221 Project Progress Report

Alexei Bastidas (alexeib@stanford.edu) & Ingerid Fosli (ifosli@stanford.edu)

Model: CSP, Algorithm: greedy backtracking

Our current model of the problem is as a constraint satisfaction problem. It uses exact predictions of fantasy points to maximize the total expected points given the constraints. The initial variables are the positions that need to be filled on a lineup, with the correct multiplicity. For example, the variables `('RB', 0)` and `('RB', 1)` represent the two running backs in the lineup. (Additional variables are created to add constraints; more on these later.)

Constraints/factors

No repeating players

Binary factor between every pair of two different variables to ensure that each player only occurs once on the lineup. This factor should only be needed explicitly for variables referring to the same position, as a player should only have one starting position. Currently, it's implemented for every pair of variables, since there are only 36 pairs, for simplicity in code, and just in case a player can have multiple starting positions.

Weights for expected fantasy points

Unary factor for each variable, which returns the expected number of points earned per player. Since the weights are combined multiplicatively to determine best solution, and the total number of fantasy points is the sum, this isn't a perfect metric for determining the best solution, but it seems to do okay. To maximize the sum instead, we can let the weight of each player be e^(expected points). That way, the total weight of a solution is e^(sum of points), so the best solution to the CSP will predict the highest number of fantasy points. It's worth checking if this will make a noticeable difference, but we'd rather try to optimize the weights first. We particularly want to take the likelihood/stability of the expected fantasy points of a player into account, to take risk into account.

Salary constraint

This uses a sum variable, as in the scheduling problem, to set a maximum on the total salary used on players. The main difficulty in implementing this is that the salary range is quite big; max salary is 60 000. I've only seen salaries be round 100s, but that still gives a range of 600 values. There are 8 variables to add up in the sum; each of which is a pair of values. Without any additional optimizations, there are 600^2 possibilities in the domain of each variable; unsurprisingly, my computer crashed when I tried to run it.

Currently, to make the code run, we use the 1000s to estimate the salary of each person, reducing the max salary to 60, and the maximum size of the domain of the extra sum variables to 60^2. Additionally, the domain of the sum variables is made smaller, using that the first coordinate must be smaller than the second, and the assumption that each player has a salary between 4000 and 10 000. From here, we use the `get_sum_variable` that we wrote for Scheduling.

To make this constraint better, we want to optimize the way ranges of numbers are stored in our code. Currently, we're using the CSP that scheduling was built on, where a domain is specified by listing all the values a variable can take on. To keep the sum more accurate, we want a larger domain than we currently have. This might be doable just by limiting the domain to values that are directly achievable, instead of generating all values that might be achievable - we've done some of this already. Or we might need to change the way the domain is stored, which will mean we need different ways of storing a domain of strings vs ints.

Team constraint

The fanduel lineup allows at most 4 players from each team in the lineup. This constraint is not yet implemented, but can be implemented using a sum variable for each team, with a max sum of 4. This should be fine in memory; there are only 32 teams in the NFL, and the range of allowed values is very small.

Algorithm

The algorithm used is a modified version of the backtracking algorithm from the Scheduling assignment. We take advantage of having the player information sorted by expected fantasy points, and use a greedy backtracking algorithm, which returns the first solution it finds, instead of checking all solutions. More work is needed to make sure the algorithm always returns the best solution, or at least returns a very good solution with high probability. It seems to be true, at least with our current implementations of constraints, but this will probably change particularly if we change the weights of players (to account for stability/expected range of points instead of expected value, or maximizing sum instead of product of individual weights)

Analyzing this model/algorithm

The model tries to solve the problem of getting the highest number of fantasy points. However, the actual problem we want to solve is to get a lineup with a good enough ranking. Our model does not take the choices of other players into account, and it also doesn't currently take the risk into account. So even if we wanted to use it for a low risk competition, where anyone in the top half of the bets earn money, our model would just gamble on what it thought best, instead of taking advantage of only trying to place in the top half.

Example

We ran the algorithm on week 9 of the current season, using our predictions from the 229 part of the project. We start by loading the information into a database, where each position is a dict with player names as keys and tuples of team, points, and salary as values.

The first step is to construct the CSP. The variables of `(position, index)` are added for each position, with index `0` if there's only one player of that position, with domains equal to the player names we have data on for this position and week. The constraints are added as implemented above, which adds additional sum variables to the graph.

Then we run `csp.backtrack()` to solve the CSP. Backtrack finds the following assignment:

```
Found assignment:  40843930040.7 {('TE', 0): 'Jimmy Graham', ('WR', 2): 'Larry
Fitzgerald', ('salary', ('WR', 0)): 7, 'sumtotal salary': 60, ('salary',
('Def', 0)): 5, "sumtotal salary('salary', ('PK', 0))": (51, 55), ('RB', 1):
'Ezekiel Elliott', "sumtotal salary('salary', ('TE', 0))": (37, 43), ('salary',
('TE', 0)): 6, ('WR', 1): 'Alshon Jeffery', ('salary', ('WR', 2)): 7, "sumtotal
salary('salary', ('WR', 1))": (23, 30), "sumtotal salary('salary', ('WR', 0))":
(16, 23), ('salary', ('RB', 1)): 8, ('WR', 0): 'Jordy Nelson', ('QB', 0): 'Tom
Brady', ('PK', 0): 'Cairo Santos', "sumtotal salary('salary', ('QB', 0))": (43,
51), ('Def', 0): 'ARI', "sumtotal salary('salary', ('WR', 2))": (30, 37),
('salary', ('QB', 0)): 8, ('salary', ('PK', 0)): 4, "sumtotal salary('salary',
('Def', 0))": (55, 60), "sumtotal salary('salary', ('RB', 0))": (0, 8),
"sumtotal salary('salary', ('RB', 1))": (8, 16), ('salary', ('WR', 1)): 7,
('salary', ('RB', 0)): 8, ('RB', 0): 'Melvin Gordon'}
```

where the additional variables are used for the sum; `('salary', (pos, index))` were passed to `get_sum_variable` and is just the salary of the selected player, `"sumtotal salary('salary', (pos, index))"` are the extra variables for calculating `sumtotal salary`, the total salary.

Then, by looking up the chosen players in the database, we print:

```
Jimmy Graham       TE     ('SEA', 17.662052158468704, 6600)
Larry Fitzgerald   WR     ('ARI', 12.064124909237618, 7200)
Ezekiel Elliott    RB     ('DAL', 16.584006072880793, 8900)
Alshon Jeffery     WR     ('CHI', 12.654198230106374, 7200)
Jordy Nelson       WR     ('GB', 14.822500691213566, 7700)
Tom Brady          QB     ('NE', 22.771763835184906, 8300)
Cairo Santos       PK     ('KC', 10.850082845242287, 4800)
ARI                Def    ('ARI', 12.712420842897982, 5200)
Melvin Gordon      RB     ('SD', 19.619457191004194, 8300)
total points expected:  139.740606776
total salary used:  64200
```

in a more readable format for the actual results. Notice that this does go above the salary cap, which is expected since we used rounding to do the salary constraint.

Next Iteration - Markov Decision Process

Currently, we are using Random Forest Regression in order to compute our predictions for a player's production, as part of our CS229 project. We will be testing a multi-class classification system next, such that we can generate individual probabilities' for a player's potential output. For example, given a player such as Jimmy Graham, we can, using our classifier, predict the probability that he will get 0-5, 5-10, 10-15, 15-20, or 20+ points given the previous 3 games, and the upcoming game's opposing team. With these ranges, along with the likelihood associated, we can convert our model from a salary based, greedy CSP, into a Markov Decision Process, wherein the the goal is to maximize the expected production, subject to the constraints, by leveraging the underlying probability for point production for any given player.

In this model, we note a state as being: ([Current Salary Cost], [Current Expected Production],[Current Position Group Counts and Names], [Players Remaining]). In order to enforce Team and Positional Constraints, we intend to implement the [Players Remaining] as a Dictionary, where the key-value pairs are Position-Group: Dictionary of Players. Once a position group is filled, we remove the position group dictionary from the Players Remaining Dictionary in the state. Similarly, once a player is chosen, he is removed from the dictionary. In order to keep track of the positions, the state will maintain an array of counts.

An action involves choosing a player. Upon choosing a player, the cost is always equal to the player's salary. For each player, however, we choose one of the 5 possible fantasy production outcomes with probabilities given by the multi-class classification model. If the salary cost is exceeded, the reward is 0. If all positions are successfully filled, the reward is equal to the current expected production.

Given this definition of the problem, we can leverage techniques as seen in the Blackjack such as Value Iteration, and QLearning. Particularly with QLearning, we can have the model 'play the game' repeatedly to find the line-up with the highest potential for points. By performing the algorithm in this way, we implicitly account for playing against others as the goal becomes to win via maximizing the production, and repeated runs of the QLearning algorithm act as simulations of other potential rosters that could be selected by other players.

We expect this framework to perform better than the current CSP in terms of providing line-ups with more predictive power. Moreover, in order to account for the size of the search space we will likely trim the players available in each position by sorting them in descending order utilizing the expectation of their predicted points, and only utilizing the top ⅓ of players (this would reduce the number of players available to 10-20 per position).