BananaSolve: a Parallel Haskell Bananagrams Solver

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. 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 some help to us, but we have veered significantly from their specific implementation (linked here).

![Figure 1: A Bananagrams Ad](image-url)
2 Algorithm

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 on letter frequencies and word length. In general, longer words that use more uncommon letters are given priority. A pseudo-code algorithm for our technique can be described as follows:

1. Given a dictionary $D$ and a hand of letters $h$:
2. Group $D$ into sub-dictionaries ($d : ds$) each containing all words of a particular word-size. Let $s$ be the wordsize of $d$, the sub-dictionary with the longest words.
3. Start BFS, using the empty board as a first BFS-node.
4. For each BFS-node, take a BFS step:
   (a) Find each available space on the board that can be used to add a new word.
   (b) For each available space, add the highest scoring word in $d$ that can be played with letters in $h$ at that space that does not invalidate the board in a new BFS-strand. Subtract the letters of that word from $h$ and reset $s$ to the length of the longest word in $d$. If no words could be placed on the board, decrement $s$ and go back to (a) with the head of $ds$.
5. If any new BFS-nodes contain a completed board (i.e., $h$ is empty), return that and halt.
6. Sort all the new BFS nodes by score. Score is calculated based on total tiles played and letter frequency values of tiles played, with more tiles and less frequently used letters scoring more. Pick the highest $n$ scoring BFS nodes to run a new BFS step on, where $n$ can be determined by the user.
7. Return to step 4. with the new BFS-nodes chosen in 6., and repeat until some limit is reached or a solution is found.

3 Some Implementation Details

3.1 Important Custom Types

BWord BWord is made up of the word’s string, starting coordinates, and direction (i.e. horizontal or vertical).

data BWord = BWord String Location Direction

OMatrix OMatrix is made up of the matrix’s origin and a CharMatrix. Keeping track of the ’virtual’ origin’s location abstracts the underlying matrix and allows for easier management of the matrix.

data OMatrix = OMatrix Location CharMatrix
Figure 2: Our algorithm’s solution for input "howareyousounbelievablyquickatbananagrams"

**Board**  Board stores an OMatrix with the list of BWords it uses.

```haskell
data Board = Board [BWord] OMatrix
```

**State**  State stores a `Board` with a `Hand` of letters yet to be played where `Hand` is a Hashmap of letters with their respective counts in the player’s hand.

```haskell
type State = (Hand, Board)
```

### 3.2 Choosing Next Word

Choosing the next word uses the algorithms in src/WordChooser.hs to pick the highest scoring word that can be played with `Hand h` at a specific tile on the board. The scoring is done based on wordlength and letter frequency. We created a `Map.Map Char Int letterPoints` to assign points to each letter based on that letter’s frequency in the English language, where the least frequent letter scores the most points. In our `letterPoints`, based on autogenerated values from chatGPT, ‘z’ scores 14926, while ‘e’ scores 2298.

The scoring algorithm is as follows:

```haskell
scoreWord :: String -> Int
scoreWord w = 15000 * length w
           + sum (map scoreChar w)

scoreChar :: Char -> Int
scoreChar ' ' = 0
scoreChar c = fromJust $ Map.lookup c letterPoints
```

### 3.3 Other Details

The code as a whole can be found in the appendix, as well as on [github](https://github.com/username/repo.git).
4 Parallel

4.1 A Test Case For Parallelization.

For our parallelization journey, we began testing with the test input “howareyousounbelievablyquickatbananagrams”. This input produced a board under the sequential algorithm in around 7.2s. The board produced is pictured in Figure 2.

4.2 Parallel BFS-Step Computation

Our first idea for parallelization was to calculate each BFS-node’s children in parallel. For example, at BFS step $x$ with 100 nodes, create a spark to calculate each node’s children.

At first we tried this with the strategy `parPar`, but this did not effect speedup. The reason was that `parPar` only goes to WHN form, and for each BFS-node, we wanted to calculate a list of its children, and WHN form for this would only actually evaluate the first child, and leave the rest as a thunk.

Therefore we switched to using the `rdeepseq` strategy, and added implementations for `instance NFData` for all our custom datatypes. This was successful. The new implementation for going from one bfs step to the next required minimal changes. We implemented it in `src/Bfs.hs` by modifying `bfsNext` to `bfsNextPar`, which ran as such:

```haskell
bfsNextPar :: DictPair -> [State] -> [State]
bfsNextPar dictpair states = concat
  $ parMap rdeepseq (playTurn dictpair) states
```

The new runtime was around 2.2s, around a 3.3x speedup from the sequential algorithm on this input. Threadscope logs are shown in Figure 3.

---

1 When run the first time, it can take up to 13s, but if run again its much faster, averaging around 7.2s. I assume this is because some of the dictionary is kept in the cache after the first run.
There was still room for improvement. Looking at the threadscope logs, we could see serious gaps of inactivity on each core. This could be caused by uneven load balancing. For example, if some of the BFS-nodes were much harder than others (for example, boards with more tiles already played require more possible new plays and more computations), one core could be given all of the harder nodes to complete.

### 4.3 Parallel Within-Step Computation

We decided to try to optimize our parallel speedup further by introducing parallelism to within BFS-node computations. Each BFS-node begins with a State containing a Hand $h$ and a Board $b$, and needs to find all open tiles on the $b$, and calculate the best possible word play with $h$ at each spot. So we can parallelize this by calculating the best possible play for each open tile in parallel.

To implement this, we changed our playTurn function in src/Bfs.hs to playTurnPar, where:

```haskell
playTurnPar :: DictPair -> State -> [State]
playTurnPar dictpair state@(_, board) =
    catMaybes $ parMap rdeepseq (playBestWordAt dictpair state)
    $ getOpenTiles board
```

The runtime for this improved algorithm was between 2.0 and 2.1s on average, which calculates to around a 3.5x speedup from the sequential algorithm. Threadscope logs, shown in Figure 4, show how core inactivities were significantly reduced in this algorithm.

One caveat is that we noticed our new algorithm resulted in a lot of fizzled sparks. Fizzled sparks are supposed to happen with a spark finds that the thunk it was supposed to evaluate was already evaluated by another part of the program. As of the due date for this project we have not been able to discover the reason for this problem in our code, as we do not think we have multiple sparks running on the same thunks.
4.4 Speed Up Results For Other Test-Cases

<table>
<thead>
<tr>
<th>Input</th>
<th>Sequential Time</th>
<th>Parallel Time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>icandothisonequickly</td>
<td>1.7s</td>
<td>0.8s</td>
<td>2.26x</td>
</tr>
<tr>
<td>howareyousogoodatpuzzlesitistrulyastounding</td>
<td>11.65s</td>
<td>3.52s</td>
<td>3.31x</td>
</tr>
<tr>
<td>whydoesntanyonewanttoplaybananagramwithmeiguessbeingthisgooodisntsofunafterall</td>
<td>59.69s</td>
<td>19.90s</td>
<td>2.99x</td>
</tr>
<tr>
<td>acomputerwoulddeservetobecalledintelligentgendifitcoulddeceiveahumanintobelievingthatitwashuman</td>
<td>76.99s</td>
<td>27.62s</td>
<td>2.79x</td>
</tr>
</tbody>
</table>

5 Appendix - Code

5.1 Main.hs

```haskell
module Main (main) where

import Bfs (bfsPar, bfsSeq)
import Data.Set (fromList)
import Types (splitDict)
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import Data.Maybe (isNothing, fromJust)
import Data.Char (isAlpha, toLower)

usage :: IO ()
usage = do
  pn <- getProgName
  die $ "$ Usage: stack exec " ++ pn ++ " -- +RTS -ls -N4 -- <algo> <tiles >\n"
    ++
    "<algo> must be 's' for sequential or 'p' for parallel. Tiles must be letters only."

main :: IO ()
main = do
  args <- getArgs
  case args of
    [_, algo, tiles] -> do
      let algoType = case algo of "s" -> Just bfsSeq
                                  "p" -> Just bfsPar
                                  _ -> Nothing
          if isNothing algoType then usage
          else do
            if any (not . isAlpha) tiles then usage
            else do
              let formattedTiles = map toLower tiles
              fcontents <- readFile "words.txt"
              let ws = lines fcontents
```

```haskell
```
dictlist = splitDict ws
dictset = Data.Set.fromList ws
putStrLn $ "Prompt: " ++ formattedTiles
let lim = 20
    stepsize = 20
res = fromJust algoType formattedTiles lim
    stepsize (dictset, dictlist)
    case res of
        Nothing -> putStrLn $ "no solution in " ++
        show lim
        Just state -> print state
    _ -> usage

5.2 Types.hs

module Types (StringSet, StringLists, splitDict, CharMatrix, Hand, Direction (H, V), flipD, Location (..), BWord (..), OMatrix (..), setElemOMatrix, Board (..), boardID, State, stateID, DictPair)
where

import Data.Set (Set)
import Data.HashMap.Strict (HashMap)
import Data.Matrix (Matrix, toList, setElem)
import Control.DeepSeq (NFData(..))
import Data.List (groupBy, sortBy)

type StringSet = Set String

type StringLists = [[String]]

splitDict :: [String] -> StringLists
splitDict dict = groupBy lengthEq $ sortBy lengthCmp dict
    where lengthCmp x y = length y `compare` length x
          lengthEq x y = length x == length y

type Hand = HashMap Char Int

data Direction = H|V deriving (Eq, Show) -- horizontal or vertical
flipD :: Direction -> Direction
flipD H = V
flipD V = H
instance NFData Direction where
    rnf d = d `seq` ()
data Location = Location Int Int deriving (Eq)
instance Show Location where
    show (Location y x) = show (y,x)
instance NFData Location where
    rnf (Location y x) = rnf y `seq` rnf x
type CharMatrix = Matrix Char
-- OMatrix stores the 'virtual' origin with a CharMatrix to allow
-- for easier usage of the CharMatrix.
data OMatrix = OMatrix Location CharMatrix
setElemOMatrix :: Char -> Location -> OMatrix -> OMatrix
    setElemOMatrix c (Location y x) (OMatrix p m) = OMatrix p new_m
        where new_m = setElem c (y,x) m
instance Show OMatrix where
    show (OMatrix p m) = show m ++ "\n" ++ show p
instance NFData OMatrix where
    rnf om = om `seq` ()
data BWord = BWord String Location Direction
    deriving (Eq, Show)
instance NFData BWord where
    rnf (BWord word p d) = rnf word `seq` rnf p `seq` rnf d
data Board = Board [BWord] OMatrix
instance Show Board where
    show (Board bwords om) = "bwords: " ++ show bwords ++ "\n"
        ++ show om
boardID :: Board -> String
boardID (Board _ (OMatrix _ m)) = toList m
instance NFData Board where
    rnf (Board bwords om) = rnf bwords `seq` rnf om
type State = (Hand, Board)
stateID :: State -> String
stateID (_, board) = boardID board
type DictPair = (StringSet, StringLists)

5.3 BananaBoard.hs

module BananaBoard (
    singleton, getSpaceAt, joinWordAt, isValidBoard
import Types (
    Direction (..),
    flipD,
    Location (..),
    OMatrix (..),
    setElemOMatrix,
    BWord (..),
    Board (..),
    StringSet)

import Data.Set (member)
import Data.Maybe (fromMaybe)
import Data.Matrix
    ( (<->), (<|>), fromLists, matrix,
    safeGet, Matrix(..), toLists, transpose )

empty :: Int -> Int -> Matrix Char
empty y x = matrix y x (\(_, _) -> ' ')

singleton :: String -> Board
singleton word = Board 
    [ BWord word ( Location 1 1) H] 
    ( OMatrix ( Location 1 1) 
    ( fromLists [word]))

-- add origin offset to coords
addO1 :: Num a => a -> a -> a
addO1 c c0 = c+c0-1
addO :: Location -> Location -> Location
addO ( Location y x) ( Location y0 x0) = Location ( addO1 y y0) ( addO1 x x0)

placeWord :: String -> Location -> Direction -> OMatrix -> OMatrix
placeWord word p@( Location y x) d om
| d == H = placeWordH word ( Location y x) sizedOM
| otherwise = placeWordV word ( Location y x) sizedOM

where
    endP = if d == H then Location y (x+length word-1)
            else Location (y+length word-1) x
    sizedOM = resizeTo endP (resizeTo p om)

placeWordH :: String -> Location -> OMatrix -> OMatrix
placeWordH [] _ m = m
placeWordH (w:ws) _p@(Location _y _x) _om@(OMatrix og _) =
    placeWordH ws (Location _y (_x+1)) $ setElemOMatrix w (add0
    _p og) _om

placeWordV :: String -> Location -> OMatrix -> OMatrix
placeWordV [] _ m = m
placeWordV (w:ws) _p@(Location _y _x) _om@(OMatrix og _) =
    placeWordV ws (Location (_y+1) _x) $ setElemOMatrix w (add0 _p
    og) _om

resizeTo :: Location -> OMatrix -> OMatrix
resizeTo _p@(Location _y _x) _om@(OMatrix og@(Location y0 x0) m) 
| yo < 1 = let yoff = 1 + abs yo in
getSpaceAt :: Location -> Int -> Direction -> OMatrix -> String
getSpaceAt p len d (OMatrix og m)
  | d == H = map getElemX $ take len [xo..]
  | otherwise = map getElemY $ take len [yo..]
where
  (Location yo xo) = addO p og
  getElemX :: Int -> Char
  getElemX x = fromMaybe ' ' $ safeGet yo x m
  getElemY :: Int -> Char
  getElemY y = fromMaybe ' ' $ safeGet y xo m

-- on top of a wordspace on a board, can we play this word?
validPlay :: String -> String -> Bool
validPlay [] _ = True
validPlay _ [] = False
validPlay (space:ss) (word:ws)
  | space == ' ' = validPlay ss ws
  | otherwise = space == word && validPlay ss ws

-- is this play both valid and also meaningful?
goodPlay :: String -> String -> Bool
goodPlay wordspace word =
  ', 'elem' wordspace && validPlay wordspace word

isValidBoard :: StringSet -> Board -> Bool
isValidBoard dict (Board _ (OMatrix _ m)) =
  areValidRows (toList m)
  && areValidRows (toList $ transpose m)
where
  isValidRow :: String -> Bool
  isValidRow row = all ('member' dict) $
    filter (\w -> length w /= 1) (words row)
  areValidRows :: [String] -> Bool
  areValidRows = all isValidRow

joinWordAt :: StringSet -> String -> Int -> BWord -> Int -> Board -> Maybe
          (Board, String)
joinWordAt \( \text{dictset s s\_ind \ (BWord \_ \ (Location \ y \ x) \ d) \ bw\_ind \ (\text{Board bwords} \ \text{om})} \)
| goodPlay boardspace s && \text{isValidBoard \ dictset \ newboard =}
| \text{Just \ (newboard, \ boardspace)}
| \text{otherwise = Nothing}
\text{where}
\text{new\_d = flipD d}
\text{boardspace = getSpaceAt p \ (length s) \ new\_d \ om}
\text{om\_new = placeWord s p new\_d \ om}
\text{newboard = Board \ (BWord s p \ new\_d:bwords) \ om\_new}
\text{p}
| \text{d == V = Location \ (y + bw\_ind) \ (x - s\_ind)}
| \text{otherwise = Location \ (y - s\_ind) \ (x + bw\_ind)}

5.4 Hand.hs

\text{module \ Hand \ (}
\text{toHand,}
\text{joinHands,}
\text{playTile,}
\text{addTile}
\text{) \ where}
\text{import \ Types \ (Hand)}
\text{import \ Data.HashMap.Strict \ (fromList, \ unionWith, \ update, \ alter)}
\text{import \ Data.List \ (group, \ sort)}
\text{toHand :: \ String -> \ Hand}
\text{toHand \ hand = fromList \$ \ map \ (\ \_ \ -> \ (head \ s, \ length \ s))}
\text{\$ \ (group . \ sort) \ hand}
\text{joinHands :: \ Hand -> \ Hand -> \ Hand}
\text{joinHands = unionWith (+)}
\text{playTile :: \ Char -> \ Hand -> \ Hand}
\text{playTile = update \ dec}
\text{where \ dec :: \ Int -> \ Maybe \ Int}
\text{dec 1 = \ Nothing}
\text{dec \ n = \ Just \ (n-1)}
\text{addTile :: \ Char -> \ Hand -> \ Hand}
\text{addTile = alter \ inc}
\text{where \ inc :: \ Maybe \ Int -> \ Maybe \ Int}
\text{inc \ Nothing = \ Just \ 1}
\text{inc \ (Just \ n) = \ Just \ (n+1)}

5.5 WordChooser.hs

\text{module \ WordChooser \ (}
\text{Hand,}
\text{buildWords,}
scoreCmp, sortWHPairs, wordsWithChar, scoreWord

where

import Data.List (sortBy)
import Data.Maybe (fromJust, mapMaybe)
import Data.HashMap.Strict (member)
import qualified Data.Map as Map
import Types (Hand)
import Hand (playTile)

-- Define a map between letters and their inverted usage frequencies generated by ChatGPT.
letterPoints :: Map.Map Char Int
letterPoints = Map.fromList
[( 'a' ,6833),( 'b' ,13508),( 'c' ,12218),( 'd' ,10747),( 'e' ,2298),( 'f' ,12772),
  ( 'g' ,12985),( 'h' ,8906),( 'i' ,8034),( 'j' ,14847),( 'k' ,14228),( 'l' ,10975),
  ( 'm' ,12594),( 'n' ,8251),( 'o' ,7493),( 'p' ,13071),( 'q' ,14905),( 'r' ,9013),
  ( 's' ,8673),( 't' ,5944),( 'u' ,12242),( 'v' ,14022),( 'w' ,12640),( 'x' ,14850),
  ( 'y' ,13026),( 'z' ,14926)]

buildWord :: String -> Hand -> Maybe Hand
buildWord [] hand = Just hand
buildWord (w:ws) hand
  | null hand || not (member w hand) = Nothing
  | otherwise = buildWord ws $ playTile w hand

buildWords :: Hand -> [String] -> [(String, Hand)]
buildWords hand = mapMaybe bw_pair
  where
    bw_pair [] = Nothing
    bw_pair word = case buildWord word hand of
      Nothing -> Nothing
      Just _hand -> Just (word, _hand)

scoreCmp :: String -> String -> Ordering
-- flip x and y so sort puts highest scorers first
scoreCmp x y = scoreWord y 'compare' scoreWord x

scoreWord :: String -> Int
scoreWord w = 15000 * length w + sum (map scoreChar w)

scoreChar :: Char -> Int
scoreChar ' ' = 0
scoreChar c = fromJust $ Map.lookup c letterPoints

sortWHPairs :: [(String, Hand)] -> [(String, Hand)]
sortWHPairs = sortBy (\x y -> scoreCmp (fst x) (fst y))
wordsWithChar :: Char -> [String] -> [String]
wordsWithChar c = filter (elem c)

5.6  Bfs.hs

module Bfs (playFirstTurn, bfsSeq, bfsPar)
where

import Data.Char (isAlpha)
import Data.Maybe ((fromJust, isJust, catMaybes, mapMaybe)
import Data.List (elemIndex, nubBy, sortBy)
import Control.Parallel.Strategies (parMap, rdeepseq)
import BananaBoard (singleton, joinWordAt)
import Hand (joinHands, playTile, addTile, toHand)
import WordChooser (buildWords, scoreCmp, sortWHPairs, wordsWithChar)
import Types (Hand, StringLists, DictPair, Board(..), BWord(..), State, stateID)

playFirstTurn :: Hand -> StringLists -> [State]
playFirstTurn _ [] = []
playFirstTurn hand (d:ds)
  | null bests = playFirstTurn hand ds
  | otherwise =
    map (\(w, h) -> (h, singleton w)) bests
    where
      bests = sortWHPairs $ buildWords hand d

playBestWordAt :: DictPair -> State -> (BWord, Int) -> Maybe State
playBestWordAt (_, []) _ _ = Nothing
playBestWordAt (dictset, d:ds) s@(hand, board) (bword@(BWord word _, _), i)
  | isJust best = best
  | otherwise =
    map (\(w, h) -> (h, singleton w)) bests
    where
      bests = sortWHPairs $ buildWords hand d
otherwise = playBestWordAt (dictset, ds) s (bword, i)
where
  c = word !! i
  bests = sortWHPairs $ buildWords (addTile c hand) $ wordsWithChar c d
  best = joinBestWord bests

joinBestWord :: [(String, Hand)] -> Maybe State
joinBestWord [] = Nothing
joinBestWord ((w, h):xs)
  | isJust res = res
  | otherwise = joinBestWord xs
where
  w_ind = fromJust (elemIndex c w)
  joinRes = joinWordAt dictset w w_ind bword i board
  res = do
    (newboard, playedOverSpace) <- joinRes
    let newhand = playTile c $ joinHands h $ (toHand . filter isAlpha) playedOverSpace
    return (newhand, newboard)

getOpenTiles :: Board -> [(BWord, Int)]
getOpenTiles (Board bwords _) = [(word, i) | word@(BWord s _ _) <- bwords, i <- [0..length s - 1]]

playTurnSeq :: DictPair -> State -> [State]
playTurnSeq dictpair state@(_, board) =
  mapMaybe (playBestWordAt dictpair state) openTiles
  where openTiles = getOpenTiles board

playTurnPar :: DictPair -> State -> [State]
playTurnPar dictpair state@(_, board) =
  catMaybes $ parMap rdeepseq (playBestWordAt dictpair state) $ getOpenTiles board

uniqueStates :: [State] -> [State]
uniqueStates = nubBy (\x y -> stateID x == stateID y)

bestStates :: Int -> [State] -> [State]
bestStates stepsize states = take stepsize $ sortBy scoreCmpState states
  where scoreCmpState :: State -> State -> Ordering
    scoreCmpState x y = scoreCmp (lettersOf x) (lettersOf y)
    lettersOf :: State -> String
    lettersOf state = filter isAlpha $ stateID state

bfsNextSeq :: DictPair -> [State] -> [State]
bfsNextSeq dictpair states = do
  state <- states
  playTurnSeq dictpair state

bfsNextPar :: DictPair -> [State] -> [State]
bfsNextPar dictpair states = concat $ parMap rdeepseq (playTurnPar dictpair) states

bfsLoop :: (DictPair -> [State] -> [State]) -> Int -> Int -> DictPair -> [State] -> Maybe State
bfsLoop _ 0 _ _ _ = Nothing
bfsLoop _ _ _ [] = Nothing
bfsLoop bfsNexter lim stepsize dictpair beginStates
  | isJust solved = solved
  | otherwise = next
  where
    solved = completeFrom beginStates
    completeFrom :: [State] -> Maybe State
    completeFrom [] = Nothing
    completeFrom (s@(hand, _):ss)
      | null hand = Just s
      | otherwise = completeFrom ss
    next = bfsLoop bfsNexter (lim - 1) stepsize dictpair $ (bestStates stepsize . uniqueStates . bfsNexter dictpair) beginStates

runBfs :: (DictPair -> [State] -> [State]) -> String -> Int -> Int -> DictPair -> Maybe State
runBfs f handstring lim stepsize d@(_, dictlist) =
  bfsLoop f lim stepsize d $ playFirstTurn (toHand handstring) dictlist

bfsSeq :: String -> Int -> Int -> DictPair -> Maybe State
bfsSeq = runBfs bfsNextSeq
bfsPar :: String -> Int -> Int -> DictPair -> Maybe State
bfsPar = runBfs bfsNextPar