

```

{-# LANGUAGE UndecidableInstances, FlexibleInstances #-}
module Partial2.Labyrinth where
  import Control.Exception
  import Control.Arrow (first, second, (&&&))
  import Control.Monad (replicateM)
  import Control.Monad.Fix
  import Data.IRef
  import Data.Tuple (swap)
  import Data.List (elemIndex, sort, nub, minimumBy, maximumBy)
  import Data.Maybe (isJust, fromJust, fromMaybe, maybeToList)
  import Data.Bits (xor)
  import Data.Either (isLeft, isRight, Either (..))
  import Data.Function (on)
  import Data.Ratio
  import Data.Set (Set, member, elemAt)
  import qualified Data.Set as Set
  import Data.Map (Map)
  import qualified Data.Map as Map
  import GHC.Exts (Down (..), sortWith)
  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
                                   }

deriving (Show, Eq)

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

```

```
labyrinthPOIs l = map ($) [initial, target]
```

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` 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 I

Misceláneo.

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)
```

3. Implementación II

Se definan las operaciones *atómicas* – sobre genes y cromosomas, y los conceptos relacionados.

3.1. Adaptación

El valor de *aptitud de adaptación* se llamará *ruta* y se define para permitir distinguir facilmente 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 *RouteFitness* =

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 RouteFitness* **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. La función misma se definirá en subsección 3.6.

3.2. Cromosomas aleatorios

Un gen aleatorio se selecciona entre todos los nodos del mapa, y se re-genera en caso de que este gen ya hubiera sido generado previamente.

```

randPoint l = first ('elemAt' nodes l)
              ◦ randomR (0, length (nodes l) - 1)
randUnique l prev = fix $
  λf g →
    let (r, g') = randPoint l g
    in if r ∈ prev then f g' else (r, g')

```

Se empieza con la generación del primer punto

```

randChain :: GA → StdGen → Int → [Point2D] → [Point2D]
randChain ga g' len prev = nextRand ga [first'] g''
  where (first', g'') = randUnique (gaLabyri ga) 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 ga = nextRand' ga False 0
nextRand' ga rev c chain@(h : t) g =
  let neighbours = gaCache ga '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' ga rev (c + 1) chain g' -- 1 / 3
      else if rev then chain -- incomplete
      else nextRand' ga True (c + 1) chain g' -- 2
    else if length chain + 1 ≡ len
      then r : chain -- return
      else nextRand' ga rev c (r : chain) g' -- next

```

La generación de cromosoma completa se presentará en subsección 3.6.

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

3.3. Sub-Rutas

Se definen los contenedores de sub-rutas. Se guardan solamente los genes extremos de la ruta y se proveen funciones de búsqueda de sub-ruta en cromosoma. Se implementa así porque las cromosomas cambian durante operaciones genéticas, afectando las sub-rutas.

El orden sobre las sub-rutas se define en contexto de dos cromosomas: donador y recipiente. Se comparan lexicográficamente los siguientes valores:

1. Número de puntos de interés que tiene el donador pero no el recipiente.
2. Diferencia entre las longitudes de donador y recipiente.

```

newtype SubRoute = SubRoute (Point2D, Point2D)
deriving Show

instance Eq SubRoute where
    (SubRoute p1) ≡ (SubRoute p2) = p1 ≡ p2 ∨ swap p1 ≡ p2
type Reversed = Bool
type SubRoutePoints = ([Point2D], Reversed)
subseq from to = take (to - from + 1) ∘ drop from
findSubRoute :: SubRoute → [Point2D] → Maybe SubRoutePoints
findSubRoute (SubRoute (x, y)) route =
    case (elemIndex x &&& elemIndex y) route of
        (Just xi, Just yi) → let rev = xi > yi
                               ids' = (xi, yi)
                               ids = if rev then swap ids'
                                   else ids'
                               in Just (uncurry subseq ids route, rev)
        _ → Nothing
-- Sub-routes, found in 2 points sequences.
data SubRoutes = SubRoutes Labyrinth2D
                  SubRoute
                  (Either SubRoutePoints SubRoutePoints)
                  (Either SubRoutePoints SubRoutePoints)

deriving Show

instance Eq SubRoutes where
    (SubRoutes _ _ (Left _) _) ≡ (SubRoutes _ _ (Right _) _) = False
    l@(SubRoutes l1 sr1 _ _) ≡ r@(SubRoutes l2 sr2 _ _) =
        l1 ≡ l2
        ∧ sr1 ≡ sr2
        ∧ same subRouteDonor
        ∧ same subRouteReceiver
    where same f = ((≡) 'on' (fst ∘ f)) l r
subRouteDonor (SubRoutes _ _ donor _) = case donor of Left x → x
                                             Right x → x

```

```

subRouteReceiver (SubRoutes _ _ _ recei) = case recei of Left x → x
                                                Right x → x

subRoutesBoth = subRouteDonor &&& subRouteReceiver
subRoutesLabyrinth (SubRoutes l _ _ _) = l
subRoutesPts (SubRoutes _ pts _ _) = pts
lPairs (f : s : t) = (f, s) : lPairs (s : t)
lPairs _ = []
subRoutesIn :: Labyrinth2D → SubRoute
              → [Point2D] → [Point2D] → [SubRoutes]
subRoutesIn l subRoute pts1 pts2 = do
  route1 ← maybeToList $ findSubRoute subRoute pts1
  route2 ← maybeToList $ findSubRoute subRoute pts2
  let valid = all (‘edgeOf’l) ∘ lPairs ∘ fst
      sRoutes = SubRoutes l subRoute
      r1 = Left route1
      r2 = Right route2
      sRoutesLeft = sRoutes r1 r2
      sRoutesRight = sRoutes r2 r1
  case (valid route1, valid route2) of
    (True, True) → [sRoutesLeft, sRoutesRight]
    (True, _) → return sRoutesLeft
    (_, True) → return sRoutesRight
    _ → []
subRoutesValue sr@(SubRoutes l _ _ _) = (pois, len)
where donor = fst $ subRouteDonor sr
      receiver = fst $ subRouteReceiver sr
      countPois = length ∘ filter (‘isPOI’l)
      pois = countPois donor - countPois receiver
      len = length receiver - length donor

instance Ord SubRoutes where compare = compare ‘on’ subRoutesValue

```

3.4. Recombinación de cromosomas

Aquí solamente se define la recombinación de dos cromosomas, su selección será descrita en la subsección 4.1.

3.4.1. Replazamiento

Se replazan los “hoyos” de la siguiente manera:

1. Se seleccionan los genes $\{c\}$, miembros de ambos cromosomas.

2. Para ambas cromosomas se encuentran *sub-rutas intercambiables*:

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

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

3. Se aplica el remplazamiento para todas las rutas intercambiables ordenadas (fue descrita en subsección 3.3). 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; 3) no disminuye el número de puntos de interes.

4. Se devuelve el par de cromosomas remplazados.

3.4.2. Extensión de extremos

Se extienden los extremos del recipiente con los del donador:

1. Se encuentran los extremos mas cortos del donador: entre todos los puntos c se seleccionan los con menor y mayor índices en el cromosoma donador. Si las sub-rutas entre los puntos extremos y los índices correspondientes están *validas* – se guardan.
2. Se encuentran los extremos, correspondientes a los puntos, seleccionados en el punto previo. Se guardan si son *invalidas*.
3. Se remplazan los extremos correspondientes del recipiente por los del donador (si fueron guardados ambos).

Se definen unas funciones de utilidad; la definición de crossover se encuentra en subsección 3.6.

```
replaceList :: (Eq a) => [a] -> [a] -> [a] -> [a]
replaceList what with l =
  let (Just il, Just ir) = ((head what `elemIndex` ) &&&
                           (last what `elemIndex`)) l
      (left, _) = splitAt il l
      (_, right) = splitAt (ir + 1) l
  in left ++ with ++ right
```

```

replaceSafe :: Labyrinth2D → [Point2D] → [Point2D] → [Point2D]
            → Maybe [Point2D]
replaceSafe lab what with l =
  let candidate = replaceList what with l
      poisTarget = length $ filter ('isPOI' lab) what
      poisSrc = length $ filter ('isPOI' lab) with
  in if poisSrc < poisTarget ∨ candidate ≠ nub candidate
      then Nothing else Just candidate

type ReplaceDebug = [(SubRoutes, Maybe [Point2D])]
tryReplace :: [Point2D]
            → (Either SubRoutePoints SubRoutePoints → Bool)
            → (Either SubRoutePoints SubRoutePoints → SubRoutePoints)
            → [SubRoutes]
            → ReplaceDebug
            → ([Point2D], ReplaceDebug)
tryReplace chrom _ [] debugAcc = (chrom, debugAcc)
tryReplace chrom thisSide getThatSide (sr : srs) debugAcc =
  tryReplace res thisSide getThatSide srs acc'
  where acc' = if mbRes ≡ Just chrom then debugAcc
               else (debug, mbRes) : debugAcc
        (mbRes, debug) = fromMaybe (Just chrom, sr) res'
        res = fromMaybe chrom mbRes
        res' = case sr of
          SubRoutes l pts src' target' | thisSide target' →
            do target ← findSubRoute pts chrom
               let src = getThatSide src'
                   rev = snd src `xor` snd target
                   debug = SubRoutes l pts
                        (fmap (const src) src')
                        (fmap (const target) target')
               return (replaceSafe l (fst target)
                           (fst src)
                           chrom
                           , debug)
          _ → Nothing

samePoints x y = let set1 = Set.fromList x
                  set2 = Set.fromList y
                  in Set.toList $ Set.intersection set1 set2

type ExtendDebug' = (Maybe [Point2D], Maybe [Point2D])
type ExtendDebug = (ExtendDebug', ExtendDebug')
tryExtend :: Labyrinth2D → [Point2D] → [Point2D]
            → ([Point2D], ExtendDebug')

```

```

tryExtend l donor receiver = (res, deb)
  where cmp x = compare 'on' ('elemIndex' x)
        valid = all ('edgeOf' l) o lPairs
        inCase x c = if c x then Just x else Nothing
        cs = samePoints donor receiver
        left  = minimumBy (cmp donor) cs
        right = maximumBy (cmp donor) cs
        Just diLeft  = left 'elemIndex' donor
        Just diRight = right 'elemIndex' donor
        drLeft  = subseq 0 diLeft donor
        drRight = subseq diRight (length donor - 1) donor
        mbdRLeft  = drLeft 'inCase' valid
        mbdRRight = drRight 'inCase' valid
        Just riLeft  = left 'elemIndex' receiver
        Just riRight = right 'elemIndex' receiver
        rrLeft  = subseq 0 riLeft receiver
        rrRight = subseq riRight (length receiver - 1) receiver
        mbrRLeft  = rrLeft 'inCase' (¬ ∘ valid)
        mbrRRight = rrRight 'inCase' (¬ ∘ valid)
        extend (Just src) (Just target) chrom =
          replaceSafe l target src chrom
        extend _ _ _ = Nothing
        extLeft  = extend mbdRLeft mbrRLeft receiver
        extRight = extend mbdRRight mbrRRight receiver
          $ fromMaybe receiver extLeft
        debug = (extLeft, extRight)
        result = (receiver 'fromMaybe' extLeft)
          'fromMaybe' extRight
        res = if null cs then receiver else result
        deb = if null cs then (Nothing, Nothing) else debug

```

3.5. Mutación de cromosomas

La mutación de cromosomas consiste de varias operaciones, que se dividen en los que cambian un gen o una sub-ruta.

```

type MutateSubRoute = [Point2D] → IO [Point2D]
type MutateGene     = [Point2D] → Point2D → IO Point2D
  -- Choose randomly an element from a list.
randChoice = fmap fst ∘ randChoice'
randChoice' xs = do ind ← randomRIO (0, length xs - 1)
  return $ (xs!!) &&& id $ ind

```

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

$$\begin{array}{l} 0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + \boxed{3-3 + 4-3} - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 + 8-6 \\ 0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - \boxed{4-2 + 4-3} - \boxed{4-5 + 6-6 + 6-5} - 7-1 + 8-0 \end{array}$$

(a) Los remplazamientos.

$$\begin{array}{l} 0-2 + 0-4 + 4-6 - 3-6 + 3-5 + \boxed{4-5 - 3-4 + 3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5} + 8-6 \\ 0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - 4-2 + 4-3 - \boxed{4-5 + 6-6 + 6-5} - 7-1 + 8-0 \end{array}$$

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

$$\begin{array}{l} 0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + \boxed{3-3 + 4-3} - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 + 8-6 \\ 0-2 + 1-2 - 2-3 + \boxed{3-3 - 4-0 + 4-1 - 4-2 + 4-3} - 4-5 + 6-6 + 6-5 - 7-1 + 8-0 \end{array}$$

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

$$\begin{array}{l} 0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + 3-3 + \boxed{4-3 - 3-2 + 4-2} + 6-1 - 7-5 + 6-5 + 8-6 \\ 0-2 + 1-2 - 2-3 + 3-3 - 4-0 + 4-1 - \boxed{4-2 + 4-3} - 4-5 + 6-6 + 6-5 - 7-1 + 8-0 \end{array}$$

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

$$\begin{array}{ll} \text{Donado:} & 4-5 + 6-6 + 6-5 \\ \text{Remplacado:} & 4-5 - 3-4 + 3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5 \\ \text{Hijo:} & 0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 + 6-6 + 6-5 + 8-6 \end{array}$$

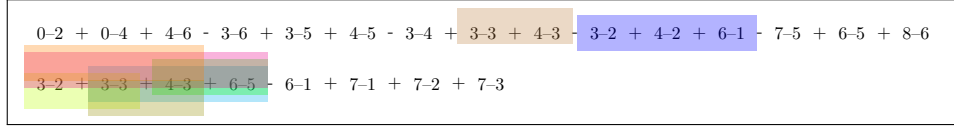
(a) Resultados de recombinación de sub-rutas en cromosoma \bullet desde \circ .

$$\begin{array}{ll} \text{Donado:} & 3-3 + 4-3 \\ \text{Remplacado:} & 3-3 - 4-0 + 4-1 - 4-2 + 4-3 \\ \text{Hijo:} & 0-2 + 1-2 - 2-3 + 3-3 + 4-3 - 4-5 + 6-6 + 6-5 - 7-1 + 8-0 \end{array}$$

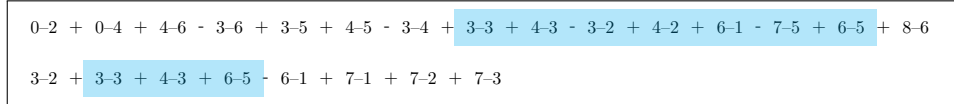
(b) Resultados de recombinación de sub-rutas en cromosoma \circ desde \bullet .

Figura 4

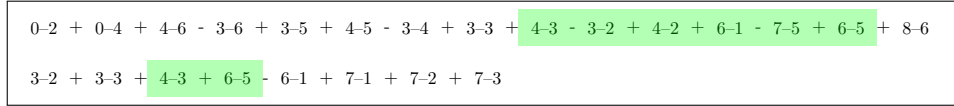
Figura 5: Recombinación de cromosomas, marcados \bullet y \bullet en la figura 2.



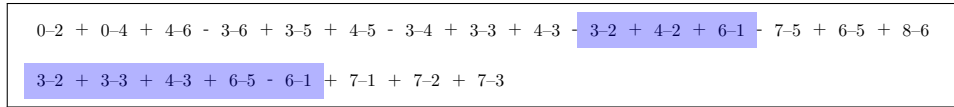
(a) Los remplazamientos.



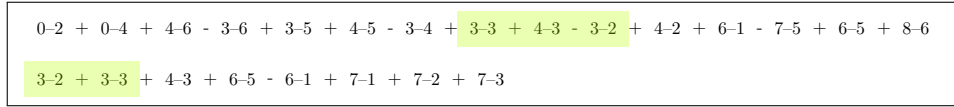
(b) Remplazamiento $\bullet \rightarrow \bullet$.



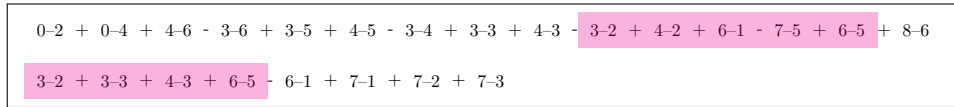
(c) Remplazamiento $\bullet \rightarrow \bullet$.



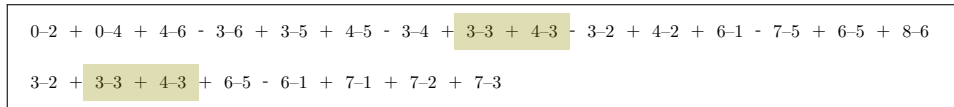
(d) Remplazamiento $\bullet \rightarrow \bullet$.



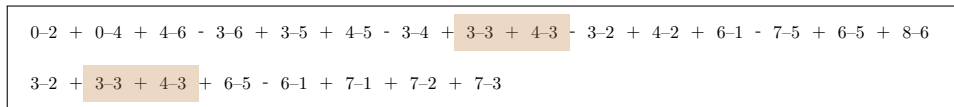
(e) Remplazamiento $\bullet \rightarrow \bullet$.



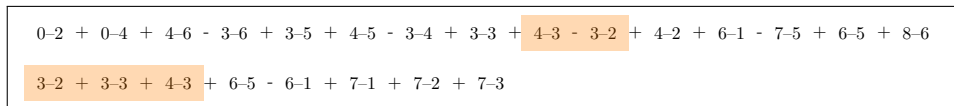
(f) Remplazamiento $\bullet \rightarrow \bullet$.



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



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



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

Donado:	3-3 + 4-3 + 6-5
Remplacado:	3-3 + 4-3 - 3-2 + 4-2 + 6-1 - 7-5 + 6-5
Hijo:	0-2 + 0-4 + 4-6 - 3-6 + 3-5 + 4-5 - 3-4 + 3-3 + 4-3 + 6-5 + 8-6

(a) Resultados de recombinación de sub-rutas en cromosoma ● desde ●.

Donado:	3-2 + 4-2 + 6-1
Remplacado:	3-2 + 3-3 + 4-3 + 6-5 - 6-1
Hijo:	3-2 + 4-2 + 6-1 + 7-1 + 7-2 + 7-3

(b) Resultados de recombinación de sub-rutas en cromosoma ● desde ●.

Figura 6

```

randChoiceSafe [] = return Nothing
randChoiceSafe xs = Just < $ > randChoice xs
randRange xs = do
  (–, i1) ← randChoice' xs
  (–, i2) ← randChoice' xs
  return $ if i2 > i1 then (i1, i2) else (i2, i1)

```

Se definen las siguientes *operaciones sobre sub-rutas*:

- Cambia una sub-ruta valida por otra aleatoria (valida), con misma longitud.

```

mutSubRouteSame :: GA → MutateSubRoute
mutSubRouteSame ga ch = do
  let srs = splitRoutes (gaLabyri ga) ch
  (sr, sri) ← randChoice' srs
  gen ← getStdGen
  let len = length sr
  rChain = randChain ga gen len []
  return ◦ concat $ subseq 0 (sri – 1) srs
    ++ rChain
    : subseq (sri + 1) len srs

```

- Cambia una sub-ruta, aleatoriamente seleccionada, por otra(s) aleatoria(s).

```

mutSubRouteAny :: GA → MutateSubRoute
mutSubRouteAny ga ch = do
  let maxGen = gaMutateMaxChainsGen $ gaParams ga
  let maxLen = gaMutateMaxChainLen $ gaParams ga
  n ← randomRIO (1, maxGen)
  cut ← flip (uncurry subseq) ch < $ > randRange ch
  paste ← sequence $ do _ ← [1..n]
  return $ do len ← randomRIO (1, maxLen)
  gen ← getStdGen
  return $ randChain ga gen len []
return $ replaceList cut (concat paste) ch

```

Se definen las siguientes *operaciones sobre genes* con la probabilidad de aplicación:

$P = 0.01$ — Cambia un gen a un de los puntos de interés (inicio/meta), si todavía no existe en el cromosoma.

```

mutGenePOI = (0.01, mutGenePOI')
mutGenePOI' :: GA → MutateGene
mutGenePOI' ga ch gene = fromMaybe gene
  < $ > randChoiceSafe notFound
where l = gaLabyri ga
  notFound = filter (¬ ∘ (∈ ch)) $ labyrinthPOIs l

```

$P = 0.005$ — Cambia un gen a un aleatorio.

```

mutGeneAny = (0.005, mutGeneAny')
mutGeneAny' :: GA → MutateGene
mutGeneAny' ga chrom gene = do
  gen ← getStdGen
  let (gene', _) = randUnique (gaLabyri ga) chrom gen
  return gene'

```

La aplicación de las mutaciones se encuentra en subsección 3.6.

3.6. Algoritmo genético

```

data GParams = GParams { gaChromGenMaxChainLen :: Int
  , gaChromGenMaxChains    :: Int
  , gaPopulationSize       :: Int
  , gaMutateMaxChainsGen   :: Int
  , gaMutateMaxChainLen    :: Int
  , gaMaxUnchangedIter     :: Int

```

```

                                , gaMaxIters           :: Int
                                , gaSelIntactFrac        :: Ratio Int
                                , gaSelCrossoverFrac     :: Ratio Int
                                , gaSelMutateFrac        :: Ratio Int
                                }
data GACache = GACache {
    cacheNeighbours  :: Map Point2D [Point2D]
    , cacheBestRepeats :: IORef Int
    , cacheBestFitness :: IORef (Maybe RouteFitness)
    , cacheIter      :: IORef Int
    , cacheSelIndexGen :: IORef (IO Int)
    }
cachedBestFit :: GACache → IO (Maybe RouteFitness)
cachedBestFit = readIORef ∘ cacheBestFitness
setCachedBestFit = writeIORef ∘ cacheBestFitness
cachedRepeat = readIORef ∘ cacheBestRepeats
affectCachedRepeat = modifyIORef ∘ cacheBestRepeats
cachedIter = readIORef ∘ cacheIter
affectCachedIter = modifyIORef ∘ cacheIter
cachedIdxGen = readIORef ∘ cacheSelIndexGen
setCachedIdxGen = writeIORef ∘ cacheSelIndexGen
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:

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

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'

```

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 = RouteFitness
```

- (3) Dirección de búsqueda – minimización.

```
type Target GA = Min
```

- (4) Información de entrada para generación de la población — el laberinto y los parametros.

```
type InputData GA = (Labyrinth2D, GAParams)
```

- (5) El resultado es el mejor cromosoma obtenido.

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

- (6) La **aptitud de adaptación** fue descrita en subsección 3.1.

```
-- 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 ∘ 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.

```

-- randomChromosome :: ga → IO (Chromosome ga)
randomChromosome ga@(GA _ params _) = do

```

Se selecciona aleatoriamente la longitud de *cadena*s.

```

chainLen ← randomRIO (1, gaChromGenMaxChainLen params)

```

Se selecciona aleatoriamente el número de *cadena*s.

```

chainCnt ← randomRIO (1, gaChromGenMaxChains params)

```

Se genera el cromosoma.

```

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

```

(8) La *recombinación* de cromosomas se enfoca en remplazar las malas sub-rutas o extender rutas existentes.

```

type CrossoverDebug GA = ([SubRoutes]
                           , (ReplaceDebug, ExtendDebug)
                           )

-- crossover' :: ga → Chromosome ga → Chromosome ga
-- → ((Chromosome ga, Chromosome ga), CrossoverDebug ga)
crossover' (GA l _ _) ch1 ch2 = (extended, debugs)
  where debugs = (subRoutes, (debugAcc, extDebug))
        cs = samePoints ch1 ch2
        sRoutes' = do x ← cs
                       y ← cs
                       let sr = SubRoute (x, y)
                       if x ≡ y then []
                       else subRoutesIn l sr ch1 ch2
        subRoutes = sortWith Down ∘ nub $ sRoutes'
        (replaced1, debugAcc') = tryReplace ch1
                                   isLeft
                                   (λ(Right x) → x)

```

```

subRoutes
[]
(replaced2, debugAcc) = tryReplace ch2
isRight
(λ(Left x) → x)
subRoutes
debugAcc'
(extended1, extDebug1) = tryExtend l replaced2 replaced1
(extended2, extDebug2) = tryExtend l replaced1 replaced2
extDebug = (extDebug1, extDebug2)
extended = (extended1, extended2)

```

(9) La *mutación* funciona en la siguiente manera:

- a) Se escoge y se aplica una de las *operaciones sobre sub-rutas*.
- b) Para cada gen del resultado del punto previo, se aplican (todas) las *operaciones sobre genes*, con su probabilidad asignada.

```

-- mutate :: ga → Chromosome ga → IO (Chromosome ga)
mutate ga chrom = do
  chrom' ← ($chrom) <<< randChoice (map ($ga) subRouteMuts)
  mutateGenes chrom' geneMuts
where subRouteMuts = [mutSubRouteSame, mutSubRouteAny]
  geneMuts          = [mutGenePOI, mutGeneAny]
  mutateGenes ch = mutateGenes' ch []
  mutateGenes' [] mutated _ = return mutated
  mutateGenes' orig@(gene : t) mutated muts =
    do g ← gene'
       mutateGenes' t (mutated ++ [g]) muts
where chrom = mutated ++ orig
      mutF (p, mut) g' = do
        d ← randomIO :: IO Double
        g ← g'
        if p < d then mut ga chrom g else return g
      gene' = foldr mutF (return gene) muts

```

(10) *Criterio de parada* se selecciona, considerando que no se conoce la longitud de ruta aceptable. Esto no permita establecer un criterio exacto.

Es porque el criterio se establece sobre el *cambio del mejor valor de adaptación en las últimas iteraciones*.

También se utiliza como criterio adicional el *número máximo de iteraciones*.

```

-- stopCriteria :: ga → [Fitness ga] → IO Bool
stopCriteria ga fitness = do
  let cache = gaCache ga
      raiseCount = cache 'affectCachedRepeat' (+1)
      best = head fitness
      best' ← cachedBestFit cache
  if Just best ≡ best' then raiseCount
  else cache 'setCachedBestFit' Just best
  repCount ← cachedRepeat cache
  iter ← cachedIter cache
  return $ repCount ≥ gaMaxUnchangedIter (gaParams ga)
      ∨ iter ≥ gaMaxIters (gaParams ga)

```

(11) Creación de instancia de GA. Se crea el cache.

```

-- newGA :: InputData ga → IO ga
newGA (labyrinth, params) = do
  let nodes' = Set.toList $ nodes labyrinth
      neighbours = Map.fromList $ do
        node ← nodes'
        let connected = filter (('edgeOf' labyrinth) ∘ (, ) node) nodes'
        return (node, connected)
  bestRepeats ← newIORef 0
  bestFitness ← newIORef Nothing
  iter ← newIORef 0
  selIndexGen ← newIORef (return (-1))
  let cache = GACache neighbours
      bestRepeats
      bestFitness
      iter
      selIndexGen
  return $ GA labyrinth params cache

```

4. Implementación III

El proyecto separa las operaciones genéticos *atómicos*, definidas en clase *GA*, de las *masivas*, definidas en clase *RunGA*.

Las operaciones *masivas* – son las que trabajar con entera población: 1) selección de cromosomas para operaciones genéticos, generación de población inicial; 2) generación de población inicial; 3) ejecución de las iteraciones.

Las últimas dos están definidas en `src/GeneticAlgorithm`, pero su código se presentara incluso en subsección 4.2.

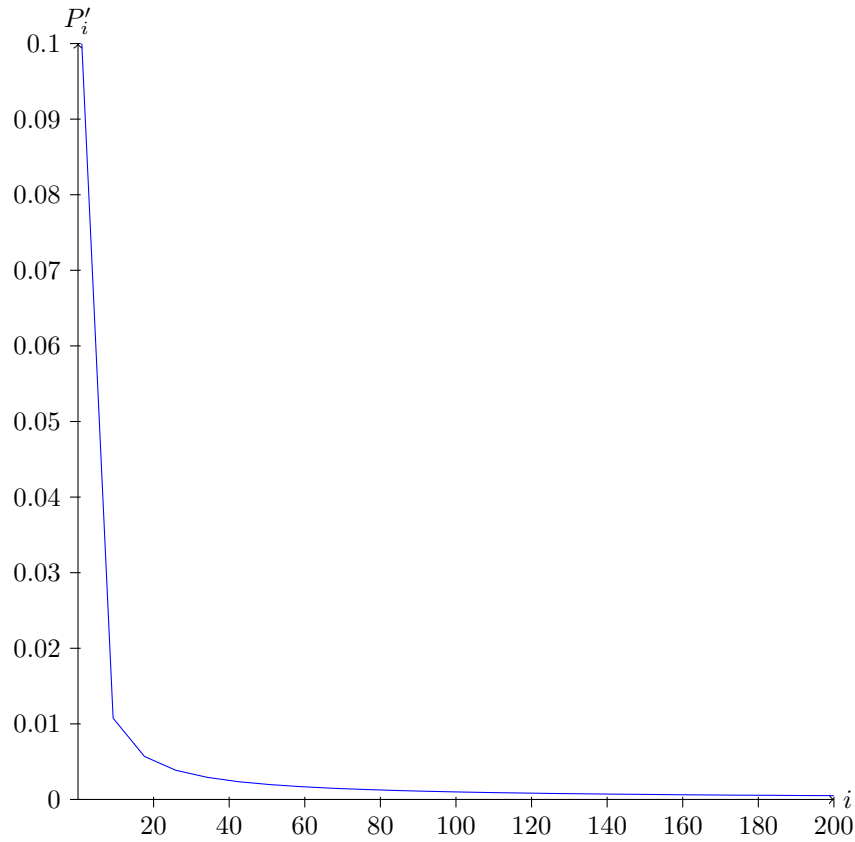
4.1. Operaciones de selección

Para preservación del tamaño de población, la unión de los tres siguientes selecciones debe siempre ser de mismo tamaño que la población, de la cual hubieron sidos seleccionados.

Se uso un concepto *Assessed*, el cual encapsula una lista de cromosomas con sus correspondientes valores de adaptación. Está siempre ordenada asendiantamente, para que los mejores cromosomas (con menor valor de adaptación) estén en el principio.

Se intente usar *RouteFitness* sin transformarlos en un valor numerico; para esto se defina la función de densidad de probabilidad de selección de un cromosoma, dependiendo de su índice en la lista.

$$P'_i = \frac{0.1}{i}$$
$$P_i = \frac{P'_i}{\sum_j P'_j}$$



```

assessedProb' i = 0.1 / fromInteger i
assessedProbs n = map (/isum) is
  where is = assessedProb' < $ > [1..n]
        isum = sum is
assessedRandIndexGen :: Integer -- Population size.
                    --> IO Int -- Random index.
assessedRandIndexGen n = randIdx
  where probs = assessedProbs n
        -- accumulated probabilities
        accProbs = snd $ foldr (\p (p', acc) → (p' + p, p' + p : acc))
                              (0, []) probs
        -- select id, given a number in [0,1]
        selIdx :: Double → Int
        selIdx d = let -- P(A | a)
                      less = (<d) 'filter' accProbs
                    in if null less then 0 -- first
                       else length less - 1 -- max of P(A | a)
        -- Generate a random assessed index.
        randIdx = selIdx < $ > randomIO
assessedRand selFrac ga assessed = do
  iGen ← cachedIdxGen $ gaCache ga
  let pSize = fromIntegral $ popSize assessed
      frac = selFrac (gaParams ga)
      count = round $ pSize * frac
  ids ← replicateM count iGen
  let ua = map fst $ unwrapAssessed assessed
  return $ map (ua!!) ids

```

```

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

```

Se define selección de cromosomas:

- (a) Que pasan al siguiente generación intactos.

La fracción establecida de la población previa se escoje aleatoriamente (con repeticiones).

```

selectIntact = assessedRand gaSelIntactFrac

```

(b) Para la recombinación.

Se escogen aleatoriamente dos fracciones establecidas (de tamaño igual) de la población previa.

```
selectCrossover ga assessed = zip < $ > rand < * > rand
where rand = assessedRand gaSelCrossoverFrac ga assessed
```

(c) Para la mutación.

Se escoge aleatoriamente una fracción establecida de la población previa.

```
selectMutate = assessedRand gaSelMutateFrac
```

El resultado es el cromosoma con mejor valor de adaptación.

```
selectResult _ (Assessed (h: _)) = (fst h, ())
```

4.2. Ejecución de algoritmo genético

Se presenta aquí parte de código desde src/GeneticAlgorithm.

- Generación de población inicial.

```
initialPopulation ga pop = sequence $ do _ <- [1..pop]
                                return $ randomChromosome ga
```

- Ejecución de las iteraciones del algoritmo.

```
runGA' ga pop = do
  let fit = assessed $ map (id &&& fitness ga) pop
  stop <- stopCriteria ga ◦ map snd $ unwrapAssessed fit
  intact <- selectIntact ga fit
  cross <- selectCrossover ga fit
  mut <- selectMutate ga fit
  mutated <- mapM (mutate ga) mut
  let pairToList (x, y) = [x, y]
  newPop = intact
           ++ concatMap (pairToList ◦ uncurry (crossover ga)) cross
           ++ mutated
  if stop then return $ selectResult ga fit
  else runGA' ga newPop
```

5. Ejecución

TODO