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…

Wanted to use constraint satisfaction. Turns out I need the best solution, not just any solution. Could have used more logic to backtrack maybe… Constraint satisfaction finds only A SOLUTION not necessarily a good solution. No what I want.

Removing bad players, because the algorithm is “dumb” doenst matter every player with the same score is equally viable in a situation, especially because algorithm looks for best score. So removing all scores of the same time and then shuffling players later.

Went with a DFS, only 9 layers deep. But that is still a huge state space 9 X 100 per position.

Tried to optimize, sorted lists and pruned after max value found. Turned out not to help.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |

Search through those lists

Originally had 6 function calls that loop through each lineup in each section. That adds up! (upto 9 cycles X 6 calls X MILLIONS of recursions). Use let statement to only calculate 3 and then compare 4 times.

Runs in reasonable time with 5-8 players per position. Make a “random” getter pull 1 out of 4/5 players from each position. Then run each pass through and compare results.

Usignd for x upto I loop will make it easy to pull. This is a very poor mans neural network, or machine learning.