```
{-# 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.List (elemIndex, sort, nub)
  import Data.Maybe (isJust, fromJust, fromMaybe, maybeToList)
  import Data.Bits (xor)
  import Data.Either (isLeft, isRight, Either (..))
  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.Exts (Down (..), sortWith)
  import Parcial2.ReadLabyrinth
  import GeneticAlgorithm
  import System.Random

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
```

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.

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

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. Adaptación

El valor de *aptitud de adaptación* 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 ($\exists$ 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 ambos puntos: inicio y meta.

  Se caracteriza por tres valores:

  1. $validez = \dfrac{\text{número de aristas existentes}}{\text{número de aristas total}}$.
     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 RouteFitness =
  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 menor que cualquier *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 en el orden.

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

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 igualidad o se termine la lista.

   El orden de 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 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
```

No se usa

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. Sub-Rutas

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

El orden sobre las sub-rutas se define en contexto de dos cromosomas: donor y recipiente. Se comparan lexográficamente los sigientes valores:

1. Número de púntos de interes que tiene el donor pero no el recipiente.
2. Diferencia entre las longitudes de donor 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
```

2.4. Algoritmo genético

```
data GAParams = GAParams { 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:

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

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.

```
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)
```

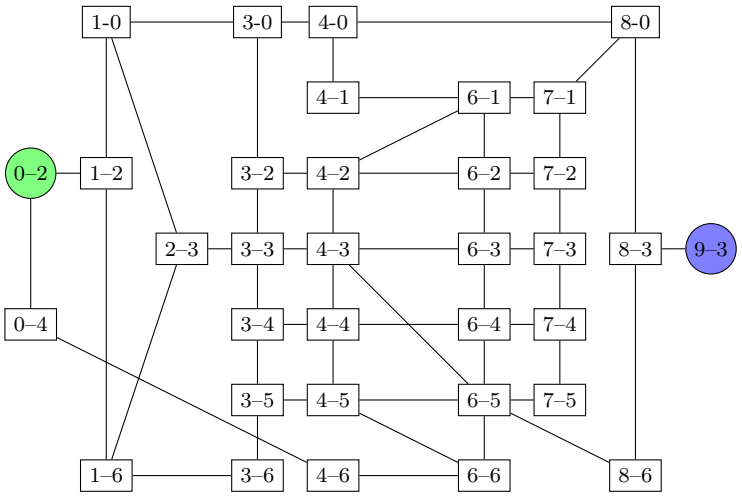


Figura 1: Ejemplo de mapa, inicio: 0–2, meta: 9–3.

```
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.

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.

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

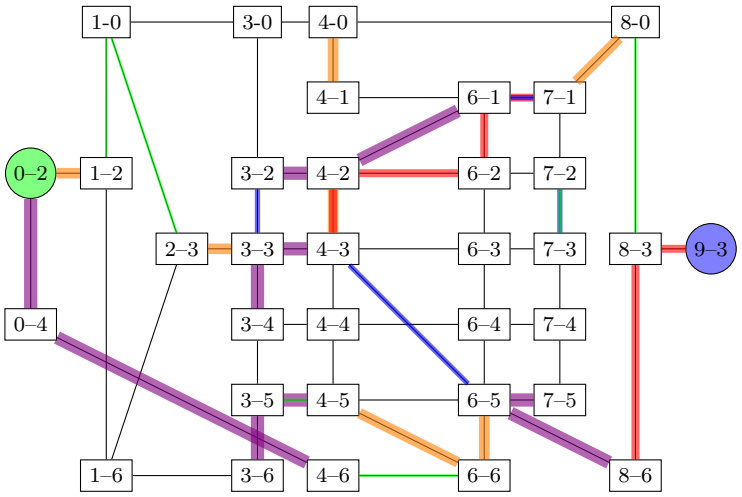


Figura 2: Se presentan algunos cromosomas en el mapa. Los cromosomas ● ● ● están compuestas de pares de genes, conectados por aristas; mientras que los cromosomas ● ● están compuestos de cadenas de genes, conectados por aristas, de longitud 3. *(Son de diferente grosor para que se ven mejor las conecciones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

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.

$$
\begin{aligned}
&\mathit{randPoint} = \mathit{first}\ (\text{`}\mathit{elemAt}\text{`}\mathit{nodes}\ l) \\
&\qquad\qquad \circ\ \mathit{randomR}\ (0, \mathit{length}\ (\mathit{nodes}\ l) - 1) \\
&\mathit{rand}\ \mathit{prev} = \mathit{fix}\ \$ \\
&\qquad\qquad \lambda f\ g \rightarrow \\
&\qquad\qquad\quad \textbf{let}\ (r, g') = \mathit{randPoint}\ g \\
&\qquad\qquad\quad \textbf{in if}\ r \in \mathit{prev}\ \textbf{then}\ f\ g'\ \textbf{else}\ (r, g')
\end{aligned}
$$

Se empieza con la generación del primer punto

$$
\begin{aligned}
&\mathit{randChain} :: \mathit{StdGen} \rightarrow \mathit{Int} \rightarrow [\mathit{Point2D}] \rightarrow [\mathit{Point2D}] \\
&\mathit{randChain}\ g'\ \mathit{len}\ \mathit{prev} = \mathit{nextRand}\ [\mathit{first}']\ g'' \\
&\quad \textbf{where}\ (\mathit{first}', g'') = \mathit{rand}\ \mathit{prev}\ g'
\end{aligned}
$$

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 longetud de *cadenas*.

$$chainLen \leftarrow randomRIO\ (1, gaChromGenMaxChainLen\ params)$$

Se selecciona aleatoriamente el número de *cadenas*.

$$chainCnt \leftarrow randomRIO\ (1, gaChromGenMaxChains\ params)$$

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 2.5.

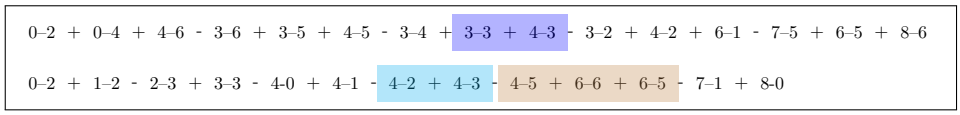
a) Se seleccionan los genes $\{c\}$, miembros de ambos cromosomas.

b) Para ambos cromosomas se encuentran *sub-rutas intercambiables*:

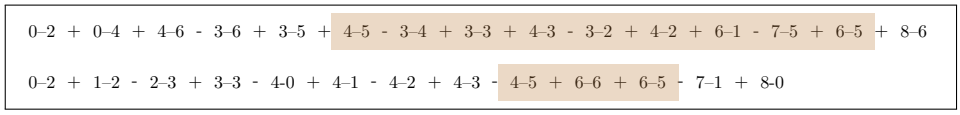
$$\begin{aligned} &\forall\ x \in \{c\} \\ &\quad y \in \{c\} \quad \Longrightarrow \\ &\begin{cases} \text{secuencia } \{r_i\}_{i=1}^{N_r} & \text{si } \begin{array}{l} \forall\ r_{j-1}, r_j \in \{r_i\}_{i=1}^{N_r} \\ \exists \text{ arista entre } r_{j-1} \text{ y } r_j \end{array} \\ \{\} & \text{en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

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

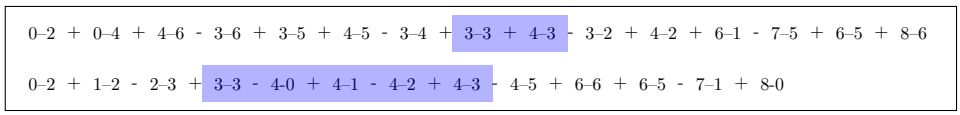
Figura 3: Recombinación de cromosomas, marcados • y • en la figura 2.



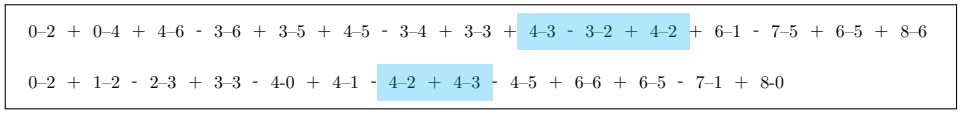
(a) Los remplazamientos.



(b) Remplazamiento • → • #1.



(c) Remplazamiento • → •.



(d) Remplazamiento • → • #2.

c) Se aplica el remplazamiento para todas las rutas intercambiables ordenadas (fue descrita en subsección 2.3). 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.

d) Se devuelve el par de cromosomas remplazados.

```
type CrossoverDebug GA = ([SubRoutes], [(SubRoutes, Maybe [Point2D])])
  -- crossover' :: ga → Chromosome ga → Chromosome ga
  -- → ((Chromosome ga, Chromosome ga), CrossoverDebug ga)
crossover' (GA l _ _) ch1 ch2 = (replaced, (subRoutes, debugAcc))
  where set1 = Set.fromList ch1
        set2 = Set.fromList ch2
        cs = Set.toList $ Set.intersection set1 set2
        sRoutes' = do x ← cs
                      y ← cs
                      let sr = SubRoute (x, y)
                      if x ≡ y then []
                      else subRoutesIn l sr ch1 ch2
```

Donado:	4–5 + 6–6 + 6–5
Remplacado:	4–5 - 3–4 + 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 + 6–6 + 6–5 + 8–6

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

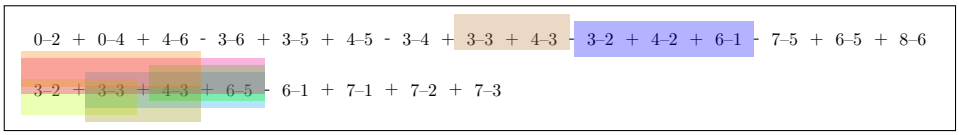
Donado:	3–3 + 4–3
Remplacado:	3–3 - 4-0 + 4–1 - 4–2 + 4–3
Hijo:	0–2 + 1–2 - 2–3 + 3–3 + 4–3 - 4–5 + 6–6 + 6–5 - 7–1 + 8-0

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

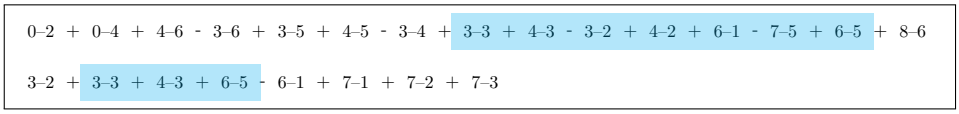
Figura 4

```
subRoutes = sortWith Down ∘ nub $ sRoutes'
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 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
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
```

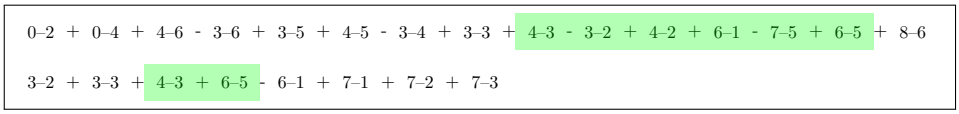
Figura 5: Recombinación de cromosomas, marcados • y • en la figura 2.



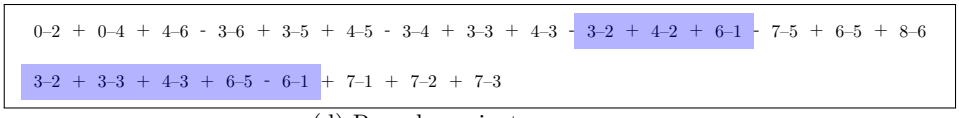
(a) Los remplazamientos.



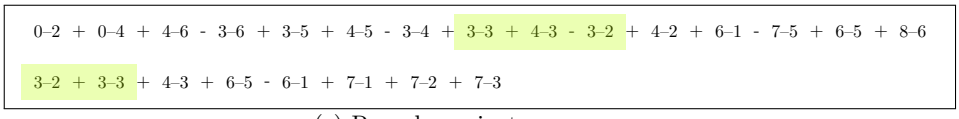
(b) Remplazamiento • → •.



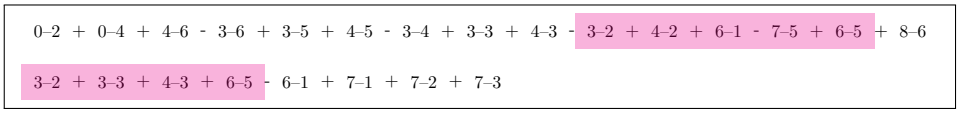
(c) Remplazamiento • → •.



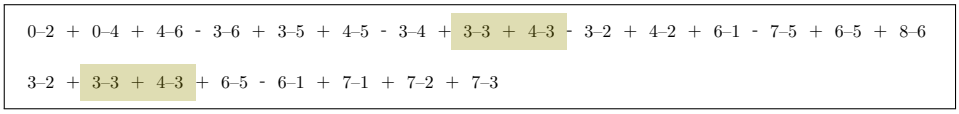
(d) Remplazamiento • → •.



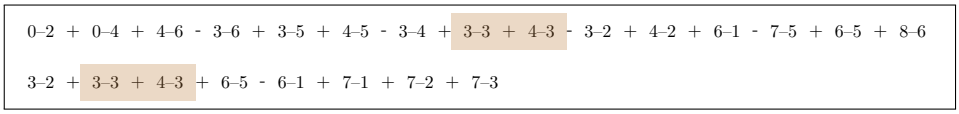
(e) Remplazamiento • → •.



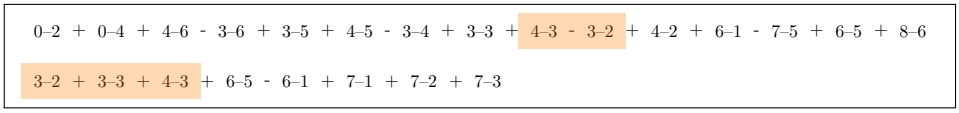
(f) Remplazamiento • → •.



(g) Remplazamiento • → •.



(h) Remplazamiento • → •.



(i) Remplazamiento • → •.

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

```
                                    debug = SubRoutes l pts
                                      (fmap (const src) src')
                                      (fmap (const target) target')
                                  return (replaceSafe l (fst target)
                                                        (fst src)
                                                        chrom
                                        , debug)
                              _ → Nothing
(replaced1, debugAcc') = tryReplace ch1
                                    isLeft
                                    (λ(Right x) → x)
                                    subRoutes
                                    []
(replaced2, debugAcc) = tryReplace  ch2
                                    isRight
                                    (λ(Left x) → x)
                                    subRoutes
                                    debugAcc'
replaced = (replaced1, replaced2)
```

(9) ?

-- mutate :: Chromosome ga → Chromosome ga

(10) ?

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

(11) ?

```
-- newGA :: InputData ga → ga
```

2.5. ??? gaRun ???

instance *RunGA GA* [*Point2D*] [*Point2D*] *RouteFitness 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.