**Summary**

The goal of my project was to create a program that would choose a fantasy hockey lineup. Each player scores points based on a nightly performance in a game, and is assigned a ‘salary’ which is the value of the player. A lineup consists of 9 players who’s combined salary is less than 55,000. Initially this seemed like an ideal candidate for a constraint satisfaction algorithm. The constraints were, 1 player per lineup slot, salary less than 55000. While constraint satisfaction algorithms excel at finding a solution to the problem, they are not as good at finding the best solution.

With the goal state being to find the highest scoring lineup each day a constraint satisfaction algorithm wasn’t the best option. (Constraint satisfaction with backtracking may have worked, more on that later) A Depth First Search (DFS) that returns all solutions was decided on, instead of a constraint satisfaction algorithm. However, the state space for a DFS was very large, 9 positions with possibly 100+ players at each position. Using recursive function calls and looping through each player at a position took too long to run with 20 players per position let along 100’s. So reducing the number of players per position to decrease the size of the state space was the only obvious solution.

Because of this the core theme of the project quickly turned from a full blow AI search algorithm, to a data manipulation, statistical averaging exercise run time optimization. Initially the search had various inefficiencies that mostly came from sub-looping through a given set while the main search looped recursively over all position lists. This added huge overhead, but was necessary to constrain the search. Several of the superfluous counts were eliminated and the number of cons cell visits decrease dramatically.

After run time optimization the DFS could return values in a reasonable time for 8-10 players per-position. At approximately 20 players per position the search ran for 10+ hours and never returned. So the search would have to run on less than 10-12 players per position. In order to pear the list of player per position down to this level may less than scientific methods were employed. For the sake of the search algorithm they made little to no statistical difference as the input models were “dumb” and didn’t take anything other than position, salary and score into account.

In order to still find decent models but have a run time that was acceptable the search was broken into several “layers” by randomly selecting 10 players for each position and searching for a best lineup from those players. This random search was done enough times that all players were incorporated into the search, although not all in the same search. This produced some results…

**Overall Goal of Project**

The overall goal of my project was to create a fantasy hockey lineup generator. Fantasy hockey is played on a daily basis, but not every team plays every night, so it is important to find the best players that are playing that night. There are many iterations of fantasy hockey for my purposes a daily fantasy league, much like you see advertised everywhere, fit the bill nicely. They played every day, and you didn’t have a set lineup of players that you ‘drafted’ at the beginning of the season. Every day is a new ‘draft’ and a new chance to select from all available players. In hindsight, project would have been much easier (and will work in its given state) for a set team of players that someone drafted at the beginning of the season.

Each player scores points based on several factors of play (which can change based on the league). There are anywhere from 2 to 2000 (or more) people who submit lineups. The lineup with the highest score total wins. Some lineups have multiple places based on how many people entered.

The rules for a lineup in the league that I targeted are:

1. A lineup is 9 players, 1 goalie and 2 each from defensemen, right wings, left wings and centers.
2. Each player is assigned a salary by the system, a lineups total salary must be below 55000.

The code I wrote for the project uses DFS to search through all of the players that are active on a given day. There are 2 main constraints that the project looks for; a full lineup having less than the maximum, 55k salary, and the highest total score, found by combining the average scores of all the players in the lineup.

Given these 2 constraints the DFS could be searching though huge state spaces to find solutions, so there were some minor constraints added when aggregating the player data. The first is that a player must have played in more than 20 games, otherwise their scoring average is skewed. The second minor constraint is that the play must sure to play that night, not possibly out due to injury.

Using this data the program is supposed to create a list of all the highest scoring lineups below the maximum salary. The reason it creates a list is so the user can choose one of several valid lineups because there is no way to know for sure which players will play the best and choosing from a list of many lineups allows some human choice to come into play.

**High-level Code Description**

The project code is split into 2 modules. For no reason other than a logical relationship between what the parts of the models do.

The first module reads in a csv file of all the players that are on teams that play in a given night. As it reads in the file it puts the stats of each player into a class that holds each stat in a separate slot to make it accessible. A list of all the data, now in player classes is stored and then parsed through to generate lists of player for each position. While this ordering by position is happening the minor constraints are applied and the players who are injured or have not played enough games are dropped.

Due to the very long processing time of the DFS and additional feature was added to module one. It accepts a number to compare to 3 modulo the players position in the list. This allowed the list of players at each position to be ‘randomly’ cut into thirds. Several smaller runs allowed the DFS to complete and made this addition crucial.

Module 2 takes the lists of players by position from module 1, sorts the lists and puts them into a 2 dimensional array, 9X~10. The outer array contains the 9 arrays that must be looped through to find a lineup. There are some helper functions that calculate the score and salary of each lineup, as well as check that a player is not in a lineup yet. The DFS is run recursively by looping through the list of players at a position and then calling the function again to go to the next position.

The code when through many lifecycles to get to its current state, nearly all to try and improve run time. First, the arrays of players by position were sorted in reverse order of salary. The initial though was that, since the DFS is running recursively that returning once the salary was known too high for 1 player would help to reduce the run time. (This did not turn out to be the case). The second lifecycle was to reduce the sub loops run in the search by calling the helper functions less often. Initially the helper functions (which all loop through a given lineup), were called 6 times for each iteration of the search. Because the search is loop through millions of times this added up, the helper function calls were reduced to only run 3 times per loop.

Even with these improvements the runtime for a list of 25-30 players per position was so long the code never returned. So looking for a way to shrink the data set I decided to randomly take only a part of the data for each run, and then combine the results for each run together. This is accomplished by running the DFS 3 times, each with a random 1/3 of the players and then storing all of the found lineups in the same global variable. The global variable can then be searched by the user to find the lineup they like best.

**Quick Run Time Analysis**

The runtimes I got for the DFS seemed to be fairly random. They held to a general pattern but seemingly for no reason some runtimes would be much much higher than others. A list of the runtimes, and cons cells is shown below for comparison.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Time cpu (sec) | Time gc | Time Real | Cons cells |
| 3 players/9 positions (sorted pruning) | 3.656 | 1.03 | 4.688 | 16,191,327 |
| 3 players/ 9 positions (no pruning) | 2.25 | .656 | 2.968 | 10,242,685 |
| 10 players/9 positions (fast load) | 784.312 | 355.14 | 1139 | 867,244,020 |
| 5-10 players/9 positions (fast load, optimized) | 49.75 | 23.75 | 74 | 60,337,170 |
| 20-30 players/9 positions | DNF | DNF | DNF | DNF |

**Instructions for Use**

The websites that host the leagues that I used provide a csv of all the players that are playing that day. I have included some of those in the zip.

To use the program define the variable full-file as the file name of a csv

(setq full-file “/path/name”)

Then simply load ProjectM1.lisp followed by loading projectM2.lisp. The code will run the DFS 3 times and generate a list of all the lineups in the global variable \*legal-lineup\* which can then be searched and worked with to find the lineup the user likes best.

WARNING: Due to the fact that the code does a random sampling to get each of the 1/3 sections of players, it is possible that there will not be any players in a given position array. This will cause the code to generate errors.

The code will output the following during runtime:

“STARTING…”

0 (the run number)

;Loading a file

4

6

8

7

RUNTIME STATISTICS

… (this row of numbers is the count of the number of players in each position array, to identify the error from above)

Once the first run is done it will give you some run time statistics from the TIME function, and then start the next run. Each run can take between 10 and 90 seconds (or more at times).

**What I learned**

I learned a lot about lisp processing and also data processing from working on this project. Below are some of the key points that were major hurdles (not all of them were crossed) that stuck with me after this project.

Data manipulation is difficult. The data I used was nicely formatted in a csv. This made it fairly straight forward to read the data in. However there were still enough edge cases in the data that the code had to very specific to these files. Especially because the code is designed to read in formatted player data, it is very likely not extensible to other projects.

And reading the data in was the easy part. Once the data is in the system working with it was much more difficult than anticipated. Again, it was fairly straight forward to create a search algorithm. I proved this was too slow for real world use and that there had to be concessions made to get performance. The aggregation of data diluted the results to a point where it was hard to say that the best lineup was found. As a matter of fact running different mutations of the data I can say that the current state of the project didn’t find the best lineup for one test set.

I learned that performance doesn’t just mean how fast a program can run, sometimes it means can it finish running at all. Runs with what I assumed were fairly small numbers of players (20-30) ran for 12+ hours before I stopped them. This highlighted for me the need to write good, clean code. More planning and design work in the beginning could save time testing and refactoring. I didn’t think it would matter how many loops ran through or how many members were in a data set. I realized that tighter code is much better, even in todays world of very fast home computers.

I also strengthened my knowledge in lisp syntax, data structures and debugging. While these are things we covered in class it always seems like doing real work on a project is the best way to learn a language.

**Project Improvements**

I am not happy with my project overall. I do not feel that it really accomplishes the goal I set out with of finding the best lineup for a fantasy hockey league. There are several things that I started trying or wanted to try to improve the project. But in the end I ran out of time and decided that producing some result was better than nothing.

I do not think lisp is slow. I think the implementation that I used for the search was very poor. I have a hunch that using loops was the slow way to do this project and it would have been faster using lambda and mapcar. Although I don’t know if that would have improved performance to a point that would yield results.

This problem is not unique in the field of computer science. DFS is a very intensive search and there are a variety of algorithms that have made improvements on it. Constraint satisfaction with backtracking would have probably worked, but I abandoned that early on as it yielded results that were bad solutions (low scoring lineups). A heuristic in the data might have helped speed the search by reducing the state space. I thought I was creating that by sorting the player and stopping the search at a non-productive point. However this didn’t speed up the search.

**Conclusion**

Ultimately I have to think of my project as partially implemented. While it produces an answer set, it doesn’t produce the answer set that I set out hoping to get. Realizing this changed the shape of my project, from a solution to a problem, into more of a research by trial and error project. Not all projects finish successfully, and some projects evolve into something else when realizations about what the code is doing change perceptions about the problem that can be solved.

Even though the initial problem was not solved does not mean there is some value in the project. Producing smaller data sets and comingling the results makes me think that it would be easy to run the same search across a combined list of all players in the results list. This would create a multi-layered search, similar to a very simple machine learning algorithm. It seems like faced with similar problems to what I am facing lead to many of the approaches that we currently see in machine learning and data science. Having more data than can be easily processed is huge field right now and completing this project highlighted for me some of the issues that surround that field.