```
{-# LANGUAGE UndecidableInstances, FlexibleInstances #-}
module Parcial2.Labyrinth where
  import Control.Exception
  import Control.Arrow (first, second)
  import Control.Monad.Fix
  import Data.Tuple (swap)
  import Data.Maybe (isJust, fromJust, fromMaybe)
  import Data.Set (Set, member, elemAt)
  import qualified Data.Set as Set
  import Data.Map (Map)
  import qualified Data.Map as Map
  import GHC.Real (infinity)
  import Parcial2.ReadLabyrinth
  import GeneticAlgorithm
  import System.Random
```

1. Introducción

El mapa (laberinto), descrito en la tarea, se define como un grafo: nodos — un conjunto de puntos (con posiciones correspondientes); aristas — la existencia de rutas directas.

```
data Labyrinth point = Labyrinth {
  nodes :: Set point,
  edges :: Set (point, point),
  initial :: point,
  target :: point
  }
edgeOf p es = any (`member` es) [p, swap p]
mapPoints f (Labyrinth ns es i t) = Labyrinth {
  nodes = Set.map f ns,
  edges = Set.map (first f ∘ second f) es,
  initial = f i,
  target = f t
  }
```

Se define la *distancia directa* entre los nodos que están conectados por una arista.

```
data DirectDistance point dist = DirectDistance {
  labyrinthDist :: Labyrinth point → point → point → Maybe dist
  }
mkDirectDistance f = DirectDistance $ λl v1 v2 →
  if (v1, v2) `edgeOf` edges l then Just (f v1 v2) else Nothing
```

El algoritmo genético abstracto está definido en src/GeneticAlgorithm.hs. Su implementación se presentará a continuación.

2. Implementación

2.1. Lectura de mapas

Se utiliza un mapa 2D:

```
newtype Point2D = Point2D (Int, Int) deriving (Eq, Ord)
instance Show Point2D where
  show (Point2D (x, y)) = show x ++ "-" ++ show y


type Labyrinth2D = Labyrinth Point2D
```

La lectura del archivo del mapa se encuentra en src/Parcial2/ReadLabyrinth.hs. Aquí se presenta la construcción del grafo a partir del mapa leido.

```
readLabyrinth2D :: FilePath → IO (Either [String] Labyrinth2D)
readLabyrinth2D file = build <$> try (readFile file)
  where
    build (Left err) = Left [displayException (err :: SomeException)]
    build (Right s) = case parseLabyrinth s of
                        Left errS → Left errS
                        Right l → Right (build' l)
    build' (LabyrinthDescription n conn (i, t) coords) =
      let get = Point2D ∘ (coords!!)
      in Labyrinth
        (Set.fromList $ map Point2D coords)
        (Set.fromList $ map (first get ∘ second get) coords)
        (get i)
        (get t)
```

2.2. Adoptación

El valor de *aptitud de adoptación* se llamará *ruta* y se define para permitir destinguir facilmente los dos tipos de rutas (encodificadas en los cromosomas) posibles:

- **Ruta completa**: es una ruta valida (∃ una conección entre cada par de genes adjuntos) que contiene en punto inicial y el punto meta.

  Se caracteriza por la *longitud de la ruta*.

  El resultado, esperado del algoritmo genético es la más corta de estas rutas.

- **Ruta parcial**: es una ruta que 1) contiene pares de genes adjuntos, los cuales no están conectados, o 2) no contiene los ambos puntos: inicio y meta.

  Se caracteriza por los tres valores:

  1. $valides = \dfrac{\text{número de aristas existentes}}{\text{número de aristas total}}$.
     aristas existentes — aristas que existen entre los pares de genes adjuntos.
  2. los *puntos de interés* que contiene la ruta.
  3. la longitud sumatoria de las sub-rutas en el cromosoma.

```
data POI = POIInit | POITarget deriving (Eq, Ord, Enum, Show)
data POIs = POINone | POISome POI | POIBoth deriving (Eq, Show)
instance Ord POIs where
  x `compare` y = val x `compare` val y where
    val POINone = 0
    val (POISome _) = 1
    val POIBoth = 2
data Route =
  CompleteRoute {routeLength :: Double}
  | PartialRoute {partialValidess :: Double
                 , partialPOI     :: POIs
                 , partialLength  :: Double
                 }
  deriving (Eq, Show)
```

Para la busqueda de la ruta mas corta, se define el orden sobre las rutas de tal manera, que una lista de *rutas*, ordenada ascendentamente, tendrá los mejores elementos en el principio.

1. Cualquiera *ruta completa* es minor que cualquiera *ruta parcial*.

$$\forall x \in \textit{ruta completa}, y \in \textit{ruta parcial} \implies x < y$$

2. Dos *rutas completas* se comparan por su longitud sin cambios de la orden.

$$\begin{array}{l} \forall x \in \textit{ruta completa}, x \sim r_1 \\ \forall y \in \textit{ruta completa}, y \sim r_2 \end{array} \implies \begin{cases} x < y & \text{si } r_1 < r_2 \\ x > y & \text{si } r_1 > r_2 \\ x = y & \text{en otro caso} \end{cases}$$

3. Dos *rutas parciales* se comparan por sus tres componentes en orden lexicográfico, que quiere decir que primero se comparan los primeros elementes, si son igual, se comparan los segundos, etc., hasta que la comparación da un resultado diferente de igualdad o se termina la lista.

El orden de la comparación se cambia al opuesto.

$$\begin{array}{l}\forall\, x \in \textit{ruta parcial} \\ \quad x \sim \langle v_x, i_x, l_x\rangle \\ \forall\, y \in \textit{ruta parcial} \\ \quad y \sim \langle v_y, i_y, l_y\rangle\end{array} \quad \Longrightarrow \quad \begin{cases} x < y & \text{si } \langle v_x, i_x, l_x\rangle > \langle v_y, i_y, l_y\rangle \\ x > y & \text{si } \langle v_x, i_x, l_x\rangle < \langle v_y, i_y, l_y\rangle \\ x = y & \text{en otro caso}\end{cases}$$

```
instance Ord Route where
  compare (CompleteRoute x) (CompleteRoute y) = compare x y
  compare (PartialRoute v1 i1 l1) (PartialRoute v2 i2 l2) =
    compare (v2, i2, l2) (v1, i1, l1)
  compare (CompleteRoute _) PartialRoute {}     = LT
  compare PartialRoute {}     (CompleteRoute _) = GT
```

Las pruebas del contenedor *Route* se encuentran en test/Parcial2/Route.hs.

2.3. Algoritmo genético

```
data GAParams = GAParams {
  gaChromGenMaxChainLen :: Int,
  gaChromGenMaxChains :: Int
  }
data GACache = GACache {
  cacheNeighbours :: Map Point2D [Point2D]
  }
neighboursOf cache point = fromMaybe []
  $ Map.lookup point (cacheNeighbours cache)
data GA = GA Labyrinth2D GAParams GACache
```

Se usa adelante un alias de tuple `(a,a)` para denotar el número de hijos de *crossover*.

```
newtype Pair a = Pair (a, a)
unwrapPair (Pair p) = p
pair2List (Pair (f, s)) = [f, s]
```

Se define la metrica sobre los puntos del grafo:

$$\text{dist}(p_1, p_2) = \begin{cases} \textit{Just } d_E(p_1, p_2) & \text{si } \exists \text{ arista, connectando } p_1 \text{ y } p_2 \\ \textit{Nothing} & \text{en otro caso}\end{cases}, \text{ donde}$$

d_E — es la distancia euclidiana entre dos puntos.

```
eDist' = mkDirectDistance $
  λ(Point2D (x1, x2)) (Point2D (y1, y2)) →
    sqrt $ fromIntegral $
    abs (x1 − x2) ↑ 2 + abs (y1 − y2) ↑ 2
eDist = labyrinthDist eDist'
```

Se define la instancia de la clase *GeneticAlgorithm* para *GA* empezando con los tipos y siguiendo con los métodos.

instance *GeneticAlgorithm GA* **where**

(1) Un *gen* se define como nodo del laberinto y un *cromosoma* como una lista de genes.

Los cromosomas no deben de tener repeticiones.

```
type Gene GA = Point2D
type Chromosome GA = [Point2D]
  -- listGenes :: Chromosome ga → [Gene ga]
listGenes = id
```

(2) Los valores de aptitud ya fueron descritos previamente.

```
type Fitness GA = Route
```

(3) Para denotar que la operación de *crossover* preserva el tamaño de población, su resultado se marca como un par de hijos.

```
type CrossoverChildren GA = Pair
```

(4) La información de entrada para generación de la población — el laberinto.

```
type InputData GA = Labyrinth2D
```

(5) El resultado es el mejor cromosoma obtenido.

```
type ResultData GA = Chromosome GA
```

(6) La **aptitud de adoptación** fue descrita en sección 2.2.

```
  -- fitness :: ga → Chromosome ga → Fitness ga
fitness (GA l _ _) genes =
  let lPairs (f : s : t) = (f, s) : lPairs (s : t)
```

```
    lPairs _ = []
    dists = map (uncurry $ eDist l) (lPairs genes)
in if isJust `all` dists
  then    -- is a valid route
    CompleteRoute ∘ sum $ map fromJust dists
  else    -- is incomplete
    let valid = filter isJust dists
      v = fromIntegral (length valid)
        / fromIntegral (length dists)
      hasInit = elem (initial l) genes
      hasFin = elem (target l) genes
      poi = case (hasInit, hasFin) of
        (True, True) → POIBoth
        (True, False) → POISome POIInit
        (False, True) → POISome POITarget
        _             → POINone
      len = sum $ map fromJust valid
      in PartialRoute v poi len
```

(7) Generación de cromosomas aleatorios.

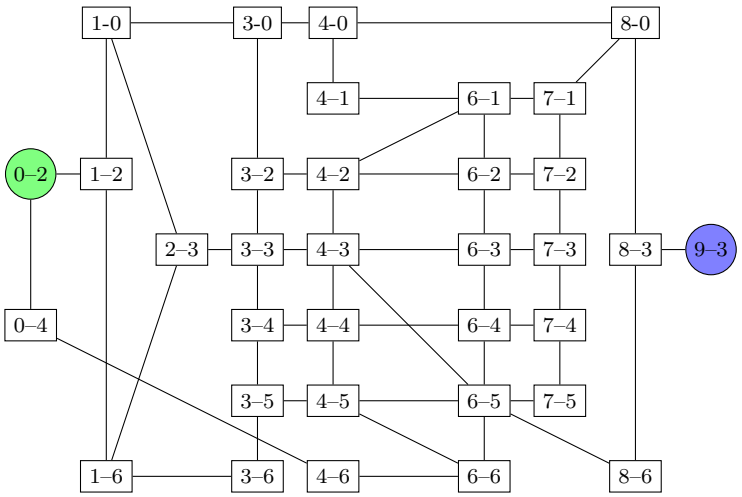

Figura 1: Un exemplo de mapa, inicio: 0–2, meta: 9–3.

Para mejorar las poblaciones iniciales, las cromosomas se componen de *cadenas* – secuencias de genes, que son sub-rutas validas de tamaños diferentes.

En la figura 1 se presenta un ejemplo de un mapa y en la figura 2 se presenta un exemplo de cromosomas generados.

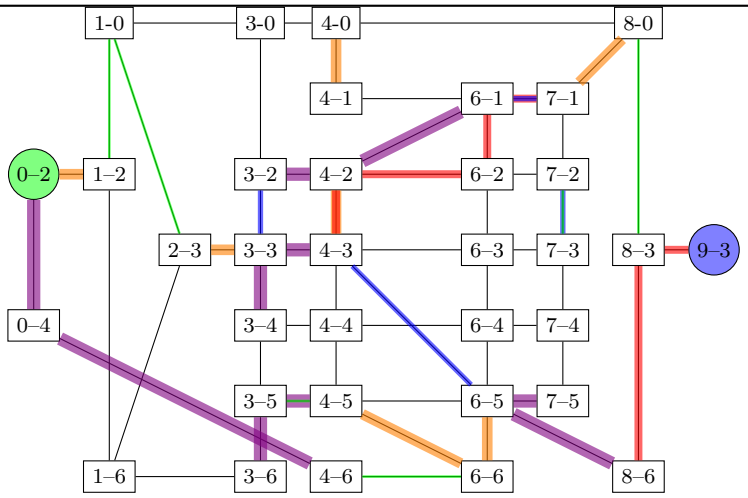

Figura 2: Se presentan algunos cromosomas en el mapa. Los cromosomas ● ● ● están compuestas de pares de genes, conectados por aristas; mientras que los cromosomas ● ● están compuestos de cadenas de genes, conectados por aristas, de longitud 3. *(Son de diferente grosor para que se ven mejor las conecciones que existen en varios cromosomas)*

```
  -- randomChromosome :: ga → IO (Chromosome ga)
randomChromosome (GA l params cache) = do
  let
```

Un gen aleatorio se selecciona entre todos los nodos del mapa, y se regenera en caso de que este gen ya fue generado previamente.

```
randPoint = first (`elemAt`nodes l)
  ∘ randomR (0, length (nodes l) − 1)
rand prev = fix $
  λf g →
    let (r, g′) = randPoint g
    in if r ∈ prev then f g′ else (r, g′)
```

Se empieza con generación del primer punto

```
randChain :: StdGen → Int → [Point2D] → [Point2D]
randChain g′ len prev = nextRand [first′] g″
  where (first′, g″) = rand prev g′
```

Los demas de genes se seleccionan desde los vicinos (los nodos directamente connectados) del gen previo.

Durante la generación de cadenas se consideran las cadenas, generadas previamente, para no permitir repeticiones de genes.

Si se encontró una repetición, se intente 1) buscar a otro vicino, que no se repita; 2) cambiar la dirección de generación; 3) buscar a otro vicino, con la nueva dirección. En caso que todas las opciones fallan, la cadena se queda de tamaño incompleto.

```
oneOf xs = first (xs!!) ∘ randomR (0, length xs − 1)
nextRand = nextRand′ False 0
nextRand′ rev c chain@(h : t) g =
  let neighbours = cache ‘neighboursOf‘ h
      (r, g′) = oneOf neighbours (g :: StdGen)
      moreTries = c < 5 ∗ length neighbours
  in if r ∈ prev ∨ r ∈ chain
     then    -- connected to some other chain
       if moreTries then nextRand′ rev (c + 1) chain g′   -- 1 / 3
         else if rev then chain   -- incomplete
           else nextRand′ True (c + 1) chain g′   -- 2
     else nextRand′ rev c (r : chain) g′   -- next
```

Se selecciona alioriamente la longetud de *cadenas*.

```
chainLen ← randomRIO (1, gaChromGenMaxChainLen params)
```

Se selecciona alioriamente el número de *cadenas*.

```
chainCnt ← randomRIO (1, gaChromGenMaxChains params)
```

Se genera el cromosoma.

```
g ← getStdGen
let f _ = randChain g chainLen
return $ foldr f [ ] [1 .. chainCnt]
```

(8) ?

```
-- crossover :: Chromosome ga → Chromosome ga
--           → CrossoverChildren ga (Chromosome ga)
```

⊥

⊥

⊥

(9) ?

```
-- mutate :: Chromosome ga → Chromosome ga
```

(10) ?

```
-- stopCriteria :: [Fitness ga] → Bool
```

(11) ?

```
-- newGA :: InputData ga → ga
```

Esto es un reporte preliminar

Nota
La intención es utilizar *crossover* para: 1) remplazar los "hoyos" en las rutas; 2) extender rutas existientes. La preferencia debe ser dada a las rutas que contienen un de los puntos de interes (inicio, meta).

La mutación debe extender/remplacar un gen al inicio/meta si $\exists$ una ruta directa.