Fantasy Football Analysis

Adam Abdulhamid
Computer Science
Stanford University
adama94@stanford.edu

Tyler Fallon
Mathematics
Stanford University
tfallon5@stanford.edu

Adam Warmoth
Engineering Physics
Stanford University
awarmoth@stanford.edu

December 11, 2015

Abstract

Fantasy football has grown into a $70 billion market, with more than thirty million Americans participating annually. The premise for fantasy football is simple. After each player has played their weekly game, their statistics are added up, and a fantasy point total is calculated. Weekly fantasy leagues like FanDuel have been growing in popularity over the past few years, largely because the commitment is only one week, unlike the entire season for traditional fantasy leagues. Each participant picks nine players (1 quarterback, 2 running backs, 3 wide receivers, 1 tight end, 1 kicker, and a team defense) from that week's games. Each player is assigned a salary correlated with how that player is expected to perform that week, and the participant must choose a lineup such that the total salary of all the players is under a specified maximum amount. The football games are then played, and each player earns a certain number of fantasy points based on their performance (number of touchdowns, yards, etc.). At the end of the week, the points from all of the players on one's team are summed up to get a score for the contestant's team. The type of game that we are looking at is the 50/50 contest, a game in which the top half of the participants win and double their entry fee.

The goal of this project is to model this game as a Constraint Satisfaction problem and try to find the optimal lineup(s) for a given week. In this paper, we will use both domain-specific knowledge as well as a few variations of backtracking search to figure out what approach is best suited to tackle this problem.

1 Task Definition

Given the structure of Fanduel, we wanted to see if we could apply our knowledge of artificial intelligence to generate lineups for these weekly fantasy leagues. Our goal is to try to generate the highest scoring lineups we can. The main success metric we will have is to generate lineups for past weeks and then compute what they would have scored in reality. FanDuel gives us the average score required to win a 50/50 game (a game in which slightly less than 50% of the entrants win), which is 111.21. We plan on using this as a benchmark to see how far above this we can get the average score of our lineups. We can also compute exactly how many of our lineups perform better than 111.21, versus the number of lineups we submitted, to get a win percentage. If we are above 50% win percentage, then our algorithm can make a return on investment.

2 Related Work

There have been plenty of other fantasy football projects, but nearly all have been an attempt at solving an entirely different problem than ours. The common problem is to try to generate weekly projections for a given player, where our project's goal is to generate optimal lineups given said different projection data. Given that this is the case, it doesn't seem as if we have anything specific to reference in terms of similar projects.

We did come across a project written by [1] an artificial intelligence PhD student. His project was similar to ours,

except that instead of American Football, his is trying to pick optimal lineups for European Football, as in soccer. His approach is to use a constraint satisfaction problem to model this and try to pick lineups that maximize the previous year's points scored. The big takeaway we can apply to our project is his mention of the large size of the domain values. He mentions that for a CSP with n unknown variables (15 positions in the case for soccer) with a domain of d possible choices (all the possible players, which is roughly a few hundred) we can work out how many possible combinations there are by calculating d^n. In his case, this is roughly 300^{15}, and in our case for American Football we have roughly 450^9. Obviously these domains are extremely large, but there are heuristics using our domain knowledge that we can apply to pick lineups without having to generate all 450^9 of them. This soccer project mentions certain heuristics such as picking the players with highest points totals first, and branching from there, which could be useful in finding successful lineups while minimizing brute force computation.

3 Approach

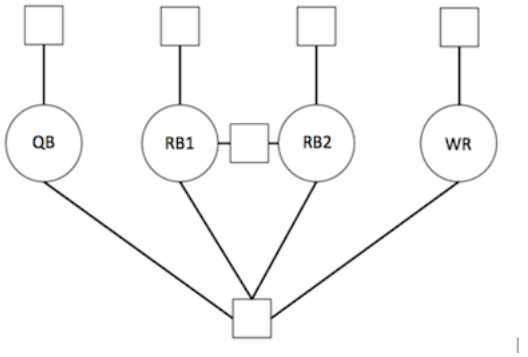
3.1 Modelling

We are going to model this as a factor graph as described in class. Intuitively, it makes sense that we want some sort of constraint satisfaction problem, since we are trying to pick the best players we can subject to a certain set of constraints on salary and position of each player.

The data we need to create this factor graph is a player object that contains the players name, position, and salary. This data is obtained from a website called rotoguru [2], which provides us with information about the salaries of each player for any given week. This website also provides us with data for evaluating our algorithm by providing the actual fantasy points that player scored in any given week. We can model this as a constraint satisfaction problem as follows:

- Variables - QB, RB1, RB2, WR1, WR2, WR3, TE, K, D
- Values - The values for these variables are tuples of player and their associated position, salary, ESPN projection, and Efficiency (we will define the latter two later).
- Domains - The domains for each variable are all (Player, Salary, Projection, Efficiency) tuples for that position
- Constraints
  - Unary - Each variable must be filled
  - Binary - RB1 and RB2 cannot be the same player
  - Ternary - WRs 1,2,3 cannot be the same player
  - This Ternary constraint can be broken down into 3 Binary constraints
  - Sum - Sum over cost of player in each variable is less than Salary Cap ($60,000)

We can apply a similar technique to the one we used in our homework about constraint satisfaction problems to generate auxiliary variables and use these to have a sum variable. We can then put a constraint on the sum variable that this is less than the salary cap. An example factor graph can be visualized in the figure below. This doesn't have all of the required variables, but we can get a good understanding from looking at it.



We tried creating this CSP but eventually ended up with a modified version, because the technique we had used in class to create the sum variable here is intractable. Rather than restrict the salary cap with a sum variable, we instead restricted it by pruning assignments that violated the cap in the backtracking search algorithm. We used our domain knowledge of fantasy football to add heuristics into the backtracking search algorithm to stop searching if certain criteria were met. The reason for doing this was due

to the huge domain size of the sum variable. Because the highest individual salary is $10000, the sum could have any value between 0 and 90000, going by increments of 100, which comes out to 900 values. Since the sum variable value is a pair of these values, each domain would have size on the order of 900^2. Constraint satisfaction problems are exponential if an exact solution is desired, so adding these variables with very large domains causes the computation to be unfeasible. Therefore, it made much more sense to prune mid-search, and this doesn't hurt the outcome, as the result is equivalent. By halting the search when it violates the salary cap constraint, we can accomplish the same goal.

The other thing we slightly modified about our constraint satisfaction problem is that instead of using a ternary constraint, which can be difficult to implement in code, we used three seperate binary constraints between the different WR variables. Each constraint enforces that the two WR positions cannot be assigned the same value. It is easy to see that these three binary constraints are equivalent to a ternary constraint between all three of the WR variables.

3.2 Algorithms

3.2.1 Backtracking Search

It is important to note here that all the constraints here simply ensure that every solution will be a valid lineup (a valid lineup is just a lineup where all positions are filled and the total salary is under the maximum allowed amount). We have done nothing yet to try to maximize our expected points scored. Now that we have introduced the model, we can look at algorithms to solve this problem. We can apply backtracking search as discussed in class, which will be an easy way for us to solve this problem. Again, note that solving the problem still has nothing to do with finding an optimal lineup. Any valid lineup, even if it consists of the worst players and ends up far under the salary cap, will be considered a solution here. Here is the basic backtracking search algorithm:

This is the base algorithm we will continue to modify throughout. Note, we can use AC-3 after line 8 and pass these new modified domains in if we choose.

Now that we have a way to generate valid lineups, we can begin to think about modifying backtracking search

Algorithm 1 Backtracking Search

```
1: procedure BACKTRACK(x, w, DOMAINS)
2:     if x is a complete assignment then
3:         update best and return
4:     choose unassigned variable X_i
5:     order values of Domain_i
6:     for each value v in that order: do
7:         δ = ∏_{f_j ∈ D(x,X_i)} f_j(x ∪ {X_i : v})
8:         if δ = 0 then continue
9:         Backtrack(x ∪ {X_i : v}, wδ, Domains)
```

to pick the ones we think will produce the most points. The first idea we had was to simply take the lineups that make the most out of the available salary. Intuitively, FanDuel gives higher costs to players which they believe will perform well this week, so it might be a good indicator to just pick the lineups that maximize the salary. We modified backtracking search to pick the player with the highest salary next, as opposed to picking based on a standard domain ordering which was decided merely by the order in which the players were loaded.

This approach has some downfalls though. Intuitively, it might not make sense for FanDuel to make the salaries the most accurate estimation of a player's projected points. If this were the case, our best lineups would simply be those that would utilize the entire $60,000 salary cap. To see alleviate this problem, we thought about maximizing an outside projection. With ESPN being the biggest name in sports, we decided to try and modify backtracking search to pick the player with the highest ESPN projection first instead.

Again, after some thought, we realized the issue with this is that although it might be an accurate projection, the ESPN projections have no knowledge of how much a player costs in FanDuel. So even if the player with the most projected ESPN fantasy points has an extremely high salary (i.e. they are overvalued by Fanduel), it will pick them at the possible expense of other strong players. To try and incorporate both pieces of information we have, we can define a new measurement which we will call efficiency. The efficiency of a player (denoted as E) is defined as follows:

$$E = \frac{\text{ESPN projected points}}{\text{FanDuel salary}/1000}$$

This measurement gives us a piece of information which incorporates that higher ESPN projected points is better, and that lower FanDuel salary is also more advantageous. In essence, it gives us how many points per thousand dollars in salary they should score. The reason for the factor of 1000 is to make the numbers easier to understand (in the range of $1 \sim 2$ rather than $.001 \sim .002$.

Now, we have three different measurements which we can use as heuristics to order our variables. However, our algorithm still needs tuning. The most noticeable downfall of the algorithm so far is that it is completely deterministic. At each step, it picks the player with the highest score (score here is FanDuel salary, ESPN projection, or the efficiency we just defined) deterministically. The reason this is an issue goes back to our large domain space. Because of the unfathomably large possible number of lineups possible in a given week, what we have done is limit backtracking search to stop after X (usually 1,000 or 10,000) number of solutions are found. This creates the problem that most of, if not all of these X lineups are very similar, because of the deterministic nature of our algorithm. What is happening is that 5 or 6 variables are chosen, and the last three get shuffled around until we reach our desired number of lineups. There is no inherent issue in this, but it could be a problem if one or more of the players in these lineups perform poorly. A poor performance by even just one player could cause all of the lineups containing this player to lose in any given week.

Our answer to this was to modify our backtracking search algorithm to use an epsilon-greedy approach, inspired by what was taught in class. Our search algorithm is now parametrized by a given probability ϵ. With probability ϵ we pick deterministically as in the algorithm before. With probability $1 - \epsilon$, we pick the next player randomly. To do this, we used a probability distribution that is generated by normalizing whatever the maximization criteria is (again, salary, projection, or efficiency). For example, say we are trying to maximize by ESPN projections, and we have the following three players and their respective projections:

$$\{\text{Player 1} : 10, \text{Player 2} : 20, \text{Player 3} : 30\}$$

In this case, we can normalize their projected scores to get probability weights as follows:

$$\left\{\text{Player 1} : \frac{1}{6}, \text{Player 2} : \frac{2}{6}, \text{Player 3} : \frac{3}{6}\right\}$$

Therefore, when sampling the next player, there is a $\frac{1}{6}$ chance Player 1 gets selected, $\frac{2}{6}$ chance Player 2 gets selected, and $\frac{3}{6}$ chance Player 3 gets selected. We would then use these probabilities in a multinomial distribution to figure out the maximum number of lineups that each player for our current variable should be included in. To do this, we would run the multinomial distribution over N trials, where N was passed in from a value of the previously filled variable and indicated the maximum number of lineups in which that value of the previously filled variable could be involved in. Note that to start backtracking search in this way, we would enter the number of total lineups we wanted to get as our first N.

In order to make sure that this algorithm converged fairly quickly, we also relaxed the constraint that all lineups must have total salary of \$60,000 and instead constrained them to have total salary between \$58,000 and \$60,000. We think this will help make the algorithm more robust, as this should avoid all the lineups being extremely similar. What this also allows us to do is play with different values of ϵ, and try to find the optimal point of being greedy versus picking semi-randomly.

This gave us a greater range of lineups and a corresponding greater range of total fantasy points. However, we were still approaching the problem with a constant ordering of the variables, namely QB, RB1, RB2, WR1, WR2, WR3, TE, K, D. This ordering makes sense from a domain knowledge perspective as it the approximate ordering of which positions affect fantasy points the most. However, ordering the variables this way meant that there would be less diversity between lineups in our quarterbacks, because once a quarterback is picked, it builds all valid lineups with that QB before backtracking search returns and picks a new quarterback. We wanted to see how our system would do if we picked variables in a random ordering as well. In order to implement this, we decided that when choosing the next variable to satisfy, we would take remaining variables to be filled and randomly choose one with equal probability.

3.2.2 Beam Search and Gibbs Sampling

In our previous algorithm, although there was some randomness involved we placed limits on how many lineups each player could be in, leading to not truly random lineups. We decided to implement a new algorithm that incorporated elements of Gibbs sampling, and beam search with K=1. This algorithm does the same thing as the other algorithm in that it generates a multinomial distribution using the normalized efficiencies. However, once it generates this distribution, it picks randomly using the multinomial distribution to pick a single player. As a result, it takes much longer to run this algorithm because it builds each lineup from scratch each time rather than building multiple lineups from each partial lineup. However, our algorithm only has to run once a week, so we were willing to take that penalty. Again, our hope was that the increase in randomness would prevent us from having big failures when the top projected players underperform.

This algorithm also resembles beam search in that it picks one player to go with and builds from there, building up the lineup in a depth first kind of way. While our regular backtracking search also goes depth first, it recurses and tries to exhaust all options before going up a level, while our algorithm, like beam search, discards all but K = 1 options for assigning a variable.

4 Data and Experiments

As we stated in the Modelling section above, we compiled data from rotoguru [2] that included each player and their corresponding salary for Weeks 1-10 of the 2015-2016 NFL season. Then, for each of these weeks, we also matched each player with their ESPN fantasy projections that we found via ESPN's website. Finally, over each week, we calculated the efficiency for each player using their salary and ESPN projection. Therefore, our dataset consisted of 10 weeks of data, and within each week, each data point was composed of a tuple of (player name, salary, projection, efficiency).

We had found this data between Week 9 and Week 10 of this NFL season. Therefore, for Weeks 1-9 of the NFL season, we have the tuples above as well as the actual scores of each of the players in each of those weeks to compare against. This meant that we split our dataset into two parts. The first would be Weeks 1-9, designated as the past dataset, which would be used to check our average win percentage with our top lineups. The second would be Week 10, designated as the future dataset, over which we would calculate average projected value of the optimal lineups found. By doing this, we were able to test our algorithms with varying inputs to quantify their effectiveness at winning a FanDuel competiton as well as their ability to find the best lineups.

4.1 Backtracking Search Results

After setting up the model and configuring the data, we wrote a baseline algorithm to test our model and generate some results. For our baseline approach, we ran basic backtracking search for the Week 9 dataset without any ordering of the players based on salary, projection, or efficiency. Given that the number of lineups that satisfy the constraints (excepting the sum constraint) is on the order of 10^{16}, we cannot compute all lineups that satisfy the constraints, so we instead stopped our search once we had generated 1000 optimal lineups. We then computed the number of fantasy points each of these lineups would have scored for the given week. Here is an example of one such assignment.

Position	Name	Salary
Defense	Atlanta	4900
Kicker	Mason Crosby	5000
QB	Aaron Rodgers	8900
RB1	Fozzy Whittaker	4700
RB2	Rashad Jennings	5500
WR1	Emmanuel Sanders	7600
WR2	Julian Edelman	8000
WR3	Martavis Bryant	6900
TE	Rob Gronkowski	8500

The total salary for this lineup is $60,000, the maximum salary, as expected, and the number of fantasy points the lineup would have scored is 95.45. Our average score across these 1000 lineups was 113.19.According to FanDuels website, the average score required to win in a 50/50 game, where winning means doubling your investment, is 111.21. In this trial, 516 of our 1000 lineups exceeded that score.

Fig 1: Week 10 Average Maximum Projections - Random Ordering

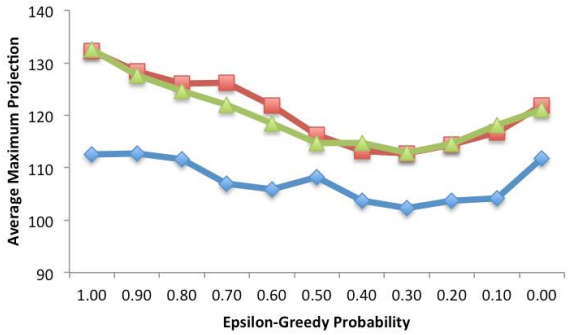


Fig 2: Past 9 Week's Average Win Percentage - Random Order

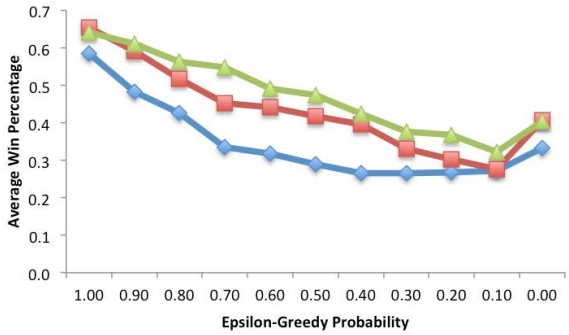


Fig 3: Week 10 Average Maximum Projections - Deterministic Order

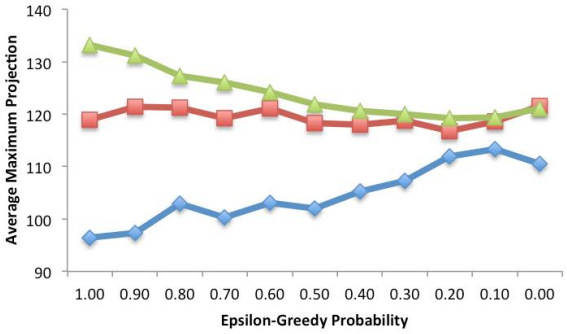
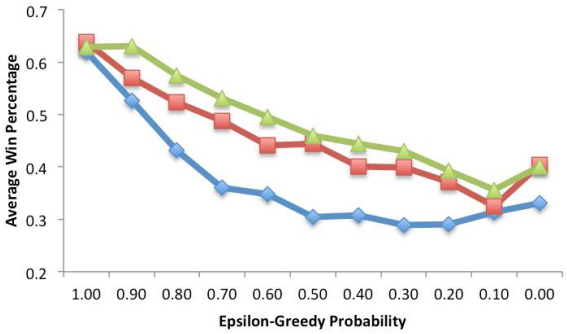


Fig 4: Past 9 Week's Average Win Percentage - Deterministic Order



1. Each data point is an average over 25 different runs, where each run finds at most 10000 possible lineups.
2. The three different lines correspond to choosing players based on maximizing salary (blue), projection (red), or efficiency (green).
3. Figures 1 and 2 have random ordering of the variables, while Figures 3 and 4 have deterministic ordering.

However, after generating these results, we went back and applied this algorithm to a larger data set. The result was that over many of the weeks, our algorithm actually performed very poorly. We knew we had to make some adjustments. In order to improve our results, we wrote a few different functions to reflect the different algorithms, stated above, which we wanted to compare. We then wrote a test suite to run the different algorithms, each with the same set of inputs, so that we could compare them. The inputs were chosen so that each of the algorithms could converge to an answer within a reasonable amount of time.

In order to be efficient, we chose to cap each algorithm at 10,000 optimal lineups, and we relaxed the constraint of the total salary of the lineups to be between $58000 and $60000 to allow the more random algorithms to converge quickly. For the epsilon-greedy algorithm, we chose to have ϵ vary by .1 over the range [0, 1] inclusive.

We chose to run each algorithm with the goal of optimizing each of the three values of salary, projection, and efficiency to get an idea of what the most useful value would be for finding the best lineups. Finally, we chose to run each set of inputs 25 times and to take an average of these 25 runs to be our output in order to reduce some of the variance of the random algorithms. Using these inputs, we ran our test suite and compiled the charts shown above.

You can see that for the most part the trend is similar in the graphs that have standard deterministic ordering and the ones that have random ordering, but there are some differences so we felt it worth to add all four figures.

For each future data point, we calculated the average maximum projection by taking the maximum projection

found in each of the 25 trials and averaging them to understand how well each of the algorithms was able to find good lineups based on varying inputs. For the points of the past data set, we took an average over 25 trials, where in each trial, we would sum the number of wins we would have had over the 9 weeks and divide that by the total number of lineups we would have submitted. Note that we only submitted the top half of the lineups generated, and that we considered a winning lineup to be one whose actual score for that past week was greater than the average score it takes to win a 50/50 game, 111.21.

4.2 Beam Search Results

Variable Ordering	Random	Not Random
Win percentage	0.393410	0.394588
Avg. max projection	125.937488	126.618332

For Beam Search, because it has a very long runtime, we chose to first run tests to maximize efficiency, typically the metric with the best results. We were then going to check what the results were after running on both random and deterministic variable ordering and decide if we wanted to continue to pursue this algorithm. The algorithm's performance ranks around the middle of all of the variations of the other algorithms that we tried. While it's selections placed it below a 50% win percentage, it was interesting to see that it is not entirely unfeasible. The algorithm could possibly be improved by changing the weights the algorithm uses to pick players, but because of the long runtime and the fairly average results, we chose to focus on the backtracking search method as it produced better results.

5 Analysis

Given these results, we can examine why these results occur, and what their interpretation is. First, we can look at the trends that the graphs follow. We can clearly see that the blue curves, obtained by maximizing with respect to salary, perform worse than the others. The red curves, obtained by maximizing with respect to ESPN projections, perform well, but not quite as well as the green, which were obtained by maximizing efficiency.

These results are reasonably expected. As mentioned in the algorithms section above, although FanDuel's salaries are probably good indicators of how a player is going to perform, they will most likely not be great indicators. This is why we see reasonably good scores (averages of right around 111.21) for the blue curve, but performance not quite as good as the others.

It makes sense that when maximizing with respect to the ESPN scores we get better results, with both a higher win percentage and a better average score of around 120. As mentioned above, we take both of these metrics into account when maximizing by efficiency, which also yields good results. We have very similar average score to the ESPN projections, but a slightly higher win percentage.

Between random and deterministic variable ordering, there doesn't seem to be a large difference between the graphs. They exhibit very similar trends, but the random ordering tend to be slightly lower and decline slightly faster as we move epsilon close to 0. This makes sense because we know that the deterministic ordering should be the best ordering of variables from a maximization perspective based on our domain knowledge. Therefore, any random ordering should not perform as well, but averaging over many trials, we see that it gets fairly close to best, deterministic ordering.

The other thing to look at is how these curves change with different values of ϵ. With $\epsilon = 1.0$, we are completely deterministic, and can see that this is actually where all of our algorithms perform the best. As we decrease ϵ, we have steady decline, until we hit $\epsilon = 0.0$ and bounce back up slightly, in both win percentage and average score. This might seem unintuitive at first, but is actually fairly reasonable. As we increase our chance of randomness by decreasing ϵ, we are giving ourselves a higher chance of picking players with lower salary/projection/efficiency. This in turn yields a lower average and therefore lower win percentage.

The natural question that arises is why did we add the epsilon-greedy strategy than? Or why did it not work? Although it seems as if it added no benefit over the strictly deterministic strategy, we still do think this strategy has some merit. This comes from the idea of a robust lineup. As mentioned before, the issue with the deterministic algorithm comes from the fact that many, if not all of, the generated lineups are going to share players. If any of these widely shared players has a bad game, it could seriously impact the performance of all of the lineups that

were generated. In other words, these lineups are not robust at all to poor performances by any of the players. Even though the random lineups might have a lower average score, they will be more robust to unusual performances by any one player.

Another good way to think about these different values of epsilon and their robustness is in terms of risk. A strictly deterministic algorithm will produce many similar lineups. Although it might not be very robust to anomalies, if most of the players perform well, the majority of the lineups will succeed. In other words, the deterministic algorithm has very high risk. If it works, it works well, but if something goes wrong, it could effect the performance of the vast majority of the lineups. The completely random algorithm on the other hand is very robust, since the lineups will share few players to begin with. If any one player has a bad game, it is likely to only effect a small portion of the lineups. This algorithm has low risk. The algorithm will likely perform worse than the deterministic one, but it should also be more consistent. Like most markets, high risk results in high reward, and low risk results in low reward. Depending on the risk preferences of the user, an epsilon could be picked to match.

6 Next Steps

There are three main topics we wish we had time to implement and research further. If given more time, these are areas of extension we think could provide a huge boost of improvement to our overall results, as well as give us a chance to test our system in real time.

6.1 Risk Profiling

The first is risk profiling. Although we mentioned risk and how our epsilon-greedy algorithm plays a role similar to risk, one application of risk to our project we think would be interesting and that would have real world value is that of a risk tolerance. The idea is that some people are very risk averse, while some actively seek being risky. This comes into play into fantasy football as well. A risky player might be willing to gamble on a rising star who may have a breakout game, while a risk averse player might be more inclined to stick to the players who are consistent and reliable.

We want to add a risk tolerance, which we'll call η, as a parameter to our system. η would take on values between [0, 1] inclusive, where $\eta = 0$ indicates a completely risk averse player, while $\eta = 1$ indicates a player actively seeking the riskiest lineups. What this would allow is for different players with different preferences to pass in their desired η. Contained in the data we received from rotoguru [2] is information on each player's "riskiness". This is calculated based on factors such as consistency of performance, as well as previous anomaly performances. We could use this to analyze a given lineups riskiness, and depending on η as parametrized by the user, we could modify the algorithm to select riskier, or less risky lineups.

6.2 Personal Projections

Another topic we think has potential to add to the overall performance of the system is incorporating our own projection analysis using either Regression or Bayesian Networks. This task in itself has enough substance to create an entire project from. Between the feature engineering required to know what pieces of information are actual useful in predicting performance to building and analyzing different algorithms for generating different models, this is no small task. Nonetheless, we think crafting a projection specific to the FanDuel scoring system might be able to improve our results, and given more time would be worth pursuing.

6.3 Real World Application

The last, and perhaps most interesting, would be to actually use our system in real competitions. We have achieved fairly good results and produced lineups that, on average, beat the average required to win a 50/50 game. Given that FanDuel allows users to enter and manage lineups programatically, it is a natural extension to actually enter the lineups we are generating into real competition. That being said, this involves real money (as little as $2 per lineup, up to as much as $1000 per lineup) and a lot of confidence in our system. We think before forging ahead and putting down our own money, we would want to test and optimize more to see if we can make the system more robust and better performing in general. Nonetheless, submitting real lineups with money behind

them is clearly a natural extension and cool application of the system we have built.

We also tried using our algorithm to enter a few lineups in free games just for fun. We generated 3 lineups using the fully greedy version of our epsilon-greedy algorithm with random variable ordering. The three lineups averaged 119.05 points, while all lineups in the leagues we entered averaged 124.28 points. While we did underperform, we did only enter 3 lineups, while our advantage really comes when entering many lineups because with a higher sample size we would be more likely to get closer to the mean. This does not show a flaw in our algorithm as much as a flaw in the projections. The lineups we entered were projected to average 131.9 points. If our lineups had performed as projected, all 3 of our entered lineups would have won in 50/50 leagues.

7 Conclusion

Overall, our system achieved fairly good results. When using our epsilon-greedy algorithm and maximizing by efficiency, our next week's average maximum projection can get as high as slightly above 130, which is nearly 20 points above the average required score to win in a 50/50 league. This corresponds to a win percentage of roughly 65 %. Getting a system, or human for that matter, to achieve even a 50% win percentage is fairly impressive, so we are pleased with our roughly 15% improvement on that.

Along with our results being fairly pleasing, this project provided us with a great chance to learn more about and apply the techniques we had learned in class and in the homework. There is often not enough time to fully explore a topic or issue presented in class, and this project gave us a chance to do so. One good example is the intractability of a CSP of the size we were dealing with. We talked briefly in class about how CSPs are exponential if exact solutions are desired, but never had to deal with this issue in practice on the homeworks. We ran into a very real problem early on about simply not being able to generate all the lineups. We had to do our best with our domain knowledge to modify backtracking search with different heuristics and pruning methods to get our best approximation. This is a great example of how this project allowed us to more intimately engage with the material taught in the course and gave us a more practical perspective of how Artificail Intelligence can be applied to subjects as seemingly far removed as Fantasy Sports.

8 References

References

[1] McDonald, Iain. "Fantasy Football Team Selector." *Life Beyond Fife*. N.p., n.d. Web. 10 Dec. 2015.

[2] "RotoGuru - Daily Blurbs." *RotoGuru - Daily Blurbs*. N.p., n.d. Web. 10 Dec. 2015.

[3] "Projections App - Fantasy Football Analytics" *Fantasy Football Analytics* N.p., n.d. Web. 10 Dec. 2015.

[4] Pisapia, Joe. "Lineup Lookback: Breaking Down Week 1's Sunday Million Winning Lineup" *Fanduel Insider* N.p., n.d. Web. 10 Dec. 2015.