

```

{-# 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.Maybe (isJust, fromJust)
  import Data.Set (Set, member)
  import qualified Data.Set as Set
  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 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 $ \l v1 v2 →
  if (v1, v2) `edgeOf` edges l then Just (f v1 v2) else Nothing

```

El algoritmo genético abstracto está definido en 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 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. Algoritmo genético

```
data GA = GA Labyrinth2D
```

Un alias para tuple (**a**, **a**).

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

Se define el valor de aptitud como uno de los dos:

- longitud de la ruta completa;
- grado de valides $\frac{\text{número de aristas existentes}}{\text{número de aristas total}}$.
aristas existentes — *aristas que existen entre los pares de genes ajustados*.

data *Route* (*dir* :: *OrdDir*) = *RouteLength Double* | *RouteValidess Double*
deriving (*Eq*, *Show*)

También tiene un parametro de tipo para establecer la dirección de búsqueda, lo que determina el orden deseado. Se define la orden sobre la aptitud de tal manera que dependiendo en la dirección:

- *Min* — $\forall x \in \text{longitud}, y \in \text{valides} \Rightarrow x < y$;

instance *Ord* (*Route Min*) **where**
compare (*RouteLength* *x*) (*RouteLength* *y*) = *compare* *x* *y*
compare (*RouteValidess* *x*) (*RouteValidess* *y*) = *compare* *x* *y*
compare (*RouteLength* *_*) (*RouteValidess* *_*) = *LT*
compare (*RouteValidess* *_*) (*RouteLength* *_*) = *GT*

- *Max* — $\forall x \in \text{longitud}, y \in \text{valides} \Rightarrow x > y$.

instance *Ord* (*Route Max*) **where**
compare (*RouteLength* *x*) (*RouteLength* *y*) = *compare* *x* *y*
compare (*RouteValidess* *x*) (*RouteValidess* *y*) = *compare* *x* *y*
compare (*RouteLength* *_*) (*RouteValidess* *_*) = *GT*
compare (*RouteValidess* *_*) (*RouteLength* *_*) = *LT*

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.

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

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 Min*

- (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 adaptación** se define como:

$$f(c) = \begin{cases} \text{Length length} & \text{si } \begin{aligned} &\forall i = \overline{[1, \text{len} - 1]} \Rightarrow \\ &\exists \text{ arista, conectando } c_{i-1} \text{ y } c_i \\ &\wedge \text{ initial} \in \{c\} \\ &\wedge \text{ target} \in \{c\} \end{aligned} \\ \text{Validess validess} & \text{en otro caso} \end{cases}$$

donde

$$\text{length} = \sum_{i=1}^{\text{len}-1} \text{dist}(c_{i-1}, c_i)$$

validess = grado de valides (se describe antes)

```
-- 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
    RouteLength o sum $ map fromJust dists
  else -- is incomplete
    RouteValidess $
      fromIntegral (length $ filter isJust dists)
      / fromIntegral (length dists)
```

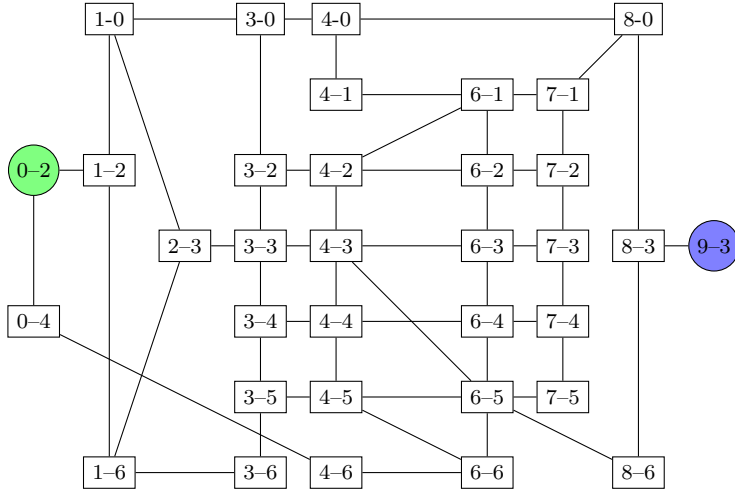


Figura 1: Un ejemplo de mapa, inicio: 0-2, meta: 9-3.

(7) Generación de cromosomas aleatorios.

Para mejorar las poblaciones iniciales, las cromosomas se componen de 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 ejemplo de cromosomas generados.

TBD ...

```
-- randomChromosome :: ga → IO (Chromosome ga)
randomChromosome (GA l) = ⊥
```

(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
```

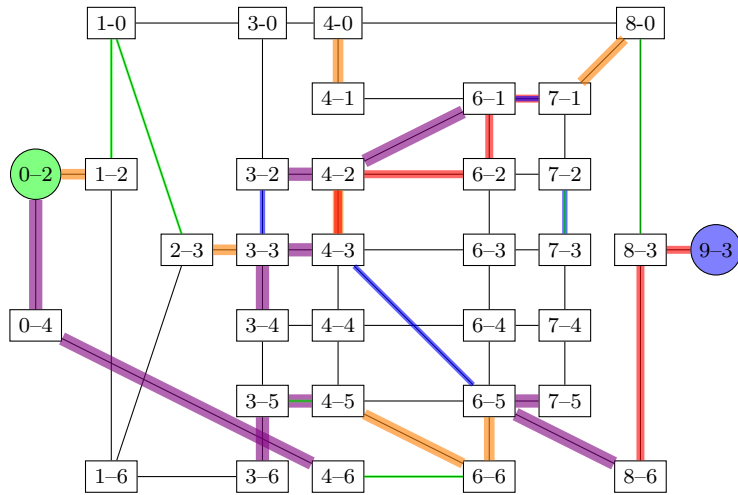


Figura 2: Se presentan algunos cromosomas en el mapa. Los cromosomas ● ● ● están compuestas de pares de genes, conectados por aristas; mientras que los cromosomas ● ● están compuestos de cadenas de genes, conectados por aristas, de longitud 3. (Son de diferente grosor para que se ven mejor las conexiones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

Esto es un reporte preliminar

Nota

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

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

! El concepto de "*valides*" va ser cambiado.