Parallel Wordle

Jennifer Wang (xw2763)

1 Introduction

The New York Times developed a Wordle puzzle game in which players are given six attempts to guess a five-letter word. Following each attempt, the Wordle game provides feedback by coloring each letter in the player's guess one of three colors: green indicates that the letter is present in the answer and at the correct position, yellow indicates that the letter is present in the answer but not in the correct position, and grey indicates that the letter is not present in the answer. If a player successfully guesses the word on their sixth attempt, they are declared the winner. If they do not, they are considered to have lost the game.
This project explores three different heuristics algorithms for solving the Wordle game: frequency, minimax, and entropy. It then paralllelizes the entropy heuristics to achieve a faster tuntime.

2 Wordle Game Configuration

2.1 File Processing

The first step of the Wordle engine is to read and process two word files: `answerlist.txt` and `wordlist.txt`. The former contains 2,309 five-letter words that are utilized by the New York Times as potential solutions to the Wordle game, while the latter contains 12,947 five-letter words that are considered valid by the NYT Wordle game. In order to optimize this file I/O process, I used ByteStrings instead of traditional Haskell strings. The file handle's entire content was read into ByteStrings in a chunk-by-chunk manner, and subsequent manipulation of the ByteString was conducted using functions from the `Data.ByteString.Char8` package. The gist of the function is presented below (`B` stands for `Data.ByteString` and `BC` stands for `Data.ByteString.Char8`. Error checking has been omitted for the sake of simplicity. Please see `Parser.hs` in the code listing section for the full implementation.

```
fileToWordList :: FilePath -> IO [String]
fileToWordList path = do
  handle <- catchAny (openFile path ReadMode) $ \_ -> do
    die $ "Could not open file: " ++ path
  contents <- B.hGetContents handle
  return $ Prelude.map BC.unpack $ BC.lines contents
```

2.2 Game Implementation

We also need a set of functions to perform the function of the NYT Wordle game. Specifically, upon each guess from the user, the Wordle game must provide feedback on the player's guess by assigning one of three colors to each of the five letters: green indicates that the letter exists in the answer and is positioned correctly; yellow indicates that the letter is present in the answer but is not placed correctly. Grey indicates that the letter does not appear in the answer.
A few subtleties are involved in this process. Namely, in the situation of repeated letters, some care needs to be taken. For instance, if the answer is `class` and the user's guess is `shoes`, then the 1st letter should return yellow, the 4th letter should return grey, and the 5th letter should return green. This shows that a counting implementation is needed instead of a simple sequential pattern matching scan.
The above is encapsulated in the function `getWordleResponse` located in `WordleGame.hs`, where a Wordle response in the format of 0's, 1's, and 2's are returned given the user's guess and the correct Wordle answer.

2.3 Player

The Wordle engine also requires a set of functions to simulate the movements of a player in the Wordle game. In contrast to the actual New York Times Wordle game, the Haskell player version implemented in this project is a bit different, tailored more towards the main goal of performance measurement rather than for entertainment. Instead, the player's role is to first make a guess based on one of the three heuristics (frequency, information entropy, or minimax) and a set of valid words, input that guess to the Wordle game engine (discussed above), filter out all possible valid words based on the response, and repeat the process until it guesses the correct word. Unlike NYT's version, no turn or depth limit is set. This allows us to obtain the full range of measurements and answer questions such as "how many games out of all possible games can the heuristics solve under a certain number of turns," or "what is the average number of turns taken by all possible games by utilizing a certain heuristic."
The set of functions in `Player.hs` implement the steps described above. Further details can be found in the code listing section.

2.4 Bonus Semi-Interactive Player

As a bonus, I also implemented a semi-interactive player to help users solve the NYT daily wordle game. The interactive function first suggests a new word from the set of valid words for the user to input into the NYT wordle game. After the user types in that word into the official game website, receives a response, and types that response to the terminal, the interactive function filters the set of valid word and repeats the word-guess suggestion process. This loop terminates when there is no remaining valid word.
Here is an example of me using the Entropy Heuristics to help me solve the Wordle game of the day. The word turned out to be "chord" and it took me three turns. Not bad!

```
$ stack exec wordleSolver-exe interactive
Please enter "soare" into NYT's Wordle game.
Enter NYT's response in 0/1/2 format:
01020
Please enter "hyoid" into NYT's Wordle game.
Enter NYT's response in 0/1/2 format:
10202
Type "chord" to win!
```

Code for this interactive loop resides in `Interactive.hs`.

3 Three Heuristics

In the last section, I introduced the logical steps taken by the player of the Wordle game. One of the key steps is to make guesses from a set of valid words based on a heuristics. The quality of the heuristics directly impacts the outcome of the Wordle game. A good heuristics leads less steps taken to reach the correct answer. This section details three types of heuristics that I implemented.

3.1 Frequency Heuristics

Frequency is a basic heuristic algorithm for selecting a word. The process involves iterating through all possible words and counting the frequency of each letter. The frequency data is stored in a map and normalized such that the sum of frequencies is equal to one. The word with the highest normalized frequency sum, taking into account the number of repeated letters in the word, is then selected.
In summary, we have a two-step process here. First, we build a normalized letter-to-frequency map:

```haskell
charFrequencyMap :: [String] -> Map.Map Char Double -> Map.Map Char Double
charFrequencyMap all_words freq_map = Map.map (/ total_letter_count) letter_to_count
  where
    letter_to_count = Prelude.foldl (Prelude.foldl insert_op) freq_map all_words
    insert_op acc char = Map.insertWith (+) char 1.0 acc
    total_letter_count = 5 * fromIntegral (length all_words)
```

Next, we choose the next-best-guess by choosing a word with the highest frequency sum, taking into account of repeated letters:

```haskell
frequencyChoose :: [String] -> Map.Map Char Double -> String
frequencyChoose word_list freq_map = fst $ word_to_score !! 0
  where word_to_score = sortedWordScores word_list freq_map

sortedWordScores :: [String] -> Map.Map Char Double -> [(String, Double)]
sortedWordScores word_list freq_map = sortBy (\(_, a) (_, b) -> compare b a) word_scores
  where
    word_scores = Prelude.map (\word -> (word, wordScore word freq_map)) word_list

wordScore :: String -> Map.Map Char Double -> Double
wordScore word freq_map = score / repeat_letter_count
  where
    score = Prelude.foldl (\acc ch -> acc + (freq_map Map.! ch)) 0.0 word
    repeat_letter_count = 5 - fromIntegral (length $ nub word) + 1
```

The full code implementation can be found in `Heuristics/Frequency.hs`.

3.2 Minimax Heuristics

The Minimax strategy assumes that when selecting the next best guess, the Wordle Game will always return us with the "hardest" word. "Hard" in this context means that the word will prune the least amount of search space. In essence, we assume that our opponent, the Wordle engine, will always maximize the remaining search space, while we, the player, attempt to minimize the remaining search space. In practice, the Wordle engine does not always play optimally, but minimax provides a good estimation. I will use a depth limit of two as this seems sufficient to achieve good results in our search.
Here is the `minimize` function used to select our next best guess. It iterates through all valid words, and for each valid word, it invokes the `maximize` function that simulates the move of the Wordle engine. The `maximize` function returns the theoretical size of the remaining search space if the current word is selected. It selects the word that will minimize the remaining search space.

```haskell
minimize :: [String] -> Int -> String -> Int -> String
minimize [] _ min_word _ = min_word
minimize valid_words@(w:ws) beta min_word min_bucket_size =
  minimize ws beta' min_word' min_bucket_size'
  where max_bucket_size = maximize w valid_words beta Map.empty 0
        min_bucket_size' = min min_bucket_size max_bucket_size
        min_word' = if max_bucket_size < min_bucket_size then w else min_word
        beta' = min beta min_bucket_size'
```

Here is the `maximize` function used to simulate the move of the Wordle engine. It iterates through all valid words, and for each word, fetches the potential Wordle response. These responses are grouped into buckets, and each bucket can be seen as a potential remaining search space size. In the end, it returns the maximum bucket size to the `minimize` layer.
I further used alpha-beta pruning to optimize for efficiency. When the minimize layer has already found a bucket smaller than or equal to the current bucket size, then the function can return immediately instead of recursing further.

```haskell
maximize :: String -> [String] -> Int -> Map.Map String Int -> Int -> Int
maximize _ [] _ _ max_bucket_size = max_bucket_size
maximize player_guess (potential_answer:vws) beta pat_to_cnt max_bucket_size =
    let pattern = getWordleResponse player_guess potential_answer
        count = fromMaybe 0 $ Map.lookup pattern pat_to_cnt
        new_count = count + 1
        pat_to_cnt' = Map.insert pattern new_count pat_to_cnt
        max_bucket_size' = max new_count max_bucket_size
    in if max_bucket_size' >= beta then max_bucket_size'
       else maximize player_guess vws beta pat_to_cnt' max_bucket_size'
```

Please see `Heuristics/Minimax.hs` in the source code listing for more details.

3.3 Entropy Heuristics

When a player makes a guess, there are $3^5 = 243$ possible responses from the Wordle game, as there are 3 outcomes (grey, yellow, and green) for each letter and 5 letters for each word. For example, "2 yellows, followed by 1 grey, followed by 2 greens" is one possible response. We can visualize these 243 responses as a histogram with 243 buckets, one for each of the possible responses.
Intuitively, we want our next guess to fall into the bucket with the least frequency, as it prunes the most of our search space. The extreme case is the all-green-response bucket: the frequency is 1 and we immediately win. On the other hand, we don't want our next guess to land in a bucket with a large frequency. In order to avoid the worse-case scenario and strive for the best-case scenario, we can't rely purely on luck. The strategy is therefore to choose a word that leads to a uniform distribution, so that we get a guaranteed reduction of 1/N of the search space.
The uniformity measurement can be conducted via information entropy, namely the function

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b(p(x_i))$$

Words with greater entropy is selected, as greater entropy indicates greater uniformity.
Below is a set of functions implementing the Entropy heuristics. To optimize for efficiency, the full entropy calculation is estimated via a frequency map. The entry point is `entropyChooseMap`. Please see the code listing section for more details.

```
entropy :: (Eq a, Floating a) => [a] -> a
entropy dist = -sum [x * log2_safe x | x <- dist]
  where log2_safe x = if x == 0.0 then 0 else logBase 2 x

entropyChooseMap :: [String] -> [String] -> Map.Map Char [Int] -> String -> String
entropyChooseMap valid_words all_words freqListMap word_so_far =
    let diffEntropies = map (\word -> (word,
          getDiffEntropyForWord valid_words word 0 freqListMap Set.empty word_so_far 0))
          all_words
        max_word = foldr1 (\(w1, e1) (w2, e2) -> if e1 >= e2 then (w1, e1) else (w2, e2)) diffEntropies
    in fst max_word

getDiffEntropyForWord :: [String] -> String -> Int -> Map.Map Char [Int] -> Set Char ->
                         String -> Double -> Double
getDiffEntropyForWord valid_words word wordIdx freqListMap seenChars wordSoFar diffEntropy
...

buildFreqListMap :: [String] -> Map.Map Char [Int]
buildFreqListMap valid_words = Map.fromList [(ch, buildFreqList valid_words ch) | ch <- ['a'..'z']]
  where buildFreqList (w:ws) ch = zipWith (+) (buildFreqListWord w ch) (buildFreqList ws ch)
        buildFreqList [] _ = [0,0,0,0,0]
        buildFreqListWord word ch = [if x == ch then 1 else 0 | x <- word]
```

4 Sequential Performance Statistics

For the three heuristics described above, which one has the best performance in terms of quality? I answer this question by simulating 2,309 Wordle games, one for each of the possible valid words used by the New York Times Wordle game. Statistics is collected and two metrics are evaluated:

1. Out of all 2,309 games, which of the games did not finish in under six turns? Recall that for the official NYT Wordle game, solving the game in under six turns is constituted as a win.
2. Out of all 2,309 games, what is the average number of turns to solve a Wordle game? We want to take as little turns as possible.

I will now answer these two questions for each of the heuristics.

4.1 Parallelization in Data Collection

Before getting started, I first parallelized the data collection process, as playing 2,309 games is a lot. To speed this up, I parallelized the game initiation process in `Main.hs`. The implementation is simple. Instead of initiating each game via `Prelude.map (->playAndGetList "entropy" word validWords allWords) validWords`, I added `` `using` parList rdeepseq `` at the end. The full implementation can be found in the code listing section. In addition, all unprocessed data collected in this section can be found in the `data/sequential_data` folder.

4.2 Frequency Heuristics

On average, it took 3.832 turns to solve each Wordle game. There are 65 words that it could not solve in under 6 turns. Here is a summary of the statistics I collected:

```
Average turn length: 3.831745344304894
Games using > 6 turns:
['goner', 'hatch', 'cater', 'waste', 'parer', 'dowel', 'eater', 'fiber', 'amaze', 'graze', 'shape',
'caper', 'comfy', 'towel', 'brick', 'witty', 'cheer', 'proof', 'baler', 'wound', 'clank', 'corer',
'baste', 'bezel', 'batch', 'witch', 'punch', 'snore', 'vaunt', 'booby', 'ferry', 'watch', 'cinch',
'waste', 'catch', 'grave', 'woozy', 'swore', 'tatty', 'river', 'rover', 'jolly', 'cover', 'furry',
'fatty', 'graze', 'swarm', 'boxer', 'flank', 'racer', 'giver', 'piper', 'riper', 'filly', 'taste',
'rarer', 'fever', 'waver', 'cower', 'wound', 'corer', 'fully', 'willy', 'roger', 'rower']
```

4.3 Minimax Heuristics

On average, it took 3.825 turns to solve each Wordle game. There are 29 words that it could not solve in under 6 turns. As we can see, compared to the frequency heuristics we reduced our loss amount by more than 50% already! Here is a summary of the statistics I collected:

```
Average turn length: 3.8254655695106106
Games using > 6 turns:
['goner', 'hatch', 'cater', 'waste', 'parer', 'dowel', 'eater', 'fiber', 'amaze', 'graze', 'shape',
'caper', 'comfy', 'towel', 'brick', 'witty', 'cheer', 'proof', 'baler', 'wound', 'clank', 'corer',
'baste', 'bezel', 'batch', 'witch', 'punch', 'snore', 'vaunt']
```

4.4 Entropy Heuristics

On average, it took 3.615 turns to solve each Wordle game. There are 8 words that it could not solve in under 6 turns. The average number of turns taken is less than those taken when we utilized the frequency or minimax heuristics. Even better, we are now losing at a much lesser extent.

```
Average turn length: 3.6149848419229103
Games using > 6 turns:
['hilly', 'bound', 'wafer', 'corer', 'tight', 'vaunt', 'willy', 'match']
```

One caveat is that since doubles are involved in the heuristics calculations, the results obtained above may vary based on slight changes in implementation. For instance, in the sequential entropy heuristics implementation discussed above, I used the `map` function to obtain the entropy for each word, then used the `foldr` function to obtain the maximum entropy. If I were to implement the same function using recursion (see `Heuristics/Entropy.hs` in the code listing section for this implementation), then I would have obtained an average turn length of `3.652` and there would be NO game taking greater than six turns! In other words, the average number of turns would increase a bit, but we would win every game, which is amazing.

5 Parallelization

According to the results obtained in the last section, the entropy heuristics had the best performance in terms of solving wordle games, as the chance of losing the game is less and on average, and it also takes the least amount of turns to solve each game. I now discuss parallelization techniques to speed up the wordle solving process via the entropy heuristics.
Looking at the sequential Entropy heuristics implementation, there are two major parallelization opportunities. First, observe how in the main Entropy Heuristics loop, we are iterating through all valid words in order to calculate the entropy for each word. We can definitely perform the word-entropy heuristics calculation for all words in parallel. Note that I decided to not parallelize the entropy calculation for each word, as this would have too fine of a granularity (after all, there are only five letters in each Wordle word).
The second major aspect that we can parallelize is not necessarily related to the Entropy heuristics calculation itself, but rather the Wordle Player. Recall that after the player obtains a response by invoking `getWordleResponse`, it must filter out invalid words from the current list of valid words in preparation for the next round of guesses. Since it is possible to have a lot of words in the initial valid words list, we can optimize for speed by conducting this filtering step in parallel.

5.1 `Par` Monad with Chunks

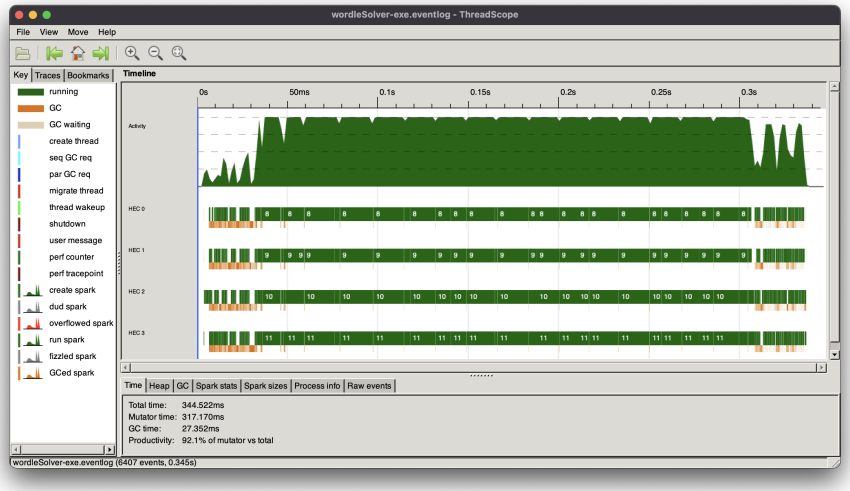
The first parallelization strategy utilizes the `Par` monad defined in `Control.Monad.Par` library. Instead of explicitly `fork`'ing new parallel tasks and `get`'ing them afterwards, I use the `parMap` function provided in the `Control.Monad.Par` library to achieve the same result. The program will first spawn one entropy calculation and one filtering task for each chunk. In the end, the solutions are concatenated. For entropy, I then select the word with the maximum entropy. It is worth nothing that `parMap` waits for all results before returning.
Here is the parallel version of the entropy calculation:

```
parMapChunk :: (NFData b) => Int -> (a -> b) -> [a] -> Par [b]
parMapChunk n f xs = fmap concat $ parMap (map f) $ chunk n xs
  where chunk _ [] = []
        chunk c_n c_xs = as : chunk c_n bs
          where (as, bs) = splitAt c_n c_xs

entropyChooseMapPar :: [String] -> [String] -> Map.Map Char [Int] -> String -> Int -> String
entropyChooseMapPar valid_words all_words freqListMap word_so_far entropy_chunk_sz =
    let diffEntropies = runPar $ parMapChunk entropy_chunk_sz (\word -> (word,
          getDiffEntropyForWord valid_words word 0 freqListMap Set.empty word_so_far 0))
          all_words
        max_word = foldr1 (\(w1, e1) (w2, e2) -> if e1 >= e2 then (w1, e1) else (w2, e2)) diffEntropies
    in fst max_word
```

Here is the parallel version of the filtering function:

```
parFilter :: (NFData a) => Int -> (a -> Bool) -> [a] -> Par [a]
parFilter n f xs = fmap concat $ parMap (filter f) $ chunk n xs
  where chunk _ [] = []
        chunk c_n c_xs = as : chunk c_n bs
          where (as, bs) = splitAt c_n c_xs

parFilterValidWords :: [String] -> String -> String -> Int -> [String]
parFilterValidWords valid_words query wordle_response chunk_sz =
  runPar $ parFilter chunk_sz (\word -> wordle_response == getWordleResponse query word) valid_words
```

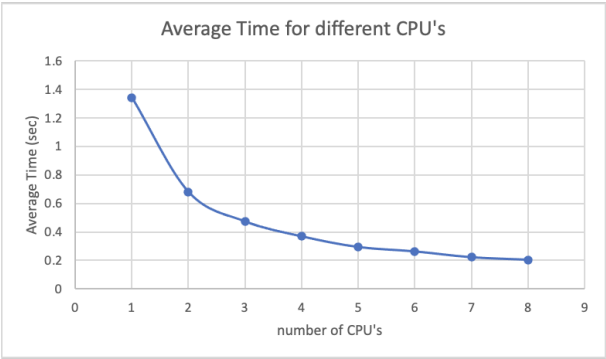
When running with four cores with an entropy chunk size of 30 and a fold chunk size of 50 on the word `baker`, I obtained the following result:

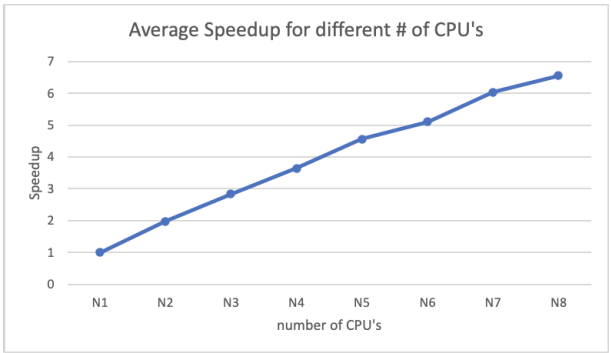


There is a good amount of parallelism involved. I further ran some experiments on a list of "hard words" (262 words that took greater than or equal to 5 turns to solve) and varied the core number. For each word and each core, I ran three trials, collected the elapsed time, and averaged the results. Filter chunk size and the entropy chunk size were both kept at 50 and 30, respectively.
Here are the results:

CPU	Average Time (s)	Average Speedup	Ideal Speedup with P = 0.97	P = (1-S) / (S/N - S)
N1	1.34	1	1	1
N2	0.68	1.97	1.94	0.99
N3	0.47	2.83	2.83	0.97
N4	0.37	3.64	3.67	0.97
N5	0.29	4.56	4.46	0.98
N6	0.26	5.10	5.22	0.96
N7	0.22	6.04	5.93	0.97
N8	0.21	6.55	6.61	0.97

According to the table above, the amount of speedup gradually decreases as the number of cores increases. This is expected, as we can never reach the ideal speedup according to Amdahl's Law. Indeed, when calculating the percentage of parallelization using the equation `P = (1 - S) / (S / N - S)`, where `S` represents the elapsed amount of time and `N` represents the number of cores used, the average percentage of parallelization obtained by using the `parMap` strategy is more than 97%! This is a huge improvement over the sequential implementation.
Below are two graphs visualizing the results obtained by the table above. One can see that as the number of cores increases, the average time to solve a Wordle Game decreases exponentially. The amount of speedup increases in a fairly linear fashion.





Note that all data can be found in `data/experiment-spreadsheet.xlsx`.

5.2 Evaluation Strategies `parListChunk`

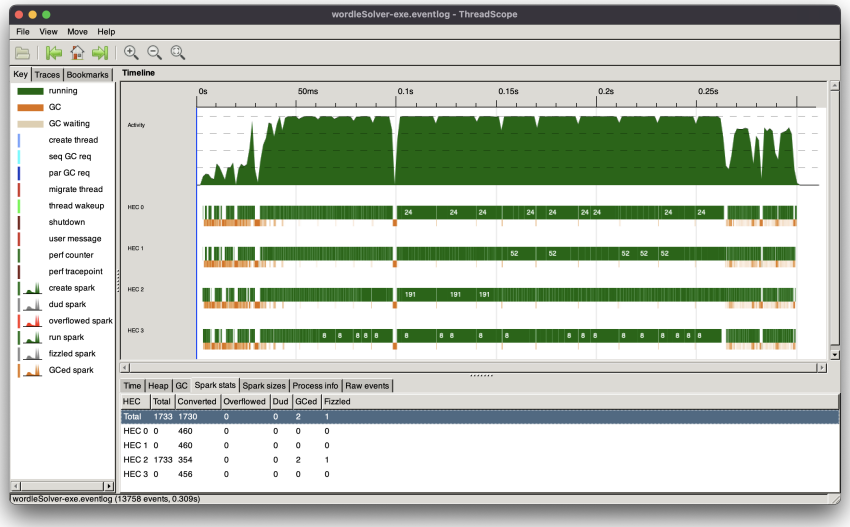
Instead of using the `Par` monad, another way to parallelize the entropy calculation and the filtering is by using evaluation strategies. Compared to the `Par` monad, there is a greater degree of code encapsulation. I will be using the `parListChunk` function and the `rdeepseq` strategy defined in `Control.Parallel.Strategies`.
Here is the parallel version of the entropy calculation:

```
entropyChooseMapChunkPar :: [String] -> [String] -> Map.Map Char [Int] -> String -> Int -> String
entropyChooseMapChunkPar valid_words all_words freqListMap word_so_far entropy_chunk_sz =
    let diffEntropies = map (\word -> (word,
          getDiffEntropyForWord valid_words word 0 freqListMap Set.empty word_so_far 0))
          all_words `using` parListChunk entropy_chunk_sz rdeepseq
        max_word = foldr1 (\(w1, e1) (w2, e2) -> if e1 >= e2 then (w1, e1) else (w2, e2)) diffEntropies
    in fst max_word
```

Here is the parallel version of the filtering function:

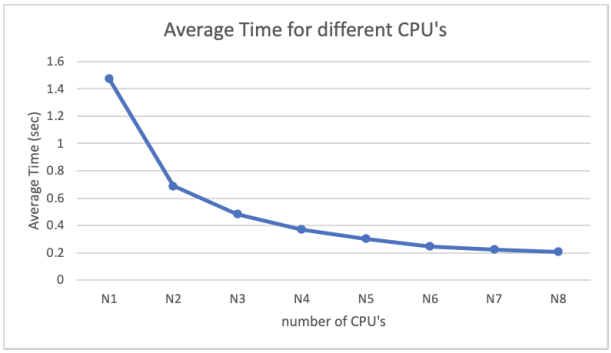
```
parFilterValidWordsChunk :: [String] -> String -> String -> Int -> [String]
parFilterValidWordsChunk valid_words query wordle_response chunk_sz=
  filter (\word -> wordle_response == getWordleResponse query word) valid_words
    `using` parListChunk chunk_sz rdeepseq
```

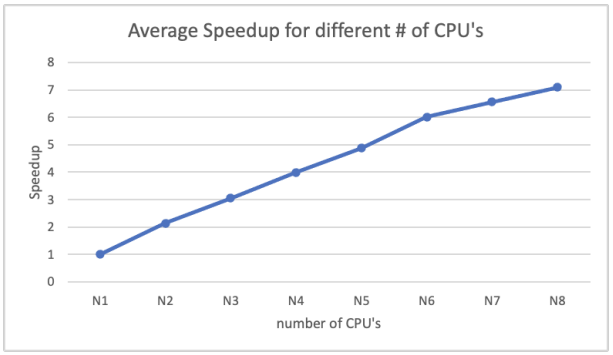
When running with four cores with an entropy chunk size of 30 and a fold chunk size of 50 on the word `baker`, we obtain the following result:



Similar to the performance of the `par` monad from the last section, there is also a great degree of parallelism here. We can further inspect on the spark stats in threadscope. By utilizing chunks to control the granularity of parallelism, there was no overflowed spark and the distribution of converted sparks is fairly evenly divided among all four cores. The same set of experiments is ran on this new parallelization strategy to test the amount of speedup with varying core numbers. 262 games with three trials for each were run. Filter chunk size and entropy chunk size were still kept at 50 and 30, respectively.

CPU	Average Time (s)	Average Speedup	Ideal Speedup with P = 0.97	P = (1-S) / (S/N - S)
N1	1.48	1	1	1
N2	0.69	2.15	1.98	1.07
N3	0.48	3.06	2.94	1.01
N4	0.37	3.99	3.88	1.00
N5	0.30	4.88	4.81	0.99
N6	0.25	6.01	5.71	1.00
N7	0.23	6.56	6.60	0.99
N8	0.21	7.09	7.48	0.98





The experimentation results were quite comparable to the Par monad method discussed in the last section. While the table and graph above may indicate a greater speed-up, if we compare the average runtime only, then we can see that these two parallelization strategies achieve mostly the same amount of reduction in time, with the percentage of ideal parallelizing hitting almost 100%.
Note that all raw data points are located at `data/experiment-spreadsheet.xlsx`.

5.3 Impact of Chunk Size

So far, I have kept the filter chunk size at 50 and the entropy chunk size at 30. Does varying the chunk size lead to further speedup? Is there an optimal chunk size. Intuitively, we know that the chunk sizes cannot be too small, as overflow of sparks will inevitably occur if this is the case:

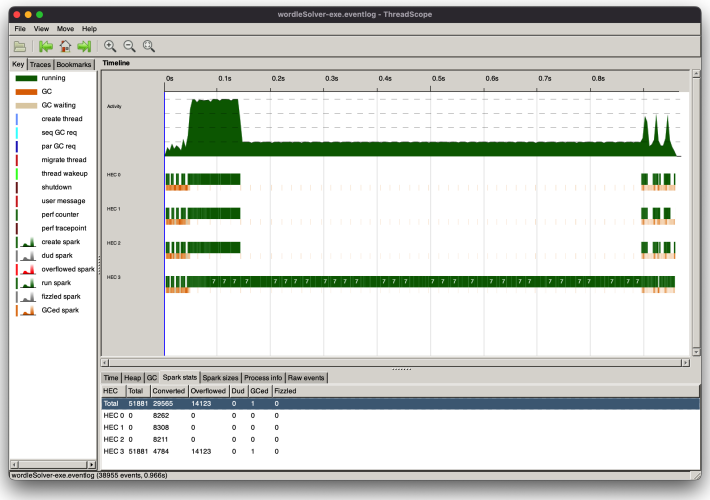


Figure 1: Solving baker with chunk size = 1 for both filtering and entropy calculation leads to a huge amount of spark overflow. In the following threadscope image, we can see that a total of 14,123 sparks overflowed, and most of the work was conducted on CPU 3 in a sequential manner.

At the same time, the chunk sizes cannot be too big, as if the goal of parallelization would be futile if that is the case: there would be essentially one spark for each entropy-calculation or filtering operation.

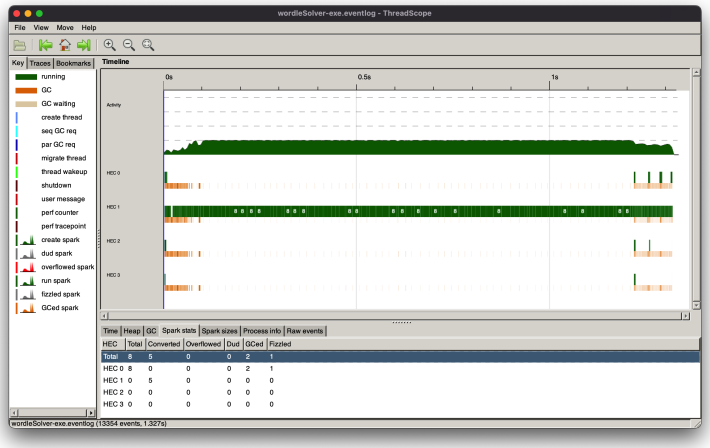
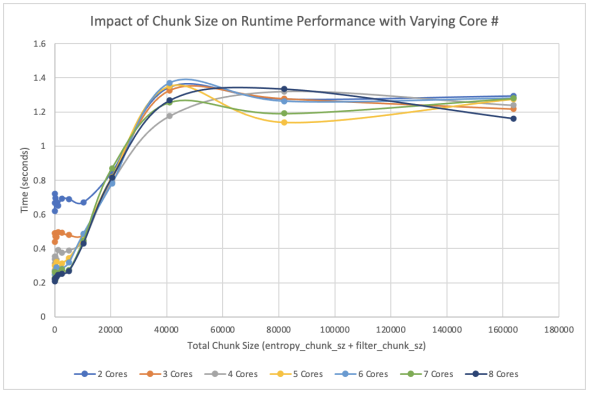


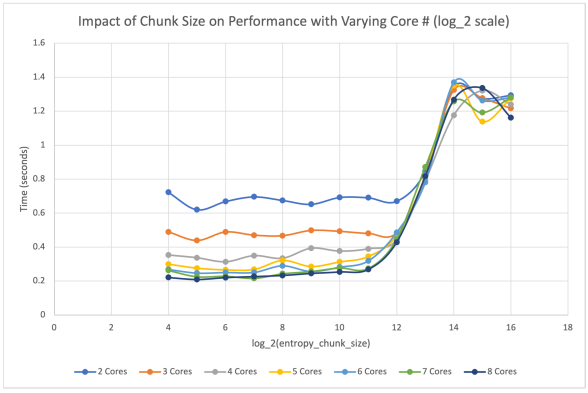
Figure 2: Solving baker with chunk size = 100000 for both filtering and entropy calculation leads to the creation of a few sparks and virtually zero parallelization.

To understand the impact of chunk size, I ran a set of experiments on the word `baker` using the 2nd parallel strategy (using `parListChunk` with `rdeepseq`). For each game, the filter chunk size is set to be 1.5 greater than the entropy chunk size. Entropy chunk sizes of 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, and 65536 were tested, with the corresponding filter chunk size of 24, 48, . . . , 98304. I ran 10 trials for each experiment and averaged the runtime.
Here are the results showing the impact of chunk size. Pink highlights show the best runtime for a specified core number.

Total Chunk Size	1 Core	2 Cores	3 Cores	4 Cores	5 Cores	6 Cores	7 Cores	8 Cores
40	1.35	0.72	0.49	0.35	0.30	0.27	0.26	0.22
80	1.30	0.62	0.44	0.34	0.28	0.25	0.23	0.21
160	1.18	0.67	0.49	0.31	0.27	0.25	0.23	0.22
320	1.37	0.70	0.47	0.35	0.27	0.25	0.22	0.23
640	1.19	0.67	0.47	0.33	0.32	0.29	0.24	0.23
1280	1.23	0.65	0.50	0.39	0.29	0.26	0.26	0.25
2560	1.27	0.69	0.49	0.38	0.31	0.28	0.28	0.25
5120	1.30	0.69	0.48	0.39	0.34	0.32	0.27	0.27
10240	1.33	0.67	0.48	0.44	0.47	0.49	0.45	0.43
20480	1.20	0.85	0.81	0.79	0.82	0.78	0.87	0.82
40960	1.22	1.34	1.33	1.18	1.35	1.37	1.26	1.27
81920	1.23	1.27	1.28	1.32	1.14	1.26	1.19	1.33
163840	1.18	1.29	1.22	1.24	1.27	1.28	1.28	1.16

If we draw a scatter plot of the data above (please see next page), it is clear that there is a certain threshold in which we should follow when setting the chunk size. In particular, beyond the 10240 total chunk size threshold, performance decreases drastically. Furthermore, if we "zoom in" by taking the log of the chunk size, a distinct pattern can be observed. In general, a total chunk size of 80 to 160 leads to the best performance (or entropy-to-filter chunk size combinations of 32-to-48 and 64-to-96). The minimum average time taken is also highlighted in the table above in pink.



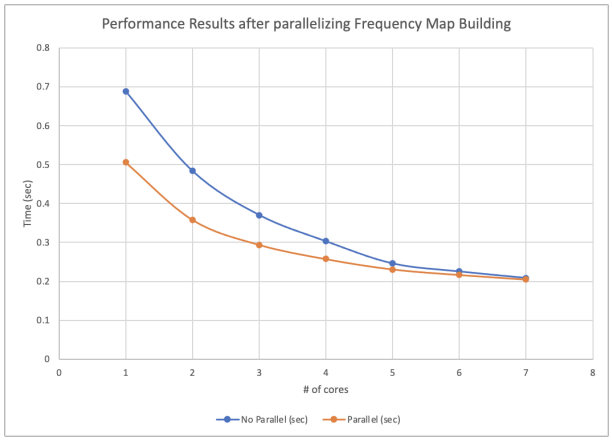


5.4 More Parallelization using the Par Monad

Taking another look at the Entropy Heuristics, another opportunity for optimization is to parallelize the frequency-map building process, specifically the `buildFreqListMap` function located in `Entropy.hs`. Given a list of valid words, this function returns the frequency of each character in a map. For each letter, it first calculates the frequency of that letter on each word in the list. It then zip these frequencies together and finally combine the frequency results for all letters a to z.
I will speed up this fold process by using a `spawnP`, `sequence`, and `mapM` get sequence. Unlike the previous approaches, chunks were not utilized since there are ultimately only 26 parallelization tasks, one for each letter.
Here is the parallel implementation:

```
buildFreqListMapPar :: [String] -> Map.Map Char [Int]
buildFreqListMapPar valid_words = Map.fromList [(ch, buildFreqList valid_words ch) | ch <- ['a'..'z']]
  where buildFreqList vw ch = runPar $ parAcc [0,0,0,0,0] [buildFreqListWord x ch | x <- vw]
        buildFreqListWord word ch = [if x == ch then 1 else 0 | x <- word]

        parAcc _ [x] = return x
        parAcc x xs = parAcc x $ runPar $ do
          let res = acc x xs
          seq_res <- T.sequence res
          T.mapM get seq_res

        acc x (y1 : y2 : ys) = spawnP (zipWith (+) y1 y2) : acc x ys
        acc x (y1 : _) = [spawnP (zipWith (+) y1 x)]
        acc _ [] = []
```

I then ran the same set of experiments as what I did for the `parListChunk` experiment, except for swapping the sequential version `buildFreqListMap` for the parallel version above. As shown in the graph below, the parallel version was indeed faster than the sequential version.



As a final note, one benefit of this Wordle game is that it can be easily extended to games with 5 or more letters. A much more noticeable speedup should be observed then. I have tried this and am omitting the experimental results for brevity. A list of words longer than 5 letter can be found in the data/dictionary folder.

6 Running Code

Below are some documentations on code execution.

6.1 Help

```
$ stack exec wordleSolver-exe -- -h True +RTS -ls -N5
Usage: wordleSolver [OPTION...]
  -i Bool    --interactive=Bool  Interactive mode
  -m String  --method=String     Heuristic method
  -p String  --parMethod=String  Parallel method
  -f Int     --filterChunk=Int   Filter chunk size
  -e Int     --entropyChunk=Int  Entropy chunk size
  -w String  --word=String       Guess word
  -h         --help              Show help
```

6.2 Default Behavior (no args)

- Initiate 2,309 Wordle games and print out the sequence of turns it took to reach the final answer
- Utilizes the entropy heuristics
- Utilizes the "parMap" parallelization strategy
- Utilizes filter_chunk_sz = 50 and entropy_chunk_sz = 30

```
$ stack exec wordleSolver-exe -- +RTS -ls -N8
"cigar"[3]: ["soare","lucid","cigar"]
"rebut"[3]: ["soare","putid","rebut"]
"sissy"[3]: ["soare","flint","sissy"]
...
```

6.3 Play 1 game at a time (-w)

Instead of playing all 2,309 games, we can choose to play only one game by specifying the -w option. For example, below is an example of playing one round of Wordle with the answer "baker".

```
$ stack exec wordleSolver-exe -- -w "baker" +RTS -ls -N5
["soare","clipt","gombo","deked","baker"]
```

6.4 Customizing Heuristics (-m)

While the default heuristics is Entropy, we can choose to use other heuristics. Here are three ways of solving the `prove` wordle game.

```
$ stack exec wordleSolver-exe -- -m "frequency" -w "prove" +RTS -ls -N5
["arose","drone","trope","probe","prove"]
$ stack exec wordleSolver-exe -- -m "minimax" -w "prove" +RTS -ls -N5
["raise","drone","probe","prove"]
$ stack exec wordleSolver-exe -- -m "entropy" -w "prove" +RTS -ls -N5
["soare","pwned","prove"]
```

6.5 Entropy Parallelization Params (`-p`, `-f`, `-e`)

We can further customize the Entropy Heuristics for it to use different parallelization strategies:

- `-p` specifies the parallelization strategy. Users can choose between "parList" and "parListChunk"
- `-f` specifies the filter chunk size. It's an integer.
- `-e` specifies the entropy chunk size. It's an integer. Recall that by default, if a user specifies none of the above, we automatically execute `-p parList -f 50 -e 30`. Here is an example of using the parListChunk strategy with filter size of 64 and entropy size of 64:

```
$ stack exec wordleSolver-exe -- -m "entropy" -p parListChunk -f 50 -e 50 -w "prove" +RTS -ls -N5
["soare","pwned","prove"]

$ stack exec wordleSolver-exe -- -m "entropy" -p parListChunk -f 50 -e 50  +RTS -ls -N5
"cigar"[3]: ["soare","lucid","cigar"]
"rebut"[3]: ["soare","putid","rebut"]
"sissy"[3]: ["soare","flint","sissy"]
...
```

6.6 Interactive Mode (`-i`)

```
$ stack exec wordleSolver-exe -- -i True +RTS -ls -N5
Please enter "soare" into NYT's Wordle game.
Enter NYT's response in 0/1/2 format:
...
```

7 Code Listing

7.1 app/Main.hs

```haskell
module Main (main) where

import Parser
import Player
import Interactive ( interactiveLoop )

import System.Environment (getArgs)
import Control.Parallel.Strategies (using, parList, rdeepseq)
import System.Console.ParseArgs ()
import System.Console.GetOpt
    ( getOpt,
      ArgDescr(NoArg, ReqArg),
      ArgOrder(Permute),
      OptDescr(..) )
import Text.Read (readMaybe)

main :: IO ()
main = do
  args <- getArgs
  opts <- wordleOpts args

  valid_words <- fileToWordList "data/answerlist.txt"
  all_words <- fileToWordList "data/wordlist.txt"

  case opts of
    (Flags {help = True}, _) -> putStrLn usageMsg
    (Flags {interactive = True}, _) -> interactiveLoop valid_words all_words
```

```
    (Flags {word = "", method = m, filter_chunk_sz = f_sz,
            entropy_chunk_sz = e_sz, par_method = pm}, _) -> do
      let arrEntropy = Prelude.map (\w ->
            playAndGetList m w valid_words all_words f_sz e_sz pm)
            valid_words `using` parList rdeepseq
      printGameRes arrEntropy valid_words
    (Flags {word = w, method = m, filter_chunk_sz = f_sz,
            entropy_chunk_sz = e_sz, par_method = pm}, _) -> do
      let answer = playAndGetList m w valid_words all_words f_sz e_sz pm
      print answer
      -- uncomment the following when doing performance testing
      -- _ <- playAndGetList m w valid_words all_words f_sz e_sz pm `seq` return ()
      -- return ()

data Flags = Flags
  {
    interactive :: Bool,
    method :: String,
    par_method :: String,
    filter_chunk_sz :: Int,
    entropy_chunk_sz :: Int,
    word :: String,
    help :: Bool
  } deriving Show

defaultFlags :: Flags
defaultFlags = Flags
  {
    interactive = False,
    method = "entropy",
    par_method = "parMap",
    filter_chunk_sz = 50,
    entropy_chunk_sz = 30,
    word = "",
    help = False
  }

flags :: [OptDescr (Flags -> Flags)]
flags =
  [ Option ['i'] ["interactive"] (ReqArg (\f opts ->
      opts { interactive = readInteractive f }) "Bool") "Interactive mode"
  , Option ['m'] ["method"] (ReqArg (\f opts ->
      opts { method = readMethod f }) "String") "Heuristic method"
  , Option ['p'] ["parMethod"] (ReqArg (\f opts ->
      opts { par_method = readParMethod f }) "String") "Parallel method"
  , Option ['f'] ["filterChunk"] (ReqArg (\f opts ->
      opts { filter_chunk_sz = readFilterChunkSz f }) "Int") "Filter chunk size"
  , Option ['e'] ["entropyChunk"] (ReqArg (\f opts ->
      opts { entropy_chunk_sz = readEntropyChunkSz f }) "Int") "Entropy chunk size"
  , Option ['w'] ["word"] (ReqArg (\f opts ->
      opts { word = f }) "String") "Guess word"
  , Option ['h'] ["help"] (NoArg (\opts ->
      opts { help = True })) "Show help"
  ]

readInteractive :: String -> Bool
readInteractive s = case readMaybe s of
```

```
  Just b -> b
  Nothing -> error "ERROR: unrecognized option -i [Bool]"

readMethod :: String -> String
readMethod s = case s of
  "entropy" -> "entropy"
  "frequency" -> "frequency"
  "minimax" -> "minimax"
  _ -> error "ERROR: unrecognized option -m [entropy | frequency | minimax]"

readParMethod :: String -> String
readParMethod s = case s of
  "seq" -> "seq"
  "parMap" -> "parMap"
  "parListChunk" -> "parListChunk"
  _ -> error "ERROR: unrecognized option -p [seq | parMap | parListChunk]"

readFilterChunkSz :: String -> Int
readFilterChunkSz s = case readMaybe s of
  Just i -> i
  Nothing -> error "ERROR: unrecognized option -f [Int]"

readEntropyChunkSz :: String -> Int
readEntropyChunkSz s = case readMaybe s of
  Just i -> i
  Nothing -> error "ERROR: unrecognized option -e [Int]"

wordleOpts :: [String] -> IO (Flags, [String])
wordleOpts args =
  case getOpt Permute flags args of
    (xs, n, []) -> return (foldl (flip id) defaultFlags xs, n)
    (_, _, errs) -> ioError (userError (concat errs ++ usageMsg))

usageMsg :: [Char]
usageMsg =
  "Usage: wordleSolver [OPTION...]" ++ "\n" ++
  "  -i Bool    --interactive=Bool  Interactive mode" ++ "\n" ++
  "  -m String  --method=String     Heuristic method" ++ "\n" ++
  "  -p String  --parMethod=String  Parallel method" ++ "\n" ++
  "  -f Int     --filterChunk=Int   Filter chunk size" ++ "\n" ++
  "  -e Int     --entropyChunk=Int  Entropy chunk size" ++ "\n" ++
  "  -w String  --word=String       Guess word" ++ "\n" ++
  "  -h         --help              Show help"
```

7.2 src/Heuristics/Frequency.hs

```
module Heuristics.Frequency (
  charFrequencyMap,
  frequencyChoose
) where

import Data.Map.Strict as Map
import Data.List (nub, sortBy)

-- given a list of words, return a normalized frequency map of the letters
charFrequencyMap :: [String] -> Map.Map Char Double -> Map.Map Char Double
charFrequencyMap all_words freq_map = Map.map (/ total_letter_count) letter_to_count
```

```
  where
    letter_to_count = Prelude.foldl (Prelude.foldl insert_op) freq_map all_words
    insert_op acc char = Map.insertWith (+) char 1.0 acc
    total_letter_count = 5 * fromIntegral (length all_words)

frequencyChoose :: [String] -> Map.Map Char Double -> String
frequencyChoose word_list freq_map = fst $ word_to_score !! 0
  where word_to_score = sortedWordScores word_list freq_map

sortedWordScores :: [String] -> Map.Map Char Double -> [(String, Double)]
sortedWordScores word_list freq_map = sortBy (\(_, a) (_, b) -> compare b a) word_scores
  where
    word_scores = Prelude.map (\word -> (word, wordScore word freq_map)) word_list

wordScore :: String -> Map.Map Char Double -> Double
wordScore word freq_map = score / repeat_letter_count
  where
    score = Prelude.foldl (\acc ch -> acc + (freq_map Map.! ch)) 0.0 word
    repeat_letter_count = 5 - fromIntegral (length $ nub word) + 1
```

7.3 src/Heuristics/Minimax.hs

```
module Heuristics.Minimax (
  minimaxChoose
) where

import qualified Data.Map as Map
import WordleGame ( getWordleResponse )
import Data.Maybe (fromMaybe)

minimaxChoose :: [String] -> String
minimaxChoose valid_words =
  if length valid_words <= 2 then head valid_words
  else minimize valid_words 1000000 "" 1000000

minimize :: [String] -> Int -> String -> Int -> String
minimize [] _ min_word _ = min_word
minimize valid_words@(w:ws) beta min_word min_bucket_size =
  minimize ws beta' min_word' min_bucket_size'
  where max_bucket_size = maximize w valid_words beta Map.empty 0
        min_bucket_size' = min min_bucket_size max_bucket_size
        min_word' = if max_bucket_size < min_bucket_size then w else min_word
        beta' = min beta min_bucket_size'

maximize :: String -> [String] -> Int -> Map.Map String Int -> Int -> Int
maximize _ [] _ _ max_bucket_size = max_bucket_size
maximize player_guess (potential_answer:vws) beta pat_to_cnt max_bucket_size =
    let pattern = getWordleResponse player_guess potential_answer
        count = fromMaybe 0 $ Map.lookup pattern pat_to_cnt
        new_count = count + 1
        pat_to_cnt' = Map.insert pattern new_count pat_to_cnt
        max_bucket_size' = max new_count max_bucket_size
    in if max_bucket_size' >= beta then max_bucket_size'
       else maximize player_guess vws beta pat_to_cnt' max_bucket_size'
```

7.4 src/Heuristics/Entropy.hs

```haskell
{-# OPTIONS_GHC -Wno-unused-top-binds #-}
module Heuristics.Entropy (
  entropyChoose,
) where
import qualified Data.Map as Map
import Data.Set as Set ( Set, insert, member, empty )
import Control.DeepSeq (NFData)
import Control.Monad.Par (Par, parMap, runPar, spawnP, get)
import Control.Parallel.Strategies (parListChunk, rdeepseq, using)
import qualified Data.Traversable as T

entropyChoose :: [String] -> [String] -> String -> Int -> String
entropyChoose valid_words all_words par_method entropy_chunk_sz
  | length valid_words <= 4 = head valid_words
  | par_method == "seq" =
      entropyChooseMap valid_words all_words freq_list_map word_so_far
  | par_method == "parMap" =
      entropyChooseMapPar valid_words all_words freq_list_map word_so_far entropy_chunk_sz
  | par_method == "parListChunk" =
      entropyChooseMapChunkPar valid_words all_words freq_list_map word_so_far entropy_chunk_sz
  | otherwise =
      entropyChooseMap valid_words all_words freq_list_map word_so_far
  where freq_list_map = buildFreqListMap valid_words
        -- freq_list_map = buildFreqListMapPar valid_words
        word_so_far = foldr1 (zipWith (\x y -> if x == y then x else '0')) valid_words

-- Sequential implementation
-- Version 1: recursive version
entropyChooseRec :: [String] -> [String] -> Map.Map Char [Int] -> String -> Double -> String -> String
entropyChooseRec _ [] _ _ _ best_word = best_word -- 2nd param is all_words
entropyChooseRec valid_words (w:ws) freqListMap word_so_far best_entropy best_word
  | length valid_words <= 2 = valid_words !! 0
  | otherwise =
    let diffEntropy = getDiffEntropyForWord valid_words w 0 freqListMap Set.empty word_so_far 0
    in if diffEntropy >= best_entropy
       then entropyChooseRec valid_words ws freqListMap word_so_far diffEntropy w
       else entropyChooseRec valid_words ws freqListMap word_so_far best_entropy best_word

-- Version 2: map & fold version
entropyChooseMap :: [String] -> [String] -> Map.Map Char [Int] -> String -> String
entropyChooseMap valid_words all_words freqListMap word_so_far =
    let diffEntropies = map (\word -> (word,
          getDiffEntropyForWord valid_words word 0 freqListMap Set.empty word_so_far 0))
          all_words
        max_word = foldr1 (\(w1, e1) (w2, e2) -> if e1 >= e2 then (w1, e1) else (w2, e2)) diffEntropies
    in fst max_word

-- Parallel version of entropyChooseMap
-- Method 1: parMap (Par monad)
parMapChunk :: (NFData b) => Int -> (a -> b) -> [a] -> Par [b]
parMapChunk n f xs = fmap concat $ parMap (map f) $ chunk n xs
  where chunk _ [] = []
        chunk c_n c_xs = as : chunk c_n bs
          where (as, bs) = splitAt c_n c_xs
```

```haskell
entropyChooseMapPar :: [String] -> [String] -> Map.Map Char [Int] -> String -> Int -> String
entropyChooseMapPar valid_words all_words freqListMap word_so_far entropy_chunk_sz =
    let diffEntropies = runPar $ parMapChunk entropy_chunk_sz (\word -> (word,
          getDiffEntropyForWord valid_words word 0 freqListMap Set.empty word_so_far 0))
          all_words
        max_word = foldr1 (\(w1, e1) (w2, e2) -> if e1 >= e2 then (w1, e1) else (w2, e2)) diffEntropies
    in fst max_word

-- Method 2: parListChunk
entropyChooseMapChunkPar :: [String] -> [String] -> Map.Map Char [Int] -> String -> Int -> String
entropyChooseMapChunkPar valid_words all_words freqListMap word_so_far entropy_chunk_sz =
    let diffEntropies = map (\word -> (word,
          getDiffEntropyForWord valid_words word 0 freqListMap Set.empty word_so_far 0))
          all_words `using` parListChunk entropy_chunk_sz rdeepseq
        max_word = foldr1 (\(w1, e1) (w2, e2) -> if e1 >= e2 then (w1, e1) else (w2, e2)) diffEntropies
    in fst max_word

getDiffEntropyForWord :: [String] -> String -> Int -> Map.Map Char [Int] ->
  Set Char -> String -> Double -> Double
getDiffEntropyForWord valid_words word wordIdx freqListMap seenChars wordSoFar diffEntropy
  | wordIdx >= 5 = diffEntropy
  | otherwise =
    let ch = word !! wordIdx
        greens = freqListMap Map.! ch !! wordIdx
        yellows = if ch `Set.member` seenChars
            then 0
            else let mask = [c /= ch | c <- wordSoFar]
                 in sum([freqListMap Map.! ch !! j | j <- [0..length wordSoFar - 1], mask !! j]) - greens
        seenChars' = Set.insert ch seenChars
        greys = length valid_words - greens - yellows
        dist = [greens, yellows, greys]
        dist' = [fromIntegral x / fromIntegral (sum dist) | x <- dist]
        diffEntropy' = diffEntropy + entropy dist'
    in getDiffEntropyForWord valid_words word (wordIdx + 1) freqListMap seenChars' wordSoFar diffEntropy'

entropy :: (Eq a, Floating a) => [a] -> a
entropy dist = -sum [x * log2_safe x | x <- dist]
  where log2_safe x = if x == 0.0 then 0 else logBase 2 x

-- sequential frequency map building
buildFreqListMap :: [String] -> Map.Map Char [Int]
buildFreqListMap valid_words = Map.fromList [(ch, buildFreqList valid_words ch) | ch <- ['a'..'z']]
  where buildFreqList (w:ws) ch = zipWith (+) (buildFreqListWord w ch) (buildFreqList ws ch)
        buildFreqList [] _ = [0,0,0,0,0]
        buildFreqListWord word ch = [if x == ch then 1 else 0 | x <- word]

-- parallel frequency map building
buildFreqListMapPar :: [String] -> Map.Map Char [Int]
buildFreqListMapPar valid_words = Map.fromList [(ch, buildFreqList valid_words ch) | ch <- ['a'..'z']]
  where buildFreqList vw ch = runPar $ parAcc [0,0,0,0,0] [buildFreqListWord x ch | x <- vw]
        buildFreqListWord word ch = [if x == ch then 1 else 0 | x <- word]

        parAcc _ [x] = return x
        parAcc x xs = parAcc x $ runPar $ do
          let res = acc x xs
          seq_res <- T.sequence res
          T.mapM get seq_res
```

```
        acc x (y1 : y2 : ys) = spawnP (zipWith (+) y1 y2) : acc x ys
        acc x (y1 : _) = [spawnP (zipWith (+) y1 x)]
        acc _ [] = []
```

7.5 src/Player.hs

```
module Player (
  playAndGetList,
  filterValidWords
) where
import qualified Data.Map.Strict as Map

import WordleGame ( getWordleResponse )
import Heuristics.Frequency ( charFrequencyMap, frequencyChoose )
import Heuristics.Entropy ( entropyChoose )
import Heuristics.Minimax ( minimaxChoose )
import Control.DeepSeq (NFData)
import Control.Monad.Par (Par, parMap, runPar)
import Control.Parallel.Strategies (parListChunk, rdeepseq, using)


playAndGetList :: String -> String -> [String] -> [String] -> Int -> Int -> String -> [String]
playAndGetList method answer valid_words all_words filter_chunk_sz entropy_chunk_sz par_method
  | method == "frequency" =
      playFreq answer valid_words freq_map []
  | method == "entropy" =
      playEntropy answer valid_words all_words [] filter_chunk_sz entropy_chunk_sz par_method
  | method == "minimax" =
      playMinimax answer valid_words (length valid_words) []
  | otherwise =
      error( "ERROR: Player.hs: playAndGetList: invalid method. " ++
             "Please use 'frequency' or 'entropy' or 'minimax'")
  where freq_map = charFrequencyMap all_words Map.empty

playFreq :: String -> [String] -> Map.Map Char Double -> [String] -> [String]
playFreq answer valid_words freq_map choices
  | guess == answer = reverse $ guess : choices
  | otherwise = playFreq answer new_valid_words freq_map (guess : choices)
  where new_valid_words = filterValidWords valid_words guess wordle_response
        guess = frequencyChoose valid_words freq_map
        wordle_response = getWordleResponse guess answer

playMinimax :: String -> [String] -> Int -> [String] -> [String]
playMinimax answer valid_words orig_valid_len choices
  | guess == answer = reverse $ guess : choices
  | otherwise = playMinimax answer new_valid_words orig_valid_len (guess : choices)
  where new_valid_words = filterValidWords valid_words guess wordle_response
        -- hardcode the first guess to speed things up
        guess = if length valid_words == orig_valid_len then "raise"
                else minimaxChoose valid_words
        wordle_response = getWordleResponse guess answer

playEntropy :: String -> [String] -> [String] -> [String] -> Int -> Int -> String -> [String]
playEntropy answer valid_words all_words choices filter_chunk_sz entropy_chunk_sz par_method
  | guess == answer = reverse $ guess : choices
  | otherwise =
```

```
      playEntropy answer new_valid_words all_words
        (guess : choices) filter_chunk_sz entropy_chunk_sz par_method
    where new_valid_words
            | par_method == "parListChunk" =
              parFilterValidWordsChunk valid_words guess wordle_response filter_chunk_sz
            | par_method == "parMap" =
              parFilterValidWords valid_words guess wordle_response filter_chunk_sz
            | par_method == "seq" =
              filterValidWords valid_words guess wordle_response
            | otherwise =
              error ("ERROR: Player.hs: playEntropy: invalid par_method. " ++
                      "Please use 'parMap' or 'parMapChunk'")
          guess = entropyChoose valid_words all_words par_method entropy_chunk_sz
          wordle_response = getWordleResponse guess answer

-- sequential filterValidWords
filterValidWords :: [String] -> String -> String -> [String]
filterValidWords valid_words query wordle_response =
  filter (\word -> wordle_response == getWordleResponse query word) valid_words

-- parallel filterValidWords
-- Method 1: use parMap (par monad)
parFilter :: (NFData a) => Int -> (a -> Bool) -> [a] -> Par [a]
parFilter n f xs = fmap concat $ parMap (filter f) $ chunk n xs
  where chunk _ [] = []
        chunk c_n c_xs = as : chunk c_n bs
          where (as, bs) = splitAt c_n c_xs

parFilterValidWords :: [String] -> String -> String -> Int -> [String]
parFilterValidWords valid_words query wordle_response chunk_sz =
  runPar $ parFilter chunk_sz (\word -> wordle_response == getWordleResponse query word) valid_words

-- Method 2: use parListChunk
parFilterValidWordsChunk :: [String] -> String -> String -> Int -> [String]
parFilterValidWordsChunk valid_words query wordle_response chunk_sz=
  filter (\word -> wordle_response == getWordleResponse query word) valid_words
    `using` parListChunk chunk_sz rdeepseq
```

7.6 src/WordleGame.hs

```
module WordleGame (
  getWordleResponse
) where

import Data.Map.Strict as Map

-- Given a user's guess and the correct answer, return the Wordle response
-- in the form of 0s, 1s, 2s, representing grey, yellow, and green, respectively.
getWordleResponse :: String -> String -> String
getWordleResponse query answer = response
  where letter_to_count = Map.fromListWith (+) $ Prelude.map (\x -> (x, 1)) answer
        green_and_grey = fillGreenAndGrey query answer letter_to_count
        -- Update letter_to_count map
        letter_to_count_updated = update_map letter_to_count green_and_grey query
        update_map ltc [] _ = ltc
        update_map ltc _ [] = ltc
        update_map ltc (x:xs) (q:qs) = if x == '2' then
```

```
          update_map (Map.adjust (\c -> c - 1) q ltc) xs qs else update_map ltc xs qs
        -- Fill in the yellow letters
        response = fillYellow query letter_to_count_updated green_and_grey

fillGreenAndGrey :: String -> String -> Map.Map Char Int -> [Char]
fillGreenAndGrey [] _ _ = []
fillGreenAndGrey _ [] _ = []
fillGreenAndGrey (q:qs) (a:as) ltc
  | q == a = '2' : fillGreenAndGrey qs as (Map.adjust (\x -> x - 1) a ltc)
  | otherwise = '0' : fillGreenAndGrey qs as ltc

fillYellow :: String -> Map.Map Char Int -> String -> [Char]
fillYellow [] _ _= []
fillYellow _ _ [] = []
fillYellow (q:qs) ltc (r:rs) | q `Map.member` ltc && r /= '2' && Map.findWithDefault 0 q ltc > 0
                               = '1' : fillYellow qs (Map.adjust (\x -> x - 1) q ltc) rs
                             | otherwise = r : fillYellow qs ltc rs
```

7.7 src/Parser.hs

```
module Parser (
  fileToWordList,
  printGameRes
) where

import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC
import GHC.IO (catchAny)
import System.IO (openFile)
import GHC.IO.IOMode (IOMode(ReadMode))
import System.Exit (die)

-- Process File Input
fileToWordList :: FilePath -> IO [String]
fileToWordList path = do
  handle <- catchAny (openFile path ReadMode) $ \_ -> do
    die $ "Could not open file: " ++ path
  contents <- B.hGetContents handle
  return $ Prelude.map BC.unpack $ BC.lines contents

-- Prints out the result of wordle games
-- Example output for one of the results:
-- "gorge"[4]: ["soare","rurus","forge","gorge"]
printGameRes :: [[String]] -> [String] -> IO ()
printGameRes [] _ = return ()
printGameRes _ [] = return ()
printGameRes (x:xs) (y:ys) = do
  putStr $ (show y) ++ "[" ++ (show $ length x) ++ "]: " ++ (show x) ++ "\n"
  printGameRes xs ys
```

7.8 src/Interactive.hs

```
-- An fun interactive player that can help you solve your NYT Wordle game of the day!
module Interactive (
  interactiveLoop,
) where
```

```haskell
import Heuristics.Entropy (entropyChoose)
import Player (filterValidWords)

-- take user input in a loop
interactiveLoop :: [String] -> [String] -> IO ()
interactiveLoop valid_words all_words = do
  let guess_word = entropyChoose valid_words all_words "parMap" 50
  putStrLn ("Please enter \"" ++ guess_word ++ "\" into NYT's Wordle game.")
  -- 0 = grey, 1 = yellow, 2 = green
  putStrLn "Enter NYT's response in 0/1/2 format:"
  wordle_response <- getLine
  if not (isValidWordleResponse wordle_response)
    then putStrLn "Invalid response. Please try again." >> interactiveLoop valid_words all_words
    else
      let new_valid_words = filterValidWords valid_words guess_word wordle_response
       in case new_valid_words of
            [word] -> putStrLn ("Type \"" ++ word ++ "\" to win!")
            [] -> putStrLn "No valid words left. You win!"
            _ ->
              case wordle_response of
                "22222" -> putStrLn "No valid words left. You win!"
                _ -> interactiveLoop new_valid_words all_words

isValidWordleResponse :: String -> Bool
isValidWordleResponse response = length response == 5 && all (`elem` "012") response
```

7.9 scripts/

There are some Python and bash scripts located in the `scripts/` folder. I used them to collect data for the experiments in this report.

7.10 data/

All data collected in this experiment are located in the `data/` folder. Here are some notable folder/files and their descriptions:

- `experiment-spreadsheet.xlsx` Contains processed data and graphs for all experiments described above
- `sequential_data/` Contains raw output for the heuristics experiment described in section 4. Sequential Performance Statistics
- `dictionary/` Although this folder was not explicitly used in the experiments described above, it contains English words that are of longer than the length of 5. I have used them to experiment with Wordle game-solving and explored their performance. I did not include the results in this report for brevity.
- `answerlist.txt` A list of 2,309 five-letter words that the NYT Wordle game uses as answers to its daily game
- `wordlist.txt` A list of 12,947 five-letter words that the NYT Wordle game accepts
- `hard_words.txt` A list of 262 five-letter words that took ¿= 5 turns to solve using the entropy heuristics