Programming assignment extra credit 2

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 have our three tasks:

1. The tree shortcut algorithm.
2. The greedy TSP algorithm as an optimization problem over independent systems:
3. Christofides' algorithm.

Here's an explanation of each algorithm.

1. The tree shortcut algorithm first looks at a graph, and creates a minimum spanning tree (MST). Then it walks along the MST and adds each nodes seen in preorder. The shortcut part comes from the preorder traversal. When walking the tree, If there's a path `x -> y -> z` and `y` has already been seen, drop the path through `y` and just take the path `x -> z`. This works because if `y` is already seen, that means it's already in the tour. And assuming that the distance between cities is a metric, the solution is still feasible by the triangle inequality. It also imposes an upper bound on the solution of being no more than twice the optimal solution. With a proof and more information given in [1].

2. The greedy algorithm:

   (a) Start with no edges
   (b) Add the edge of minimum distance that does not create a short cycle or increase any vertex degree to larger than 2.
   (c) Repeat until a full tour is formed.

   That's pretty much the entirety of the algorithm.

3. Christofides' algorithm.

   (a) Find a minimum spanning tree T.
   (b) Find a perfect matching M among vertices with odd degree.
   (c) Combine the edges of M and T to make a multigraph G.
   (d) Find an Euler cycle in G by skipping vertices already seen.
   (e) Convert the Euler cycle to a Hamilton cycle.

   More information is provided in [2].

   However, we take a slightly different approach. We skip creation of an Euler cycle, and go straight to the Hamilton cycle by shortcutting. A proof that this is still a valid implementation of Christofides' algorithm is provided in [3].

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 PX2 where

> import Data.Either (rights)
> import Data.List (sort)
>
> import PX2.Algorithm (christofides, greedy, treeShortcut)
> import PX2.Distance (Distance(..), distancesVertices, distancesEdges)
> import PX2.Graph (Graph(..))
> import PX2.Printing (optimalTours, prettyTable)
> import PX2.Parser (parseData)
> import PX2.Tour (Tour(..), tourLength)
>
> import System.FilePath (takeBaseName)
> import System.FilePath.Glob (glob)
>
> import Text.Parsec.String (parseFromFile)
```

We're only going to run on a few distance files, since I don't have all day to run these algorithms.

```
> main :: IO ()
> main = do
>     -- First glob all the distance file names.
>     tspFiles <- sort <$> glob "distances/*.txt"
>     -- Parse them all to lists of distances.
>     tsps <- mapM (parseFromFile parseData) tspFiles
>     let tsps' = rights tsps
>     let algo1 = runAlgorithm treeShortcut <$> tsps'
>     let algo2 = runAlgorithm greedy <$> tsps'
>     let algo3 = runAlgorithm christofides <$> tsps'
>     let files = takeBaseName <$> tspFiles
>     let table = prettyTable [ files
>                             , optimalTours
>                             , fmap show algo1
>                             , fmap show algo2
>                             , fmap show algo3
>                             ]
>     writeFile "comparison.md" table

> distances2Graph :: [Distance] -> Graph Int Int
> distances2Graph = Graph <$> distancesVertices <*> distancesEdges

> runAlgorithm :: (Graph Int Int -> [Int]) -> [Distance] -> Int
> runAlgorithm f ds = tourLength ds $ Tour $ f $ distances2Graph ds
```

Looking at the comparison between the algorithms. Neither the tree shortcut nor Christofides' algorithm seem to give enough of an increase over the greedy version to warrant the amount of effort that goes into them. Christofides' actually seems to give pretty bad tours.

If we look at the implementation of `treeShortcut` and `christofides`, they're both pretty complex. `greedy` on the other hand is pretty understandable, and makes for a decent tour.

Table 1: Comparison of algorithms

Filename	Optimal	Shortcut	Greedy	Christofides
berlin52Distances	7542	10402	10034	17318
burma14Distances	3323	4003	3832	3903
gr96Distances	55209	75225	66737	107179
ulysses16Distances	6859	7788	7938	8470
ulysses22Distances	7013	8308	8185	11355

Here we talk about all the algorithms we need.

```
> {-# LANGUAGE OverloadedLists #-}
> {-# LANGUAGE ViewPatterns #-}
> module PX2.Algorithm where
```

```
> import Control.Lens ((|>), (^..), _1, _2, _3, each, view)
>
> import Data.Function (on)
> import Data.List (group, sort, sortOn)
> import Data.List.Split (chunksOf)
> import Data.Tree
>
> import GHC.Exts
> import GHC.Natural
>
> import PX2.Graph
> import PX2.Orphan
>
> import qualified Data.Map as M
> import qualified Data.MultiSet as MS
> import qualified Data.Set as S
```

```
> treeShortcut :: (Ord v, Ord w) => Graph v w -> [v]
> treeShortcut = treeShortcut' kruskal
```

```
> treeShortcut' :: Ord v => MSTFunc v w -> Graph v w -> [v]
> treeShortcut' f graph = walk $ f graph
```

We don't convert this to an actual tree, so the walk is a bit odd. It's possible that it's not even correct, but it seems to do the job.

```
> walk :: Ord v => MST v w -> [v]
> walk mst = go S.empty (S.toAscList . unweightedEdges $ mst)
>     where
>     go :: Ord v => S.Set v -> [(v, v)] -> [v]
>     go vs []              = []
>     go vs es@((i, j):_) = snd (go' vs i es)
>         where
>         -- This is pretty terrible...
>         go' vs i []
>             | S.member i vs = (vs, [])
```

```
>             | otherwise     = (S.insert i vs, [i])
>         go' vs i ((j, k):es')
>             | not (S.member i vs)          =
>                 (i:) <$> go' (S.insert i vs) i ((j, k):es')
>             | i == k && not (S.member j vs) =
>                 let (vs', xs) = (j:) <$> go' (S.insert j vs) j es
>                 in  (xs ++) <$> go' vs' i es'
>             | i == j && not (S.member k vs) =
>                 let (vs', xs) = (k:) <$> go' (S.insert k vs) k es
>                 in  (xs ++) <$> go' vs' i es'
>             | otherwise                     =
>                 go' vs i es'

> kruskal :: (Ord v, Ord w) => MSTFunc v w
> kruskal = go S.empty S.empty . sorted . edges
>     where
>     go :: (Ord v, Ord w) => Family v -> MST v w -> [(v, v, w)] -> MST v w
>     go us a []                 = a
>     go us a (e@(i, j, _):es) = case unioned us i j of
>         (Just vs, Just ws)
>             | vs == ws     -> go us                  a              es
>         (Just vs, Just ws) -> go (reunion us vs ws) (S.insert e a) es
>         (Nothing, Just ws) -> go (reunion' us ws i) (S.insert e a) es
>         (Just vs, Nothing) -> go (reunion' us vs j) (S.insert e a) es
>         (Nothing, Nothing) -> go (reunion'' us i j) (S.insert e a) es
>     sorted :: Ord w => S.Set (v, v, w) -> [(v, v, w)]
>     sorted = sortOn (view _3) . S.toList

> unioned :: Ord v => Family v -> v -> v -> (Maybe (S.Set v), Maybe (S.Set v))
> unioned us i j = (lookupFamily us i, lookupFamily us j)
> lookupFamily :: Ord v => Family v -> v -> Maybe (S.Set v)
> lookupFamily us u = fst <$> S.minView (S.filter (S.member u) us)
> reunion :: Ord v => Family v -> S.Set v -> S.Set v -> Family v
> reunion us vs ws = S.insert (S.union vs ws) $ S.delete ws $ S.delete vs us
> reunion' :: Ord v => Family v -> S.Set v -> v -> Family v
> reunion' us vs w = reunion us vs (S.singleton w)
> reunion'' :: Ord v => Family v -> v -> v -> Family v
> reunion'' us v w = reunion' us (S.singleton v) w

> cycles :: Ord v => Graph v w -> Bool
> cycles g = vSize g > 2 && go S.empty (S.toList $ unweightedEdges $ edges g)
>     where
>     go :: Ord v => Family v -> [(v, v)] -> Bool
>     go us [] = False
>     go us ((i, j):es) = case unioned us i j of
>         (Just vs, Just ws) | vs == ws     -> True
>         (Just vs, Just ws)                -> go (reunion us vs ws) es
>         (Just vs, Nothing)                -> go (reunion' us vs j) es
>         (Nothing, Just ws)                -> go (reunion' us ws i) es
>         (Nothing, Nothing)                -> go (reunion'' us i j) es

> incidents :: Ord v => S.Set (v, v, w) -> M.Map v Natural
> incidents = foldMap (M.singleton <$> head <*> fromIntegral . length)
```

```
>           . group
>           . sort
>           . (^..traverse.each)
>           . S.toList
>           . unweightedEdges

> tooIncident :: Ord v => S.Set (v, v, w) -> Bool
> tooIncident = tooIncident' 2

> tooIncident' :: Ord v => Natural -> S.Set (v, v, w) -> Bool
> tooIncident' n es = n < maximum (incidents es)

> greedy :: (Ord v, Ord w) => Graph v w -> [v]
> greedy g = go S.empty . sortOn (view _3) . S.toList . edges $ g
>     where
>     go ps _ | vSize g - 1 == S.size ps = walk ps
>     go ps []                           = walk ps
>     go ps (e:es)                       = let ps' = S.insert e ps in
>         if cycles g {edges = ps'} || tooIncident ps'
>           then go ps es
>           else go ps' es

> christofides :: (Ord v, Ord w) => Graph v w -> [v]
> christofides = christofides' kruskal

> christofides' :: (Ord v, Ord w) => MSTFunc v w -> Graph v w -> [v]
> christofides' f g = shortcut $ multigraph mst perfects
>     where
>     mst = f g
>     odds = oddIncidents mst
>     perfects = perfectMatching g odds
>     multigraph = MS.union `on` MS.fromSet . unweightedEdges
```

We can eschew creating an Eulerian Tour, and just shortcut the multigraph.

```
> type Missed v = [(v, v)]
> type Edges v = [(v, v)]
> shortcut :: Ord v => MS.MultiSet (v, v) -> [v]
> shortcut []                    = []
> shortcut (toList -> (i, j):es) = go [i, j] [i, j] es
>     where
>     -- This is pretty terrible...
>     go :: Ord v => S.Set v -> [v] -> [(v, v)] -> [v]
>     go ws vs [] = vs
>     go ws vs ((i,j):es)
>         | S.member i ws && S.member j ws
>             = go                ws   vs             es
>         | S.member i ws
>             = go (S.union [j]    ws) (vs |> j)      es
>         | S.member j ws
>             = go (S.union [i]    ws) (vs |> i)      es
>         | otherwise
>             = go (S.union [i, j] ws) (vs |> j |> i) es
```

```
> perfectMatching :: (Ord v, Ord w) => Graph v w -> [v] -> S.Set (v, v, w)
> perfectMatching g vs = foldl go S.empty
>                      . filter ((== 2) . length)
>                      . chunksOf 2
>                      . sort
>                      $ vs
>     where
>     es = edges g
>     go acc [x, y] = S.union acc $ S.filter (equivalent x y) es
>     equivalent x y (i, j, _) = i == x && j == y
```

```
> oddIncidents :: Ord v => S.Set (v, v, w) -> [v]
> oddIncidents = M.keys . M.filter odd . incidents
```

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

```
> module PX2.Distance where
>
> import qualified Data.Set as S
>
> data Distance = Distance
>     { i        :: Int
>     , j        :: Int
>     , distance :: Int
>     } deriving (Eq, Ord, Show)
```

```
> distanceEdge :: Distance -> (Int, Int, Int)
> distanceEdge = (,,) <$> i <*> j <*> distance
```

```
> distancesEdges :: [Distance] -> S.Set (Int, Int, Int)
> distancesEdges = S.fromList . fmap distanceEdge
```

```
> distancesVertices :: [Distance] -> S.Set Int
> distancesVertices ds = isjs
>     where
>     is = S.fromList $ i <$> ds
>     js = S.fromList $ j <$> ds
>     isjs = is `S.union` js
```

Since the graph and tree packages on hackage are pretty much terrible, we roll our own graph.

```
> {-# LANGUAGE DeriveAnyClass #-}
> {-# LANGUAGE DeriveDataTypeable #-}
> {-# LANGUAGE DeriveFoldable #-}
> {-# LANGUAGE DeriveGeneric #-}
> {-# LANGUAGE TupleSections #-}
> module PX2.Graph where
```

```
> import Control.Lens
>
> import Data.Data
> import Data.Traversable (for)
```

```haskell
>
> import GHC.Generics
>
> import Test.QuickCheck.Arbitrary
>
> import qualified Data.Set as S
```

```haskell
> data Graph v w = Graph
>     { vertices :: S.Set v
>     , edges    :: S.Set (v, v, w)
>     } deriving (Data, Eq, Foldable, Generic, Ord, Show, Typeable)
> type MST v w = S.Set (v, v, w)
> type MSTFunc v w = Graph v w -> MST v w
> type Family v = S.Set (S.Set v)
>
> instance (Arbitrary v, Arbitrary w, Ord v, Ord w) => Arbitrary (Graph v w) where
>     arbitrary = do
>         vs <- S.fromList <$> arbitrary
>         let es = [(i, j) | i <- S.toList vs, j <- S.toList vs, i < j]
>         es' <- for es $ \(i, j) -> (i, j, ) <$> arbitrary
>         pure $ Graph vs (S.fromList es')
```

```haskell
> unweightedEdges :: Ord v => S.Set (v, v, w) -> S.Set (v, v)
> unweightedEdges =
>     S.fromList . (map $ (,) <$> view _1 <*> view _2) . S.toList
```

```haskell
> cartesian :: (Ord a, Ord b) => S.Set a -> S.Set b -> S.Set (a, b)
> cartesian xs ys = S.fromList $ (,) <$> S.toList xs <*> S.toList ys
```

```haskell
> vSize :: Graph v w -> Int
> vSize = S.size . vertices
> eSize :: Graph v w -> Int
> eSize = S.size . edges
```

An orphan instance for `Data.MultiSet`.

```haskell
> {-# LANGUAGE TypeFamilies #-}
> module PX2.Orphan where
```

```haskell
> import GHC.Exts (IsList(..))
>
> import qualified Data.MultiSet as MS
```

```haskell
> instance Ord a => IsList (MS.MultiSet a) where
>     type Item (MS.MultiSet a) = a
>     fromList = MS.fromList
>     toList = MS.toList
```

We need to parse in the distance files.

```
> module PX2.Parser (parseData) where
>
> import PX2.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.

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

And of course we want some way to print this stuff out prettily.

```
> module PX2.Printing where

> import Text.PrettyPrint.Boxes

> optimalTours :: [String]
> optimalTours =
>     [ "7542"
>     , "3323"
>     , "55209"
>     , "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"
>                        , "Shortcut"
>                        , "Greedy"
>                        , "Christofides"
>                        ]
```

We want to verify some facts using tests that are a bit too time consuming to verify with the type system.

```
> {-# LANGUAGE OverloadedLists #-}
> module PX2.Test where

> import Control.Lens
>
> import GHC.Natural
>
> import PX2.Algorithm
> import PX2.Graph
>
> import Test.QuickCheck
>
> import qualified Data.Map as M
> import qualified Data.MultiSet as MS
> import qualified Data.Set as S
```

Generated graphs should have the right number of edges.

```
> prop_ArbitraryEdgeNumbers :: Graph Natural Natural -> Bool
> prop_ArbitraryEdgeNumbers g = (vSize g * (vSize g - 1) `div` 2) == eSize g
```

Given a graph G = (V, E), E should be a subset of V x V.

```
> prop_EdgesAreSubset :: Graph Natural Natural -> Bool
> prop_EdgesAreSubset g = unweightedEdges es `S.isSubsetOf` cartesian vs vs
>     where
>     vs = vertices g
>     es = edges g
```

A graph with less than three vertices cannot have cycles.

```
> prop_SmallGraphNoCycles :: Graph Natural Natural -> Property
> prop_SmallGraphNoCycles g = vSize g < 3 ==> not (cycles g)
```

Given a graph G = (V, E), an MST of G should have exactly max(0, |V| - 1) edges.

```
> prop_MSTEdges :: MSTFunc Natural Natural
>               -> Graph Natural Natural
>               -> Bool
> prop_MSTEdges f g = max 0 (vSize g - 1) == S.size (f g)
```

Given a graph G = (V, E), an MST of G should be exactly V.

```
> prop_MSTVertices :: MSTFunc Natural Natural
>                  -> Graph Natural Natural
>                  -> Property
> prop_MSTVertices f g = vSize g > 1 ==> vertices g == vs
>     where
>     vs = S.fromList $ (^..traverse.each) $ S.toList $ unweightedEdges $ f g
```

Given a graph G = (V, E), a tree shortcut should be exactly V.

```
> prop_treeShortcut :: Graph Natural Natural -> Property
> prop_treeShortcut g = vSize g > 1 ==> vertices g == vs
>     where
>     vs = S.fromList $ treeShortcut g
```

Given a graph G = (V, E), a walk of the MST of G should give exactly V.

```
> prop_walkVertices :: MSTFunc Natural Natural
>                   -> Graph Natural Natural
>                   -> Property
> prop_walkVertices f g = vSize g > 1 ==> vertices g == vs
>     where
>     vs = S.fromList $ walk $ f g
```

Given a complete graph G = (V, E), the incidents of E should have a maximum of |V| - 1.

```
> prop_IncidentsMaximum :: Graph Natural Natural -> Property
> prop_IncidentsMaximum g = vSize g > 1 ==>
>     maximum (incidents $ edges g) == fromIntegral (vSize g - 1)
```

Given a complete graph G = (V, E), the incidents of E should be exactly V.

```
> prop_IncidentsKeys :: Graph Natural Natural -> Property
> prop_IncidentsKeys g = vSize g > 1 ==>
>     M.keysSet (incidents $ edges g) == vertices g
```

Given a graph G = (V, E), a greedy G should be exactly V.

```
> prop_GreedyVertices :: Graph Natural Natural -> Property
> prop_GreedyVertices g = vSize g > 1 ==> S.fromList (greedy g) == vertices g
```

Given a graph G = (V, E), a christofides G should be exactly V.

```
> prop_ChristofidesVertices :: Graph Natural Natural -> Property
> prop_ChristofidesVertices g = vSize g > 1 ==>
>     S.fromList (christofides g) == vertices g
```

Helper for unit tests/TDD.

```
> quickAssert :: Testable prop => prop -> IO ()
> quickAssert = quickCheckWith stdArgs { maxSuccess = 1 }

> main :: IO ()
> main = do
```

```
>     quickCheck prop_ArbitraryEdgeNumbers
>     quickCheck prop_EdgesAreSubset
>     quickCheckWith stdArgs {maxDiscardRatio = 100} prop_SmallGraphNoCycles
>     quickCheck $ prop_MSTEdges kruskal
>     quickCheck $ prop_MSTVertices kruskal
>     quickCheck $ prop_walkVertices kruskal
>     quickCheck prop_treeShortcut
>     quickCheck prop_IncidentsMaximum
>     quickCheck prop_IncidentsKeys
>     quickCheck prop_GreedyVertices
>     quickCheck prop_ChristofidesVertices
```

Use some TDD to figure out the walk.

```
>     quickAssert (walk (S.empty :: MST Natural Natural) == [])
>     quickAssert (walk [(1,2,10)] == [1,2])
>     quickAssert (walk [(1,2,10),(2,3,2)] == [1,2,3])
>     quickAssert (walk [(2,3,0),(2,5,0),(5,6,0)] == [2,3,5,6])
>     quickAssert (walk [(2,10,1),(10,11,2),(8,11,0)] == [2,10,11,8])
```

Use some TDD to figure out cycles.

```
>     quickAssert (cycles $ Graph [1,2,3] [(1,2,0),(1,3,0),(2,3,0)])
>     quickAssert (not $ cycles $ Graph [1,2,3] [(1,3,0),(2,3,0)])
>     quickAssert (not $ cycles $ Graph [1..7] [(1,3,0),(2,4,0),(3,6,0),(4,5,0),(5,7,0)])
>     quickAssert (not $ cycles $ Graph [1..7] [(1,2,0),(1,3,0),(2,4,0),(3,6,0),(4,5,0),(5,7,0)])
>     quickAssert (cycles $ Graph [1..7] [(1,2,0),(1,3,0),(2,4,0),(3,6,0),(4,5,0),(5,7,0),(6,7,0)])
```

Use some TDD to figure out the shortcut.

```
>     quickAssert (shortcut ([] :: MS.MultiSet (Natural, Natural)) == [])
>     quickAssert (shortcut [(1,2)] == [1,2])
>     quickAssert (shortcut [(1,2),(2,3)] == [1,2,3])
>     quickAssert (shortcut [(2,3),(2,5),(5,6)] == [2,3,5,6])
>     quickAssert (shortcut [(2,10),(10,11),(8,11)] == [2,10,11,8])
>     quickAssert (shortcut [(2,10),(8,11),(10,11)] == [2,10,11,8])
```

We run the tours here.

```
> {-# LANGUAGE DeriveTraversable #-}
> {-# LANGUAGE GeneralizedNewtypeDeriving #-}
> {-# LANGUAGE OverloadedLists #-}
> {-# LANGUAGE TypeFamilies #-}
> module PX2.Tour where
>
> import Control.Lens
> import Control.Applicative (Alternative)
> import Control.Monad (MonadPlus)
>
> import Data.Foldable (find)
> import Data.Traversable (for)
> import Data.Tuple (swap)
```

```
>
> import GHC.Exts (IsList(..))
>
> import PX2.Distance
```

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)
```

References

1. L. Lovász, J. Pelikán, K. Vesztergombi. *Discrete mathematics : elementary and beyond.* pp 162-163. Springer, 2003.
2. Paul E. Black, "Christofides algorithm", in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, eds. 28 October 2008. (May 31, 2015) Available from: http://www.nist.gov/dads/HTML/christofides.html
3. Jung-Sheng Lin, "Course Notes - IEOR 251 – Facility Design and Logistics" [online], February 16, 2005. (May 31, 2015) Available from: http://ieor.berkeley.edu/~kaminsky/ieor251/notes/2-16-05.pdf