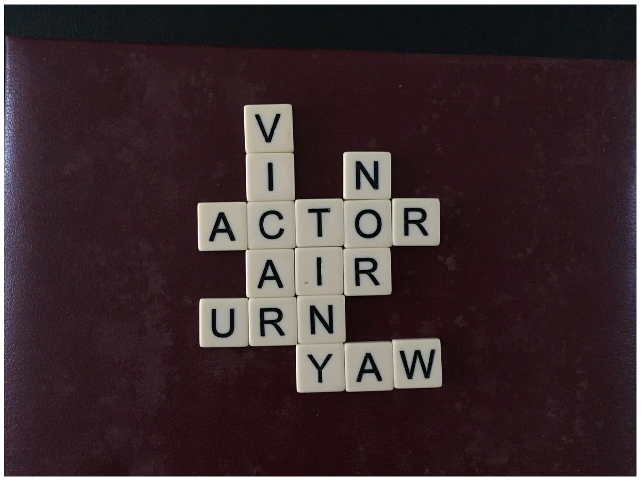
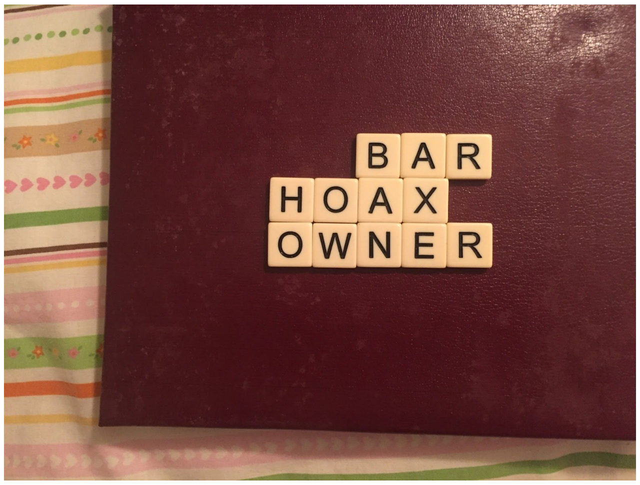
Minimal Area Bananagrams

Nathaniel Saxe





Problem Statement

Bananagrams is a game where you take letter tiles and make configurations similar to Scrabble boards where all the rows and columns must contain valid words. Players race to incorporate all tiles into their board and draw new tiles from a central pile. Whoever first makes a valid configuration after the central pile is completely exhausted wins the game.

As a daily puzzle for myself, I used to pick some tiles at random out of the bag and try to arrange them into the most dense possible configuration. It took me about 15-20 minutes to find the above two arrangements. In this project, I wrote a program to find these arrangements for me.

To be more precise, the program takes as input an unordered list of *n* letter tiles and a list of *w* legal words, and searches the space of Bananagrams configurations using those tiles for one that is valid and has minimal bounding box area.

As examples, the above left configuration has a bounding box of 5x6 = 30 and the right configuration has a bounding box of 3x5 = 15.

Algorithm

The problem is a messy one full of constraints. Given the scope of the project, I decided not to improve the algorithm asymptotically above brute force, instead using a simple state space search with some clever optimizations to improve speed. The basic idea is to place a word and some information about the word onto a stack, then recurse on every word that can be placed on the new board state, until all tiles have been used. The overall structure of the program looks like this:

```
-get input word file and letter tiles
-filter the words down to those which can be constructed with
the given letter tiles
-findMinimumAreaConfiguration()

function findMinimumAreaConfiguration():
     -pull from stack
     -place a preassigned word onto the grid in some position
     if(out of tiles):
          all tiles used, so this is a solution
          if the solution is valid:
               if the solution has bounding box area
               less than or equal to the current minimum:
                    -this solution is the new minimum
          -pop top off stack
          -return
     for each possible word placement in the
     current board:
          -push that word onto the stack
          -findMinimumAreaConfiguration()
     -pop top off stack
     -return
```

By itself, this algorithm would simply scan over all possible sequences of word placements and run in time $O(v^d)$ where v is the number of valid words which can be made from the letter tiles and d is the average recursion depth (the number of words you must place on the board on average before running out of tiles). Worst case, every word in the dictionary can be made with the letter tiles, and all words are 1 character long, so the worst case runtime is $O(w^n)$

As stated before, the acutal algorithm used does not improve upon this asymptotic runtime but uses heuristics to narrow the state space and generally speed things up. In particular, it uses several preemptive checks on the solution before deciding whether to recurse, as follows:

-It is guaranteed that once a word is placed, the bounding box area of the configuration cannot decrease below its old value (either it stays the same or increases). Therefore, if the bounding box area is above the minimum known area at any point, pop from the stack and return early.

-While the algorithm is guaranteed to try each ordered sequence of word placements exactly once, many of these may accidentally create the exact same board configuration repeatedly. Each board state encountered is therefore hashed and checked to see if it has been encountered before before continuing. This creates the possibility that some correct solutions will be skipped because of hash collisions, but the probability of this is quite slight as long as good hashing is used.

-After all tiles have been used up, a function called checkValidBananagrams() is called on the board to see if all rows and columns consist of valid words as per the rules. If this function is also called on every intermediate step of the recursion, it will skip over many configurations that are unlikely to lead to valid solutions with more word placements. It turns out that adding this check drastically increases the search speed of the algorithm. However, with this preemptive checking the search will only discover a solution if it has the property that if can be constructed word by word and at every intermediate step is a valid bananagrams configuration, which is not guaranteed in general and especially not for the very densest configurations which we are trying to find. However, it has held true for every solution I've found without preemptive checking. I therefore included as a command line argument -c.

Another optimization used was reordering the list before searching through it. Interestingly, sorting the list of valid words by shortest first and by longest first both led to solutions faster than leaving it in alphabetical order. This is because with longest words first the algorithm would use more tiles per layer of recursion and the recursion depth was lower, so it reached finished configurations faster. With shortest words first the algorithm had higher average recursion depth, but was also placing many short combinations of letters down and so was iterating through more diverse boards arrangements faster and discovering compact solutions sooner.