ByteBracket: an efficient method for encoding, sharing, and scoring prediction brackets for single elimination tournaments

Andy Brown[1*]

[1]Research and Development, Udacity

[*]To whom correspondence should be addressed; email: pursuingpareto@gmail.com.

ByteBracket is a protocol for efficiently encoding and decoding predictions and results of single elimination tournaments. In a tournament with N teams, ByteBracket requires $N - 1$ bits to uniquely specify a set of predictions and can score predictions against a completed results bracket in $O(\log N)$.

1 Introduction and Motivation

The NCAA Division 1 basketball tournament, colloquially referred to as "March Madness", is a 64 team single-elimination tournament. Every year, millions of fans painstakingly predict the outcome of each of the 63 games by filling out their "bracket".

These "brackets" are often pooled into office tournaments, friendly competitions, etc... but the sharing of these brackets is awkward and highly dependent on the proprietary encoding / decoding techniques of various bracket making services[1].

Moreover, much of this bracket sharing occurs over character-limiting social networks like Twitter where an entire ecosystem of character compressing services has emerged to cater to

[1]One such bracket making tool can be found at http://www.udacity.com/bytebracket

the social network's character-conscious user base.

Without a succinct and standardized system for representing prediction brackets, users of these networks have had to rely on closed-source bracket sharing solutions offered by existing bracket-maker providers, making it both clumsy to share prediction brackets and almost impossible to compare brackets made with competing providers.

ByteBracket aims to solve these problems while also giving bracket-maker providers an optimally compressed representation of a bracket.

2 The ByteBracket Protocol

The ByteBracket protocol exploits the binary nature of a basketball game. Since ties are not permitted in basketball each game has exactly two possible outcomes. By agreeing on an ordering for games and a convention for specifying outcomes, a 63 game tournament can be encoded into 63 bits or 8 bytes. The naming conventions can be understood by examining a small eight

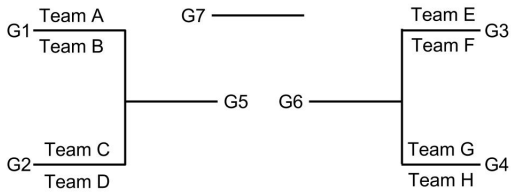

Figure 1: Diagram for eight team tournament with seven games.

team tournament as shown in figure 1. In this tournament the seven games are assigned the numbers 1 - 7 according to the following rules (listed in descending priority):

1. The championship game is always assigned the number $N - 1$ for a tournament with N teams.

2. Games in earlier rounds have lower numbers than games in later rounds.

3. Games on the left side of the diagram have lower numbers than the right.

4. Games are numbered from top to bottom.

With this numbering convention in place, we can encode predictions into 7 bits, one for each game, according to the following rules:

1. Each game is encoded according to its game number.

2. A prediction for the upper team to win is encoded as a 1.

3. A prediction for the lower team to win is encoded as a 0.

4. The winner of game $N-3$ is considered the **upper** team for the championship game. The winner of game $N-2$ is the **lower** team.

According to these rules, a prediction bracket like the one shown in figure 2 would be encoded as **1101110**.

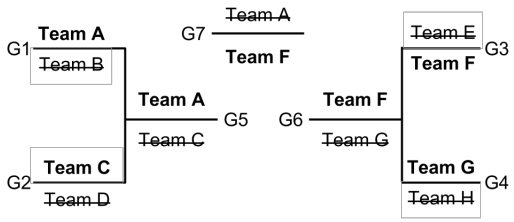

Figure 2: Completed prediction "bracket" with Team F winning the tournament. This bracket would be encoded as 1101110.

2.1 Hexadecimal character minimization

While an $N-1$ bit representation of a bracket is optimally compact for *storage* of a prediction bracket, it's not optimized for *sharing* on social networks where 1s and 0s are represented as characters which require a full byte to encode.

The number of characters required can be reduced by a factor of 4 by converting the bitstring representation to hexadecimal. Using the example from figure 2, we can convert the first four bits, 1101, into the hexadecimal equivalent of D and the last three bits (with a zero added to the end), 1100, into C. This gives the shareable bracket code DC.

3 ByteBracket Scoring Algorithm

The ByteBracket scoring algorithm compares a prediction bracket against a "results" bracket in $O(\log N)$ time where N is the number of teams in the tournament.

3.1 Python Summary

The summary is given as annotated Python code. Note that for pedagogical purposes and readability this code has not been fully optimized, though will still score a bracket in $O(\log N)$ time.

```
def score(bracket, results, filt,
    teams_remaining, blacklist, round_num=0):
    """
    Recursively calculates the score of a prediction
    bracket against a results bracket.

    - bracket
        A bitstring representing a prediction bracket. For
        a 64 game tournament this would be 63 bits.

    - results
        A bitstring representing the actual outcome of
        a tournament.

    - filt
```

```
    With the exception of the first round in a tournament,
    its not possible to score a round by just comparing
    the bits in bracket to the bits in results. For example,
    correctly predicting the championship game requires not
    only the correct bit for that game, but also the correct
    prediction for every game the tournament winner had
    won before the final round.

    The filt parameter is a one time pre-computed bitstring
    used to indicate which games in a round must be
    correctly predicted in order to correctly predict
    successive games. For a 64 game tournament filt would
    contain 62 bits.

- teams_remaining
    score is a recursive function where each call
    represents another tournament round.

    teams_remaining gives the number of teams left in the
    tournament as of this function call.

- blacklist
    This parameter is a sequence of N bits where N is
    the number of games in the current round. It uses
    the accuracy of predictions from previous rounds to
    "remember" which games are possible to correctly
    predict. When calling the score function initially
    these bits should all be set to 1.

- round_num
    A number representing the current round. For a 64 team
    tournament this would take the values 0,1,2,3,4, and 5
"""

# First check if there is a winner
if teams_remaining == 1 :
    return 0

# compute constants for round
# round_mask is a bitstring with all bits set to 0
# except the bits corresponding to the current round
num_games = teams_remaining / 2
round_mask = 2 ** num_games - 1

# the current round is encoded in the num_games
# least significant bits. Likewise for results
# and filter
round_predictions = bracket & round_mask
bracket = bracket >> num_games
```

```
    round_results = results & round_mask
    results = results >> num_games
    round_filter = filt & round_mask
    filt = filt >> num_games

    # The overlap between the prediction bits and the
    # results bits is calculated by XORing the two and
    # then flipping the result.
    overlap = ~(round_predictions ^ round_results)

    # In all rounds except the first, overlap will tend
    # to overestimate a bracket's correctness. This is
    # corrected by ANDing the overlap with the blacklist
    scores = overlap & blacklist

    # the points for this round are calculated by counting
    # the number of 1s in the scores bitstring and then
    # multiplying by 2 ^ round_num (this multiplication
    # is used to weigh predictions in later rounds more
    # heavily than earlier rounds)
    points = popcount(scores) << round_num

    # with the points calculated we can now use the
    # pre-computed filter to figure out which of these
    # predictions may impact future predictions
    relevant_scores = scores & round_filter

    # For each pair of games in this round, look for a 1
    # in either of the bits to compute the blacklist
    # for the next round.
    even_bits, odd_bits = get_odds_and_evens(relevant_scores)
    blacklist = even_bits | odd_bits

    # recursively call score function with updated params
    return points + score(bracket, results, filt,
        teams_remaining / 2, blacklist, round_num + 1)

def get_odds_and_evens(bits):
    """
    Separates the even and odd bits by repeatedly
    shuffling smaller segments of a bitstring.
    """
    tmp = (bits ^ (bits >> 1)) & 0x22222222;
    bits ^= (tmp ^ (tmp << 1));
    tmp = (bits ^ (bits >> 2)) & 0x0c0c0c0c;
    bits ^= (tmp ^ (tmp << 2));
    tmp = (bits ^ (bits >> 4)) & 0x00f000f0;
    bits ^= (tmp ^ (tmp << 4));
    tmp = (bits ^ (bits >> 8)) & 0x0000ff00;
```

```
    bits ^= (tmp ^ (tmp << 8));
    evens = bits >> 16
    odds = bits % 0x10000
    return evens, odds

def popcount(x):
    """
    Counts the number of 1s in a bitstring.
    """
    x -= (x >> 1) & 0x5555555555555555
    x = (x & 0x3333333333333333) + ((x >> 2) & 0x3333333333333333)
    x = (x + (x >> 4)) & 0x0f0f0f0f0f0f0f0f
    return ((x * 0x0101010101010101) & 0xffffffffffffffff ) >> 56
```

4 Acknowledgments and Apologies

First I'd like to thank Udacity and the incredible people who have worked there. Four years ago I had never written so much as a print statement and since then I have used Udacity to acquire skills I had once thought were unobtainable. Udacity has changed me and I'm forever grateful for not only the skills I've obtained but also the incredible new lens through which I can now view the world. There is something beautiful to me about seeing a March Madness bracket as a sequence of 63 bits and were it not for the education Udacity has given me that beauty would still be hidden.

Finally, I also want to extend my thanks and apologies to you, reader, for making it this far through this absurd exploration of unnecessary optimization. And hey, if you wanted to go to www.udacity.com/bytebracket, make a bracket and tweet it to the world with the ByteBracket hashtag, that would be great! Thanks for reading.