Programming assignment 4

Hardy Jones

Professor Köppe

This program is a series of literate Haskell modules split across multiple files. So it could be run directly if that were required.

We can use most of our previous work.

Again, we first require some pragmas to let us know if something slipped by.

```
> {-# OPTIONS_GHC -Wall #-}
> {-# OPTIONS_GHC -Werror #-}
> {-# OPTIONS_GHC -fno-warn-unused-do-bind #-}
```

Now on to our actual program:

```
> module P4 where
```

We import a whole mess of helper things

```
> import Control.Lens
>
> import Data.Either (rights)
> import Data.List (sort)
>
> import P4.Morphism
> import P4.Parser (parseData)
>
> import System.FilePath (takeBaseName)
> import System.FilePath.Glob (glob)
>
> import Text.Parsec.String (parseFromFile)
> import Text.PrettyPrint.Boxes
```

And off we go.

Unfortunately, I didn't plan time acccordingly, and do not have enough time to run all results. Only burma14 and the ulysses tours are run

```
> main :: IO ()
> main = do
>     -- First glob all the distance file names.
>     tspFiles <- sort <$> glob "distances/*.txt"
>     let tspFiles' = take 1 (drop 3 tspFiles) ++ drop (length tspFiles - 2) tspFiles
>     let opts = take 1 (drop 3 optimalTours) ++ drop (length optimalTours - 2) optimalTours
>     -- Parse them all to lists of distances.
>     tsps <- mapM (parseFromFile parseData) tspFiles'
>     let tsps' = rights tsps
>
>     -- First we compute the canonical tour.
```

```haskell
>     let d1 = tsps' <&> (\xs -> tourLength xs . canonical $ xs)
>     -- Next we compute the Nearest Neighbor tour.
>     let d2 = tsps' <&> (\xs -> tourLength xs . nearestNeighbors $ xs)
>     -- Next we compute the Nearest Neighbor followed by twoOpt.
>     let d3 = tsps' <&> (\xs -> tourLength xs . twoOpt xs . nearestNeighbors $ xs)
>     -- Next we compute the Nearest Neighbor followed by lin'Kernighan.
>     let d4 = tsps' <&> (\xs -> tourLength xs . lin'Kernighan xs . nearestNeighbors $ xs)
>     -- Next we compute the Farthest Insertion tour.
>     let d5 = tsps' <&> (\xs -> tourLength xs . farthestInsertion $ xs)
>     -- Next we compute the Farthest Insertion followed by twoOpt.
>     let d6 = tsps' <&> (\xs -> tourLength xs . twoOpt xs . farthestInsertion $ xs)
>     -- Next we compute the Farthest Insertion followed by lin'Kernighan.
>     let d7 = tsps' <&> (\xs -> tourLength xs . lin'Kernighan xs . farthestInsertion $ xs)
>     let table = prettyTable [ takeBaseName <$> tspFiles'
>                             ,                  opts
>                             , show         <$> d1
>                             , show         <$> d2
>                             , show         <$> d3
>                             , show         <$> d4
>                             , show         <$> d5
>                             , show         <$> d6
>                             , show         <$> d7
>                             ]
>     writeFile "comparison.md" table

> optimalTours :: [String]
> optimalTours =
>     [ "2579"
>     , "202310"
>     , "7542"
>     , "3323"
>     , "69853"
>     , "40160"
>     , "134602"
>     , "171414"
>     , "294358"
>     , "55209"
>     , "80369"
>     , "36905"
>     , "6859"
>     , "7013"
>     ]

> prettyTable :: [[String]] -> String
> prettyTable = prettyCols
>
> -- The `boxes` api isn't so great to use, but quite powerful.
> prettyCols :: [[String]] -> String
> prettyCols = render . hsep 2 left . mkHeaders . map (vcat left . map text)
>     where
>     mkHeaders = zipWith mkHeaders' headers
>     mkHeaders' h c = h // separator (max (cols h) (cols c)) // c
>     separator n = text (replicate n '-')
>     headers = map text [ "Filename"
```

```
>                         , "Optimal"
>                         , "Canonical"
>                         , "NN"
>                         , "NN 2-opt"
>                         , "NN LK"
>                         , "FI"
>                         , "FI 2-opt"
>                         , "FI LK"
>                         ]
```

We can compare the distances. Notice that sometimes nearest neighbors can return a worse distance than even the naïve straight line tour. But it's also never better than the optimal.

Table 1: Comparison of distances

Filename	Optimal	Canonical	NN	NN 2-opt	NN LK	FI	FI 2-opt	FI LK
burma14Distances	3323	4562	4048	3371	4048	5240	3336	5162
ulysses16Distances	6859	9665	9988	6909	9988	10634	6870	8693
ulysses22Distances	7013	12198	10586	7083	10586	11254	7294	9313

We want to represent the distance as a pair of node numbers and the distance between them.

```
> {-# LANGUAGE TemplateHaskell #-}
> module P4.Distance where
>
> import Control.Lens (makeLenses)
>
> data Distance = Distance
>     { _i        :: Int
>     , _j        :: Int
>     , _distance :: Int
>     } deriving (Eq, Ord, Show)
```

We use a little lens magic to make things easy on us.

```
> makeLenses ''Distance
```

Here's the brunt of the program.

We implement the different algorithms here.

```
> {-# LANGUAGE DeriveTraversable #-}
> {-# LANGUAGE GeneralizedNewtypeDeriving #-}
> {-# LANGUAGE OverloadedLists #-}
> {-# LANGUAGE TypeFamilies #-}
> module P4.Morphism where
>
> import Control.Lens
> import Control.Applicative (Alternative, empty)
> import Control.Monad (MonadPlus, guard)
>
```

```
> import Data.Bits (xor)
> import Data.Foldable (find, fold, foldl')
> import Data.Function ((&), on)
> import Data.List (groupBy, maximumBy, minimumBy, sort)
> import Data.Maybe (fromMaybe, isJust, maybeToList)
> import Data.Monoid (First(..), (<>))
> import Data.Ord (comparing)
> import Data.Traversable (for)
> import Data.Tuple (swap)
>
> import GHC.Exts (IsList(..))
>
> import P4.Distance
>
> import qualified Data.Set as S
```

We need some way to express tours, a [Int] might work, but let's give it a newtype just to make things easier.

```
> newtype Tour a = Tour { unTour :: [a] }
>     deriving ( Alternative, Applicative, Eq, Functor, Foldable, Monad
>              , MonadPlus, Monoid, Ord, Show, Traversable
>              )
> instance IsList (Tour a) where
>     type Item (Tour a) = a
>     fromList = Tour
>     toList = unTour
```

We can compute the `tourLength` of any list of distances.

```
> tourLength :: [Distance] -> Tour Int -> Int
> tourLength xs = sum . maybe [] (fmap _distance) . tourDistance xs

> tourDistance :: [Distance] -> Tour Int -> Maybe [Distance]
> tourDistance xs ys = for (pair $ toList ys) $ \(i', j') ->
>     find (\d -> _i d == min i' j' && _j d == max i' j') xs

> pair :: [a] -> [(a, a)]
> pair = zip <*> (uncurry (++) . swap . splitAt 1)

> canonical :: [Distance] -> Tour Int
> canonical xs = fmap _i (fromList front) <> [loop]
>     where
>     front = fmap head . groupBy ((==) `on` _i) . sort $ xs
>     loop = _j $ last front

> nearestNeighbors :: [Distance] -> Tour Int
> nearestNeighbors [] = []
> nearestNeighbors xs = fromList $ constructTour $ go xs (_i $ head xs)
>     where
>     go :: [Distance] -> Int -> [Distance]
>     go [] _ = []
>     go xs m = let d = shortest m xs
```

```
>               in  d:go (filter (not . neighbor m) xs) (next m d)
>     constructTour = reorder Nothing . fmap ((,) <$> _i <*> _j)
>     reorder _ [] = []
>     reorder Nothing [(x, y)] = [x,y]
>     reorder (Just x') [(x, y)] = if x' == x then [x,y] else [y,x]
>     reorder Nothing ((x, y):zs) = x:reorder (Just y) zs
>     reorder (Just x') ((x, y):zs) = if x' == x then x:reorder (Just y) zs else y:reorder (Just x) zs
> shortest :: Int -> [Distance] -> Distance
> shortest n = minimumBy (comparing _distance) . filter (neighbor n)
> longest :: Int -> [Distance] -> Distance
> longest n = maximumBy (comparing _distance) . filter (neighbor n)
> next :: Int -> Distance -> Int
> next n (Distance i j _) = if i == n then j else i
> neighbor :: Int -> Distance -> Bool
> neighbor n (Distance i j _) = i == n || j == n
```

We codify the “Farthest Insertion” algorithm

```
> data Mini2Tour a = Mini2Tour a a   (S.Set a)
> data Mini3Tour a = Mini3Tour a a a (S.Set a)
> data MiniTour  a = MiniTour [a]    (S.Set a)

> farthestInsertion :: [Distance] -> Tour Int
> farthestInsertion [] = []
> farthestInsertion xs = fromList $ init $ go $ miniTour $ mini3Tour
>     where
>     mini2Tour ::Mini2Tour Int
>     mini2Tour = Mini2Tour x y (S.fromList [1..nodes xs] S.\\ S.fromList [x, y])
>     mini3Tour :: Mini3Tour Int
>     mini3Tour = Mini3Tour x y x (S.delete x us)
>         where
>         Mini2Tour x y us = mini2Tour
>         filtered = (filter (xor <$> neighbor x <*> neighbor y) xs)
>         Distance i' j' _ = maximumBy (comparing _distance) filtered
>         z = if i' == x || i' == y then j' else i'
>     miniTour :: Mini3Tour Int -> MiniTour Int
>     miniTour (Mini3Tour x y z us) = MiniTour [x, y, z] us
>     go :: MiniTour Int -> [Int]
>     go (MiniTour vs us)
>         | S.null us = vs
>         | otherwise = go $ MiniTour vs' (S.difference us $ S.fromList vs')
>         where
>         filtered = filter (\d -> 1 == (length $ filter (`neighbor` d) vs)) xs
>         Distance i' j' _ = maximumBy (comparing _distance) filtered
>         vs' = foldl' (inject i' j') [] vs
>     inject i' j' acc v
>             | v == i'   = acc ++ [v] ++ [j']
>             | v == j'   = acc ++ [v] ++ [i']
>             | otherwise = acc ++         [v]
>     farthest = maximumBy (comparing _distance) xs
>     (x, y) = (_i farthest, _j farthest)

> nodes :: [a] -> Int
> nodes xs = floor $ (1 + sqrt (fromIntegral (1 + 8 * length xs))) / 2
```

```haskell
> twoOpt :: [Distance] -> Tour Int -> Tour Int
> twoOpt xs = go
>     where
>     go :: Tour Int -> Tour Int
>     go ys = case improvement xs ys of
>         []  -> ys
>         ys' -> go (minimumBy (comparing (tourLength xs)) ys')

> improvement :: [Distance] -> Tour Int -> [Tour Int]
> improvement xs ys = do
>     let bestDistance = tourLength xs ys
>     guard (0 < bestDistance)
>     i <- [0..length ys - 1]
>     k <- [i + 1..length ys]
>     let ys' = fromList (take 1 (toList ys) <> twoOptSwap (drop 1 (toList ys)) i k)
>     let newDistance = tourLength xs ys'
>     guard (0 < newDistance)
>     if newDistance < bestDistance then pure ys' else empty
> twoOptSwap :: [a] -> Int -> Int -> [a]
> twoOptSwap xs i k = front ++ reverse middle ++ back
>     where
>     (front, (middle, back)) = splitAt (k - i) <$> splitAt (i - 1) xs

> lin'Kernighan :: [Distance] -> Tour Int -> Tour Int
> lin'Kernighan xs path =
>     fromMaybe path $ getFirst $ improvePath xs path 1 S.empty

> alpha :: Int
> alpha = 5
> improvePath :: [Distance] -> Tour Int -> Int -> S.Set Int -> First (Tour Int)
> improvePath xs path depth restricted
>     | depth < alpha =
>         let pathList = toList path
>             filtered = filter (flip S.notMember restricted . fst) $ pair pathList
>             gs = fmap g filtered
>             g (x, y) = (x, y, weight xs x y - weight xs (last pathList) x)
>             goodGs = filter ((> 0) . view _3) gs
>             choices = fmap choose goodGs
>             choose (x, y, _) =
>                 let swapped = fromList $ replace pathList y (last pathList)
>                     swappedLength = tourLength xs swapped
>                     pathLength = tourLength xs path
>                 in  if swappedLength < pathLength
>                       then pure swapped
>                       else improvePath xs swapped (depth + 1) (S.insert x restricted)
>         in  fold choices
>     | otherwise =
>         let pathList = toList path
>             (x, y, d) = maximumBy (comparing $ view _3) $ fmap g $ pair pathList
>             g (x, y) = (x, y, weight xs x y - weight xs (last pathList) x)
>             swapped = fromList $ replace pathList y (last pathList)
>             swappedLength = tourLength xs swapped
>             pathLength = tourLength xs path
```

```haskell
>         in  if d > 0
>               then if swappedLength < pathLength
>                       then pure swapped
>                       else improvePath xs swapped (depth + 1) (S.insert x restricted)
>               else mempty
> replace path y e =
>     let (front,  back) = span ((/=) y) path
>         (middle, rest) = span ((/=) e) $ drop 1 back
>     in  front ++ [e] ++ middle ++ [y] ++ drop 1 rest
> weight :: [Distance] -> Int -> Int -> Int
> weight xs x y = maybe 0 _distance $ find ((&&) <$> neighbor x <*> neighbor y) xs
```

We need to parse in the distance files.

```haskell
> module P4.Parser where
>
> import P4.Distance
>
> import Text.Parsec (count, spaces)
> import Text.Parsec.String (Parser)
> import Text.ParserCombinators.Parsec.Char (CharParser)
> import Text.ParserCombinators.Parsec.Number (nat)
>
> parseData :: Parser [Distance]
> parseData = do
>     m <- nat'
>     let n = m * (m - 1) `div` 2
>     count n parseDistance
>
> parseDistance :: Parser Distance
> parseDistance = Distance <$> nat' <*> nat' <*> nat'
```

We want to parse `nats` with spaces around them.

```haskell
> nat' :: Integral i => CharParser st i
> nat' = spaces *> nat <* spaces
```