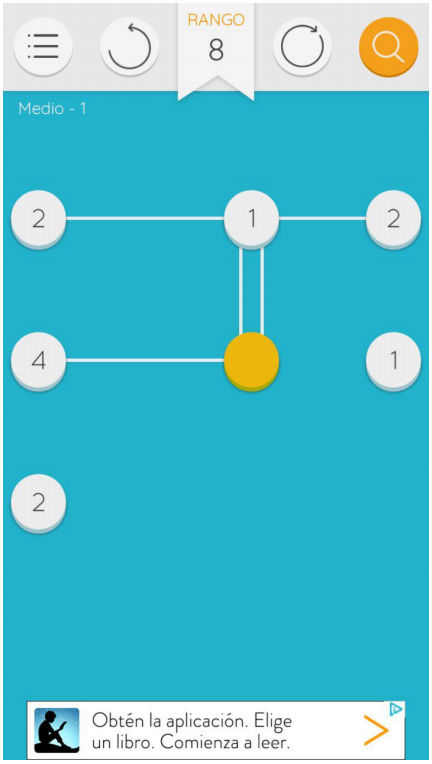
Bridges

Roberto Hueso Gómez

1 Descripción

El puzle consiste en varias islas que se pueden conectar entre ellas por puentes. Dadas dos islas, se pueden conectar entre ellas por 0, 1 o 2 puentes. Cada isla tiene asociado un número $x \in \mathbb{N}$ y cada vez que conectamos un puente a esa isla $x_n = x_{n-1} - 1$ el objetivo es hacer que $x_n = 0$ en todas las islas. Cuando cumplimos el objetivo, hemos ganado la partida.

Figura 1: Captura de pantalla durante el juego.



2 Representación

Todo el problema se ha representado como un grafo dirigido en el que los vértices son las islas y las aristas los puentes.

Obviaré los detalles del código puesto que para algo detallado es mas simple leer el código.

2.1 Grafo

El grafo se define como

```
data Graph = Empty | G vertice [vertices_conectados] (Graph)
```

Dónde *vertice* se define como

```
data Vertex = V etiqueta (x, y)
```

Donde la *etiqueta* es el número asociado a la isla y (x,y) son las coordenadas x,y en el plano euclideo de la isla. El campo *vertices_conectados* indican los vertices a los que se conecta *vertice* de manera que queda representado la arista dirigida *vertice→vertice_conectado_1*

El tipo arista es puramente auxiliar y se define como

```
type Edge = (vertice_origen, vertice_destino)
```

2.2 Juego

He decidido que la *etiqueta* de los nodos sea un entero (los puentes restantes) por simplicidad. Los nodos estarán en puntos discretos del plano, por lo que he decidido que las *coordenadas* se expresen como duplas de enteros.

```
type Coordinates = (Int, Int)
```

El tipo *Mundo* también será una dupla pero en este caso será entre un grafo y unas coordenadas (como método auxiliar) que representan las coordenadas de origen de la arista. Las coordenadas de destino las tomaremos en el manejador de acciones.

```
type Mundo = (Graph, Coordinates)
```

Para simplificar el código al crear los grafos haré las coordenadas (x,y) de los nodos sean múltiplos de 3 partiendo del origen (0,0). También haré que los puentes no se puedan cruzar para evitar tener que comprobar la planaridad del grafo.

2.3 Búsqueda de solución en anchura

El tipo *Estado* quedará definido como

```
type Estado = Graph
```

Dónde *Graph* representa el estado actual del mapa de islas.

Los movimientos que podremos aplicar a cada situación del mapa vendrán determinados por *Movimiento* que se define como

```
data Movimiento = Arriba vertice
  | Abajo vertice
  | Izquierda vertice
  | Derecha vertice
```

dónde cada dirección representa desplazarse en esa dirección 3 unidades del plano euclideo desde el *vertice*.

3 Código

3.1 Ejemplos

Ejecutar resolución en CodeWorld

```
:l Bridges.hs
main
-- Para probar otros niveles cambiar la constante "main_game"
```

Ejecutar y visualizar BFS

```
:l Bridges_Solver.hs
dibuja_resultado $ busqueda estado1
```

4 Referencias

- Programación declarativa - US[1]
- Wikipedia - BFS[2]
- Aprende Haskell por el bien de todos[3]
- Haskell reference[4]
- CodeWorld reference[5]
- Haskell Text Package[6]
- Haskell List Package[7]

[1] https://www.cs.us.es/cursos/pd/
[2] https://en.wikipedia.org/wiki/Breadth-first_search
[3] http://aprendehaskell.es/main.html
[4] http://zvon.org/other/haskell/Outputglobal/index.html
[5] https://code.world/doc-haskell/CodeWorld.html
[6] https://hackage.haskell.org/package/text-1.2.3.0/docs/Data-Text.html
[7] https://hackage.haskell.org/package/base-4.10.1.0/docs/Data-List.html