Making Sense of the Madness: A Bilevel Programming Approach to Bracket Generation

1. Introduction

March Madness, the annual NCAA basketball tournament to determine the college basketball champion, is a single elimination tournament with 64 teams and 63 games (after the play-in games). It is one of the largest single elimination tournaments in sports. Given the size of this tournament, it is high likely that favorites will lose somewhere along the way. That's where the "madness" in March madness comes from and it's this "madness" that captivates sports enthusiasts and casual observers alike.

Over the years it has become tradition to guess the outcome of the tournament before the tournament starts. How captivated are sports enthusiasts and casual observers? Statista estimates that in 2023, 56.3 million adults in the U.S filled out a March Madness bracket [1]. The American Gaming Association estimates that in 2023, 70 million brackets were filled out in the U.S [2].

Correctly selecting the outcome of the tournament is no easy task given the sheer combinatorics of the problem. There are roughly 9.2 quintillion or ~9.2×10^{18} possible brackets. If the 56.3 million adults coordinated with each other to fill out enough unique brackets to reach 9.2 quintillion, each adult would have to fill out around 163 billion brackets. It would take roughly 5200 years for each adult to fill out their brackets if each of the 56.3 million adults could fill out 1 bracket per second and did so for the entirety of the 5200 years. This is a challenging problem.

This has not deterred statisticians and data scientists from attempting to find some order in the madness. Statisticians such as FiveThirtyEight and Ken Pomeroy may use box score statistics, adjusted box score statistics, compositions of other computer ratings, travel distance, and many other features to predict the outcome of a single game [3]. Then the probability a team makes any round is the probability they make the previous round multiplied by the probability they win against any team they could face in that round weighted by the likelihood that the other team makes it to that round. Given how sophisticated these methods are, they tend to perform well, outperforming median scores quite often. However, once again, the combinatorics make it such that if a person follows the most probable outcome (now referred to as chalk) as defined by these techniques, you may still not win your bracket challenge pool. The most likely scenario is still very unlikely to occur.

So, is this the best that can be done in a predictable fashion? Is there any way to deviate from chalk predictably and still outperform chalk? There is one popular method introduced by Bradley Carlin and co-authors in their 2005 paper, *Identifying and Evaluating Contrarian Strategies for NCAA Tournament Pools* [4]. The general methodology is to account for group size and behavior. After all, the end goal might be to win a pool and winning a pool means beating the competition. If there are many people in a group selecting the favorite to win the championship, it may not be advantageous to also select the same champion. For example, if a person selects the favorite that most of the pool selected and that team does win the championship, then it is required that the person beats most of the group on all prior rounds as well. However, if a person does not select the favorite and their selection wins the championship, that person will most likely win the pool out right.

While this prior work is excellent, the focus is shifted back to studying the probabilistic projections themselves. Could there be something about the projections that could be exploited? Is there a way to predictably deviate from chalk while still outperforming chalk?

2. Methodology

2.1 Single bracket generation problem

As defined in the introduction, the goal is to select a bracket that deviates from chalk while still outperforming chalk. It's clear that chalk performs well so a starting point for formulating the problem could be to select the most probable outcome that is some distance from chalk. This problem statement lends itself to an optimization problem where the objective function is the probability of the bracket occurring, the decision variable is the bracket itself, and the problem is constrained such that the bracket that is selected is a real bracket, mimicking a tournament structure, and that the bracket generated is some distance from chalk. This is formalized below.

$$\underset{\boldsymbol{X}}{\text{maximize}} \quad \sum_{i\in\mathcal{T}, j\in\mathcal{R}} \boldsymbol{X}_{i,j} \cdot \boldsymbol{P}_{i,j} \tag{1}$$

Subject to:

$$\boldsymbol{X}_{i,j+1} \leq \boldsymbol{X}_{i,j} \qquad \forall\, i,j \tag{2}$$

$$\sum\nolimits_{i,j\,\in node\ in} \boldsymbol{X}_{i,j} = 1 \qquad \forall\ node\ in\ tourney \tag{3}$$

$$\frac{\sum_{i\in\mathcal{T}} \boldsymbol{X}_{i,j}\boldsymbol{P}_{i,j}}{\sum_{i\in\mathcal{T}} \boldsymbol{C}_{i,j}\boldsymbol{P}_{i,j}} \leq \boldsymbol{d}_j \qquad \forall j \tag{4}$$

$$\boldsymbol{X} \in \{0,1\}^{|\mathcal{T}|\times|\mathfrak{R}|}$$

Where: $\boldsymbol{X}_{i,j}$ *is the selected outcome of team i in round j* $-\ \boldsymbol{X} \in \{0,1\}^{|\mathcal{T}|\times|\mathfrak{R}|}$

$\boldsymbol{P}_{i,j}$ *is the probability of team i winning round j* $-\ \boldsymbol{P} \in [0,1]^{|\mathcal{T}|\times|\mathfrak{R}|}$

$\boldsymbol{C}_{i,j}$ *is the chalk outcome for team i in round j* $-\ \boldsymbol{C} \in \{0,1\}^{|\mathcal{T}|\times|\mathfrak{R}|}$

$\boldsymbol{p}_j$ *is the points awarded for winning round j* $-\boldsymbol{p} \in \mathbb{R}^{|\mathcal{R}|}$

$\boldsymbol{d}_j$ *is the distance from chalk in round j* $-\boldsymbol{d} \in [0,1]^{|\mathcal{R}|}$

$\mathcal{T}$ *is the set of teams*

$\mathcal{R}$ *is the set of rounds*

The objective function, or equation (1), attempts to maximize the probability of the selected bracket occurring. This is not the actual probability of the bracket occurring but a proxy. $\boldsymbol{P}_{i,j}$ are not conditional probabilities and do not change when a different team than expected is playing team i in round j nor are the probabilities multiplied by one another as expected in a probability

calculation. The objective function is a linear simplification that captures the essence of higher probability teams being selected.

The decision variable, $\boldsymbol{X}_{i,j}$, is 1 when team i wins round j and 0 otherwise. The problem needs to be constrained such that the selection of $\boldsymbol{X}$ mimics a real tournament. If the tournament is structured as a graph with vertices or nodes representing a game in round j and edges into the nodes that represent the outcome of the game for team i in round j, then it becomes clear how to completely specify the behavior of a tournament for $\boldsymbol{X}$. First, any team that loses in round j cannot possibly win in round $j+1$. This is reflected in equation (2). Then for any game, there can only be one winner. This is reflected in equation (3). Figure 1 is an example of a 4-team tournament as a graph.

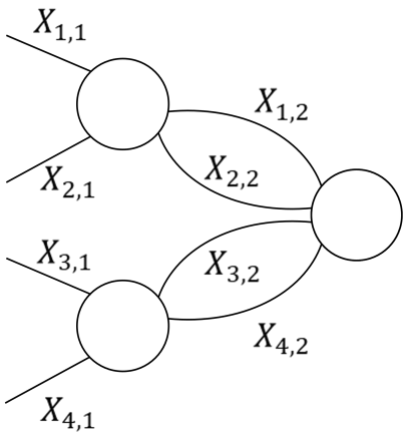

Figure 1: 4-team tournament graph

The last component that is needed is some notion of "distance" from chalk. For each round, the proxy for probability (sum of winning team probabilities) will be at max, the chalk outcome. This is an artifact of the fact that these probabilities are not conditional probabilities and will not change when facing different teams than expected. This lends itself to a nice distance metric defined as the fraction of the generated bracket probability to the chalk probability. This is nice for carrying out the optimization because now the distance for any round is bounded to a maximum of 1. Equation (4) reflects this formulation.

2.2 Bilevel formulation

Now the overall goal of outperforming chalk must be addressed. Firstly, a distance vector must be selected that when applied to the single bracket generation problem, results in a score the outperforms chalk. Secondly, for this problem, the distance vector should not change from year-to-year. The goal is to find a *predictable* distance from chalk that could outperform chalk. So finally, the overall formulation is that the distance vector that is selected, when used in the single-bracket generation problems year-over-year, should result in more brackets that outperform chalk over those years. This can be formulated as another optimization problem.

$$\underset{\boldsymbol{d}}{\text{maximize:}} \quad F(\boldsymbol{X}_1^*, \boldsymbol{X}_2^*, \dots, \boldsymbol{X}_K^*) \tag{5}$$

$$\textit{subject to:} \quad \boldsymbol{X}_k^* \in \underset{\boldsymbol{X}_k}{\text{argmax}}\, f(\boldsymbol{X}_k, \boldsymbol{d}) \tag{6}$$

$$\boldsymbol{d} \in [0,1]^{|\mathcal{R}|}$$

Where: $F(\cdot)$ *is the fitness function*

$f(\cdot)$ *is the objective of the single bracket generation problem*

$\boldsymbol{X}_k^*$ *is the most likely tournament matrix for year* k *some distance from chalk*

$\boldsymbol{d}$ *is the distance vector*

The structure of equations (5)-(6) above show a bilevel structure with upper and lower-level optimization problems that depend on one another. The lower-level problems (single bracket generation problems) are aiming to maximize their own probabilities given what they view as a fixed distance. The upper-level problem must maximize the number of lower-level problems that outperform chalk by changing the distance vector which in turn, changes the lower-level problems' behavior. The upper-level problem can be solved only when the lower-level problems are solved. These types of problems are not uncommon. For example, the toll setting problem. A government may operate a network of highways and may seek to maximize revenue (higher level problem) by setting tolls. The revenue will be a function of the drivers who drive through the tolls however, the drivers seek to minimize their travelling costs (lower-level problem) and may avoid tolls. The government can understand the revenue gain from a toll-setting strategy only once they set the toll and solve the lower-level problem. See Figure 2 below for a visual representation of the bilevel programming problem.

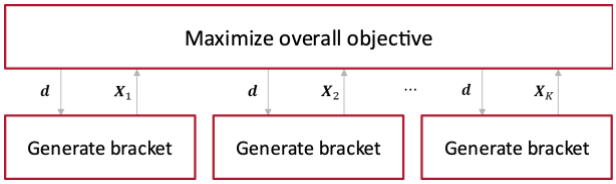

Figure 2: Bilevel Programming Problem

Solving this problem will be discussed in further detail in section 2.3.

2.2.1 Fitness Function

The upper-level objective function or "fitness" function still needs to be defined. At first glance, it seems fair to define the fitness function as the total number of times that the brackets generated by the single bracket generation problem outperform chalk over the years in the dataset. Formally:

$$F(\boldsymbol{X}_1^*, \boldsymbol{X}_2^*, \dots, \boldsymbol{X}_K^*) = \sum_{k=1}^{K} g(\boldsymbol{X}_k^*) = \begin{cases} 1, & S(\boldsymbol{X}_k^*) \geq S(\boldsymbol{C}_k) \\ 0, & o.t.w \end{cases} \tag{7}$$

Where: $S(\cdot)$ *is the score of a bracket given the actual outcome*

This may work for a large dataset but for ~10 datapoints, this is a very discrete function. When changing the distance vector, there may be changes in $S(\boldsymbol{X}_k^*)$ but it may not be enough to change $F(\cdot)$. The function space is very flat with spikes. Searching for a solution is like blindly searching for needles in a haystack. See Figure 3. Optimizing up to the maximum for (a) is as simple as moving in the direction of the gradient; the optimal direction is clear. If a change in X resulted in an increase in $F(X)$, then continue in that direction. However, for (b), going from $X = 2$ to $X = 3$ results in an increase in $F(X)$ but going from 3 to 4 results in a reversion back to the baseline. Which direction is the correct one to explore now?

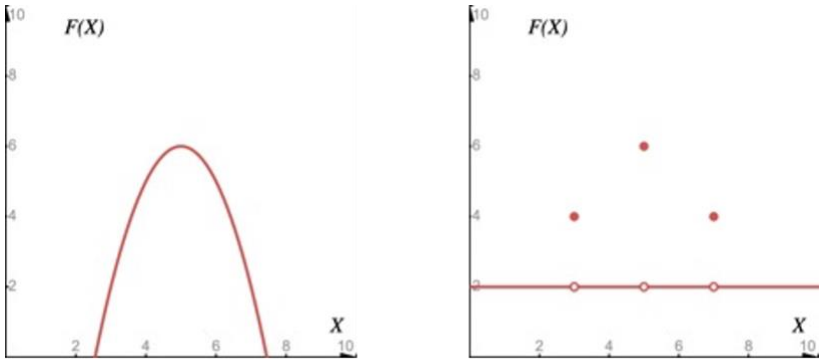

Figure 3: Visual of optimization space where (a) is smooth and continuous and (b)is not

The function space needs more topography such that changes in the distance vector give a clearer direction of heading closer or further from the maximum. Defining the fitness function as the difference in score between the bracket generation problem and chalk rather than the binary outperform or not would provide more topography. An average over the years could be taken resulting in a fitness function that looks like:

$$F(\boldsymbol{X}_1^*, \boldsymbol{X}_2^*, \dots, \boldsymbol{X}_K^*) = \frac{1}{K}\sum_{k=1}^{K} S(\boldsymbol{X}_k^*) - S(\boldsymbol{C}_k) \tag{8}$$

This succeeds in giving the function space a more variable topography but might not accomplish the goal of maximizing the number of years that the single bracket generation problems outperform chalk. For example, take a maximization over two years. In one scenario the single bracket generation problem outperforms chalk by 1000 points in year 1 and underperforms chalk by 200 points in year 2. In a second scenario, the single bracket generation problem could outperform chalk by 300 points both years. In the first scenario, the outcome of equation (8) is 400 and in the second scenario the outcome of equation (8) is 300. However, the second scenario is the only one where chalk was outperformed both years.

To put an emphasis on outperforming chalk while making the solution space more continuous, the difference in performance between the single bracket generation problem and chalk is passed through a sigmoid function.

$$F(\boldsymbol{X}_1^*, \boldsymbol{X}_2^*, \ldots, \boldsymbol{X}_K^*) = \frac{1}{K}\sum_{k=1}^{K} \frac{1}{1 + e^{-\left(s(\boldsymbol{X}_k^*) - s(\boldsymbol{C}_k)\right)/w}} \tag{9}$$

Where: *w is a weight that determines how much performance over chalk matters*

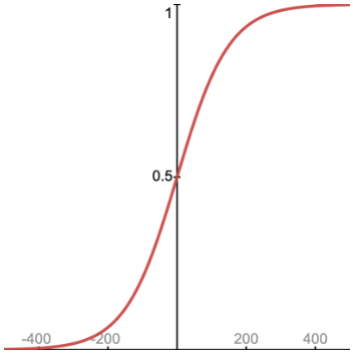

Figure 4: Sigmoid function where w=75

A weight is selected such that differences over a certain threshold provide significant diminishing returns to the fitness function. This makes it such that the highest incremental gain to the fitness function is flipping from underperforming to outperforming chalk. So not only does it prevent from optimizing the outperformance in one year from 800 to 1200 points (for example), but it also discourages "minimizing the bleeding" of a scenario by pushing the underperformance from -1200 to -800 (for example). Those steps will only be taken after all years have been flipped from underperforming to outperforming.

With the bilevel programming problem formulated, focus is turned to attaining a solution to the problem.

2.3 Solving the Bilevel Programming problem

The bilevel programming problem is challenging to solve given that the lower-level problems need to be solved before the upper-level problem can be solved. Bilevel programs with convex lower programs can be reformulated into a single level by replacing the lower-level problem with Lagrangian multipliers and KKT conditions.

Unfortunately, the single bracket lower-level problems are discrete programming problems and are not convex so there is no straightforward way to reformulate the problem and come to an exact solution. An approximate solution must be attained heuristically by efficiently searching over distance vectors and their corresponding fitnesses. Genetic algorithms are a good option to do just that.

Given its namesake, genetic algorithms take inspiration from evolution and genetics. They generally work by initializing a set of distance vectors as candidates. Each distance vector is evaluated for its fitness. Then a "parent" distance vector is selected using a process that makes it more likely for higher fitness distances to be selected. A common method is tournament selection where k distance vectors are selected at random from the population and the highest fitness vector is selected with some probability p, the second highest fitness is selected with some probability $p * (1 - p)$, the third highest fitness is selected with some probability $p * (1 - p)^2$, and so on. This is done twice to

create two parents and a child is generated through crossover and mutation. Crossover is when values at the same index in two vectors are swapped. Mutation is when a value in the vector is perturbed. Child creation continues until enough children are made to make a new generation at which the cycle starts over with the children becoming the population. The algorithm is laid out more formally:

Algorithm 1 Genetic algorithm

Randomly initialize $population$ with distance vectors
For $generation$ in $number_{generations}$:
  Evaluate the fitness of the distance vectors in $population$
  Initialize empty list $children$
  For i in $size_{population}$
    Select 2 parents from $population$ through tournament selection
    Create distance vector $child$ through crossover and mutation
    Add $child$ to $children$
  End for
  Set $population = children$
End for
Select highest fitness individual in $population$ and terminate

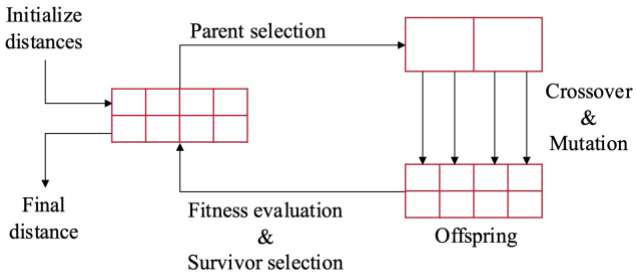

Figure 5: Genetic Algorithm Overview

In the case of this problem, evaluating the fitness is the most computationally expensive portion of the loop given that it is K discrete programming problems that must be solved using a branch and bound method. There are efficiency gains in evaluating the fitnesses of the population in parallel or synchronously. This class of evaluation is common in genetic algorithms and is called a synchronous genetic algorithm. However, for this problem, the full effect of the efficiency gain from synchronous fitness evaluation is not realized. Synchronous evaluation works well when fitness function computation time is fixed. If it takes one second to evaluate the fitness of a distance vector and two distances need to be evaluated, either it takes two seconds to do the evaluation in series or it takes 1 second if both the distance vectors are evaluated in parallel on two separate processors. However, this changes if fitness function computation time is variable. For example, now the fitness function takes 1 second to solve *on average* where the evaluation time distribution is the uniform distribution from 0.5 seconds to 1.5 seconds. There is a $50\% * 50\% = 25\%$ chance that both distance vectors' fitnesses are evaluated under the 1 second average. This means there is a 75% chance that at least one of the processes runs long. If one of the processes runs long, the processor where the process has already completed must sit idle and wait for the other process to complete.

This idle time erodes the efficiency meant to be gained from synchronous evaluation. More details on this are in Appendix A.

Given that this fitness function involves solving discrete programming problems, the time to a solution is not constant and synchronous evaluation will not see full gain. The distance vectors must be evaluated asynchronously to remove the idle time. Rather than complete the genetic algorithm in generations or batches, a running population is held and updated. Processors complete the evaluation of children asynchronously and when evaluation on a processor is complete, the child is compared to the population. If the child has a higher fitness than the individual in the population with the worst fitness, that individual is replaced with the child. If not, the child is rejected. Upon replacement or rejection, the processor immediate evaluates a new child. The new asynchronous algorithm looks like:

Algorithm 2 Asynchronous genetic algorithm

Randomly initialize $population$ with distance vectors
Evaluate the fitness of the distance vectors in $population$
For $birth$ in $max\ births$ (executed whenever a process is free):
    Select 2 parents from $population$ through tournament selection
    Create distance vector $child$ through crossover and mutation
    Evaluate $fitness_{child}$
    If $fitness_{child} > \min(fitness_{population})$:
        Remove individual with lowest fitness from $population$
        Add $child$ to $population$
    Else
        Reject $child$
    End if
End for
Select highest fitness individual in $population$ and terminate

The generations are replaced with births as the population is no longer created in batches but continuously updated. The birth loop can be completed on any number of processors. A new birth occurs when a processor has completed evaluation of a child, removing any idle time.

3 Results

The problem was solved over the last 9 years using projections from FiveThirtyEight.com [5]. The asynchronous genetic algorithm was defined with the hyper parameters listed in Table 1. The library used to execute the asynchronous genetic algorithm was leap-ec [6]. The algorithm took 45 minutes to run on a 10th gen Intel Core i7. The algorithm came to the solution in Table 2. For comparison, Table 2 also contains the distance vectors corresponding to the actual outcomes for the years included in the dataset. Table 3 contains a comparison of the scores that would have been generated by the distance vector solved for in Table 2. The scores are calculated using ESPN's tournament scoring methodology.

Table 1: Asynchronous Genetic Algorithm Hyperparameters

Hyperparameter	Specification
Number of processors	4
Maximum births	2048
Population size	16
Parent selection	Tournament selection
Tournament selection mechanics	Select 2 individuals at random. Select best individual out of the two 100% of the time
Crossover mechanic	Crossover each element with 20% probability
Mutation mechanic	Add random variable with distribution $\mathcal{N}\left(0, \frac{1}{30}\right)$, bounds $= [0.8, 1]$, to two elements in the vector on average

Table 2: Distance Vectors

	Round 64	Round 32	Sweet 16	Elite 8	Final 4	Championship
Final solution	0.999	0.909	0.929	0.915	0.826	0.983
2014	0.894	0.779	0.706	0.484	0.118	0.045
2015	0.922	0.792	0.787	0.718	0.380	0.141
2016	0.838	0.811	0.8	0.639	0.673	0.331
2017	0.962	0.789	0.632	0.527	0.847	0.467
2018	0.868	0.637	0.553	0.65	0.631	1.000
2019	0.879	0.979	0.868	0.52	0.583	0.891
2021	0.842	0.678	0.639	0.752	0.955	0.475
2022	0.876	0.703	0.447	0.425	0.324	0.33
2023	0.856	0.751	0.492	0.176	0.137	0.139

Table 3: Results

Year	Total		Outperformance by round					
	D score	Chalk	Round 64	Round 32	Sweet 16	Elite 8	Final 4	Champ
2014	**780**	690	**10**	**40**	**40**	0	0	0
2015	**870**	860	**10**	**40**	-40	0	0	0
2016	**1050**	990	0	**20**	-40	**80**	0	0
2017	**860**	700	0	0	0	0	**160**	0
2018	710	1140	**10**	**40**	0	0	-160	-320
2019	**1250**	950	**20**	-80	-40	**80**	0	**320**
2021	**1020**	870	**10**	**20**	**40**	**80**	0	0
2022	**1020**	790	-10	-40	-40	0	0	**320**
2023	510	520	-10	-40	**40**	0	0	0

4 Discussion

4.1 Reflections/considerations

Before investigating the final scores, it is interesting to look at the distance vectors. Table 2 contains the distance vector solution from running the asynchronous genetic algorithm and the distance vectors for the outcomes that actually occurred over the years included in the dataset. It is interesting to note that the elements of the distance vectors for the actual outcomes trend downwards as the rounds carry-on. This makes sense in one sense; as more rounds occur, more upsets occur and there are fewer teams in the denominator making it more likely that the teams in the final rounds are those that won an upset. There is also a competing dynamic that cinderellas should be "flaming out" by the final rounds. This does occur in some years where the final round fractions jump up. The most interesting observation is that the solution that comes from the algorithm is quite close to chalk when compared to the actual outcomes. This shows, at least for the strategy discussed in this paper, that while deviating from chalk is necessary, doing so in a predictable fashion year-over-year is challenging. The best strategy is to stick close to chalk. This follows convention and confirms that these projections are very good.

Even with very good projections, the problem formulated and solved as is generates brackets that outperform chalk 7 out of the last 9 years. This is an unexpected and interesting outcome. Given these projections are probabilistic in nature, it would be expected that picking a predictable distance from chalk might outperform chalk in a couple years. But to outperform chalk more often than not – and in this case, almost every time – could say something about the projections and the nature of the tournament.

It's important to be skeptical of the data and to inspect it critically. Firstly, a degree-of-freedom analysis can provide some context as the result. The distance vector has 6 elements, one for each round. It could be possible that each element could affect the outperformance of only one round for one year. For that to be the case, it would need to be assumed that deviating from chalk in one round does not affect the difference in score from the downstream rounds. This could happen like in a scenario where an upset occurs in round j but that same team loses in round $j + 1$ to the favorite. It would also need to be assumed that deviating from chalk in one year does not cause deviations from chalk in other years, which cannot be true given the problem is structured such that the deviations are the same across all years. But even under these extreme assumptions, it would be expected that this method would outperform chalk in 6 years – one for each degree of freedom in the distance vector.

Secondly, it's good to look for "outlier" scenarios that may be overrepresented in this small dataset and may not scale to the distribution of a larger dataset. There are two years, 2019 and 2022, where the distance vector generation method is underperforming chalk all the way until the final round but is saved by the championship. It might be rare that this strategy could correctly select the winner and if scaled to a larger dataset, the +320-point boost might not occur as often as it does in this small dataset. Even if those years are counted as underperforming chalk, this bracket generation method still outperforms chalk more than half the time. In almost every round, except for the sweet 16, this method outperforms chalk more times than not across the years. It does not seem as if the strategy is to underperform chalk until the final rounds and "get lucky" in the high scoring rounds.

After looking at the results through a critical lens, there does seem to be some underlying pattern for selecting brackets some distance from chalk that will outperform chalk. Could it be that there is something unique about the structure of the tournament that needs to be accounted for in probabilistic projections?

4.2 Next Steps

These findings are on a very limited dataset. Next step would be to run this algorithm on a larger sample size. To do this, extra projections are needed. If the trend holds true on a larger sample size, the following step is to attempt to find the predictable pattern and incorporate them into the projections themselves.

Another next step is to find other scenarios in sports where this bilevel programming approach could be useful. It probably isn't useful in the middle of a game given its computational cost, but it could be used in pre-game planning and strategy. There are probably other scenarios in sports where a higher-level goal must be attained while individuals are solving their own lower-level, individual problems.

5 Conclusion

In conclusion, selecting a bracket some distance from chalk is required to win a bracket challenge, however deviating from chalk should result in fewer points year-over-year. To test this hypothesis, the bracket generation problem was structured as a bilevel programming problem where the lower-level problem is to select the most probable bracket some distance from chalk and the upper-level problem is to select the distance that results in the maximum number of brackets outperforming chalk. The problem was solved using an asynchronous genetic algorithm and the solution resulted in brackets that outperformed chalk 7 out of the last 9 years.

Appendix A: Asynchronous gains

As stated in section 2.3, the synchronous genetic algorithm is most effective when evaluation of the fitness function is constant. The bracket generation problem does not have constant solve time as the lower-level problems require solving discrete programming problems meaning the gain from synchronous fitness evaluation is eroded. Figure 6 shows a distribution of fitness evaluation times for 250 runs. The data roughly follow a shifted exponential distribution. The distribution was fit using the distfit package in python.

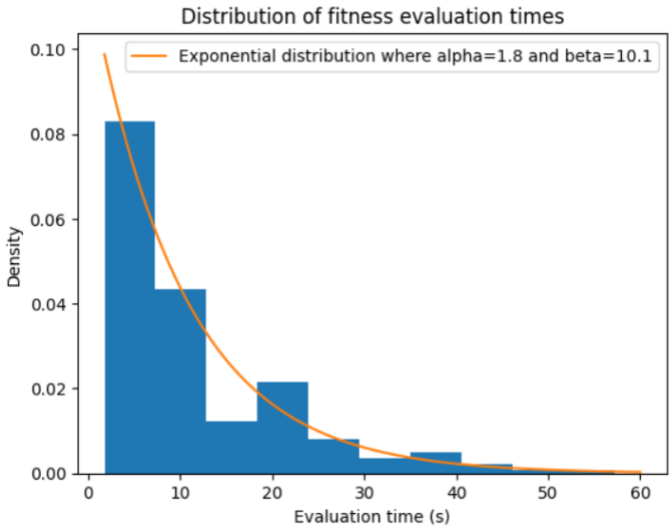

Figure 6: Fitness evaluation distribution

Now, the effect of having two processors evaluating the fitness function synchronously will be illustrated to show the erosion of efficiency gain under random processes. As discussed in section 2.3, the issue is that one processor may run long and, in that scenario, the first processor will have to sit idle until the second processor completes. Given this issue, the quantity of interest is how much time it takes the second processor to finish, on average.

To start, the probability density function for the exponential distribution $p(t)$ is integrated to arrive at a cumulative distribution $P(t)$. This cumulative distribution describes the probability that the process will complete in a certain amount of time.

$$p(t) = \frac{1}{\beta} e^{-(t-\alpha)/\beta} \tag{10}$$

$$P(x \leq t) = \int p(x)\, dx = \int_{\alpha}^{t} \frac{1}{\beta} e^{-(x-\alpha)/\beta}\, dx \tag{11}$$

$$P(x \leq t) = \int_{\alpha}^{t} \frac{1}{\beta} e^{-(x-\alpha)/\beta}\, dx = -e^{-(x-\alpha)/\beta}\Big|_{\alpha}^{t} = 1 - e^{-(t-\alpha)/\beta} \tag{12}$$

Then the probability that two processors will complete in a certain amount of time is the square of the cumulative distribution function assuming the solve times are independent.

$$P(x_1 \le t, x_2 \le t) = P(x_1 \le t) \cdot P(x_2 \le t) = \left(1 - e^{-(t-\alpha)/\beta}\right)^2 \tag{13}$$

Finally, the goal is to find the average time it takes for two processors to complete. The cumulative distribution will need to be converted back to a probability density function, done in equation 14, and then an average must be computed over the domain, done in equation 15.

$$\frac{dP}{dt} = \frac{2}{\beta}\left(1 - e^{-(t-\alpha)/\beta}\right)e^{-(t-\alpha)/\beta} \tag{14}$$

$$E[t] = \int_{\alpha}^{\infty} \frac{2t}{\beta}\left(1 - e^{-(t-\alpha)/\beta}\right)e^{-(t-\alpha)/\beta}\, dt \tag{15}$$

Evaluating the integral is easiest done by multiplying through and splitting the integral in two terms:

$$E[t] = 2\left[\int_{\alpha}^{\infty} \frac{t}{\beta}e^{-(t-\alpha)/\beta}\, dt - \int_{\alpha}^{\infty} \frac{t}{\beta}e^{-2(t-\alpha)/\beta}\, dt\right] \tag{16}$$

Both terms in the integral will need to be evaluated using integration by parts defined in equation 17.

$$\int u\, dv = uv - \int v\, du \tag{17}$$

Equations 18 and 19 evaluate the first term of equation 16 using integration by parts.

$$\begin{aligned} u &= t & v &= -e^{-(t-\alpha)/\beta} \\ du &= dt & dv &= \frac{1}{\beta}e^{-(t-\alpha)/\beta}\, dt \end{aligned} \tag{18}$$

$$-te^{-(t-\alpha)/\beta}\Big|_{\alpha}^{\infty} - \int_{\alpha}^{\infty} -e^{-(t-\alpha)/\beta}\, dt = -(t+\beta)e^{-(t-\alpha)/\beta}\Big|_{\alpha}^{\infty} = \alpha + \beta \tag{19}$$

Equations 20 and 21 evaluate the second term of equation 16 using integration by parts.

$$\begin{aligned} u &= t & v &= -\frac{1}{2}e^{-2(t-\alpha)/\beta} \\ du &= dt & dv &= \frac{1}{\beta}e^{-2(t-\alpha)/\beta}\, dt \end{aligned} \tag{20}$$

$$-\frac{t}{2}e^{-2(t-\alpha)/\beta}\Big|_{\alpha}^{\infty} - \int_{\alpha}^{\infty} -\frac{1}{2}e^{-2(t-\alpha)/\beta}\, dt = -\left(\frac{t}{2} + \frac{\beta}{4}\right)e^{-2(t-\alpha)/\beta}\Big|_{\alpha}^{\infty} = \frac{\alpha}{2} + \frac{\beta}{4} \tag{21}$$

Then focus is returned to completing the evaluation of the expectation. The result is equation 22.

$$E[t] = 2\left[\alpha + \beta - \frac{\alpha}{2} - \frac{\beta}{4}\right] = \alpha + \frac{3\beta}{2} \tag{22}$$

This is the average time required for two processors to each evaluate the fitness of an individual if evaluation started at the same time and finished when the second processor completed evaluation. It's straight forward to evaluate the time it takes for one processor to evaluate the fitness of one individual. This is done by integrating $tp(t)$ over the domain which was done in the first term of equation 16 and solved in equation 19.

Now if the processes are done synchronously and the solve time is constant at the average solve time, then adding a processor would mean that both processes still complete in $\alpha + \beta$ time or 11.9 seconds for the bracket generation problem. However, equation 22 shows that there is a penalty of $\frac{\beta}{2}$ or 5.05 seconds for the bracket generation problem. This penalty comes from the possibility of one of the processes running long. It makes sense that the penalty is related to β as this parameter describes the spread of the distribution. The higher the spread, the more likely a processor will run long, and the worse the penalty.

This phenomenon can be examined at scale as well for n processors. All that's required is to adapt equation 13 to n processors, compute its derivative, and then take the expectation. The result is equation 23.

$$E[t] = \int_{\alpha}^{\infty} \frac{nt}{\beta}\left(1 - e^{-(t-\alpha)/\beta}\right)^{n-1} e^{-(t-\alpha)/\beta}\ dt \tag{23}$$

This integral is not as straight forward to solve for analytically, but it can be approximated using the scipy package in python. Figure 7 shows the average time required to complete n fitness function evaluations when waiting for the last one to finish.

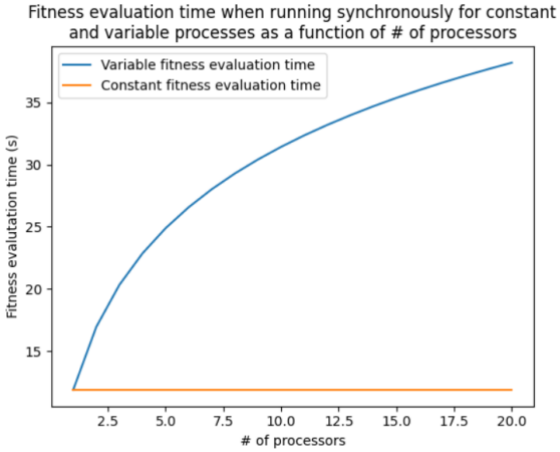

Figure 7: Average evaluation time for n processors

The horizontal line represents the time it would take for n processors to complete the task if they all started at the same time and the solve time was fixed at the average for the exponential

distribution. It's clear that the penalty accumulates as more processors are included. This seems intuitive as increasing the number of solves done synchronously should increase the likelihood that one of them will take a long time. It is interesting to note that the curve is convex. The incremental penalty per processor is decreasing as more processors are included. This most likely comes from the shape of the exponential distribution. Very long evaluation times are very unlikely so including another processor is worth the risk of a longer evaluation occurring.

This is better illustrated when compared to a completely unparallel evaluation of fitness functions (standard genetic algorithm). In the extreme scenario, one where the processors evaluate fitness functions in series, rather than in parallel, the evaluation time would just be $n * mean\ eval\ time$ or in the case of this problem $11.9n$. It would be interesting to take the fitness evaluation times from Figure 7 and divide them by the unparallel scenario.

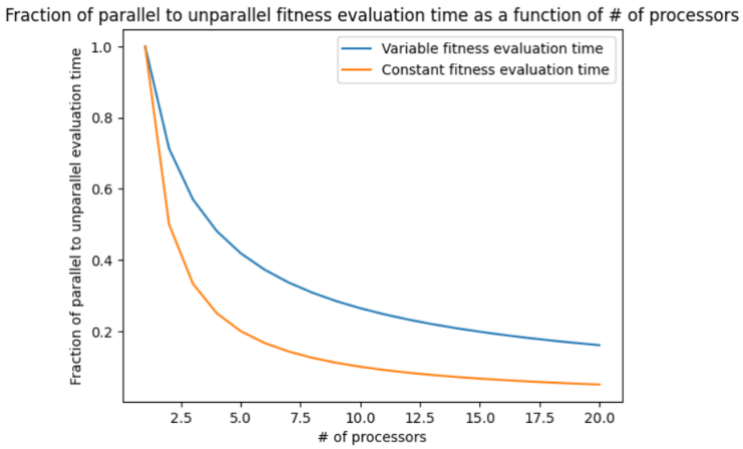

Figure 8: Fraction of unparallel evaluation as a function of # processors

From Figure 8, it is clear that even though there is a penalty for the variable evaluation times, it is still better than doing no parallelization at all. It even looks like the gap between variable and constant evaluation times is closing as the number of processors increases which also reinforces the diminishing penalty from including more processors.

Although there are diminishing penalties from an increase in processors, the number of processors available to most is limited thus the asynchronous algorithm, which removes idle time, saves valuable time.

References

[1] Published by Statista Research Department and S. 7, "Number of bettors filling out a march madness bracket US 2023," Statista, https://www.statista.com/statistics/1223099/bracket-march-madness-intention/ (accessed Nov. 24, 2023).

[2] D. Holmes, "How Many Different March Madness Combinations Are There?," *BIL Betting*, Mar. 10, 2023. https://ballislife.com/betting/news/how-many-different-march-madness-combinations-are-there/#:~:text=There%20are%209.2%20quintillion%20possible (accessed Nov. 24, 2023).

[3] Boice, Jay. "How Our March Madness Predictions Work." FiveThirtyEight, 17 Mar. 2019, fivethirtyeight.com/methodology/how-our-march-madness-predictions-work-2/. Accessed 24 Nov. 2023.

[4] Niemi, Jarad, et al. Identifying and Evaluating Contrarian Strategies for NCAA Tournament Pools Identifying and Evaluating Contrarian Strategies for NCAA Tournament Pools. 2005

[5] M. A. Coletti, E. O. Scott, and J. K. Bassett, "Library for Evolutionary Algorithms in Python (LEAP)," in Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, 2020, pp. 1571–1579. doi: 10.1145/3377929.3398147.

[6] J. Boice, N. Silver , 2023, "FiveThirtyEight NCAA Forecasts", [Online]. Available: https://www.kaggle.com/datasets/raddar/ncaa-men-538-team-ratings