1 Introduction

The core of the geofencing problem is searching through a set of boundaries to find which subset contains a query point. Inspired by how Uber deals with the geofencing problem, we aim to solve geospatial problems parallelly in Haskell by building an R-tree structure.

2 Implementation

Data Parsing and Preprocessing
We used two open-source datasets, one containing polygons outlining boundaries for all countries in the world, and another containing polygons outlining 4000+ states over the world. In order to handle JSON format data, we used the Aeson library which is the most widely used library for parsing JSON. We wrote the following modules for data parsing and preprocessing.

GeoJSONParser.hs
All the polygon data we downloaded is in the GeoJSON geospatial data interchange standard. It has ‘type’ and ‘features’ fields on the outermost
layer of JSON data and for each feature, it has ‘type’, ‘properties’, and
‘geometry’ fields. So we created a data type ‘GeoJSONFeatureCollection’
and ‘GeoJSONFeature’ along with corresponding FromJSON instances. We
opted to derive generic FromJSON instances with a customized field label
modifier.

{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}

module GeoJSONParser (
    parseFeatureCollection,
    GeoJSONFeatureCollection (..),
    GeoJSONFeature (..)
) where

import qualified Data.ByteString.Lazy as B
import GHC.Generics
import Data.Aeson
import Data.Char (toLower)
import qualified Data.Map.Strict as Map
import Geometry (Geometry)

parseFeatureCollection :: B.ByteString -> Maybe GeoJSONFeatureCollection
parseFeatureCollection = decode

data GeoJSONFeatureCollection =
    GeoJSONFeatureCollection { fcType :: String
    , fcFeatures :: [GeoJSONFeature]
    } deriving (Show, Generic)

instance FromJSON GeoJSONFeatureCollection where
    parseJSON = genericParseJSON defaultOptions {
        fieldLabelModifier = defaultFieldLabelModifier 
    }
data GeoJSONFeature =
  GeoJSONFeature { ftType :: String
                   , ftProperties :: Map.Map String Value
                   , ftGeometry :: Geometry
               } deriving (Show, Generic)

instance FromJSON GeoJSONFeature where
  parseJSON = genericParseJSON defaultOptions {
    fieldLabelModifier = defaultFieldLabelModifier }

defaultFieldLabelModifier :: String -> String
defaultFieldLabelModifier = map toLower . drop 2

Geometry.hs

Geometry data consists of ‘type’ and ‘coordinates’ attributes. It could be
‘Polygon’ or ‘MultiPolygon’ type and its ‘coordinates’ attribute describes
one or more polygons as a list of linear rings. The first element in the list
represents the exterior ring and any subsequent elements represent interior
rings (or holes). Each linear ring is composed of a list of points on the
map. In order to correctly parse geometry data and eliminate any possibil-
ity of malformed input, we wrote the datatype ‘GeoError’ and ‘Geometry’.
In ‘GeoError’, we defined several potential formatting issues for geometry
data, including ‘ClockwiseOuterRing’ or ‘LineStringNotClosed’, etc. Then
we defined a custom FromJSON instances for ‘Geometry’ instead of deriving
a generic instance because the mapping is not straightforward. For point
queries, we used the winding number algorithm to check whether a polygon
or multipolygon contains a point.

{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE NamedFieldPuns #-}

module Geometry (Geometry (..), LinearRing (..), GeoError, containsP, fromLineString, Point) where

import Data.Aeson
import BoundingBox (BoundingBox(..), Boundable, getBoundingBox, enlarge)
import GHC.Generics (Generic)
import Control.DeepSeq

data GeoError =
    ClockwiseOuterRing { badRing :: LinearRing }
  | CounterClockwiseInnerRing
  | LineStringTooShort
  | LineStringNotClosed
  | UnknownGeometryType

instance Show GeoError where
  show ClockwiseOuterRing { badRing } =
    "Polygon has invalid clockwise outer ring: " ++ show badRing
  show CounterClockwiseInnerRing =
    "Polygon has invalid counterclockwise inner ring(s)."
  show LineStringTooShort = "LineString too short."
data Geometry =
    Polygon { pOuterRing :: LinearRing
             , pInnerRings :: [LinearRing] }
  | MultiPolygon { mPolygons :: [Geometry] }
  deriving (Show, Eq, Generic)

instance NFData Geometry

instance Boundable Geometry where
    getBoundingBox Polygon { pOuterRing } = getBoundingBox pOuterRing
    getBoundingBox MultiPolygon { mPolygons } = foldl1 enlarge $
                                                                                 map getBoundingBox mPolygons

instance FromJSON Geometry where
    parseJSON = withObject "Geometry" $ \obj -> do
        _type <- obj .: "type"
        case _type of
            String "Polygon" ->
                do linearRings <- obj .: "coordinates"
                   return $ unwrap $ fromLinearRings linearRings
            String "MultiPolygon" ->
                do linearRingsList <- obj .: "coordinates"
                   let polygons = fromLinearRings <$> linearRingsList
                       mPolygons = fmap unwrap polygons
                   return $ MultiPolygon { mPolygons = fmap unmap polygons }
            _ -> error $ show UnknownGeometryType

unwrap :: Show a => Either a b -> b
unwrap (Left e) = error $ show e
unwrap (Right p) = p
A linear ring MUST follow the right-hand rule with respect to the area it bounds, i.e., exterior rings are counterclockwise, and holes are clockwise.

fromLinearRings :: [LinearRing] -> Either GeoError Geometry
fromLinearRings rings
    | isClockwise outerRing = Left $ ClockwiseOuterRing { badRing = outerRing }
    | anyCounterClockwise innerRings = Left CounterClockwiseInnerRing
    | otherwise = Right $ Polygon { pOuterRing = outerRing
                                         , pInnerRings = innerRings }

    where outerRing = head rings
          innerRings = tail rings
          anyCounterClockwise = any (not . isClockwise)

isClockwise :: LinearRing -> Bool
isClockwise = (> 0) . sum . map transformEdge . makeEdges . getLineString
          where transformEdge ((x1, y1), (x2, y2)) = (x2 - x1) * (y2 + y1)
                  makeEdges = zip <$> id <*> tail

-- Check whether a polygon contains a point using winding number algo
windNum :: LinearRing -> Point -> Bool
windNum rs (x, y) = (/= zero) . sum $ map checkOneEdge edges

    where zero = 0 :: Int
          edges = makeEdges $ getLineString rs
          makeEdges ls = zip ls (tail ls)
          isLeft (x1, y1) (x2, y2)
              | y1 < y2 = crossProduct > 0
              | y1 > y2 = crossProduct < 0
              | otherwise = False
          where crossProduct = (((x2 - x1) * (y - y1))
                               - ((x - x1) * (y2 - y1))

          checkOneEdge (p1@(_, y1), p2@(_, y2))
              | y1 <= y && y2 > y && isLeft p1 p2 = 1
containsP :: Point -> Geometry -> Bool
containsP p (Polygon {pOuterRing}) = windNum pOuterRing p
containsP p (MultiPolygon {mPolygons}) = any (containsP p) mPolygons

newtype LinearRing = LinearRing { getLineString :: LineString }
                    deriving (Show, Eq, Generic)

instance NFData LinearRing

instance Boundable LinearRing where
  getBoundingBox LinearRing { getLineString }
    | minX > maxX || minY > maxY = error "Invalid BoundingBox"
    | otherwise = BoundingBox minX minY maxX maxY
    where minX = minimum $ xs
          maxX = maximum $ xs
          minY = minimum $ ys
          maxY = maximum $ ys
          xs = map fst getLineString
          ys = map snd getLineString

instance FromJSON LinearRing where
  parseJSON jsn = do
    ls <- parseJSON jsn
    return $ unwrap $ fromLineString ls

  -- A linear ring is a closed LineString with four or more positions.

fromLineString :: LineString -> Either GeoError LinearRing
fromLineString ls
  | length ls < 4 = Left LineStringTooShort
  | not $ isClosedLineString ls = Left LineStringNotClosed
| otherwise = Right $ LinearRing ls

isClosedLineString :: LineString -> Bool
isClosedLineString ls
    | [] <- ls = True
    | [[]] <- ls = True
    | [x, y] <- ls, x /= y = False
    | [x, y] <- ls, x == y = True
    | x:_:rest <- ls = isClosedLineString (x:rest)

type LineString = [Point]

type Point = (Double, Double)

Entities.hs
Since our data contains both countries and states information and we plan to build an R-tree using all of those geometry data, we decided to give the parsed data a common type, ‘Entity’. Each Entity data could be either a country or a state along with its name, admin and geometry data. After parsing GeoFeature data from the original JSON file, ‘parseCountries’ and ‘parseStates’ would extract geometry and particular attributes from ‘GeoJSONFeatureCollection’ data to generate a list of Entities.

{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE DeriveGeneric #-}

module Entities (  
    Entity,  
    parseStates,  
    parseCountries,  

containsPoint,
buildEntityWithGeo
)

import Geometry
import GeoJSONParser ( GeoJSONFeatureCollection(..)
, GeoJSONFeature(..)
)
import qualified Data.Map.Strict as Map
import Data.Aeson.Types (Value, Value (String))
import qualified Data.Text as T
import BoundingBox (area, Boundable, getBoundingBox)
import GHC.Generics (Generic)
import Control.DeepSeq

data Entity =
  Country { cGeometry :: Geometry
   , cName :: String
   , cAdmin :: String }
| State { sGeometry :: Geometry
   , sName :: Maybe String
   , sAdmin :: String } deriving (Eq, Generic)

instance NFData Entity

instance Ord Entity where
e1 `compare` e2 = a1 `compare` a2
  where a1 = area $ getBoundingBox e1
        a2 = area $ getBoundingBox e2

instance Show Entity where
  show Country { cName } = "Country{ " ++ show cName ++ " }
  show State { sName } = "State{ " ++ show sName ++ " }"
instance Boundable Entity where
    getBoundingBox Country { cGeometry } = getBoundingBox cGeometry
    getBoundingBox State { sGeometry } = getBoundingBox sGeometry

parseCountries :: GeoJSONFeatureCollection -> Maybe [Entity]
parseCountries = mapM featureToCountry . fcFeatures

featureToCountry :: GeoJSONFeature -> Maybe Entity
featureToCountry GeoJSONFeature { ftProperties, ftGeometry } = do
    name <- extractText <$> Map.lookup "NAME" ftProperties
    admin <- extractText <$> Map.lookup "ADMIN" ftProperties
    return $ Country { cGeometry = ftGeometry
        , cName = name
        , cAdmin = admin
    }

parseStates :: GeoJSONFeatureCollection -> Maybe [Entity]
parseStates = mapM featureToState . fcFeatures

featureToState :: GeoJSONFeature -> Maybe Entity
featureToState GeoJSONFeature { ftProperties, ftGeometry } = do
    name <- extractMaybeText <$> Map.lookup "name" ftProperties
    admin <- extractText <$> Map.lookup "admin" ftProperties
    return $ State { sGeometry = ftGeometry
        , sName = name
        , sAdmin = admin
    }

extractText :: Value -> String
extractText (String t) = T.unpack t
extractText _ = error "not text"
R-tree Implementation

Instead of using Uber’s two-level hierarchy model, we implement an R-tree data structure to index polygons based on containment, the node at the root of a subtree spatially contains nodes below it. To build the R-tree, we use a bounding box for each polygon which is defined by the minimum and maximum coordinates to generate sequences of input entities. Searching the R-tree for which polygon’s bounding boxes contain a point improves time complexity from $O(n)$ to $O(\log Mn)$ where $M$ is the user-defined constant of the maximum children a node can have. Followings are the modules we created for implementing the R-tree structure.

BoundingBox.hs

BoundingBox is composed of 2 (long, lat) coordinates, representing the bottom left and top right corners of the rectangle. And we used BoundingBox to generate ordered sequences of Entities we parsed from datasets to build R-tree. We should be able to get a bounding box for any geometry element based on the minimum and maximum coordinates of constituent coordinates. Therefore, we defined a type class ‘Boundable’ which has just the function
‘getBoundingBox’ which returns a bounding box for that element. Then we implemented ‘Boundable’ instances for both Geometry and Entity data type by calculating the maximum and minimum longitude and latitude coordinates among the list of points. We also added some other helper functions between bounding boxes. ‘Enlarge’ function returns the smallest bounding box that contains 2 bounding boxes supplied as input. ‘Area’ computes the area of a bounding box, ‘containsPoint’ checks whether a given point falls within a bounding box.

{-# LANGUAGE DeriveGeneric #-}

module BoundingBox where

import Data.List (intersperse)
import GHC.Generics (Generic)
import Control.DeepSeq

data BoundingBox = BoundingBox { x1 :: !Double
, y1 :: !Double
, x2 :: !Double
, y2 :: !Double }

instance NFData BoundingBox

instance Ord BoundingBox where

    bb1 `compare` bb2 = area bb1 `compare` area bb2

class Boundable a where

    getBoundingBox :: a -> BoundingBox

type Point = (Double, Double)
-- Get the smallest bounding box that contains the two input bounding boxes

enlarge :: BoundingBox -> BoundingBox -> BoundingBox

enlarge b1 b2 = BoundingBox (min x1' x1'') (min y1' y1'')
  (max x2' x2'') (max y2' y2'')

where BoundingBox x1' y1' x2' y2' = b1
   BoundingBox x1'' y1'' x2'' y2'' = b2

-- Compute the area of a bounding box

area :: BoundingBox -> Double

area (BoundingBox x1' y1' x2' y2') = (x2' - x1') * (y2' - y1')

-- Check whether a bounding box contains a point

containsPoint :: BoundingBox -> Point -> Bool

containsPoint bb (px, py) = px > x1' && px < x2' && py > y1' && py < y2'

where BoundingBox x1' y1' x2' y2' = bb

instance Show BoundingBox where
  show (BoundingBox x1' y1' x2' y2') = "BB [" ++ points ++ "]"

    where points = concat $ intersperse "," $ map show [x1', y1', x2', y2']

RTree.hs

Our implementation of R-tree data type includes ‘Empty’, ‘Node’ which contains a bounding box and a list of children nodes, and ‘Leaf’ which contains a bounding box and a specific entity. The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree. We implemented NFData, Boundable and Show instances to RTree data type. Insertion and searching are the two main functions we were working on. For insertion, we traversed the tree from root to bottom. At each step, all bounding boxes in the current layer are examined and we choose the node that requires least enlargement to insert the new entry. Upon reaching the second last layer of the tree, we
directly append the new entry to the children list and then check whether the length of children exceeds the maxChildren we set. If the node is full, we split the node into 2 subnodes by regrouping its children. In order to find the best split, we used an algorithm that Guttman proposed in his paper called QuadraticSplit. The algorithm searches for the pair of rectangles that is the worst combination to have in the same node, and makes them the initial objects of the two new groups. It then searches for the child node which has the strongest preference for one of the groups (in terms of area increase) and assigns the object to this group until all objects are assigned. For searching, we wrote a function called contains which accepts a Rtree and a point and returns all leaf nodes that contain the point as a list. The ‘contains’ function traverses the tree from top to bottom and at each level, it will recursively call ‘contains’ function at those children whose bounding box contains the point till the bottom of the tree. The time complexity of searching is $O(\log Mn)$. In order to improve the performance of building tree, we added the function of union two subtrees into one single tree. Its implementation is pretty similar to the insertion.

```haskell
{-# LANGUAGE DeriveGeneric #-}

module RTree where

import BoundingBox
import Data.List (sortBy, maximumBy)
import GHC.Generics (Generic)
import Control.DeepSeq

minChildren :: Int
minChildren = 2
```
maxChildren :: Int
maxChildren = 4

data RTree a =
  Node BoundingBox [RTree a]
| Leaf BoundingBox a
| Empty
  deriving (Eq, Generic)

instance NFData a => NFData (RTree a)

instance Boundable (RTree a) where
  getBoundingBox (Node bb _) = bb
  getBoundingBox (Leaf bb _) = bb
  getBoundingBox Empty = error "getBoundingBox on Empty"

instance Show a => Show (RTree a) where
  show Empty = "Empty"
  show (Leaf _ e) = show e
  show (Node _ children) = show children

newTree :: RTree a
newTree = Empty

gETCHILDREN :: RTree a -> [RTree a]
gETCHILDREN (Node _ children) = children
  _ = []

gETELEM :: Boundable a => RTree a -> a
gETELEM Empty = error "getElem on Empty"
  (Leaf _ e) = e
  (Node _ _) = error "Node does not have elem"
singleton :: Boundable a => a -> RTree a
singleton a = Leaf (getBoundingBox a) a

-- Generate a node which has this list of nodes as its children
generateNode :: Boundable a => [RTree a] -> RTree a
generateNode [] = Empty
generateNode children = Node newBB children
    where newBB = mergeBB' $ getBoundingBox <$> children
          mergeBB' bbs = foldr1 enlarge bbs

insert :: Boundable a => RTree a -> a -> RTree a
insert Empty e = singleton e
insert n@(Leaf _ _) e = Node (mergeBB n e) [singleton e, n]
insert n@(Node _ _ ) e
    | length (getChildren newN) > maxChildren = generateNode $ splitNode newN
    | otherwise = newN
    where newN = addToNode n $ singleton e

-- Merge two subtrees into one
union :: Boundable a => RTree a -> RTree a -> RTree a
union Empty right = right
union left Empty = left
union (l@(Leaf bb1 _)) r@(Leaf bb2 _)
    | bb1 == bb2 = l -- if two leaves have the same bounding box, return left
    | otherwise = generateNode [l,r]
union left right
    | depth left > depth right = union right left
    | depth left == depth right = foldr1 union $ (getChildren left) ++ [right]
    | length (getChildren newN) > maxChildren = generateNode $ splitNode newN
    | otherwise = newN
    where newN = addToNode right left

-- Add new node to a tree
addToNode :: Boundable a => RTree a -> RTree a -> RTree a
addToNode old new = Node newBB newChildren
    where newBB = unionBB old new
          oldChildren = getChildren old
          directAdd = new : filter (bbNotSame new) oldChildren
          bbNotSame n c = getBoundingBox c /= getBoundingBox n
          newChildren
            | depth old == depth new + 1 = directAdd
            | otherwise = insertIntoBestChild oldChildren new

fromList :: Boundable a => [a] -> RTree a
fromList xs = foldl insert newTree xs

toList :: RTree a -> [a]
toList Empty = []
toList (Leaf _ a) = [a]
toList (Node _ ts) = concatMap toList ts

-- Merge bounding box of given node with element
mergeBB :: Boundable a => RTree a -> a -> BoundingBox
mergeBB Empty e = getBoundingBox e
mergeBB t e = enlarge (getBoundingBox t) (getBoundingBox e)

{- Insert a new node into the best child of a list of tree nodes by finding
the child that needs to expand its bounding box the least to accommodate
the new node. -}
insertIntoBestChild :: Boundable a => [RTree a] -> RTree a -> [RTree a]
insertIntoBestChild [] _ = []
insertIntoBestChild children@(x:xs) new
    | getBoundingBox x == getBoundingBox best = (inserted best) ++ xs
    | otherwise = x : insertIntoBestChild xs new
    where (best:_)= sortBy compare' children
compare\ x\ y = \text{diffBB}\ x' \ \text{`compare`}\ \text{diffBB}\ y

\text{diffBB}\ x' = \text{area}\ \left(\text{unionBB}\ x'\ \text{new}\right) - \text{originalArea}\ x'

\text{originalArea} = \text{area} \cdot \text{getBoundingBox}

inserted node

| length (getChildren newNode) > maxChildren = splitNode newNode
| otherwise = [newNode]

\text{where}\ \text{newNode} = \text{addToNode}\ \text{node}\ \text{new}

\hspace{1cm} \text{-- Split a tree node into 2 nodes by regrouping its children into 2 groups}

\text{splitNode} :: \text{Boundable}\ a \rightarrow \text{RTree}\ a \rightarrow [\text{RTree}\ a]

\text{splitNode}\ \text{Empty} = \text{error} \ "\text{cannot split empty node}"

\text{splitNode}\ \text{(Leaf } _{-} _) = \text{error} \ "\text{cannot split leaf node}"

\text{splitNode}\ \text{(Node } _{-} \text{ children)} = \left[\text{generateNode}\ \text{group1}, \text{generateNode}\ \text{group2}\right]

\text{where}\ \left(1, r\right) = \text{worstPair}\ \text{children}

\text{toAdd} = \text{filter}\ \text{notLOrR}\ \text{children}

\text{notLOrR}\ e = \text{getBoundingBox}\ e /= \text{getBoundingBox}\ 1 \ \&\&

\text{getBoundingBox}\ e /= \text{getBoundingBox}\ r

\left(\text{group1},\ \text{group2}\right) = \text{partition}\ \left[1\ \left[r\ \text{toAdd}\right]\right]

\hspace{1cm} \text{-- Find the pair of child nodes which form the biggest enlarged bounding box}

\text{worstPair} :: \text{Boundable}\ a \rightarrow [\text{RTree}\ a] \rightarrow \left(\text{RTree}\ a, \text{RTree}\ a\right)

\text{worstPair}\ \text{children} = \text{result}

\text{where}\ \text{result} = \text{snd}\ \$\ \text{maximumBy}\ \left(\lambda m\ n \rightarrow \text{compare}\ \left(fst\ m\right)\ \left(fst\ n\right)\right)\$

\left[\text{combinedArea},\ \text{pair}\right]

| x \leftarrow \text{indexedC}
| y \leftarrow \text{indexedC}

\text{let}\ \left(c1,\ idx1\right) = x

\left(c2,\ idx2\right) = y

\text{let}\ \text{bb1} = \text{getBoundingBox}\ c1

\text{bb2} = \text{getBoundingBox}\ c2

\text{combinedArea} = \text{area}\ \$\ \text{enlarge}\ \text{bb1}\ \text{bb2}

\text{pair} = \left(c1,\ c2\right)
```haskell

indexedC = zip children ([1..] :: [Int])

-- Get the enlarged bounding box containing two nodes
unionBB :: Boundable a => RTree a -> RTree a -> BoundingBox
unionBB n1 n2 = enlarge (getBoundingBox n1) (getBoundingBox n2)

-- Compute the area diff when merging a node with another
areaDiffWithNode :: Boundable a => RTree a -> RTree a -> Double
areaDiffWithNode newNode old = newArea - oldArea
  where newArea = area $ unionBB newNode old
        oldArea = area $ getBoundingBox old

-- Partition the third list of nodes into either the first
-- or the second group of nodes returning (group1, group2)
partition :: Boundable a => [RTree a] -> [RTree a] -> [RTree a] -> ([RTree a], [RTree a])
partition l r [] = (l, r)
partition l r toAdd
  | length toAdd + length l <= minChildren = (l ++ toAdd, r)
  | length toAdd + length r <= minChildren = (l, r ++ toAdd)
  | otherwise = assign nextNode l r
  where nextNode = snd $ maximumBy (
m n -> compare (fst m) (fst n)) $ [(diff e, e) | e <- toAdd]
      lNode = generateNode l
      rNode = generateNode r
      leftDiff e = areaDiffWithNode e lNode
      rightDiff e = areaDiffWithNode e rNode
      diff e = abs (leftDiff e - rightDiff e)
      assignToLeft = partition (nextNode : l) r remain
      assignToRight = partition l (nextNode : r) remain
      remain = filter notNextNode toAdd
```

19
notNextNode n = getBoundingBox n /= getBoundingBox nextNode
assign nextN l' r'
  | leftDiff nextN < rightDiff nextN = assignToLeft
  | leftDiff nextN > rightDiff nextN = assignToRight
  | areaL < areaR = assignToLeft
  | areaL > areaR = assignToRight
  | length l' < length r' = assignToLeft
  | otherwise = assignToRight
where areaL = area $ getBoundingBox lNode
  areaR = area $ getBoundingBox rNode

default :: Boundable a => RTree a -> Int
default Empty = 0
default (Leaf _ _) = 1
default (Node _ children) = 1 + (maximum $ map default children)

-- Get all leaf nodes as a list that contain the point
contains :: Boundable a => RTree a -> Point -> [RTree a]
contains Empty _ = []
contains l@(Leaf bb _) p
  | containsPoint bb p = [l]
  | otherwise = []
contains (Node bb children) p
  | containsPoint bb p = foldr (\x acc -> contains x p ++ acc) [] children
  | otherwise = []

printTree :: (Boundable a, Show a) => String -> RTree a -> IO ()
printTree header Empty = putStrLn $ header ++ "Empty"
printTree header (Leaf bb x) = putStrLn $ header ++ "Leaf " ++ (show bb) ++ " " ++ (show x)
printTree header (Node bb children) =
do putStrLn $ header ++ "Node " ++ (show bb) ++ "" mapM_ (printTree $ header ++ space) children
```haskell
putStr "\n"
where space = replicate 9 ' '  

Evaluation

Evaluate.hs

Contains helper functions for performing evaluations which allows different sections of the program to be selectively run in parallel or sequential modes, different numbers of randomly generated test points and different numbers of randomly generated geofences.

module Evaluate where

import qualified Entities as E
import Geometry (Point)
import GeoJSONParser (parseFeatureCollection)
import qualified RTree as RT
import Control.Parallel.Strategies (using, parList, rdeepseq)
import qualified Generator as G
import qualified Data.ByteString.Lazy as B
import Control.DeepSeq
import Data.List.Split (chunksOf)
import BoundingBox (BoundingBox(..), Boundable(..))
import System.Directory
import Control.Concurrent.ParallelIO.Local
import Data.Maybe (fromJust)

data Execution = Parallel | Sequential deriving (Eq, Show)

type Path = String

countryJson :: Path
countryJson = "data/full/countries.json"
```
stateJson :: Path
stateJson = "data/full/states_provinces.json"

chunkedJsonPath :: Path
chunkedJsonPath = "data/separate/"

evaluate :: Execution -> Execution -> Execution -> Int -> Int -> IO ()
evaluate e1 e2 e3 numPoints additionalEntities = do
  putStrLn ("Starting Evaluation with " ++ show numPoints ++ " points and " ++ show additionalEntities ++ " additional entities")
  putStrLn "Generating test points using "
  let points = generateTestPoints numPoints
  putStrLn $ "Generated " ++ (show $ length points) ++ " points"
  putStrLn ("Loading test entities using " ++ show e1 ++ " mode")
  seedEntities <- loadTestEntities e1
  putStrLn $ "Loaded " ++ (show $ length seedEntities) ++ " test entities"
  putStrLn "Generating additional entities"
  let generatedEntities = generateNewEntities e1 seedEntities additionalEntities
      entities = seedEntities ++ generatedEntities
  putStrLn $ (show $ length entities) ++ " total entities"
  putStrLn ("Constructing RTree using " ++ show e2 ++ " mode")
  let tree = makeTree e2 entities
  putStrLn $ "Constructed RTree of depth " ++ (show $ RT.depth tree)
  putStrLn $ "Query points using " ++ show e3 ++ " mode"
  let results = case e3 of
      Sequential -> op
      Parallel -> op `using` parList rdeepseq
      where op = map (enclosingFences tree) points
  putStrLn "Length of results:"
  putStrLn $ length results

enclosingFences :: RT.RTree E.Entity -> (Double, Double) -> [RT.RTree E.Entity]
enclosingFences tree p = filter (doesContain p) $ RT.contains tree p
    where doesContain p' leaf = E.containsPoint (RT.getElem leaf) p'

evaluateList :: Execution -> [Point] -> IO ()
evaluateList e points = do
    entities <- loadTestEntities e
    let tree = makeTree e entities
        result = case e of
            Sequential -> op
            Parallel -> op `using` parList rdeepseq
                where op = map (enclosingFences tree) points
        mapM_ print $ zip points result

loadTestEntities :: Execution -> IO [E.Entity]
loadTestEntities Sequential = do
    countries <- loadCountries countryJson
    states <- loadStates stateJson
    return (countries ++ states)

loadTestEntities Parallel = do
    filePaths <- listDirectory chunkedJsonPath
    let paths = filter (
        path -> path `notElem` [".DS_Store"])
        filePaths
    es <- withPool 4 $ \pool -> parallelInterleaved pool (map load paths)
    return $ concat es

load :: String -> IO [E.Entity]
load path@(\'s\': _) = loadStates $ chunkedJsonPath ++ path
load path@(\'c\': _) = loadCountries $ chunkedJsonPath ++ path
load _ = error $ "unknown path"

loadStates :: String -> IO [E.Entity]
loadStates path = do
    x <- B.readFile path
    return $ fromJust $ E.parseStates $ fromJust $ parseFeatureCollection x
loadCountries :: String -> IO [E.Entity]
loadCountries path = do
  x <- B.readFile path
  return $ fromJust $ E.parseCountries $ fromJust $ parseFeatureCollection x

generateTestPoints :: Int -> [Point]
generateTestPoints n = G.genPoints world n

generateNewEntities :: Execution -> [E.Entity] -> Int -> [E.Entity]
generateNewEntities e bounds numEntities = genList ++ remList
  where num = numEntities `quot` length bounds
        r = numEntities `mod` length bounds
        remList
          | e == Sequential = concat $ map (generateEntity 1) (take r bounds)
          | otherwise = concat (map (generateEntity 1) (take r bounds)
                           `using` parList rdeepseq)
        genList
          | e == Sequential = concat $ map (generateEntity num) bounds
          | otherwise = concat (map (generateEntity num) bounds
                           `using` parList rdeepseq)

generateEntity :: Int -> E.Entity -> [E.Entity]
generateEntity n entity = E.buildEntityWithGeo <$> polygons
  where polygons = G.genPolygons n $ getBoundingBox entity

makeTree :: (Boundable a, NFData a) => Execution -> [a] -> RT.RTree a
makeTree Sequential xs = RT.fromList xs
makeTree Parallel xs = let chunks = split numChunks xs in makeTreePar chunks
  where numChunks = 10

makeTreePar :: (Boundable a, NFData a) => [[a]] -> RT.RTree a
makeTreePar entitiess = foldr1 RT.union (map RT.fromList entitiess)
split :: Int -> [a] -> [[a]]
split numChunks xs = chunksOf (length xs `quot` numChunks) xs

world :: BoundingBox
world = BoundingBox { x1 = longMin
, y1 = latMin
, x2 = longMax
, y2 = latMax
}
where latMin = -90
      latMax = 90
      longMin = -180
      longMax = 180

Generator.hs
Contains helper functions for generating random geofence polygons. Poly-
gons are generated by first generating a set of random points within some
region specified by the provided bounding box, and then computing the con-
vex hull of these “seed” points.

{-# LANGUAGE NamedFieldPuns #-}

module Generator where

import qualified Geometry as GM
import qualified ConvexHull as CH
import BoundingBox (BoundingBox(..))
import System.Random
import Data.List.Split (chunksOf)
import qualified RTree as RT
genRandomNumbersBetween :: Int -> Int -> (Double, Double) -> [Double]
genRandomNumbersBetween n seed (a, b) = take n $ (randomRs (a, b) myGenerator) where
    myGenerator = mkStdGen seed

define getPair :: [a] -> (a, a)
define getPair [x, y] = (x, y)
define getPair _ = error "shouldn't happen"

define genPoints :: BoundingBox -> Int -> [GM.Point]
define genPoints bb n = zip xs ys
    where xs = genRandomNumbersBetween n seedX (xMin, xMax)
          ys = genRandomNumbersBetween n seedY (yMin, yMax)
            BoundingBox {x1, y1, x2, y2} = bb
            [xMin, yMin, xMax, yMax] = [x1, y1, x2, y2]
            seedX = 100
            seedY = 120

define genPolygons :: Int -> BoundingBox -> [GM.Geometry]
define genPolygons n (BoundingBox {x1,y1,x2,y2}) = map makePoly chunks
    where chunks = chunksOf numPts $ zip xs ys
            xs = genRandomNumbersBetween (numPts * n) seedX (x1, x2)
            ys = genRandomNumbersBetween (numPts * n) seedY (y1, y2)
            numPts = 20
            seedX = 100
            seedY = 120

define genSampleTree :: RT.RTree GM.Geometry
define genSampleTree = RT.fromList polygons
    where polygons = concatMap (genPolygons 10) quadrants
            quadrants = [ BoundingBox { x1 = 0, x2 = 0.49, y1 = 0, y2 = 0.49 } , BoundingBox { x1 = 0.5, x2 = 1, y1 = 0, y2 = 0.49 } , BoundingBox { x1 = 0, x2 = 0.49, y1 = 0.5, y2 = 1 } , BoundingBox { x1 = 0.5, x2 = 1, y1 = 0.5, y2 = 1 } ]

26
\[
\text{makePoly :: } [(\text{Double, Double})] \rightarrow \text{GM.Geometry}
\]

\[
\text{makePoly } \text{pts} = \begin{cases} 
\text{Right } x & \rightarrow \text{GM.Polygon} \{ \text{GM.pOuterRing} = x, \text{GM.pInnerRings} = [] \} \\
\text{Left } m & \rightarrow \text{error } \langle \text{show } m \rangle
\end{cases}
\]

\[
\text{where } \text{lr} = \text{GM.fromLineString} \langle \text{ch} ++ [\text{head } \text{ch}] \rangle
\]

\[
\text{ch} = \text{map } \text{getPair} \langle \text{CH.convexHull} \cdot \text{map } (\lambda p \rightarrow [\text{fst } p, \text{snd } p]) \rangle \text{pts}
\]

Testing

Main.hs

\[
\text{import Evaluate}
\]

\[
\text{import System.Environment}
\]

\[
\text{import System.Exit(die)}
\]

\[
\text{import System.IO(readFile)}
\]

\[
\text{import Geometry(Point)}
\]

\[
\text{import Data.List.Split (splitOn)}
\]

\[
\text{main :: IO ()}
\]

\[
\text{main} = \text{do}
\]

\[
\text{args} \leftarrow \text{getArgs}
\]

\[
\text{case args of}
\]

\[
[\text{filename, } "s"] \rightarrow \text{do}
\]

\[
\text{contents} \leftarrow \text{readFile } \text{filename}
\]

\[
\text{let points} = \text{getPoints} \langle \text{lines } \text{contents} \rangle
\]

\[
\text{evaluateList Sequential points}
\]

\[
[\text{filename, } "p"] \rightarrow \text{do}
\]

\[
\text{contents} \leftarrow \text{readFile } \text{filename}
\]

\[
\text{let points} = \text{getPoints} \langle \text{lines } \text{contents} \rangle
\]

\[
\text{evaluateList Parallel points}
\]

\[
- \rightarrow \text{do}
\]

\[
\text{pn} \leftarrow \text{getProgName}
\]

\[
\text{die } \langle "\text{Usage: } " + pn + " \rangle
\]

27
getPoints :: [String] -> [Point]
getPoints lines' = map toPoint lines'
    where toPoint l = helper $ splitOn "," l
          helper [xs, ys] = (read xs :: Double, read ys :: Double)
          helper _ = error $ "unknown formatting"

evaluatePerformance.hs

import Evaluate (Execution(..), evaluate)
import System.Environment
import System.Exit (die)

main :: IO()
main = do
    args <- getArgs
    case args of
        [loadMode, buildTreeMode, queryMode, numPoint, numPolygon] -> do
            let lm = getMode loadMode
                tm = getMode buildTreeMode
                qm = getMode queryMode
                numPoint' = read numPoint :: Int
                numPolygon' = read numPolygon :: Int
                evaluate lm tm qm numPoint' numPolygon'
        _ -> do
            pn <- getProgName
            die $ "Usage: " ++ pn ++ " <loadFileMode> <makeTreeMode> <queryPointMode> <numPoint> <numPolygon>\n" ++ "XMode: s --sequential, p --parallel"

getMode :: String -> Execution
getMode "s" = Sequential
getMode "p" = Parallel
getMode _ = error "Invalid mode"

3 Performance

Main.hs
Test on testPoints.txt

Sequential run with 1 core
Total time: 8.35s

Parallel run with 4 cores

Sparks total converted GC’d overflowed fizzled
21 16 0 0 5

Total time: 5.92s
Speedup: 1.4

evaluatePerformance.hs
Test on 10000 points and 10000 additional polygons.

LoadData, BuildTree, QueryPoint all Seq with 1 core
### Parallel Loading Data, Rest Seq

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>converted</th>
<th>GC’d</th>
<th>overflowed</th>
<th>fizzled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5164</td>
<td>5159</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total time:** 19.08s  
**Speedup:** 1.14

### Parallel Building Tree, Rest Seq

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>converted</th>
<th>GC’d</th>
<th>overflowed</th>
<th>fizzled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

**Total time:** 8.61s  
**Speedup:** 2.22
Parallel Querying Data, Rest Seq

Sparks

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>converted</th>
<th>GC’d</th>
<th>overflowed</th>
<th>fizzled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10000</td>
<td>39</td>
<td>8191</td>
<td>1770</td>
<td>0</td>
</tr>
</tbody>
</table>

Total time: 20.05s

Speedup: 0.95

All Parallel

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>converted</th>
<th>GC’d</th>
<th>overflowed</th>
<th>fizzled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15175</td>
<td>5155</td>
<td>8193</td>
<td>1808</td>
<td>19</td>
</tr>
</tbody>
</table>

Total time: 6.27s

Speedup: 3.04
4 References

https://eng.uber.com/go-geofence/
https://medium.com/@buckhx/unwinding-uber-s-most-efficient-service-406413c5871d
https://hackage.haskell.org/package/aeson-1.4.6.0/docs/Data-Aeson.html
http://geomalgorithms.com/a03-inclusion.html
http://hackage.haskell.org/package/data-r-tree-0.0.5.0/docs/Data-RTree.html
http://rosettacode.org/wiki/ConvexHullHaskell