

```

{-# LANGUAGE UndecidableInstances, FlexibleInstances #-}
module Partial2.Labyrinth where

import Control.Exception
import Control.Arrow (first, second, (&&&))
import Control.Monad.Fix
import Data.Tuple (swap)
import Data.List (elemIndex, sortBy, nubBy)
import Data.Maybe (isJust, fromJust, fromMaybe)
import Data.Function (on)
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 Partial2.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 l = any ('member' edges l) [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
}

isPOI p l = p ≡ initial l ∨ p ≡ target l

```

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 $ \l v1 v2 →
    if (v1, v2) `edgeOf` 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 leído.

```

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. Adaptación

El valor de *aptitud de adaptación* se llamará *ruta* y se define para permitir distinguir fácilmente los dos tipos de rutas (encodificadas en los cromosomas) posibles:

- **Ruta completa:** es una ruta valida (\exists una conexión 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 ambos puntos: inicio y meta.

Se caracteriza por tres valores:

1. $validez = \frac{\text{número de aristas existentes}}{\text{número de aristas total}}$.
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 búsqueda 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 menor que cualquier *ruta parcial*.

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

2. Dos *rutas completas* se comparan por su longitud sin cambios en el orden.

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

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

El orden de comparación se cambia al opuesto.

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

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
```

Función de utilidad: separación de las sub-rutas que se encuentran dentro de un cromosoma. Separa los puntos de interes como sub-rutas.

```
splitRoutes :: Labyrinth2D → Chromosome GA → [[Gene GA]]
splitRoutes l = reverse ∘ map reverse ∘ splitRoutes' [] [] l
splitRoutes' accSplit [] [] = accSplit
splitRoutes' accSplit accRoute _ [] = accRoute : accSplit
splitRoutes' accSplit accRoute l (h : t) =
  case accRoute of
    _ | h 'isPOI' l → splitRoutes' (addR accRoute accSplit) [h] l t
    prev: _ | prev 'isPOI' l → splitRoutes' (addR accRoute accSplit) [h] l t
    prev: _ | (prev, h) 'edgeOf' l → splitRoutes' accSplit (h : accRoute) l t
    _ → splitRoutes' (addR accRoute accSplit) [h] l t
  where addR [] s = s
        addR r s = r : s
```

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

2.3. Algoritmo genético

```
data GParams = GParams {
  gaChromGenMaxChainLen :: Int,
  gaChromGenMaxChains :: Int,
  gaPopulationSize      :: Int
}
```

```

data GACache = GACache {
  cacheNeighbours :: Map Point2D [Point2D]
}
neighboursOf cache point = fromMaybe []
  $ Map.lookup point (cacheNeighbours cache)
data GA = GA { gaLabyri :: Labyrinth2D
  , gaParams :: GAParams
  , gaCache :: GACache
  }

```

Se define la metrica sobre los puntos del grafo:

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

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'

```

```

lPairs (f : s : t) = (f, s) : lPairs (s : t)
lPairs _ = []

```

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) Dirección de búsqueda – minimización.

```
type Target GA = Min
```

- (4) 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 adaptación** fue descrita en subsección 2.2.

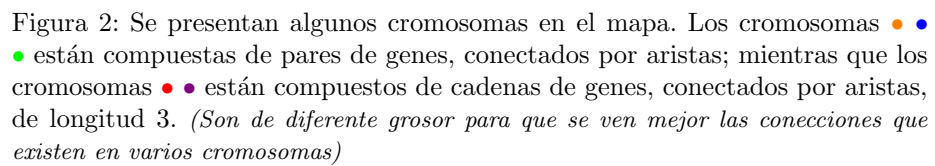
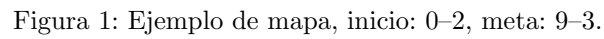
```
-- fitness :: ga -> Chromosome ga -> Fitness ga
fitness (GA l -) genes =
  let dists = map (uncurry $ eDist l) (lPairs genes)
  in if isJust 'all' dists
    then -- is a valid route
      CompleteRoute o sum $ map fromJust dists
    else -- is incomplete
      let valid = filter isJust dists
          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.

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

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

```
-- 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 hubiera sido 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 la generación del primer punto

```

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

```

Los demas genes se seleccionan desde los vecinos (los nodos directamente conectados) 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 intenta 1) buscar otro vecino, que no se repita; 2) cambiar la dirección de generación; 3) buscar a otro vecino, con la nueva dirección. En caso que todas las opciones fallen, 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 if length chain + 1 ≡ len
  then r : chain -- return
  else nextRand' rev c (r : chain) g' -- next

```

Se selecciona aleatoriamente la longitud de *cadenas*.

```

chainLen ← randomRIO (1, gaChromGenMaxChainLen params)

```

Se selecciona aleatoriamente 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) La *recombinación* de cromosomas se enfoca en remplacar las malas sub-rutas o extender rutas existentes.

Aquí solamente se define la recombinación de dos cromosomas, su selección será descrita en la subsección 2.4.

- a) Se separan las rutas con función *splitRoutes* (fue descrita en subsección 2.2) para ambos cromosomas.
- b) Se seleccionan los genes $\{c\}$, miembros de ambos cromosomas.
- c) Para ambos cromosomas se encuentran *sub-rutas intercambiables*:

$$\begin{aligned}
& \forall x \in \{c\} \\
& y \in \{c\} \implies \\
& \begin{cases} \text{secuencia } \{r_i\}_{i=1}^{N_r} & \text{si } \forall r_{j-1}, r_j \in \{r_i\}_{i=1}^{N_r} \\ & \exists \text{ arista entre } r_{j-1} \text{ y } r_j \\ \{\} & \text{en otro caso} \end{cases}
\end{aligned}$$

Se guarda también la dirección de las sub-rutas para ambos cromosomas.

- d) Las sub-rutas intercambiables se dividen dependiendo si se encuentran en ambos cromosomas o solamente en uno. El último caso tiene más prioridad.

Uno – se ordenan por su longitud.

Ambos – se ordenan por la diferencia absoluta en sus dos longitudes.

- e) Se aplica el remplazamiento para todas las rutas intercambiables ordenadas. Se remplazan las sub-rutas no existentes por las existentes; y se remplazan las existentes por mas cortas.

El remplazamiento se aplica solamente si 1) los genes en cuestion no fueron eliminados con los remplazamientos previos; 2) remplazamiento no creará genes duplicados.

- f) Se devuelve el par de cromosomas remplazados.

```

type CrossoverDebug GA = [(
  (Point2D, Point2D),
  (Maybe ([Point2D], Bool), Maybe ([Point2D], Bool))
)]

```

Figura 3: Recombinación de cromosomas, marcados \bullet y \circ en la figura 2.

0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + 3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 + 8-6
0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - 4-2 + 4-3 - 4-5 + 6-6 + 6-5 - 7-1 + 8-0

(a) Los remplazamientos.

0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + 3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 + 8-6
0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - 4-2 + 4-3 - 4-5 + 6-6 + 6-5 - 7-1 + 8-0

(b) Remplazamiento $\circ \rightarrow \bullet$ #1.

0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + 3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 + 8-6
0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - 4-2 + 4-3 - 4-5 + 6-6 + 6-5 - 7-1 + 8-0

(c) Remplazamiento $\circ \rightarrow \bullet$ #2.

0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + 3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 + 8-6
0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - 4-2 + 4-3 - 4-5 + 6-6 + 6-5 - 7-1 + 8-0

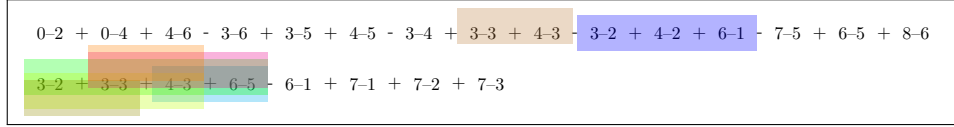
(d) Remplazamiento $\bullet \rightarrow \circ$.

```

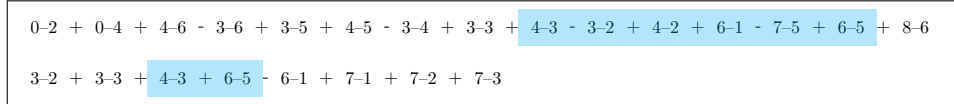
-- crossover' :: ga → Chromosome ga → Chromosome ga
-- → ((Chromosome ga, Chromosome ga), CrossoverDebug ga)
crossover' (GA l _) ch1 ch2 =
  (replaceRoutes ch1 ch2 &&& id) (subRoutes isrs)
  where
    rs1 = splitRoutes l ch1
    rs2 = splitRoutes l ch2
    set1 = Set.fromList ch1
    set2 = Set.fromList ch2
    cs = Set.toList $ Set.intersection set1 set2
    subseq l r = take (r - l + 1) ∘ drop l
    -- replacable sub-routes (unordered)
    isrs = do x ← cs
              y ← cs
    let valid = any ((∧) < $ > (x ∈) < * > (y ∈))
        valid1 = valid rs1
        valid2 = valid rs2
        valid' ch = (subseq left right ch, rev)
    where
      Just ix = x 'elemIndex' ch
      Just iy = y 'elemIndex' ch

```

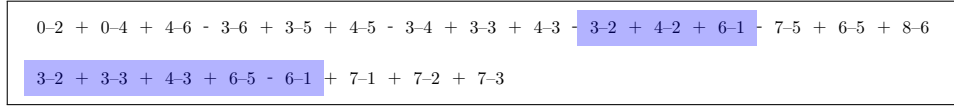
Figura 4: Recombinación de cromosomas, marcados \bullet y \bullet en la figura 2.



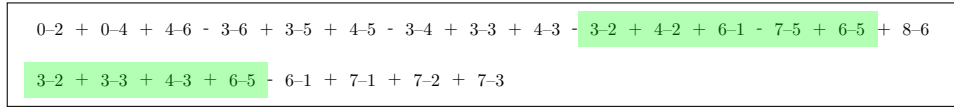
(a) Los remplazamientos.



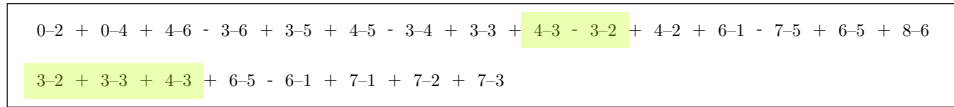
(b) Remplazamiento $\bullet \rightarrow \bullet$.



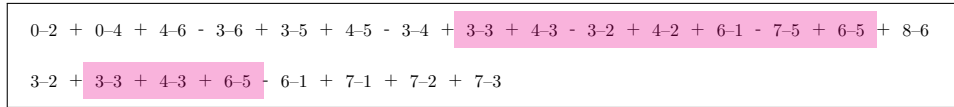
(c) Remplazamiento $\bullet \rightarrow \bullet$.



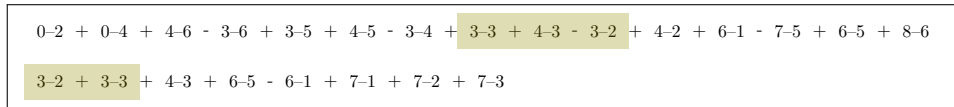
(d) Remplazamiento $\bullet \rightarrow \bullet$.



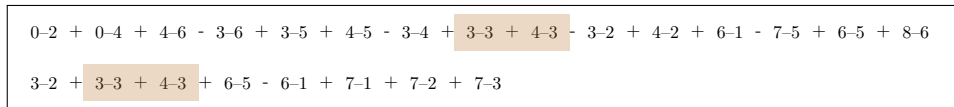
(e) Remplazamiento $\bullet \rightarrow \bullet$.



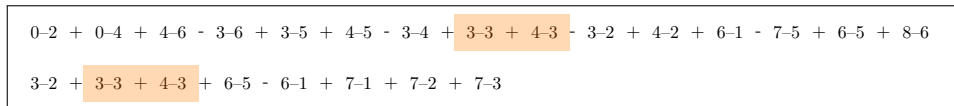
(f) Remplazamiento $\bullet \rightarrow \bullet$.



(g) Remplazamiento $\bullet \rightarrow \bullet$.



(h) Remplazamiento $\bullet \rightarrow \bullet$.



(i) Remplazamiento $\bullet \rightarrow \bullet$.

```

(left, right, rev) =
  if ix < iy then (ix, iy, False) else (iy, ix, True)
valid1' = if valid1 then Just $ valid' ch1 else Nothing
valid2' = if valid2 then Just $ valid' ch2 else Nothing
if x ≠ y ∧ (valid1 ∨ valid2)
then return ((x, y), (valid1', valid2'))
else []
sameEdge par1 par2 = par1 ≡ par2 ∨ swap par1 ≡ par2
subRoutes = sortBy cmpSubRoute ∘ nubBy (sameEdge `on` fst)
cmpSubRoute (_, (Just _, Nothing)) (_, (Just _, Just _)) = LT
cmpSubRoute (_, (Just _, Just _)) (_, (Just _, Nothing)) = GT
cmpSubRoute (_, p1) (_, p2) = rt p1 `compare` rt p2
  where rt (Just x, Nothing) = length x
        rt (Nothing, Just x) = length x
        rt (Just x, Just y) = abs (length x - length y)
replaceRoutes c1 c2 (((start, end), (mbFst, mbSnd)) : rt) = ⊥
  where source = ⊥
        target = ⊥
        -- srcGenes = do x1 j- start elemIndex

```

(9) ?

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

(10) ?

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

(11) ?

```
-- newGA :: InputData ga → ga
```

2.4. ??? gaRun ???

```

instance RunGA GA [Point2D] [Point2D] Route Min where
  type DebugData GA = ()

```

Esto es un reporte preliminar

Nota

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.