KnapKings: DFS Optimizer

CS701 Final Project Report

Greg Dray
Middlebury College
Middlebury, VT
gdray@middlebury.edu

Dylan Mortimer
Middlebury College
Middlebury, VT
dmortimer@middlebury.edu

ABSTRACT

In our project we focused on the problem of daily fantasy sports (DFS) lineup optimization and the open source accessibility of such optimization. DFS websites are generally dominated by a very small percentage of users who pay a majority of entry fees and account for the majority of winnings. These players presumably often use optimization methods themselves. The ultimate goal of our project was to create a collection of accessible and user friend software tools to provide users looking to play DFS casually a better opportunity for success. In creating our own optimizer, we took a dynamic programming approach, implementing a customized version of the 0-1 Knapsack Algorithm. The algorithm was able to take player projections, player salaries, positions to be filled, and budget as inputs and output a lineup optimized with respect to total projected points. In an attempt to make our optimizer accessible and easy to use for anyone we built a collection of functions into a Python package and uploaded it to pip, called KnapKings. The package includes a driver program that allows users to see their customized daily optimized lineups after choosing parameters like sport, fantasy platform and projection system. Using the driver program, users can also directly influence the outcome of lineups by 'locking' or 'dropping' players, all without needing to interact with the actual code. In order to make our optimizer useful for casual players we aggregated the necessary input data ourselves by scraping projections, prices, and other necessary data from various online resources. In about three weeks of testing the long-term success of entering our optimizer's lineups in single-entry $1 competitions, we found the optimizer appeared to win money in competitions more often than not, leading to long term net-winnings. We are encouraged by these results and are hopeful that we have in fact created a set of tools that can fairly easily allow inexperienced daily fantasy players (e.g. ourselves) to enjoy success while still allowing for user input and control.

Keywords

CS701; LaTeX;

1. INTRODUCTION

Since their rise to popularity, DFS platforms DraftKings and Fanduel have revolutionized the fantasy sports industry. Unlike the traditional fantasy sports format where players join a league and draft a team at the beginning of the year, sticking with that same team all year with the ability to add, drop, and trade players, DraftKings and Fanduel have given players the chance to treat each day as a new season with a chance at instant gratification and monetary prizes. Daily fantasy's cash prizes have enticed many to contribute to the $3 billion plus that the industry takes in as entry fees each year. Not all that surprisingly thanks to the huge potential payouts, DFS are dominated by a minority of 'sharks' who make up a majority of the entry fees and win a disproportionately large percentage of prizes. The dominance of these sharks make it notoriously difficult for casual players to enjoy success on sites like DraftKings and Fanduel.

The inspiration for this project came from wanting to contribute to a package of tools that could level the playing field for casual DFS players to any nontrivial degree. In recent years many have suggested that anyone (the sharks excluded) playing DFS is a 'sucker' and destined to lose in the long run. We've attempted to give the so-called 'suckers' a chance by providing functionality to optimize daily fantasy lineups with respect to total projected fantasy points. This isn't a novel idea in and of itself. Many lineup optimizers already exist for public use. The problem is that the majority of these tools cost a subscription fee or have 'premium features'. In our eyes, 'casual' players shouldn't need to pay $300 a year to have a fighting chance at breaking even. We've created our own lineup optimizer made up of an automated data scraping/aggregating process, user-driven player pool customization functionality, and a dynamic programming 0-1 Knapsack optimizing algorithm. This paper will cover each part of our optimization pipeline in greater detail, discuss the actual success that lineups generated by our optimizer have had in competitions, and evaluate the viability of our optimizer as a useful daily fantasy tool.

2. PROBLEM STATEMENT

Our project aimed to create a set of tools to algorithmically generate winning DFS lineups. We focused on NBA competitions because there were consistently NBA competitions being held to test on. We also focused on single-entry Tournament competitions and single-entry 50/50 competitions. Single-entry Tournament competitions involve each entrant being able to submit one lineup with winnings being awarded based on where each lineup places. In single-entry 50/50 competitions each entrant can submit one lineup and the top 50% of lineups win a certain amount. We tested on only $1 competitions so payouts for 50/50 competitions were always $1.80. We decided to focus on single-entry and $1 competitions due to funding constraints (we didn't get any).

To be valid entries, lineups need to follow three main con-

straints. The sum of salaries of all players in a lineup must be less than or equal to the competition's 'salary cap'. The salary cap is generally $50,000 for DraftKings and $60,000 for Fanduel. All positions must be filled. Lineups cannot contain more than four players from one team. Our project focused on implementing a modified dynamic programming 0-1 Knapsack algorithm to generate the optimal lineup, given these constraints. As input, our algorithm needed to take in the list of players available for each lineup position, each player's salary, and each player's projected fantasy points. We also wanted to be able to generate multiple optimized lineups to assist users in multi-entry competitions.

In order to utilize our algorithm we needed to collect and clean the data necessary to input. We needed to find data sources with player salaries, projections, and eligible positions. We then needed to find a way to scrape or request the data and clean and package it into a format digestible by our implemented knapsack algorithm.

A final but important sub problem was giving users the ability to influence the lineup generating process by 'locking' players into their lineups or excluding them from the player pool fed to our algorithm.

All of the functionality described in this section also needed to be packaged in a user-friendly but malleable form. We decided to aim to build a Python package with functionality to let the user generate lineups with minimal execution of code while also allowing the user to implement the use of alternative data sources or to optimize for different sports and different types of competitions.

3. RELATED WORK

As we mentioned in our introduction, DFS lineup optimizers are already everywhere. One of the most successful and well known is run by Rotogrinders[6]. Rotogrinders is a DFS website that partners with DraftKings and provides DFS players more information about a day's players and matchups than they could possibly fully synthesize in time to set their lineups. Some of this information is free to all and some is privy only to premium members. Rotogrinders' lineup optimizer allows users to choose a sport, slate of games, and DFS platform, allows users to lock and exclude players, set minimum and maximum percentage of generated lineups they want each player to appear in, and generate up to twenty optimized lineups for free. The premium features allow up to 150 lineups to be generated and many other DFS strategies to be employed that were outside of the scope of our project. Rotogrinders served as somewhat of a model for our lineup optimizer, as we often compared our lineups to theirs. Rotogrinders' daily player projections were also one of the data sources we decided to use to feed our own algorithm. Rotogrinders' lineup optimizer is a black box as far as users know. We aimed to achieve as similar functionality as possible while staying open source and transparent.

Eliot Johnson uses a variation of the 0-1 Knapsack algorithm to optimize NFL DraftKings lineups in his Medium article Hacking the Optimal DraftKings Lineup[10], although he only considers a greedy approach. He discusses how incredibly long his runtime will be in taking a pure greedy approach. To mitigate this obstacle he basically adds constraints to the player pool using his best judgment, in the end considering only a fraction of the total players. Our dynamic programming approach is much faster than any greedy approach and we estimate that any advantage in accuracy that a greedy approach has in theory is probably negated when so many constraints have to be put in place to make its runtime reasonable.

4. METHODS

4.1 Web Scraping and Data Processing

The first implementation step was to get player data. Specifically, we needed the following:

- Name: Essential for returning a coherent lineup to the user.
- Salary: how much the player costs to hire. This depends on his recent performance and other proprietary metrics. Set by each fantasy contest website.
- Position: what position the player plays in the game. This depends on the fantasy contest website, as DraftKings allows players to be hired for more than one position (but not be hired twice in the same lineup), but Fanduel only allows players one specific position.
- Projection: expected number of fantasy points the player will score in a specific slate of games. Based on proprietary data/algorithms and no objective measure exists.
- Team: required to catch lineups containing over four players from the same team, a violation of some contest rules.

We used two well-known and respected projection websites for scraping this data: NumberFire[4] (https://www.numberfire.com) and RotoGrinders (https://rotogrinders.com/). While both contain a lot of the same information, each has its own methods for producing player projections. We tested each website's projections against one another to see which were more accurate.

For the scraping process, we used the Python requests[7] library and BeautifulSoup[5]. requests simply retrieves raw webpage data from each projection site. BeautifulSoup parses the html into a format that is easy to search. Through scanning the html of each projection website, we found the names of the tags we were interested in, such as 'cost' or 'player-info–position', and passed these to BeautifulSoup's findAll() function.

Once we had all the relevant data, the next step was to organize it in a way that would it easy to use. BeautifulSoup would return several lists: one containing costs, another containing positions, etc. Since the format of the websites was consistent, we knew that these lists were in parallel, i.e. each index pertains to the same player. So, we iterated through all the lists at the same time, compiling the data into one Player object. While iterating, we sorted each Player object into a certain list by position. This allowed us to return a list of these position-lists, where the order of positions was the same every time. This concludes the scraping and processing portion of our program.

4.2 Lineup Optimization

The most important and complex aspect of our program is the optimization step. We solved this problem using dynamic programming. We got the inspiration for the general version of this problem from an Algorithms textbook[8] and used pseudo-code online as a starting point for our algorithm[1]. The abstracted (and simplified) version of the problem can be construed as follows: You have P players that you need to hire for N positions. You have a budget of $X. Each player has a cost, value and position associated with him. Your goal is maximize the total value of the team while staying within budget.

For our purposes, however, we had additional constraints:

- Each position must be filled
- No more than four players from the same team on a lineup
- Players play multiple positions (but cannot be hired twice), DraftKings contests only
- Must hire two players for certain positions, FanDuel only

While the simplified problem is relatively straightforward to solve with dynamic programming, our solution become more complicated once we factored in our constraints. As such, we will not provide pseudo-code here, but we will provide a recurrence relation and description. First, let us define a few mathematical terms that serve to elegantly split this problem into smaller sub-problems:

- $V[i, x]$: value of the optimized lineup when only considering positions i through N with a budget of $x
- $S(n, x)$: subset of players who play position n and whose cost <= $x
- p: player with attributes name, cost, projected points, position, team

Given these constraints, the problem can be split into subproblems by considering only certain positions with a certain amount of money left to spend. This allows for the algorithm to only solve each sub-problem once, increasing the computational efficiency. It is guided by the following recurrence relation:

$$v[i,x] = \begin{cases} \max\limits_{p \in S(N,x)} p.value & if\ i = N \\ \max\limits_{p \in S(i,x)} (p.value + v[i+1, x - p.cost]) & if\ i < N \end{cases}$$

The algorithm works off of a Numpy[3] table V containing the solution to V at each cell, with i, x as the row, column. The algorithm begins by filling in the bottom row of the table where i=N left to right, as this row pertains to the top line of the recurrence, which can be filled out immediately. This line ensures that when only considering the final position N, the algorithm hires the most valuable player that can be afforded. Once the algorithm fills in the bottom row, it moves up by one row such that i=N-1. The algorithm now must implement the bottom line of the recurrence, which guarantees that the algorithm hires the player that results in the highest value total overall lineup. The algorithm hires the player whose value plus the value of the best possible lineup with the remaining positions/money is the greatest.

Once the algorithm fills out all the cells in the table, it must recreate the solution. Each time the algorithm decided it was optimal to hire a player in a given cell in V, it saved that Player object in a parallel table, Who. Thus, to find the optimal lineup, the algorithm goes Who[N, X] and hires that player, call him player A. Then it hires the player in cell Who[N-1, X-A.cost], and continues this process iteratively until players for all positions have been hired.

Unfortunately, the algorithm is complicated by the additional constraints listed above. The most costly are constraints 2 and 3, which require that the algorithm recreate the current solution every time it considers a new cell in V, to ensure it is not hiring the same player twice or hiring a fifth player on the same team. The recreation of the solution is simple to do once, but it becomes costly to perform this operation every time. Luckily, there is one aspect of fantasy contest rules which we were able to take advantage of and resulted in noticeably faster runtime: each player's salary must be a multiple of 100. This allowed us to divide all salaries by 100, and thus to create table V with size (N, $\frac{X}{100}$).

Runtime: The final algorithm has runtime of $O(N^3 * \frac{X}{100})$. The algorithm must fill out all $N\frac{X}{100}$ cells in table V and Who, and at each cell, it takes worst case N^2 to recreate the solution. While the runtime is inefficient with respect to N, N will never be higher than the 9 positions DraftKings requires. In practice, it returns the optimal lineup or a near-optimal lineup in a few seconds.

4.3 User Interaction

With the data obtained and an optimization algorithm implemented, the last step was to allow the program to be flexible so that users could utilize it but retain control over many of its functions. We wanted our algorithm to be used for at least two sports, different fantasy contest websites and different projection websites. Finally, we wanted the user to be able to individually select players he/she wanted to ensure were included or excluded in the resultant lineup. Each of these controls added its own complications to the implementation, which are listed in greater detail in the results section.

Once the functionality was implemented for changing these parameters, we needed to make a decision on how users would interact with the program. Due to time constraints, we knew that an app of any type was not possible. We decided that the second best would be a command line interface. We designed an easy to use interface that asked the user which options they would like to select for each of the parameters listed. The interface includes a significant amount of error checking to ensure the user does not receive a error message they cannot interpret.

5. RESULTS

5.1 Package Methods

Our package includes fourteen functions that contribute to either the data aggregation process, the optimization process, or are wrapper functions for simplified use. There are scraping functions that scrape all necessary player data from numberfire.com for Fanduel basketball and baseball competitions and there are functions that scrape all necessary player data from rotogrinders.com for Fanduel and DraftKings basketball competitions. In the case of the DraftKings scraping functions, the data outputted is ready to be in-

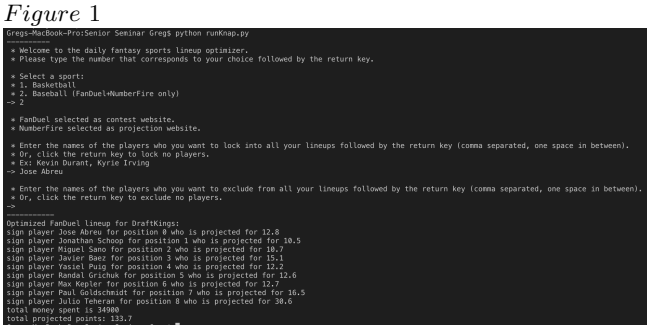
putted to our optimize function. In the case of the Fanduel scraping functions, the data outputted must be slightly altered in format by the formatPlayersFD() function and is then ready to be input for our optimize function. The reason we do not have scraping functionality for both projection systems with all platforms and sports is that we ran into many barriers in the html of the webpages we were trying to scrape from. Many of these pages had custom html elements that we could not scrape from or alter to display the desired information.

We have one function that does our dynamic programming optimization called optimize(). This function takes in a list of lists of Player objects with attributes: name, value, cost, team, id, and it takes in a lineup's budget/100 (so 600 if $60,000). This function returns the filled out dynamic programming table from which our optimized lineup can be derived using the getLineup() function. This function has powerful standalone capability that could be applied to many more optimization problems than just DFS lineups. We also have a function called multipleOptimize() that will return up to 20 unique lineups using altered code from the optimize() function. This functionality is important for ente

5.2 Driver Program

We recognized that many potential users might have little to no coding experience. Because of this possibility we developed a driver program that can be run using solely command-line interaction. This program, runKnap.py, prompts the user on the sport they would like to generate a lineup for. After the user selects a sport, they are queried on whether they would like to lock or drop any players from the optimizer's player pool. Following this, if the user chooses baseball, the interaction is over, since our package only provides functionality for generating Fanduel baseball lineups using Numberfire projections. If the user instead chooses basketball, the user is then prompted for their desired fantasy platform, their desired projection system, and the number of lineups they would like generated. Upon completion of the user interaction, the program prints all desired lineups to the terminal, listing their projected fantasy points. A summary of each of these parameters and what actions they require of the program are described in Figure 2. A picture of the command-line interface in action can also be found in Figure 1 below.

Figure 1



5.3 Competition Results

To test the effectiveness of our optimized lineups we entered $1 single entry tournament competitions on DraftKings and Fanduel as well as 50/50 (top 50% win) competitions on Fanduel daily. Since mid-April, our net profits have been $10.84 on DraftKings, and $18.74 on Fanduel. As you can see in Figures 3 and 4, in Fanduel competitions, we found using rotogrinders' projections to be about 5x as effective as numberfire's projections. We also found tournament competitions to be about 5x as fruitful as 50/50 competitions. We were never able to implement functionality for NumberFire projections for DraftKings competitions so we do not know their effectiveness for them, but would expect their success relative to Rotogrinders' to be similar to Fanduel competitions. We were also unfortunately unable to test baseball lineups or multiple lineup competitions (in either sport) due to implementing the functionality late in the project.

Figure 2

Parameter	**Options**	**Complications**
Sport	Basketball Baseball	Additional data scraping required
Contest website	FanDuel DraftKings	Different contest rules, overlapping players, etc
Projection website	NumberFire Rotogrinders	Additional data scraping required
Lock player(s)	All players	Hire locked players before passing to algorithm, checking for user error, e.g. nonexistent player
Drop player(s)	All players	Remove player from player list before passing to algorithm, checking for user error, etc.
No. Lineups	1-20	Add randomization to optimization algorithm to create additional lineups

Figure 3

		Projection System	
	Total	Rotogrinders	Numberfire
Fanduel	$18.74	$15.58	$3.20
DraftKings	$10.84	$10.84	N/A

Figure 4

		Competition Type	
	Total	Tournament	50/50
Fanduel	$18.74	$15.38	$3.40
DraftKings	$10.84	$10.84	N/A

6. DISCUSSION

With customization and ease-of-use in mind, we built a Python package that users can use to create strong daily fantasy lineups. With numerous options to choose from, we hope we have designed something that people with a variety of interests can gain enjoyment from.

One important insight our package provides is that DFS are not purely luck-based[2]. The fact that an algorithm

nets positive on average suggests that player projections are better than intuition and can be used to win. That said, our winning rates of 53% for Fanduel and 58% for DraftKings show that projections are not incredibly reliable, as we won only slightly over half the time. It requires a long-term strategy to win using projections, as they are not highly likely to win individual contests. Furthermore, our project demonstrates that DFS can be made to be more accessible. There certainly exist many sports fans who do not have the means to pay for a premium optimizer like the one Rotogrinders provides. Our project aimed to bridge this gap by creating a tool that performs many of the functions of an expensive optimizer but costs nothing.

There are several parts of our project that we are proud of. First, we are happy that with only the knowledge of how to run a Python program, anybody could take advantage of our optimizer, thanks to the command line interface ew designed. The vast user base this entails makes us hope that we can make DFS less intimidating to some players. Next, we believe the optimization process we built is highly accurate - when checked against the Rotogrinders optimizer, the lineups were usually the exact same. Finally, we are excited that an extremely simple (in concept, not implementation) optimization strategy resulted in net winnings over time. This gives us faith in the merits of DFS as an institution and inspires us to improve our algorithm further to improve our winning percentages.

Our project does have a few limitations. First, we recognize that an interface that does not interact directly with the terminal or Python code would have opened the door to many more users. Unfortunately we lacked the time to build an app. Second, our optimization function is not completely guaranteed to give the best lineup. The simplified version of the problem outlined in section 4.2 guarantee the optimal solution, but to deal with the additional constraints, in certain edge cases the solution may be sub-optimal. However, the solutions have never been significantly lower in projected points than the Rotogrinders optimizer. Third, we were unable to test the code that checks that no more than four players from the same team have been hired. We thus commented it out to be safe, so this constraint is not fully implemented. Lastly, more error checking/catching would be useful. We included as much as possible in the time allowed, but in some edge cases (e.g. no games that day) the program will give an undesired/unclear output.

These limitations provide insight on where the program might be improved in the future. The biggest change would be to completely change the interface from the command line to an app, whether a web-app or for mobile devices. We believe this would result in much more use and further our goal of making DFS more accessible. A second possible improvement would be increase the number of lineups the program will output. This would open the door to a different contest entry strategy: instead of entering a few lineups, enter many (on the order of 100) that are uncorrelated with each other. This way, it becomes more likely that at least one lineup will place and win a lot more. This strategy comes from an MIT paper[9]. Other minor changes would be to ensure all fantasy contest constraints are met and improve error catching.

7. CONCLUSION

Our programs gives users customizable DFS lineups that result in positive winnings over the long term. Granted, our testing has been limited to low-stakes competitions with limited payouts, but this has given us hope that we have created a tool that can facilitate the success of casual DFS players who cannot build their own optimization tools or afford to spend a significant amount of money on a premium optimizer/entry fees. We are optimistic that the functionality we provide allows enough human input to allows users to make the lineups their own.

8. ACKNOWLEDGMENTS

We would like to thank Professor Ananya Christman who took time during her sabbatical to talk through our algorithm via Skype. We also would like the thank Professor Grant for advice throughout the semester.

9. REFERENCES

[1] 15-12 signing free-agent baseball players. *CLRS Solutions*.
[2] Are daily fantasy sports gambling? *IOS Press*.
[3] Beautiful soup documentation. *Crummy*.
[4] Numberfire. *NumberFire*.
[5] Numpy developer guide. *SciPy*.
[6] Rotogrinders. *RotoGrinders*.
[7] The user guide. *Python-Requests*.
[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*, 3 ed.
[9] Hunter, D. S., Vielma, J. P., and Zaman, T. Picking winners in daily fantasy sports using integer programming.
[10] Johnson, E. Hacking the optimal draftkings lineup. *Medium* (Jan 2019).