Battleship and Hangman Theory

Grant Yang Atharv Goel

I. Battleship

Playing battleship, we can query one of the 100 possible squares to get a result of 'hit' 'miss', or 'you sank my battleship'. To simplify this problem, we will ignore the 'you sank my battleship' possibility and assume that each turn will at maximum yield 1 bit of information about the configuration of the opponent's ships. To maximize information gained, we should choose the square such that the probability of hitting a ship is closest to 50%. However, because one must sink all ships to win the game, we should instead choose the square with the highest probability of hit. The calculation of the probability of hitting a ship can be calculated with two algorithms.

The first algorithm is uniform random sampling. From the current game state, we can randomly place battleships in arrangements that satisfy the observed hits and misses up to this point. We will try to sample uniformly despite the fact that arrangements where two ships are touching are probably less likely in reality due to that not being a good strategy (on the opponent's part). However, efficient and unbiased random sampling of battleship positions becomes very inefficient once multiple hits have been observed. ¡DESCRIBE ALGORITHM¿ For this reason, we will switch to our second strategy once two hits have been observed.

Our second strategy is simply full enumeration of all possible battleship states given the current observed state for the current turn. ¡DESCRIBE ALGORITHM AND OPTIMIZATIONS¿

II. Hangman

We consider Hangman as two different subproblems. The first is picking the optimal letter given a current game. The second is updating the game after receiving information from the user. We begin with the latter.

Since the user can lie, we are uncertain about the game state. Hence, let the current game be the random variable S with probability mass function $p(s)$. Each different possible value of S now represents the simplified game state given that the user lied on a specific turn, or not at all. At the beginning of the game, there is only one possible value for S: the *truth node*. Note that the user can only lie once. Thus, as the game progresses, S gains more possible values as the *truth node* splits into a new *truth node* and a *lie node* that assumes the user lied on that turn. However, all the *lie nodes* from previous turns can get filtered by the new information as if it was telling the truth. $p(s)$ is defined as a geometric distribution with $p = 0.25$. It is a little arbitrary, but it works.

For picking the optimal letter, we merely choose whichever gives the most expected information, across all possibilities for S. Each word has a relative frequency sourced from the web. Since the specific value of S does not matter, we condense it into a single state, with each word frequency scaled by $p(s)$ where s is the node that the word is in. Identical words in different nodes are combined such that their $p(s)$ scaled frequencies are added. The phrases are flattened as well. Thus, we obtain a single structure with a random variable W for each word, with a probability mass function obtained from the word frequencies. For a given letter, the expected information is found by summing $-p\log(p)$ over every different pattern in which the letter can appear, where p is the sum of the probabilities of the words to which the pattern corresponds. This is looped over every letter, and whichever one yields the highest expected information is guessed.