

Programming assignment 2

Hardy Jones

999397426

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 only have to make slight modifications to our previous program in order to handle EUC_2D and GEO files.

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 P2 where
```

We import a whole mess of helper things

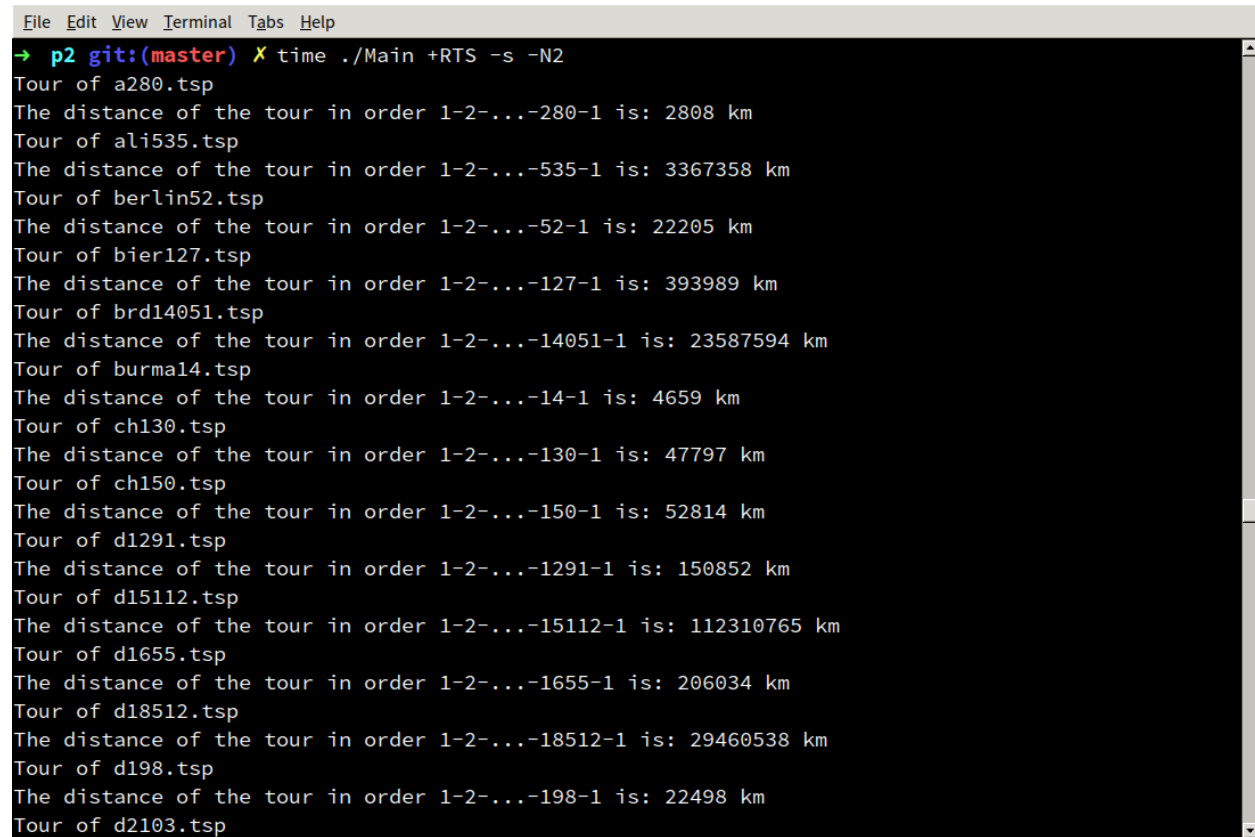
```
> import Data.Foldable (for_)  
> import Data.List (sort)  
>  
> import P2.Parser (parseData)  
> import P2.Printing (prettyDirect, prettyIncPairs, prettyNearest, prettyTable)  
>  
> import System.FilePath.Glob (glob)  
>  
> import Text.Parsec.String (parseFromFile)
```

And off we go.

```
> main :: IO ()  
> main = do  
>   -- First glob all the tsp file names.  
>   tspFiles <- sort <$> glob "all_tsp/*.tsp"  
>   -- Parse them all and only grab the ones that are valid (`EUC_2D` or `GEO`).  
>   tsps <- mapM (parseFromFile parseData) tspFiles  
>   let tsps' = [(fp, ns) | (fp, Right ns) <- zip tspFiles tsps]  
>   -- We first need to write the increasing distance pairs for each file.  
>   for_ tsps' $ uncurry prettyIncPairs  
>   -- Next we want to take compute the direct tour from 1 to n.  
>   for_ tsps' $ uncurry prettyDirect  
>   -- Now we need to do the nearest neighbor.  
>   for_ tsps' $ uncurry prettyNearest  
>   writeFile "comparison.md" $ prettyTable tsps'
```

We can see the result of running this program.

First the direct tour.



```
File Edit View Terminal Tabs Help
→ p2 git:(master) X time ./Main +RTS -s -N2
Tour of a280.tsp
The distance of the tour in order 1-2-...-280-1 is: 2808 km
Tour of ali535.tsp
The distance of the tour in order 1-2-...-535-1 is: 3367358 km
Tour of berlin52.tsp
The distance of the tour in order 1-2-...-52-1 is: 22205 km
Tour of bier127.tsp
The distance of the tour in order 1-2-...-127-1 is: 393989 km
Tour of brd14051.tsp
The distance of the tour in order 1-2-...-14051-1 is: 23587594 km
Tour of burma14.tsp
The distance of the tour in order 1-2-...-14-1 is: 4659 km
Tour of ch130.tsp
The distance of the tour in order 1-2-...-130-1 is: 47797 km
Tour of ch150.tsp
The distance of the tour in order 1-2-...-150-1 is: 52814 km
Tour of d1291.tsp
The distance of the tour in order 1-2-...-1291-1 is: 150852 km
Tour of d15112.tsp
The distance of the tour in order 1-2-...-15112-1 is: 112310765 km
Tour of d1655.tsp
The distance of the tour in order 1-2-...-1655-1 is: 206034 km
Tour of d18512.tsp
The distance of the tour in order 1-2-...-18512-1 is: 29460538 km
Tour of d198.tsp
The distance of the tour in order 1-2-...-198-1 is: 22498 km
Tour of d2103.tsp
```

Screenshot of direct tour

Then the nearest neighbor tour.

```
File Edit View Terminal Tabs Help
Tour of vm1748.tsp
The distance of the tour in order 1-2-...-1748-1 is: 10005342 km
Tour of a280:
[1,280,2,3,279,278,4,277,276,275,274,273,272,271,16,17,18,19,20,21,128,127,126,125,30,31,32,29,28,27,26,22,25,23,24,14,13,12,11,10,8,7,9,6,5,260,259,258,257,254,253,208,207,210,209,252,255,256,249,248,247,244,241,240,239,238,231,232,233,234,235,236,237,246,245,243,242,250,251,230,229,228,227,226,225,224,223,222,219,218,215,214,211,212,213,216,217,220,221,203,202,200,144,143,142,141,140,139,138,137,136,135,134,270,269,268,267,266,265,264,263,262,261,15,133,132,131,130,129,154,155,153,156,152,151,177,176,181,180,179,178,150,149,148,147,146,145,199,198,197,194,195,196,201,193,192,191,190,189,188,187,185,184,183,182,161,162,163,164,165,166,167,168,169,101,100,99,98,93,94,95,96,97,92,91,90,89,81,80,79,76,75,74,73,72,71,70,67,66,65,64,63,62,118,61,60,43,42,41,40,39,38,37,36,35,34,33,124,123,122,121,120,119,157,158,159,160,175,174,173,106,105,104,103,102,170,171,172,107,108,109,110,111,112,88,83,82,84,85,86,116,115,114,113,87,117,59,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,68,69,77,78,186,204,205,206]
Distance of tour: 3157

Tour of ali535:
[1,497,315,391,490,389,133,405,461,136,470,50,88,191,397,30,413,193,76,189,396,68,122,18,16,362,101,501,340,118,185,416,529,455,445,147,479,450,449,454,98,373,128,506,231,232,409,40,261,57,145,348,22,27,176,359,228,331,514,249,95,509,94,209,301,530,412,493,507,401,494,512,65,275,314,337,492,89,212,345,203,170,293,505,324,481,262,474,400,70,81,114,129,385,495,259,369,377,112,99,408,91,270,332,290,338,175,127,319,93,533,465,460,192,116,167,172,419,28,273,372,329,437,458,534,272,280,286,210,271,83,491,90,140,179,234,414,354,55,160,107,356,111,174,258,284,297,196,321,344,171,202,384,58,166,453,366,67,25,87,214,215,358,75,2,14,131,320,498,54,477,466,265,266,162,395,279,469,335,355,277,51,115,36,217,53,317,278,155,480,487,220,21,226,20,510,531,295,511,375,431,364,276,182,515,464,15,208,264,17,485,403,104,126,402,13,195,9,282,472,475,350,156,71,119,190,415,74,374,349,241,346,60,157,380,484,260,100,188,254,105,183,351,288,428,197,194,291,78,287,325,334,238,148,132,56,8,267,285,130,281,4,483,200,435,102,482,24,31,292,35,158,250,33,143,423,11,236,46,204,10,52,169,244,72,342,239,300,387,532,146,311,438,328,486,318,124,77,294,84,246,304,168,441,39,159,41,153,420,253,3,187,64,432,478,42,468,79,242,421,150,251,110,142,535,407,73,503,390,252,443,237,161,326,163,392,247,80,467,108,306,308,12,123,135,19,339,417,360,489,245,219,103,218,440,347,436,370,221,499,353,378,177,471,323,463,263,216,32,459,352,206,198,184,199,371,62,462,6,154,113,343,205,305,327,207,47,388,368,296,452,473,429,243,406,85,527,522,517,63,520,240,523,518,519,225,97,528,521,165,164,120,386,125,138,139,230,117,365,457,316,302,363,330,227,525,69,451,256,383,425,257,269,444,439,434,524,526,516,151,59,23,361,496,144,152,223,427,173,5,333,66,37,500,224,513,149,43,382,178,235,357,268,49,86,381,141,303,367,312,341,248,376,433,448,29,398,186,504,61,393,109,255,137,38,299,44,134,394,399,447,446,310,476,424,211,313,309,7,121,82,502,213,274,283,34,229,488,410,336,96,180,418,430,426,508,201,411,92,45,456,404,106,422,298,48,442,233,307,289,322,379,222,26,181]
Distance of tour: 265685

Tour of berlin52:
[1,22,49,32,36,35,34,39,40,38,37,48,24,5,15,6,4,25,46,44,16,50,20,23,31,18,3,19,45,41,8,10,9,43,33,51,12,28,27,26,47,13,14,52,11,29,30,21,17,42,7,2]
Distance of tour: 8980
```

Screenshot of nearest tour

And finally the time of the whole program.

```

File Edit View Terminal Tabs Help
474,24,204,23,147,771,802,1038,1039,1694,1483,1529,1345,1140,1155]
Distance of tour: 408101

4,454,726,393,896 bytes allocated in the heap
236,978,149,832 bytes copied during GC
  54,144,216 bytes maximum residency (17046 sample(s))
  1,548,200 bytes maximum slop
    125 MB total memory in use (0 MB lost due to fragmentation)

           Tot time (elapsed)  Avg pause  Max pause
Gen  0      8569098 colls,    0 par   195.246s   194.016s    0.0000s    0.0025s
Gen  1      17046 colls,    0 par    82.950s    82.930s    0.0049s    0.0620s

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.001s ( 0.001s elapsed)
MUT     time 1305.708s (1323.415s elapsed)
GC      time 278.196s (276.946s elapsed)
EXIT    time    0.001s ( 0.000s elapsed)
Total   time 1583.906s (1600.362s elapsed)

Alloc rate    3,411,731,520 bytes per MUT second

Productivity  82.4% of total user, 81.6% of total elapsed

gc_alloc_block_sync: 0
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0
./Main +RTS -s 1557.95s user 25.96s system 98% cpu 26:40.37 total

```

Screenshot of total time

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 | Direct | Nearest Neighbor | Optimal |
|----------|-----------|------------------|-------------------|
| a280 | 2808 | 3157 | [2579] |
| ali535 | 3367358 | 265685 | [202310] |
| berlin52 | 22205 | 8980 | [7542] |
| bier127 | 393989 | 135737 | [118282] |
| brd14051 | 23587594 | 585382 | [468942,469445] |
| burma14 | 4659 | 4501 | [3323] |
| ch130 | 47797 | 7579 | [6110] |
| ch150 | 52814 | 8191 | [6528] |
| d1291 | 150852 | 60214 | [50801] |
| d15112 | 112310765 | 1960503 | [1564590,1573152] |
| d1655 | 206034 | 74022 | [62128] |
| d18512 | 29460538 | 799220 | [644650,645488] |
| d198 | 22498 | 18240 | [15780] |
| d2103 | 141309 | 86653 | [79952,80450] |
| d493 | 113542 | 41661 | [35002] |
| d657 | 232152 | 61626 | [48912] |
| eil101 | 2062 | 803 | [629] |
| eil51 | 1308 | 511 | [426] |
| eil76 | 1969 | 642 | [538] |

| Filename | Direct | Nearest Neighbor | Optimal |
|----------|----------|------------------|-----------------|
| fl1400 | 172735 | 27447 | [20127] |
| fl1577 | 51304 | 27996 | [22204,22249] |
| fl3795 | 169398 | 35285 | [28723,28772] |
| fl417 | 55445 | 15013 | [11861] |
| fnl4461 | 5872302 | 229963 | [182566] |
| gil262 | 26298 | 3208 | [2378] |
| gr137 | 97312 | 98781 | [69853] |
| gr202 | 59536 | 54092 | [40160] |
| gr229 | 180306 | 162109 | [134602] |
| gr431 | 234813 | 206628 | [171414] |
| gr666 | 425823 | 364429 | [294358] |
| gr96 | 81283 | 74939 | [55209] |
| kroA100 | 191387 | 27807 | [21282] |
| kroA150 | 287844 | 33633 | [26524] |
| kroA200 | 373938 | 35859 | [29368] |
| kroB100 | 157190 | 29158 | [22141] |
| kroB150 | 273239 | 34499 | [26130] |
| kroB200 | 327456 | 36980 | [29437] |
| kroC100 | 183466 | 26227 | [20749] |
| kroD100 | 170990 | 26947 | [21294] |
| kroE100 | 188351 | 27460 | [22068] |
| lin105 | 36480 | 20356 | [14379] |
| lin318 | 119872 | 54019 | [42029] |
| linhp318 | 119872 | 54019 | [41345] |
| nrw1379 | 712343 | 68964 | [56638] |
| p654 | 107737 | 43457 | [34643] |
| pcb1173 | 123837 | 71978 | [56892] |
| pcb3038 | 295793 | 176310 | [137694] |
| pcb442 | 221440 | 61979 | [50778] |
| pr1002 | 349403 | 331103 | [259045] |
| pr107 | 62752 | 46680 | [44303] |
| pr124 | 98941 | 69297 | [59030] |
| pr136 | 287028 | 120769 | [96772] |
| pr144 | 93526 | 61652 | [58537] |
| pr152 | 160980 | 85699 | [73682] |
| pr226 | 110417 | 94683 | [80369] |
| pr2392 | 378032 | 461170 | [378032] |
| pr264 | 77977 | 58023 | [49135] |
| pr299 | 83506 | 59890 | [48191] |
| pr439 | 270646 | 131281 | [107217] |
| pr76 | 150781 | 153462 | [108159] |
| rat195 | 4030 | 2752 | [2323] |
| rat575 | 12934 | 8605 | [6773] |
| rat783 | 72134 | 11054 | [8806] |
| rat99 | 2124 | 1554 | [1211] |
| rd100 | 50560 | 9938 | [7910] |
| rd400 | 215558 | 19183 | [15281] |
| rl11849 | 86621276 | 1125239 | [920847,923368] |
| rl1304 | 3231694 | 335779 | [252948] |
| rl1323 | 3088190 | 332103 | [270199] |
| rl1889 | 6601280 | 389270 | [316536] |
| rl5915 | 10145025 | 695602 | [565040,565530] |

| Filename | Direct | Nearest Neighbor | Optimal |
|-----------|------------|------------------|---------------------|
| rl5934 | 9861324 | 672412 | [554070,556045] |
| st70 | 3410 | 830 | [675] |
| ts225 | 276540 | 152493 | [126643] |
| tsp225 | 10308 | 4722 | [3919] |
| u1060 | 260174 | 308980 | [224094] |
| u1432 | 183070 | 188807 | [152970] |
| u159 | 43381 | 54675 | [42080] |
| u1817 | 71460 | 72030 | [57201] |
| u2152 | 81704 | 79260 | [64253] |
| u2319 | 281496 | 278765 | [234256] |
| u574 | 40197 | 50459 | [36905] |
| u724 | 157485 | 52942 | [41910] |
| ulysses16 | 9693 | 8081 | [6859] |
| ulysses22 | 12316 | 8248 | [7013] |
| usa13509 | 1590833042 | 24973197 | [19947008,19982889] |
| vm1084 | 5350742 | 301475 | [239297] |
| vm1748 | 10005342 | 408101 | [336556] |

We want to talk abstractly about the different distances. Since they (so far) seem to have the same types, we abstract over them as a typeclass.

```
> {-# LANGUAGE DataKinds #-}
> {-# LANGUAGE KindSignatures #-}
> module P2.Distance (Distance(..)) where

> import Control.Lens ((^.), to)
>
> import Data.Int (Int32)
>
> import P2.Node (EdgeWeightType(..), Node(..), r1, r2)
```

For each tagged type, we need a different distance function. Since this is the only thing that changes in the whole algorithm, we abstract over it with a typeclass.

```
> class Distance (s :: EdgeWeightType) where
>     distance :: Node s -> Node s -> Int32
```

Then we can provide an implementation for each Node synonym.

The algorithm given in the TSPLIB pdf. This is basically a direct copy. Any other implementation ends up with slightly different values.

```
> instance Distance 'GEO where
>     distance i j = floor $ rrr * acos (((1 + q1) * q2 - (1 - q1) * q3) / 2) + 1
>     where
>         rrr, q1, q2, q3, latI, latJ, longI, longJ, pi' :: Double
>         q1 = cos $ longI - longJ
>         q2 = cos $ latI - latJ
>         q3 = cos $ latI + latJ
>         latI = radians i latDegrees latMinutes
```

```

> latJ = radians j latDegrees latMinutes
> longI = radians i longDegrees longMinutes
> longJ = radians j longDegrees longMinutes
> radians node d m = pi' * (node^.d + 5 * node^.m / 3) / 180
> latDegrees = r1.to (fromInteger . round)
> latMinutes = r1.to ((-) <*> fromInteger . round)
> longDegrees = r2.to (fromInteger . round)
> longMinutes = r2.to ((-) <*> fromInteger . round)
> pi' = 3.141592
> rrr = 6378.388

```

Euclidean distance on nodes.

```

> instance Distance 'EUC_2D where
>   distance i j = round $ sqrt $ dx ** 2 + dy ** 2
>   where
>     dx = i^.r1 - j^.r1
>     dy = i^.r2 - j^.r2

```

Here we collect different morphisms on nodes.

```

> module P2.Morphism ( incPairsDists, nearestNeighbors
>                     , nearestDistance, tourDistance
>                     ) where
>
> import Control.Lens ((^..))
>
> import Data.Foldable (minimumBy)
> import Data.Int (Int32)
> import Data.List (delete)
> import Data.Ord (comparing)
> import Data.Tuple (swap)
>
> import P2.Distance (Distance(..))
> import P2.Node (Node(..), number)
>
> import qualified Data.Set as S

```

We construct a tour by pairing up each node with its immediate successor, and wrapping the end of the list around.

```

> tour :: [a] -> [(a, a)]
> tour = zip <*> (uncurry (++) . swap . splitAt 1)
>
> tourDistance :: (Distance s) => [Node s] -> Int32
> tourDistance = sum . fmap (uncurry distance) . tour

```

We want to be able to make increasing pairs of nodes.

```

> incPairs :: [Node s] -> [(Node s, Node s)]
> incPairs ns = [(i, j) | i <- ns, j <- ns, i^.number < j^.number]
>

```

```

> incPairsDists :: (Distance s) => [Node s] -> [(Int, Int, Int32)]
> incPairsDists = fmap go . incPairs
>   where go (i, j) = (i^.number, j^.number, distance i j)

```

Compute the nearest neighbors and then run the tour.

```

> nearestDistance :: Distance s => [Node s] -> Int32
> nearestDistance = tourDistance . nearestNeighbors

```

Make nearestNeighbors' slightly less polymorphic.

```

> nearestNeighbors :: Distance s => [Node s] -> [Node s]
> nearestNeighbors = nearestNeighbors' distance

```

Codify the nearest neighbor algorithm as a list of visited nodes, and a set of unvisited ones.

```

> data Nearest a = Nearest
>   { visited    :: [a]
>     , unvisited :: S.Set a
>   }

```

The general Nearest Neighbors algorithm.

```

> nearestNeighbors' :: (Ord a, Ord b) => (a -> a -> b) -> [a] -> [a]
> nearestNeighbors' d ns = go $ mkNearest ns
>   where
>     -- Our main iteration.
>     -- We're done when there's no more unvisited nodes.
>     go (Nearest vs          us) | S.null us = reverse vs
>     -- Push the nearest onto the front and go with the rest.
>     go (Nearest vs@(current:_) us) =
>       let (next, rest) = nearest current us
>       in go $ Nearest (next:vs) rest
>     -- If we haven't visited anything yet,
>     -- grab the first thing out of the set.
>     go (Nearest []          us) =
>       go $ Nearest [S.findMin us] (S.deleteMin us)
>     -- Compute the nearest distance.
>     nearest i us =
>       let ns = S.map ((,) <*> d i) us
>       in n = fst $ minimumBy (comparing snd) ns
>       in (n, S.delete n us)
>
> mkNearest :: Ord a => [a] -> Nearest a
> mkNearest xs =
>   let (vs, us) = splitAt 1 xs in Nearest vs (S.fromList us)

```

We want is to model the data somehow, so we create nodes.

```

> {-# LANGUAGE DataKinds #-}
> {-# LANGUAGE KindSignatures #-}
> {-# LANGUAGE TemplateHaskell #-}
> module P2.Node where

```



```

> import Control.Lens (makeLenses)
>
> import Data.Typeable (Typeable)

```

We want to be able to talk about the different types of input formats. At the moment we have to handle EUC_2D and GEO. Since there's no actual difference between the format of the data, we can use `Unit` types to represent the different formats. We promote them to the kind level with `DataKinds` so that we can talk about which “kind” of node we're dealing with.

```

> data EdgeWeightType = EUC_2D | GEO deriving (Eq, Show)

```

Nodes are parameterized by a phantom `EdgeWeightType`. This is because there's no actual difference between the representation of nodes, but the way we calculate the distance changes depending on which `EdgeWeightType` we have.

```

> data Node (s :: EdgeWeightType) = Node
>   { _number :: Int
>   , _r1      :: Double
>   , _r2      :: Double
>   } deriving (Eq, Ord, Show, Typeable)

```

And of course generate some lenses so we don't end up doing too much work ourselves.

```

> makeLenses ''Node

```

We also want a sum of the possible lists of nodes. This will help when we get to parsing.

```

> data Nodes = GEOS      [Node 'GEO]
>             | EUC_2DS  [Node 'EUC_2D]

```

Now we need some way to parse the file. Since we don't have to concern ourselves with the entire format at this time, we do the most simplistic approach possible.

```

> module P2.Parser (parseData) where
>
> import Control.Applicative ((<|>), empty)
>
> import P2.Node (EdgeWeightType(..), Node(..), Nodes(..))
> import P2.TSPLIB (TSPLIBKeywords(..))
>
> import Text.Parsec ( anyChar, endOfLine, eof, choice, count
>                     , manyTill, spaces, string, try
>                     )
> import Text.Parsec.String (Parser, parseFromFile)
> import Text.ParserCombinators.Parsec.Number ( floating2, nat
>                                                , sign
>                                                )

```

First, we ignore everything up to the `DIMENSION`. We use the `DIMENSION` to tell the parser how many nodes to expect. The formatting doesn't work out well enough in general to allow a parser that doesn't depend on `DIMENSION`. Next, we need the `EDGE_WEIGHT_TYPE` so we know what kind of nodes we're dealing with, then ignore everything up until `NODE_COORD_SECTION`, then parse `DIMENSION` nodes and tag them with their .

```

> parseData :: Parser Nodes
> parseData = do
>   dim <- parseDimension
>   ewt <- parseEdgeType
>   anyChar `manyThen` `` NODE_COORD_SECTION
>   -- This should be done better,
>   case ewt of
>     Just GEO    -> GEOS    <$> parseNodes dim
>     Just EUC_2D -> EUC_2DS <$> parseNodes dim
>     Nothing     -> empty

> parseNodes :: Int -> Parser [Node s]
> parseNodes = (`count` parseNode)

> parseDimension :: Parser Int
> parseDimension = anyChar `manyThen` `` DIMENSION *> colon *> nat

> parseEdgeType :: Parser (Maybe EdgeWeightType)
> parseEdgeType = do
>   anyChar `manyThen` `` EDGE_WEIGHT_TYPE
>   colon
>   choice [ Just GEO    <$ try (string' GEO)
>            , Just EUC_2D <$ try (string' EUC_2D)
>            , pure Nothing
>          ]

```

The format for the nodes is: <integer> <real> <real>.

We create a parser for this schema.

```

> parseNode :: Parser (Node s)
> parseNode =
>   Node <$> (spaces *> nat)
>           <*> (spaces *> (sign <*> floating2 True))
>           <*> (spaces *> (sign <*> floating2 True))
>           <*> spaces

```

And some helpers that probably should exist somewhere else.

```

> colon :: Parser ()
> colon = spaces *> string ":" *> spaces
> manyThen :: Parser a -> Parser b -> Parser [a]
> manyThen p = manyTill p . try
> manyThen' :: Show s => Parser a -> s -> Parser [a]
> manyThen' p = manyThen p . string'
> string' :: Show s => s -> Parser String
> string' = string . show

```

Here we want to have different ways of printing the nodes. This is kind of a conglomeration of IO a functions and evaluations. Not a whole lot of purity in here.

```

> {-# LANGUAGE RankNTypes #-}
> module P2.Printing where

```

```

> import Control.Lens (view)
>
> import Data.Foldable (foldl')
> import Data.Int (Int32)
> import Data.List (transpose)
>
> import P2.Distance (Distance)
> import P2.Morphism (incPairsDists, nearestDistance, nearestNeighbors, tourDistance)
> import P2.Node (Node, Nodes(..), number)
> import P2.Table (augment)
>
> import System.FilePath ((</>), replaceExtension, takeBaseName, takeFileName)
>
> import Text.Printf (PrintfType, printf)
> import Text.PrettyPrint.Boxes
>
> import qualified Data.Map as M

```

First, make a helper catamorphism that unwraps the Nodes.

```

> cataNode :: PrintfType a
>           => (forall s. (Distance s) => ([Node s] -> a))
>           -> Nodes
>           -> a
> cataNode f (GEOS ns) = f ns
> cataNode f (EUC_2DS ns) = f ns

> cataNode' :: (forall s. (Distance s) => ([Node s] -> a))
>            -> Nodes
>            -> a
> cataNode' f (GEOS ns) = f ns
> cataNode' f (EUC_2DS ns) = f ns

```

We can print a list of nodes directly.

```

> prettyDirect :: FilePath -> Nodes -> IO ()
> prettyDirect tsp n = do
>   printf "Tour of %s\n" (takeFileName tsp)
>   cataNode direct' n
>
> direct :: PrintfType a => Int -> Int32 -> a
> direct = printf
>   "The distance of the tour in order 1-2-...-%d-1 is: %d km\n"
>
> direct' :: (Distance s, PrintfType a) => [Node s] -> a
> direct' xs = direct (length xs) . tourDistance $ xs

```

We can construct pairs in increasing order and take the distances.

```

> prettyIncPairs :: FilePath -> Nodes -> IO ()
> prettyIncPairs = writeIncPairs . mkIncPairsName
>

```

```

> writeIncPairs :: FilePath -> Nodes -> IO ()
> writeIncPairs fp = cataNode (writeFile fp . prettyInc)
>
> prettyInc :: Distance s => [Node s] -> String
> prettyInc xs = unlines . (len:) . fmap go . incPairsDists $ xs
>   where
>     go = uncurry3 $ printf "%d %d %d"
>     len = show $ length xs
>     uncurry3 f (x, y, z) = f x y z
>
> mkIncPairsName :: FilePath -> FilePath
> mkIncPairsName fp = incPairsDir </> replaceExtension (takeFileName fp) incPairsExt
>
> incPairsDir :: FilePath
> incPairsDir = "inc_pairs"
>
> incPairsExt :: FilePath
> incPairsExt = "inc"

```

We can find the nearest neighbors and take the distances.

```

> prettyNearest :: FilePath -> Nodes -> IO ()
> prettyNearest fp = cataNode (prettyNearest' fp)
>
> prettyNearest' :: Distance s => FilePath -> [Node s] -> IO ()
> prettyNearest' fp ns = do
>   let nearest = nearestNeighbors ns
>   printf "Tour of %s:\n" $ takeBaseName fp
>   print $ fmap (view number) nearest
>   printf "Distance of tour: %d\n\n" $ tourDistance nearest

```

We want to make things easier on ourselves for creating a table.

```

> prettyTable :: [(FilePath, Nodes)] -> String
> prettyTable = prettyTable' . augment . allBothDistances
>
> prettyTable' :: M.Map String (Int32, Int32, [Int32]) -> String
> prettyTable' = prettyCols . M.toList
>
> prettyCols :: [(String, (Int32, Int32, [Int32]))] -> String
> prettyCols = render . hsep 2 left . mkHeaders . map (vcat left) . transpose . map go
>   where
>     go (name, (x, y, z)) = [text name, text' x, text' y, text' z]
>     mkHeaders = zipWith mkHeaders' headers
>     mkHeaders' h c = h // separator (max (cols h) (cols c)) // c
>     separator n = text (replicate n '-')
>     headers = map text ["Filename", "Direct", "Nearest Neighbor", "Optimal"]
>     text' :: Show a => a -> Box
>     text' = text . show
>
> allBothDistances :: [(FilePath, Nodes)] -> M.Map String (Int32, Int32)
> allBothDistances =
>   foldl' (\acc (x, y) -> uncurry M.insert (bothDistances x y) acc) M.empty

```

```

>
> bothDistances :: FilePath -> Nodes -> (String, (Int32, Int32))
> bothDistances fp = cataNode' (bothDistances' fp)
>
> bothDistances' :: Distance s => FilePath -> [Node s] -> (String, (Int32, Int32))
> bothDistances' fp ns = (takeBaseName fp, (tourDistance ns, nearestDistance ns))

```

Since we have to compare the values, we build up some stuff for working with the table.

```

> module P2.Table where

> import Data.Int (Int32)
>
> import qualified Data.Map as M

> augment :: M.Map String (a, b) -> M.Map String (a, b, [Int32])
> augment = M.intersectionWith (\z (x, y) -> (x, y, z)) bounds

```

We statically know the optimum bounds, so let's just store that and be done with it.

```

> bounds :: M.Map String [Int32]
> bounds = M.fromList
>   [ ("a280",      [2579])
>     , ("ali535",   [202310])
>     , ("bayg29",   [1610])
>     , ("bays29",   [2020])
>     , ("berlin52", [7542])
>     , ("bier127",  [118282])
>     , ("brd14051", [468942, 469445])
>     , ("burma14",  [3323])
>     , ("ch130",    [6110])
>     , ("ch150",    [6528])
>     , ("d198",     [15780])
>     , ("d493",     [35002])
>     , ("d657",     [48912])
>     , ("d1291",    [50801])
>     , ("d1655",    [62128])
>     , ("d2103",    [79952, 80450])
>     , ("d15112",   [1564590, 1573152])
>     , ("d18512",   [644650, 645488])
>     , ("eil51",    [426])
>     , ("eil76",    [538])
>     , ("eil101",   [629])
>     , ("fl1417",   [11861])
>     , ("fl1400",   [20127])
>     , ("fl1577",   [22204, 22249])
>     , ("fl3795",   [28723, 28772])
>     , ("fnl4461",  [182566])
>     , ("gil262",   [2378])
>     , ("gr96",     [55209])
>     , ("gr137",    [69853])
>     , ("gr202",    [40160])

```

```

> , ("gr229", [134602])
> , ("gr431", [171414])
> , ("gr666", [294358])
> , ("kroA100", [21282])
> , ("kroB100", [22141])
> , ("kroC100", [20749])
> , ("kroD100", [21294])
> , ("kroE100", [22068])
> , ("kroA150", [26524])
> , ("kroB150", [26130])
> , ("kroA200", [29368])
> , ("kroB200", [29437])
> , ("lin105", [14379])
> , ("lin318", [42029])
> , ("linhp318", [41345])
> , ("nrw1379", [56638])
> , ("p654", [34643])
> , ("pcb442", [50778])
> , ("pcb1173", [56892])
> , ("pcb3038", [137694])
> , ("pr76", [108159])
> , ("pr107", [44303])
> , ("pr124", [59030])
> , ("pr136", [96772])
> , ("pr144", [58537])
> , ("pr152", [73682])
> , ("pr226", [80369])
> , ("pr264", [49135])
> , ("pr299", [48191])
> , ("pr439", [107217])
> , ("pr1002", [259045])
> , ("pr2392", [378032])
> , ("rat99", [1211])
> , ("rat195", [2323])
> , ("rat575", [6773])
> , ("rat783", [8806])
> , ("rd100", [7910])
> , ("rd400", [15281])
> , ("rl1304", [252948])
> , ("rl1323", [270199])
> , ("rl1889", [316536])
> , ("rl5915", [565040, 565530])
> , ("rl5934", [554070, 556045])
> , ("rl11849", [920847, 923368])
> , ("st70", [675])
> , ("ts225", [126643])
> , ("tsp225", [3919])
> , ("u159", [42080])
> , ("u574", [36905])
> , ("u724", [41910])
> , ("u1060", [224094])
> , ("u1432", [152970])
> , ("u1817", [57201])
> , ("u2152", [64253])

```

```

> , ("u2319", [234256])
> , ("ulysses16", [6859])
> , ("ulysses22", [7013])
> , ("usa13509", [19947008, 19982889])
> , ("vm1084", [239297])
> , ("vm1748", [336556])
> ]

```

Since we seem to be parsing the whole file eventually, let's start encoding the grammar. We might not use it all at the moment, but better to have it done now.

```

> module P2.TSPLIB (TSPLIBKeywords(..)) where

> data TSPLIBKeywords = NAME
> | TYPE
> | COMMENT
> | CAPACITY
> | DIMENSION
> | EDGE_WEIGHT_TYPE
> | EDGE_WEIGHT_FORMAT
> | EDGE_DATA_FORMAT
> | NODE_COORD_TYPE
> | DISPLAY_DATA_TYPE
> | EOF
> | NODE_COORD_SECTION
> | DEPOT_SECTION
> | DEMAND_SECTION
> | EDGE_DATA_SECTION
> | FIXED_EDGES_SECTION
> | DISPLAY_DATA_SECTION
> | TOUR_SECTION
> | EDGE_WEIGHT_SECTION
> deriving (Eq, Show)

```