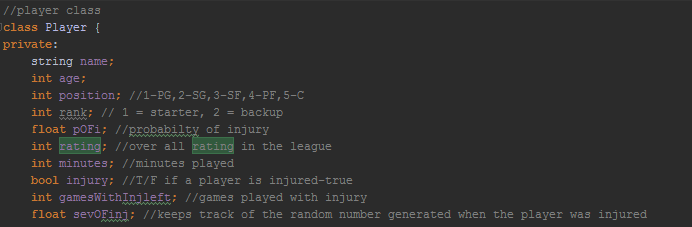


Figure 1. Overview of Player class data members.

The player class contains information that applies to each player of the game, such as their age, their overall ranking, their position, their probability of getting injured, as well as the minutes of gameplay they have acquired during the season. Though the simplest class, the player class demonstrates some complexity as the minutes played in past games affects their probability of getting injured in any given game. Additionally, the player class contains functions that allow the class user to update the minutes played, as well as update their probability of injury.

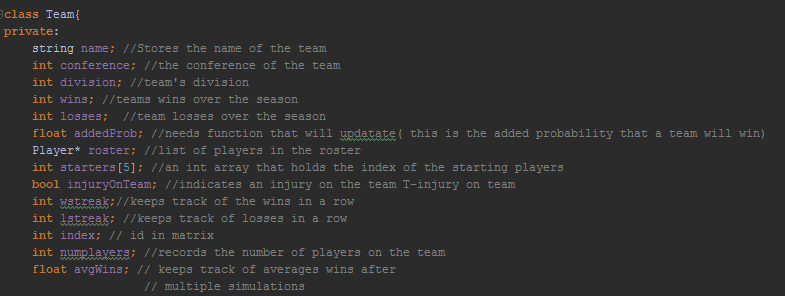


Figure 2. Overview of Team class data members.

Similar to the player class, the team class contains salient information about each team such as the roster of players, if there is an injury on the team, its win loss record, as well as its index in the probability matrix. The team class also contains functions that dynamically update the team's data members in the case of a win or a loss. The number of minutes played by each of the team's players as well as its win loss record are some of the members updated after every game.

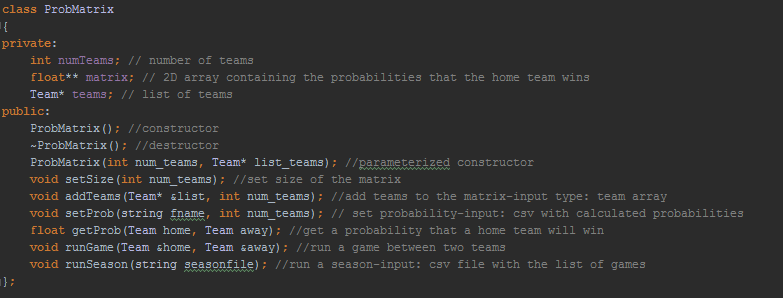


Figure 3. Overview of probability matrix class.

The last and most complex data type is the probability matrix data type. Derived from information gathered from the NBA and ESPN, the probability matrix data type holds the information containing the probability of each team winning over the other. The most important functionality of the probability matrix class is its ability to grab the probability that one team wins over the other, account for deviations from this standard probability by taking into account each teams injury and win/loss history and output a winner. The probability matrix class functionality, however, extends much further. Not only can it simulate a season by simulating multiple games in sequence, but it also accepts \*.csv files that initializes the teams being considered, initialized the players on each of the teams, and imports respective probabilities of each team winning at home versus a given opponent.

Without object orientation and data abstraction, the designers of this program would have to manually update each object in the program every time something changed. With object orientation, a simple function call will perform a preprogrammed sequence of events that corresponds to the appropriate dynamics of the system at that particular time without having to update each data member. Additionally, the data abstraction that object orientation allows for makes it easy to group together program variables and only access the ones that make sense.

*Class Types*

**"Players.h/cpp"**

Though each of the data structures involved in this program were mentioned in brief above, it is now important to delve deeper into the classes, understand their functionality, describe the processes in which they interact with other data structures, and highlight their key features.

The "players" class makes use of a slew of functions important for updating and returning the status of its data members. Much of the functions in this class are related to accessing and mutating data members such as the "rating" of a player, and updating the minutes played in a season as shown below.

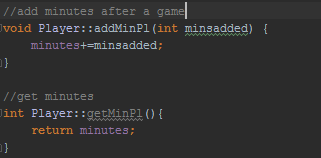
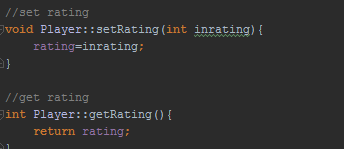


Figure 4a and 4b. The screenshots illustrate a few get and set functions for the player class.

Functions like "setRating" are only set at the beginning of the program when the players are imported via a csv file. Other functions, such as "addMinPl" are updated after every game is played.

One of the more interesting functions in "Players.h" is the "set injury probability" function and "if injured in game" function. The "set injury probability must be updated after every game because it depends, not only on a players age (a constant variable), but also on the minutes played throughout the course of the season. Though we did not research the correlation between age and probability of getting injured playing basketball we assumed a logarithmic correlation for the purposes of this simulation. In this simulation, as a player gets older, his chances of getting injured increases quickly when the player is young and slower than when he is old. When a more accurate equation if found, the program function can be easily altered. Additionally, we assumed that if a player exceeds 2400 minutes of playing time, an additional few points based on how far they exceeded 2400 minutes will be added to the probability of getting injured. Again, though these equations are only premonitions, they work for the purposes of this simulation, and can be easily adjusted.

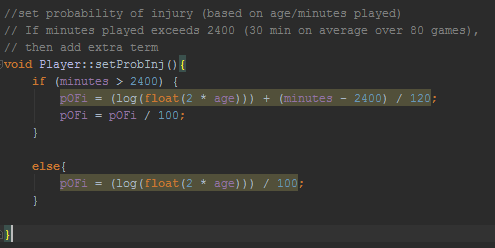


Figure 5. Overview of set probability of injury function for players class.

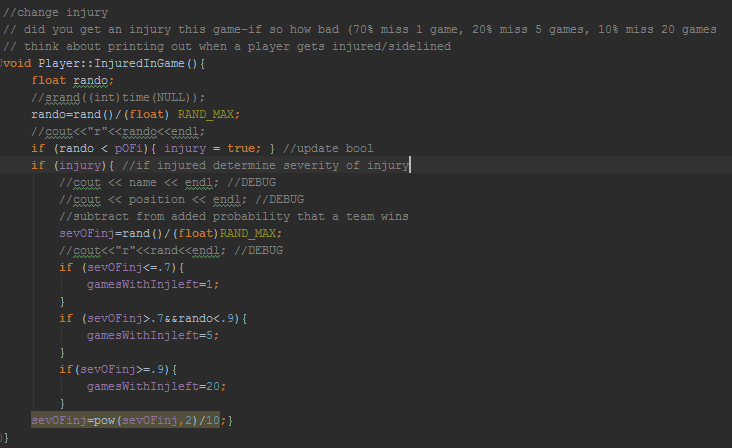
The function that determines if a player is injured after a game first generates a random number. If that number is greater than the player's probability of injury, then another random number will be generated to determine the severity of the injury. If the new random number is less than 0.7, then the player will only be injured (or sit out) for one game. If the number is higher than 0.7 and less than 0.9, then the player will be injured for 5 games. If the number is larger than 0.9, then we assume that the player has had a severe injury and does not play for 20 games. These numbers are more or less proxies, but allow for injury to happen relatively often-semi-realistic of a real basket ball season. The appearance of commented out debug "cout" statements demonstrates how we checked a realistic frequency of injuries in a game. This particular function updates the boolean "injury" variable, "severityOfInjury" variable, as well as the "gamesWithInjLeft" variable which are all accessed by various functions in other classes within the program. 

Figure 6. Screen shot of injured in game function for players class.

**"Team.h/cpp"**

The team class is the second level of abstraction in the program. The most important data members of the Teams class is the roster, the starter array, the average wins per season, and the team's index in the probability matrix. The index, and the roster are all imported at runtime via a csv file, the starter array is updated before every game in order to account for team injuries, and the average wins per season is updated after every simulated season.

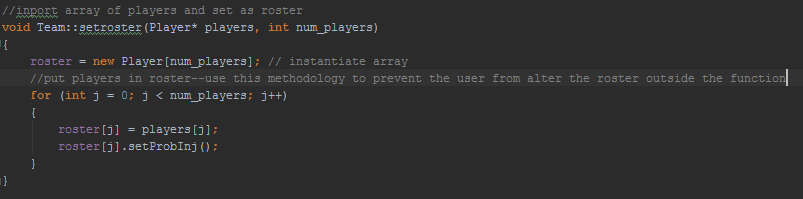
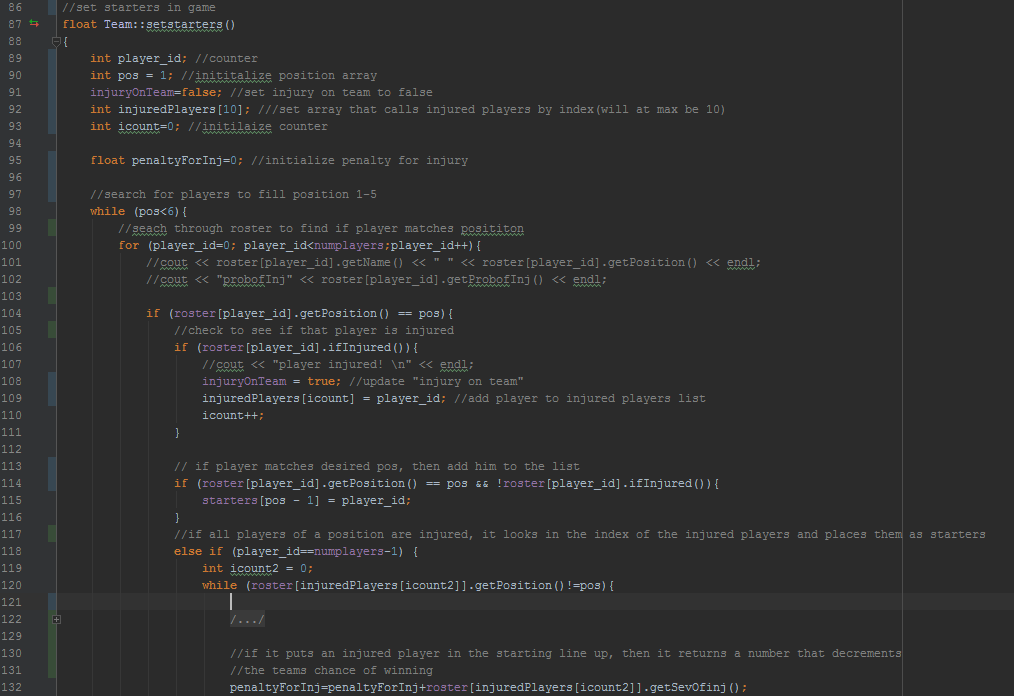


Figure 7. Overview of set roster function for Team class.

The set roster function is more or less self explanatory: it takes an array of players as well as the number of players on the team as an input, then it sets the players in that array to the rosters array data member in the class. We chose to implement in this way, which is perhaps less time efficient in order to ensure that the roster was not getting altered by what happens in other parts of the code.



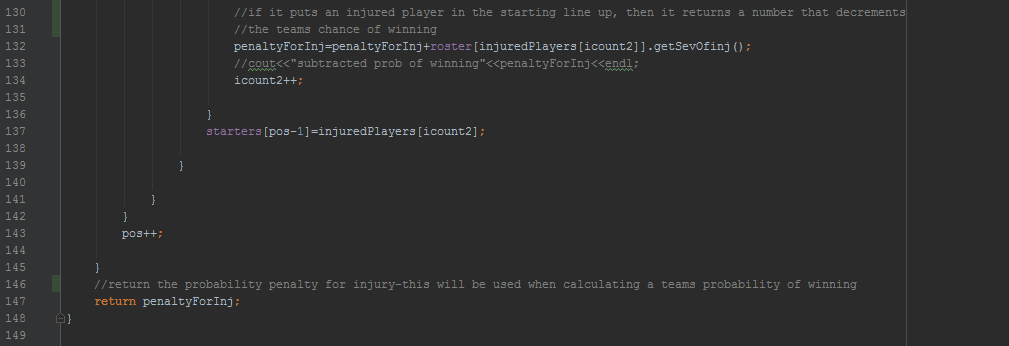
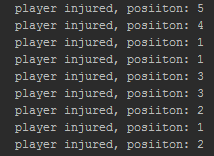
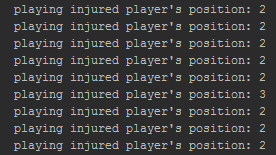


Figure 8. Set starters function of Team class.

The set starters function was one of the more challenging functions to implement, and though it stands partially functional at this point (ie it is not detrimental to the functioning of the program as a whole), it does have a drawback which will be discussed in a moment. The nature of the algorithm proceeds as such. Before every game this function is called. First the function searches through all of the players on the roster looking for players to fill position '1'. If the program comes across a player whose position is '1', and that player is not injured, then the player's index in the roster will be placed in the starters array. Because the players in the roster are organized by position and sorted with the best players at the top of the list, the first player whose position matches the one the program is looking for will automatically be the starter. If the program comes across a player who had been injured in a previous game, then it adds that players index to an injured players array. If the program has looked through all of the players in the team's roster and still hasn't found an uninjured player whose position matches the one looking for, then the program searches through the injured players array to find an injured player whose position matches the one that the program is looking for. When the program finds the injured player who fills that position, it puts the index of that player in the starting players array, but also adds a probability penalty for having an injured player play in a game. At the end of this function, the pentaltyForInjury is returned for use when calculating the final probability that the home team wins.

Because players get injured during every game and other players get better after every game, this function will be run before every game starts. One drawback of this algorithm is that it does not account for the interchangeability of some players. Additionally, for some reason, that has yet to be discovered by us, this function seems to favor putting injured player's whose position is '2' in the game as can be seen in the picture to the left. This is odd, especially as there is a relatively even distribution of injured players. As can be seen in the figure below and to the right.

Figures 9a and 9b. A snapshot of player injuries over the course of the season and in some circumstances, injured players forced to start.

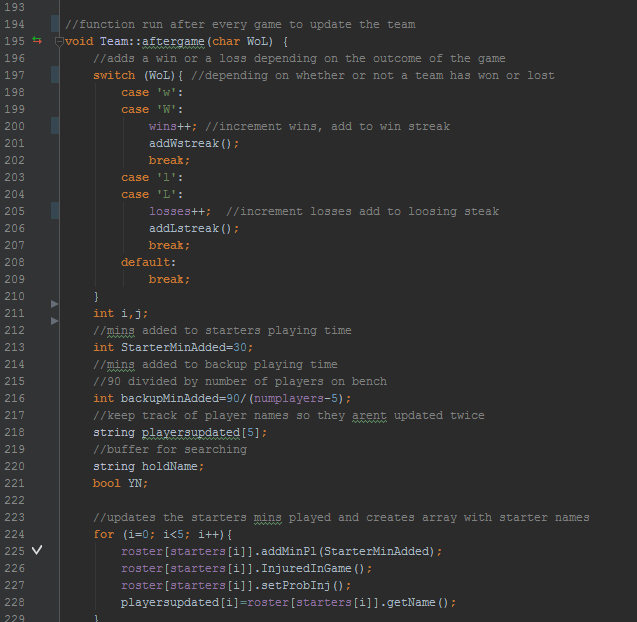


Figure 10a. Top half of after game function.

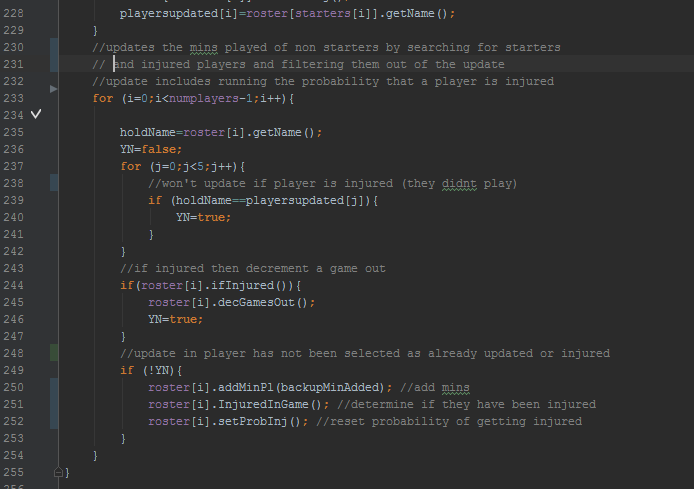


Figure 10b. Bottom half of after game function.

The "aftergame" function is another integral function in the "team.h" class. Run after every game when it is called by the probability matrix class, 'aftergame" takes in a character to determine if the team won or loss and then updates the team's wins and losses. "Aftergame" first updates the minutes of the players who started in the previous game and determines if they have received and injury. Then "aftergame" searches through the roster, identifies those players who have yet to be updated. If a player was injured, then their games left with injury decrements and no minutes are added to their minutes of "game play". The other players in the game have the appropriate amount of minutes added to their "minutes of season game play" and we determine if each player was injured during a game using the function call "injured in game." In this way, the the "Team" class updates both itself and the player class as it is executed.

***ProbMat.h/cpp***

Because the probability matrix class holds the probability that one team beats another team in any given game (at home or away), we decided that it was also the most convenient way to store the entire NBA team and player information.

This class has the ability to find the probability that one team beats another as well as the ability to run a game between two teams AND run an entire season. In order to set up a simulation, a list of teams must be imported as well as the probability that one team wins over another. "addTeams" accepts an array of teams and creates the list of teams for the class. "SetProb" accepts a .csv file and imports the probability data into the "matrix" data member of the class.

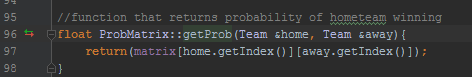


Figure 11. Overview of get probability function for matrix calss.

The function, "getProb" accepts pointers to the two teams that are being compared, finds their indexes in the matrix, then reaches into the matrix to identify the probability that the home team will win. This function will be utilized in the "runGame" function. Which will be discussed next. Finally, a game between two teams can be simulated.

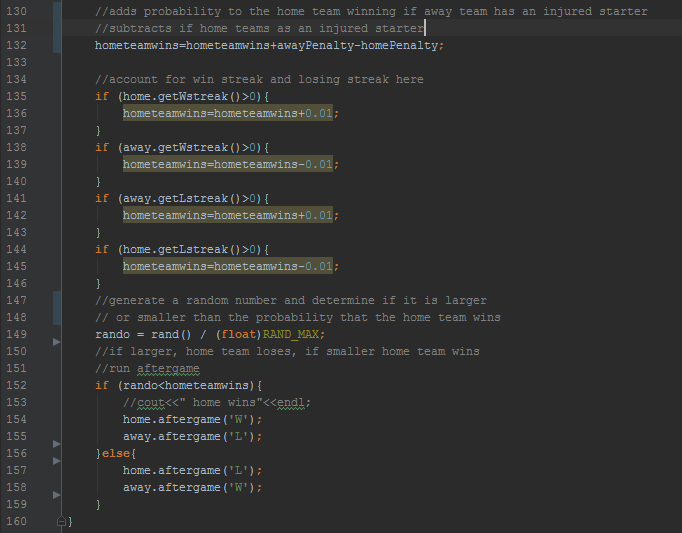
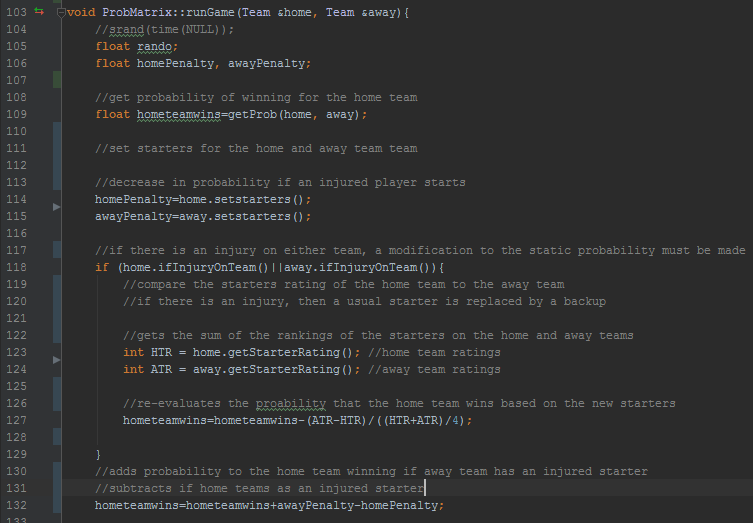


Figure 12. Run game function overview from probability matrix class.

"RunGame" is one of the more dynamic functions in this program. The function accepts pointers to the two teams that are playing each other and finds the probability that the home team will win. Then the function calls "setStarters" for the two teams. Set starters returns a predetermined value that corresponds to the decrease in probability that a team withstands because of an injured player starting in the game. This probability decrease is reconciled in line 130. Then, the function looks to see if there is an injury on the team. If there is an injury on the team, then the team's default players are not in the game and the default probability no longer stands. The function compares the sum of each teams starter rankings and subtracts accordingly (line 127). Next, the function looks at each teams win and loss record. We assume that if a team is on a winning streak, they are more likely to win the next game. Likewise, if a team is on a losing streak, then they are more likely to lose the next next game. These probabilities are added and subtracted in lines 134-136. Finally, the function generates a random number between 1 and 0 and compares that number to the probability that the home team wins. If the random number is higher than the probability, then the home team has lost. If the random number is lower than the probability of it winning, then the home team has won. Depending on the outcome of the game, "aftergame" is run for both the home team and away team. As mentioned before, this function updates both teams accordingly.

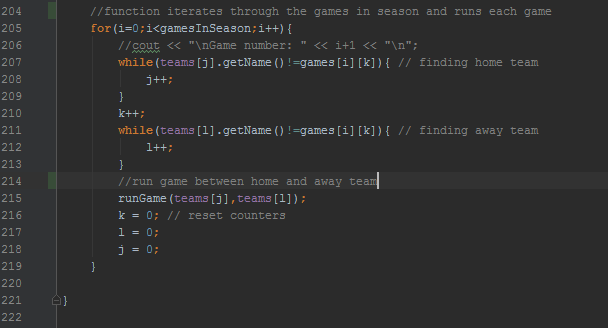


Figure 13. Overview of run season function from probability matrix class.

The "runSeason" function accepts a csv file holding the sequence of games in a season. The bulk of the code "runSeason" function deals with importing and storing this data. Once the schedule has been imported, the function iterates through the games, looks for teams that match the name of the team in the csv file, then calls "runGame" and runs the game between the two teams.

**Program Design & Evaluation**

Our program did not require a high degree of complex abstraction and our implementation was built from fundamental data types. Beyond the use of our three classes, our second most complex data structure were arrays. Our program made efficient use of dynamically allocated arrays, as the size of some these objects was determined until run time, and more specifically when a particular csv file was read. The use of pointer objects proved to be beneficial when it came time to fill in the arrays, for it wasn’t necessary to use pass by reference. In some scenarios where we were passing non-pointer objects whose values we needed to update, it became necessary to pass them by reference. These instances weren’t very common but were applied when appropriate. The possibility of using vectors over arrays was briefly explored, only because they appeared to be easier to instantiate and manipulate, but once we determined how to appropriately allocate an array, we decided to stick with arrays.

Many of our higher level functions such as "runGame" and "runSeason", did not return any values, as it was only necessary to update certain data members of the Team and player classes. And when it came time to rank teams for each conference to determine which ones made the playoffs, it was only necessary to print out the results. As there was no need to deal with those classes after that operation. Had we decided to predict playoff winners, then it be necessary to return an array of teams after sorting the top 8 teams from each conference.

*More about generate the standings for each conference.*

Given the NBA’s standards for determining playoff seeds, we needed to sort multiple groups of teams, particularly the top four playoffs seeds, which consisted of the division winners and the team with the next best record, and the remaining 11 teams. To sort theses different subarrays for a given conference, we implemented a very simple algorithm in bubble sort. While, it may not be the most efficient or sophisticated algorithm, we felt given its simplicity and our small number of object, which was 15 at most, simplicity was preferred over complexity. The first grouping of teams was by conference: eastern and western, thus leaving two groups of 15 teams. We furthered divided those teams into their respective divisions. This results in 6 groups of 5. Our first sorting operation arranged teams within a division in descending order by average wins per season. We then divided the divisions into 5 groups for eastern divisions and 5 groups for the western division. Each group was based on what place a team finished in its own division, thus we had a group of 1st place division winners, 2nd place finishers, ect… for each conference. From those grouping it was possible to separate the top four teams from the remaining eleven. The top four would consist of the 1st place division winners plus the team with the best record from the second place finishers for a conference.

*Testing the Code*

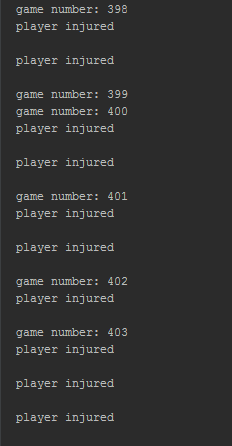


Figure 14. Print out of player injuries for a sample of games.

The snapshot above shows how often a player is injured per game. This output was generated using a cout statement within the "injuredInGame" function. It is observable that there are about 2 injuries per game. This figure is a bit large, but also mostly expected with the algorithms used to evaluate if a player is injured. Considering that the algorithm takes into account players that have to sit out a game or two, these results make sense.

The snapshot below was generated by adding a cout statement right below the "runGame" call in the "runSeason" function. It is clear to see that the modifications on home team winning probability are working correctly. The probability modifications vary around 5 percent and sometimes, if neither team has sustained any injuries, then the probability remains the same.

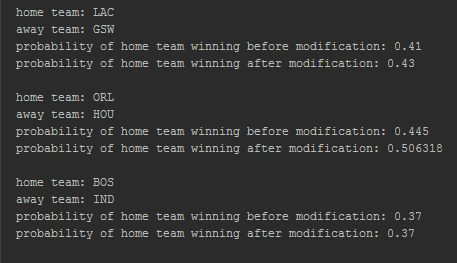


Figure 15. Comparing home team probabilities before and after any potential modificaitons.

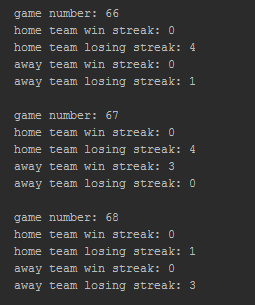


Figure 16. Print out of winning and losing streaks of teams mid-season.

The output to the left was generated by adding cout statements after the "runGame" call in the "runSeason" function. Here, it is clear to see that there is a fair distribution of wins and losses. More importantly, there are no teams that are on a winning and losing streak.

There are certain areas where we could improve the design and implementation of program, starting with the modularity of our code.Making changes to certain aspects of our classes took some effort (e.g. adding implications on team probability if an injured player was forced to start.) In certain scenarios it is hard to predict what functions in particular will require flexibility, but in other cases that should always be part of the design. Another aspect to address would be the readability of our code, and this goes inline with modularity by implementing more helper functions. An increase in readability will lead to decrease in complexity too.

*Results of program (see figure 17-19 in appendix)*

We ran our program at multiple trial lengths including 50, 250, and 1000 season simulations. It appears that our simulation has teams’ winning records regressing toward the mean (i.e around 41 wins per season). Even when simulating for just 50 games there appeared to be little variance among playoff contending teams, with the best record in the league averaging slightly over 47 wins. And these results are attributable to our simplistic model, which only accounts for a small sample size of wins and point differentials between two teams to determine the probability of the home team winning, if both sides are healthy. A more sophisticated model would account for more advanced team statistics and even player analytics that would have more implications from game to game. Nevertheless, our model does have some element of consistency to it as teams rarely jump up or down a seed or two between 50 and 1000 simulations.

Comparing our results, particularly after simulating 1000 season, there are some predictions that match well with the playoff racing unfolding now, but there is no consistency. First off, for the eastern conference, we predict that the Charlotte Bobcats will finish first with roughly 46 wins for the season, followed by Chicago with 45 wins. Referring to the eastern conference standings we observed that has the potential to obtain 40+ wins for the season but is unlikely to finish in first. And likewise, Chicago over performed in our simulation, however, it is important to know that Chicago’s best player has been injured for the past month, which has caused the team to drop to the 9th seed. Other interesting observations are the Cleveland Cavaliers, the presumed favorites to come out of the eastern conference finals come playoff time for this year and last, rank near the bottom in our simulations. Also, the Golden State Warriors, from the western conference, are favored to finish near the top of their conference standings in our program, in reality are indisputably the best team in the league right with nearly 60 wins already, and are on pace to have the best regular season in NBA history.

**Challenges & Limitations**

Undoubtedly, the greatest challenge for our group was the short span of we had to complete the project. We had initially decided to create a food recommendation app for students looking to eat on campus, but after a couple weeks we changed project ideas to one for more in line with the project objectives. Nevertheless, we were able to produce program with a fair level of complexity. While scripting programs such as R and Python are more efficient for prototyping, giving the high level of class-based interaction, using a language like C++ allowed for a wider range of programmability in manipulating our objects.

This project gave us valuable programming experience at the expense of debugging various functions and operation throughout the development of the program. One big challenge was importing pre-processed data from R into C++. While handling csv files in a script language if fairly straightforward, trying to handle them in C++ proved to a struggle at first. We had to learn about string buffers (i.e. string streams), how to read a file row by row, and then element by element. After reviewing the code, the process of reading csv doesn’t seem very daunting but the first time doing so proved to be a non-trivial task. Additional challenges came when our conditional statements for different operations were not programmed properly, and thus we kept getting accessing random parts of memory or getting erroneous values. That largely comes with programming such a large project with hundreds of lines of code, so commenting and print out various lines of code was crucial to get an understanding of how our program was functioning.

As to the results of our project, the low level of predictability can be attributed to several assumptions and simplifications we made in our model. Several include assuming all starters average the same amount of minutes each game and the same for role players on a given team. While it is expected that a starter will typically play more than half the game, the true number varies by age and team. Some teams, with limited talent, expect their starters to carry a higher load but will distribute more minutes to younger starters. To add complexity on this part of our model, we should gather information on the averages minutes per game each starter has in the league and then have our probability of getting injured based on minutes vary by age. Accounting for the games played in previous seasons by a player would also be useful for predicting the likelihood of a player missing a game or a stretch of games.

We did not account for overtime games in minutes played per season, as that would increase the complexity of our model. And given our limited amount of time, there was only so much we could account for. Another over-simplification was forcing injured players to start if there wasn’t anyone healthy at that position, while in reality guards and forwards are fairly interchangeable among themselves. Other simplifications made in the program were never updating a player’s probability of getting injured after an injury, accounting for changing rosters, and the concept of resting high contributing players at the tail end of the season when team has a fairly high chance of getting home court during the start of the playoffs. Another limited assumption was using 2K ratings to assess the value of a player. We used NBA 2k16 player ratings to quantify the impact of substituting one starter for a role player. A more accurate measure would be player efficiency rating (PER), and perhaps other measures that account for the defensive value added (or subtracted) by player. Lastly, we didn’t consider many intra-team and inter-team dynamics, which is a limitation of only accounting for team statistics in our simulation. In addition to teams having winning and losing streaks, so do players. These can be accounted for in terms of increased point and rebound production over a stretch a games. We largely didn’t account for such statistics, as we determined predictive analytics at this level was out of scope for us. We were predicting season long trends rather than game to game trends.

**Conclusions/Future Work**

From a object oriented standpoint, we are satisfied the with outcome of the project in terms of implementing our abstract data types for NBA teams and players as well as capturing multiple class interactions. This project provided us with valuable coding experience in C++ and has undoubtedly improved our debugging skills. While we incorporated several in-class concepts, we did learn how to read and parse data from csv files, which is good use for any data related work.

There was certainly a refinement process that took time to first get all our code to run together to simulate one game: read a csv file first and instantiate a list of teams and players. Get our probability matrix to fill in properly. Then make sure we had a fully functioning random number generator. After which, it came time to run a single game simulation, followed by simulating an entire season, and then multiple seasons. Once we got that going, then it came time to address some of the simplifications of our model. This last part is a continual process that is limited by time constraints. But given the scope of the project we were satisfied with the program’s complexity.

In terms of future work on this project there is much to look for in increase the sophistication and robustness of our simulation probabilities and analyzing the results generated from our simulation to provide insight for biases/naive assumptions. Fivethirtyeight.com, for example, provides excellent posts on basketball analytics, so that would be a good starting point, if we wanted to learn more about advance metrics used in the NBA and how statisticians are incorporating them to make predictions. They in fact have their own model for ranking players and teams which they to predict game winners.[[4]](#footnote-4),[[5]](#footnote-5)And more complexity in our statistical analysis would translate to a more dynamic program. Perhaps the best way to predict NBA games would be to do them week by week, so we could web scrape recent data such as up to date player and team statistics and recent trades. And then late into the season (e.g. 20 games left in the season) we would generate predicted wins for the seasons and rank teams accordingly for playoff seeds. With more data particularly on intra-team dynamics, we could allow for richer and more complex class interaction in our program. So perhaps with future coursework in statistical models and more programming experience this would be a more feasible challenge.

**Appendix**

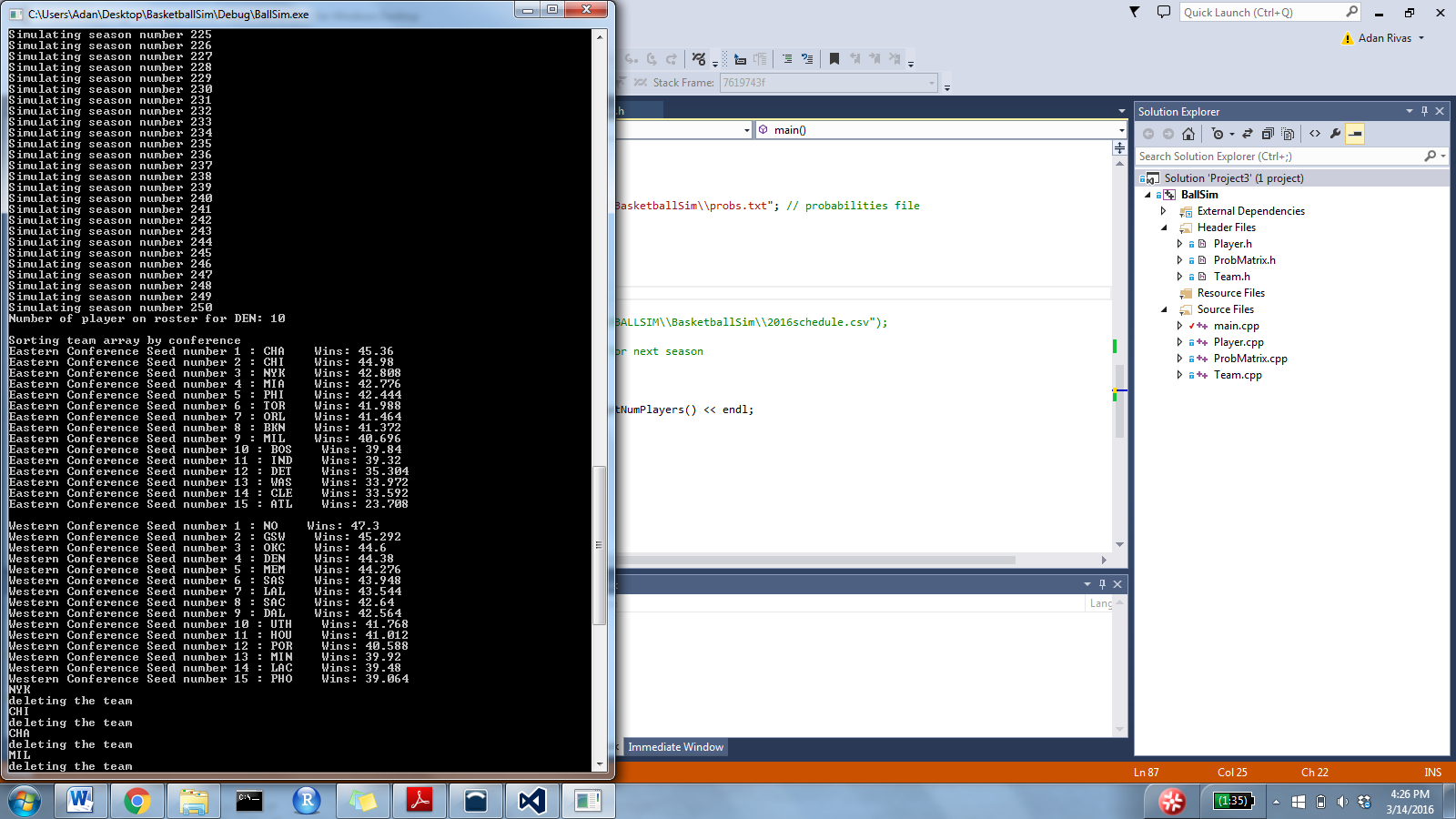


Figure 17. Results after 1000 season simulations.

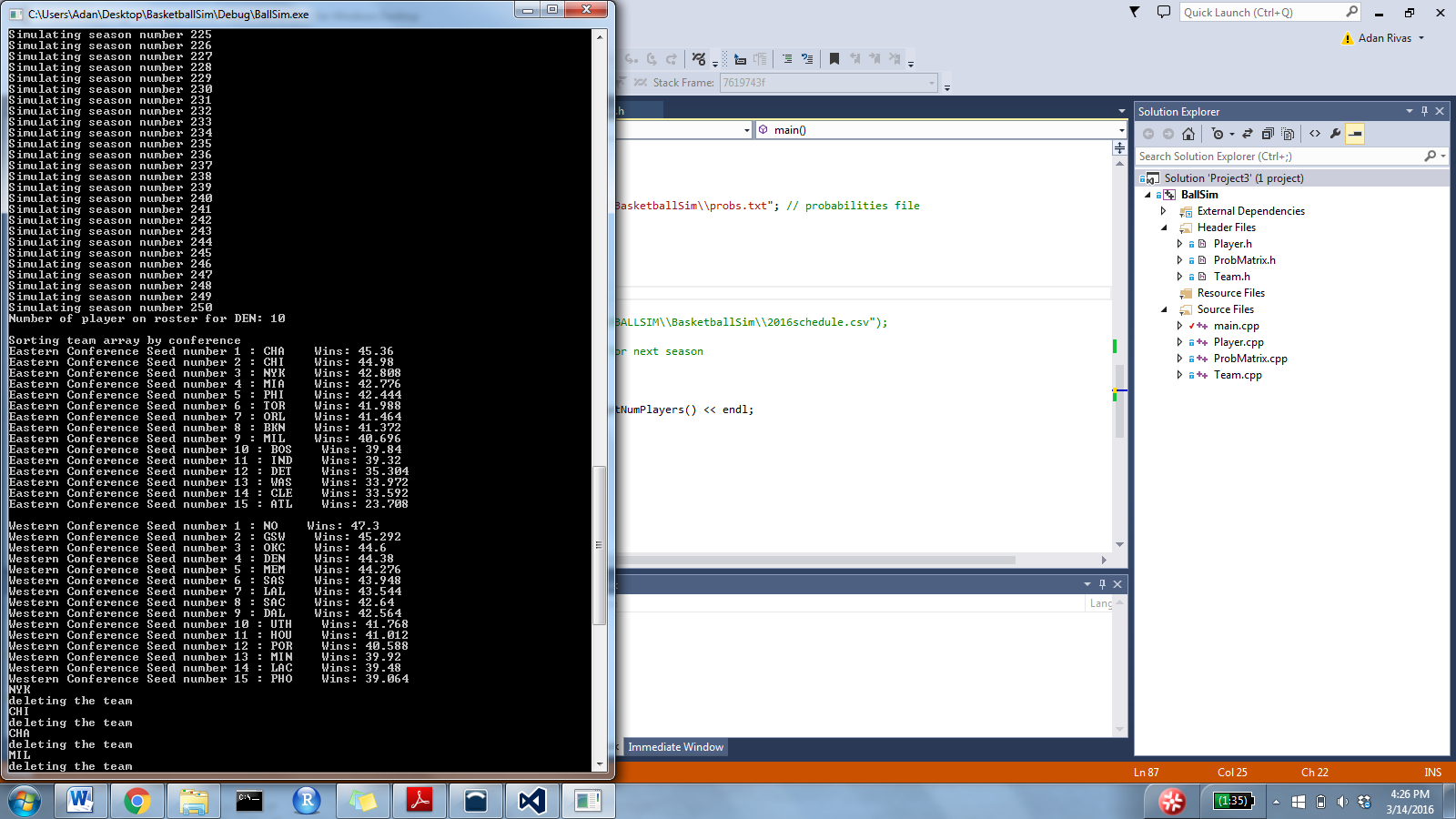


Figure 18. Results after 250 season simulations

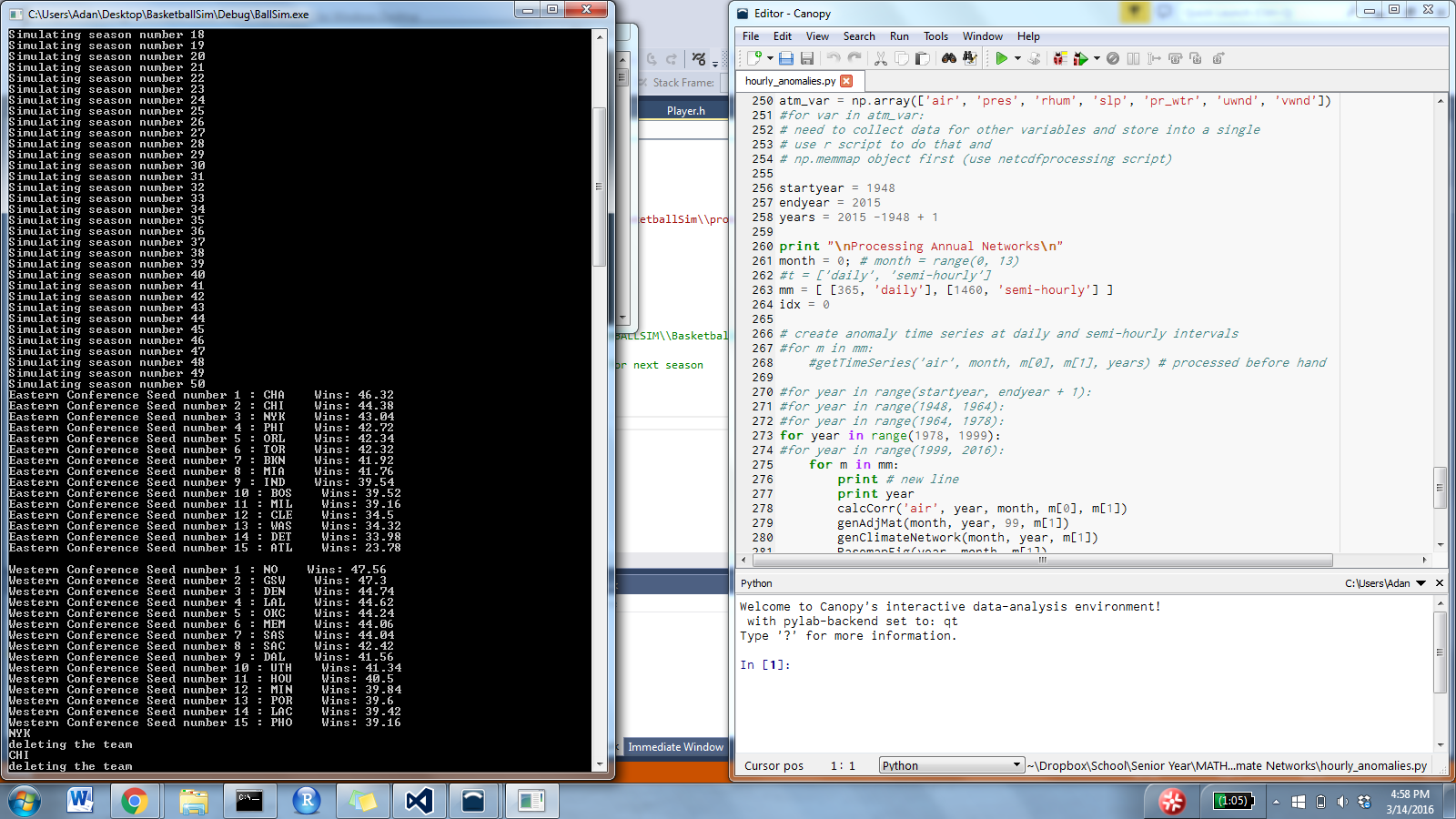


Figure 19. Results after 50 season simulations

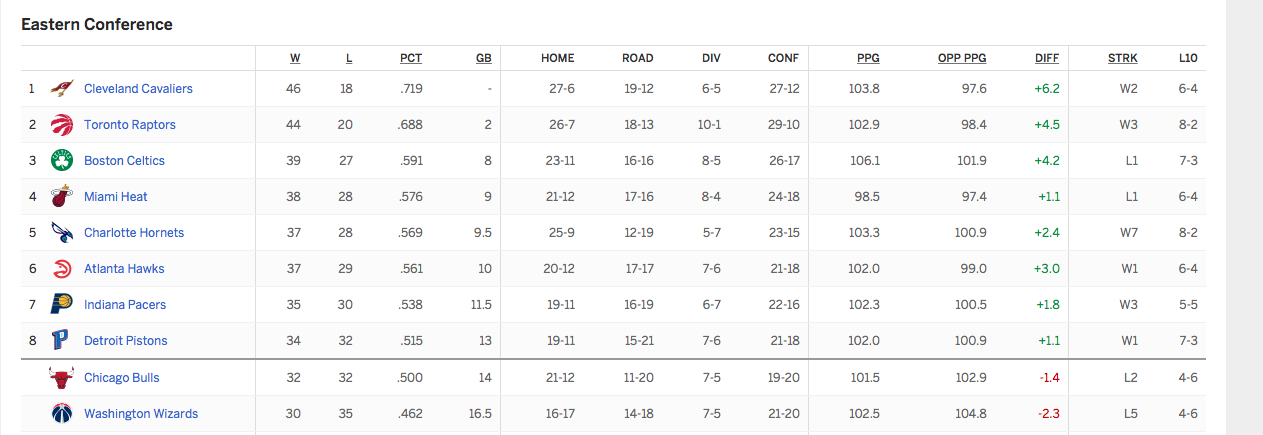


Figure 20. Eastern conference standings (as of 3/13/16)

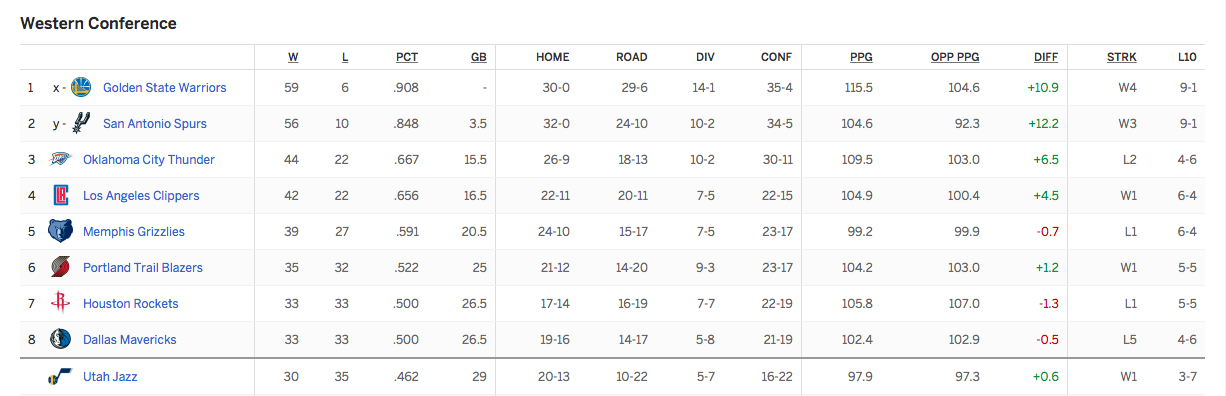


Figure 21. Western Conference Standings (as of 3/13/16)

Source for figures 20 and 21:

NBA. "2015-2016 Conference Regular Season Standings." Standings. March 13, 2016. <http://www.nba.com/standings/team_record_comparison/conferenceNew_Std_Cnf.html>

1. "Basketball." Wikipedia. March 10, 2016. Accessed March 11, 2016. <https://en.wikipedia.org/wiki/Basketball>. [↑](#footnote-ref-1)
2. "NBA Depth Charts." ESPN. Accessed March 01, 2016. http://espn.go.com/nba/depth/type/full [↑](#footnote-ref-2)
3. Hype, Hoops. "These Are the Ratings of All Players in NBA 2k16." HoopsHype. November 05, 2015. Accessed March 01, 2016. http://hoopshype.com/2015/11/05/ratings-of-all-players-in-nba-2k16/. [↑](#footnote-ref-3)
4. to Boice, Jay, Reuben Fischer-Baum, and Nate Silver. "2015-16 NBA Predicitons." FiveThirtyEight. March 3, 2016. <http://projects.fivethirtyeight.com/2016-nba-picks/> [↑](#footnote-ref-4)
5. Boice, Jay. "How Our 2015-16 NBA Predictions Work." FiveThirtyEight. December 7, 2015. http://fivethirtyeight.com/features/how-our-2015-16-nba-predictions-work/ [↑](#footnote-ref-5)