BananaSolve: Bananagrams Solver Project Proposal

Shai Goldman (stg2126), Aaron Priven (ahp2154)

1 Overview

Context Bananagrams is an entertaining scabble-like game in which the players are all given a set number of letter tiles and asked to construct a totally connected scrabble-like board using all of their tiles. Whoever finishes their board fastest wins. (In the real game, after a player completes a board, all players draw a new tile and have to incorporate it into their boards, until all tiles are gone, but for our project we will be focusing on the initial board creation stage of the game.) See Figure 1.

Project Goal Our goal is to write a Haskell algorithm that, given a set number of tiles, will construct a valid bananagrams board using those tiles. The algorithm will return the first valid board it finds. We will then speed up the program using parallel strategies.

Challenges Banagrams is a relatively new game. It has been around since 2006. For this reason it lacks well-known sequential solving algorithms. An algorithm written by college students in python has been of help to us (linked here).

2 Algorithms and Strategies

A Sequential Algorithm Our sequential algorithm plays one word onto the board at a time until all tiles have been used. The best word to play is assessed using a heuristic based

Figure 1: A Bananagrams Ad

on letter frequencies and word length. In general, longer words that use more uncommon letters are given priority.

One key restriction to simplify and speed up the algorithm is to only play words that connect with a single letter on a single existing word on the board. For example, the board in Figure 1 does not conform to this restriction: "Quart" touches "Mean" at three letters, each of which are configured to spell different words in the crossword. By restricting our algorithm to use only single-point connections, we avoid the need to verify that the newly added word does not create an invalid word in another part of the board. Since each word it plays will not affect other parts of the board, the board always remains valid after adding a valid word.

A pseudo-code algorithm for our technique can be described as follows:

1. Given a dictionary d and a hand of letters h:
2. Go through d once and record the length of each word.
3. Let k be the length of the longest word in the dictionary.
4. Repeat the following in a BFS-based algorithm (alternating orientations of horizontal and vertical) until one strand of the BFS succeeds (i.e. no more letters in h, or some limit is reached:
   (a) For all words of length k, filter out words from d that can't be made from letters in h.
   (b) Find each available space on the board that can be used to add a new word.
   (c) For each available space, add the highest scoring word available in a new BFS-strand. (Words that intersect with other words are considered invalid.) Subtract the letters of that word from h and reset k to the length of the longest word in d. If no words could be placed on the board, decrement k and go back to (a).

Parallel Opportunities. We will try to find the best depth in the BFS at which to parallelize. We suspect the best place to parallelize may be at the first word placement on the board (Step 2 in the algorithm above).