99 questions/Solutions/94

From HaskellWiki

< 99 questions | Solutions

(***) Generate K-regular simple graphs with N nodes

In a K-regular graph all nodes have a degree of K; i.e. the number of edges incident in each node is K.

This solution generates all possible graphs with n nodes and n * k / 2 edges, filters the k regular graphs, then collects all non-isomorphic graphs using graph canonization. It is somewhat of a slow solution, taking >10 s to run
`regular 6 3`
.

```
data Graph a = Graph [a] [(a, a)]
               deriving (Show, Eq)

data Adjacency a = Adj [(a, [a])]
                   deriving (Show, Eq)

regular :: Int -> Int -> [Graph Int]
regular n k | r == 1 || n <= k || n < 0 || k < 0 = []
            | otherwise =
               map (adjToGraph . fst) $
               foldr (\x xs -> if any ((==) (snd x) . snd) xs then xs else x : xs) [] $
               zip a $ map canon a
   where
      a = filter (\(Adj a) -> all ((==) k . length . snd) a) $
          map (graphToAdj . Graph [1..n]) $ perm e q
      e = map (\xs -> (head xs, last xs)) $ perm [1..n] 2
      (q, r) = (n * k) `quotRem` 2
      perm n k = foldr (\x xs ->
                        [i : s | i <- n, s <- xs, i `notElem` s, asc (i : s)])
                        [[]] [1..k]
      asc xs = all (uncurry (<)) $ zip xs $ tail xs

graphToAdj :: (Eq a) => Graph a -> Adjacency a
graphToAdj (Graph [] _)      = Adj []
graphToAdj (Graph (x:xs) ys) = Adj ((x, ys >>= f) : zs)
   where
      f (a, b)
         | a == x = [b]
         | b == x = [a]
         | otherwise = []
      Adj zs = graphToAdj (Graph xs ys)

adjToGraph :: (Eq a) => Adjacency a -> Graph a
adjToGraph (Adj [])          = Graph [] []
```

```
adjToGraph (Adj ((v, a):vs)) = Graph (v : xs) ((a >>= f) ++ ys)
   where
      f x = if (v, x) `elem` ys || (x, v) `elem` ys
            then []
            else [(v, x)]
      Graph xs ys = adjToGraph (Adj vs)

canon :: (Eq a, Ord a) => Adjacency a -> String
canon (Adj a) = minimum $ map f $ perm n
   where
      n = length a
      v = map fst a
      perm n = foldr (\x xs -> [i : s | i <- [1..n], s <- xs, i `notElem` s]) [[]] [1..n]
      f p = let n = zip v p
            in show [(snd x,
                      sort id $ map (\x ->
                         snd $ head $ snd $ break ((==) x . fst) n) $ snd $ find a x)
                    | x <- sort snd n]
      sort f n = foldr (\x xs -> let (lt, gt) = break ((<) (f x) . f) xs
                                 in lt ++ [x] ++ gt) [] n
      find a x = let (xs, ys) = break ((==) (fst x) . fst) a in head ys
```