Agar.io

Instituto Tecnológico de Buenos Aires

Buenos Aires, Argentina

2017

- Gastón Rodríguez grodriguez@itba.edu.ar 56020

Introducción

El objetivo de este trabajo práctico es utilizar las técnicas aprendidas a lo largo de la materia sobre el paradigma funcional y el lenguaje de programación Haskell para implementar un motor de simulación del juego agar.io.
El juego consiste en un tablero habitado por células, éstas pueden ser jugadores o comida. Los jugadores pueden consumir la comida del tablero para hacerse más grandes, y al obtener tamaño pueden absorber a los jugadores que sean más pequeños que ellos, siendo el objetivo del juego ser la última célula viva, habiendo comido todo lo disponible en el mapa.

Modelado del juego

Nuestro juego diferirá del agario real en un número de cosas:

- El agario real es un único partido contínuo que está siempre siendo jugado, donde los jugadores entran y salen del partido cuando les plazca, mientras que en nuestra simulación se creará un tablero con todos los jugadores, y terminará cuando se alcance una situación de equilibrio que se dará cuando quede una única célula viva, o no quede más *plankton* y ninguna de las células se puedan comer entre ellas (todas tienen un tamaño igual o similar).
- En nuestra simulación los jugadores no serán jugadores en línea, sino que serán autómatas que seguirán alguna estrategia definida al momento de empezar el partido, decidiendo su próximo movimiento en cada paso del juego.
- En nuestra simulación habrá una cantidad finita de comida o *plankton*, y no se irá regenerando a lo largo del tiempo como en el juego real.
- Los jugadores tendrán una velocidad constante y no será en función de su tamaño, mientras que en el juego real las células más grandes se mueven más lento.

Motor del simulador

Definiciones

Antes de describir el motor del juego y cómo está programado, voy a asentar las definiciones de tipos y de estructuras de datos más importantes que se usan en el programa para luego referenciarlas a lo largo del informe.

En el archivo *GameDefinitions.hs* se encuentran las definiciones de los tipos y estructuras de datos que son referenciados a lo largo del programa, entre los más importantes están:

```
data Cell =
  Player Position Radius Integer StrategyFunction |
  Plankton Position Radius Integer
```

La estructura de datos *Cell* es la que representa a una entidad en el tablero, puede ser tanto un plankton como un jugador. Ambos tipos de célula tienen una posición (dos *doubles*), un radio (*Double)* y un Id, mientras que los jugadores también cuentan con una función de estrategia, que decidirá hacia dónde se moverá en cada paso del partido.

```
type Players   = [Cell]
type Planktons = [Cell]
data GameContainer = GC BoardSize Players Planktons [OutputFunction]
```

GameContainer es la representación de un tablero en un momento dado, contiene el tamaño del tablero (*Integer*) y una lista de jugadores y planktons. La lista de *output functions* ya veremos para que sirve.

```
data GameHistory = ([Players],[Planktons])
```

GameHistory es el historial del partido, guardando todos los estados por los que va pasando el partido hasta su conclusión, para eventualmente extraer la información que uno quiera del partido entero a través de los *outputters* (ya vienen).

```
type GameState = (Players, Planktons)
```

GameState representa el estado del partido en un paso temporal determinado.

```
type StrategyFunction = Cell -> GameState -> Vector
type OutputFunction = GameHistory -> IO ()
```

StrategyFunction es la encargada de decidir hacia dónde se moverá el jugador (primer parámetro) dado el estado actual del juego.
OutputFunction es una función que, dado el historial de la partida, ejecuta una acción *IO ()* para representarlo de alguna manera. Son la forma que tiene el motor del juego de comunicar el resultado de la simulación al mundo exterior.

Motor.hs

El archivo Motor.hs contiene el método principal de la simulación. El método *startSimulation* se encarga de, dados unos parámetros iniciales, inicializar el tablero creando con una posición aleatoria las células dentro del tamaño indicado, y comenzar el partido del juego. Los pasos a seguir en cada etapa de la simulación son:

1. Asignar a las células velocidades según sus estrategias
2. Mover el mundo un paso temporal, avanzando a todas las células
3. Revisar los solapamientos de células con plankton y realizar las absorciones en caso de solapamiento
4. Revisar los solapamientos de células con células y realizar las absorciones en caso de solapamiento
5. Revisar la condición de equilibrio para determinar si el juego terminó
6. En caso de que haya terminado el juego, llamar a todos los *outputters* del *GameContainer*, para guardar el resultado de la simulación.

La función *runGame* es la encargada de ejecutar estos pasos:

```
runGame :: GameContainer -> GameHistory -> IO [()]
runGame (GC bS players planktons os) (playersHistory, planktonsHistory) = do
  let playersAfterMoving = movePlayers (players, planktons) bS
  let newGC = processOverlappingCells bS (playersAfterMoving, planktons) os
  if equilibriumReached newGC then
    (mapM (callOutputter ((playersHistory++[getPlayers newGC]), (planktonsHistory++[getPlanktons
newGC]))) os)
  else
    runGame newGC ((playersHistory++[getPlayers newGC]), (planktonsHistory++[getPlanktons newGC]))
```

Como se puede ver, el método se llama recursivamente a sí mismo acumulando cada paso temporal en GameHistory, hasta que se cumpla la condición de equilibrio mencionada anteriormente, donde llamará a la función *mapM*.
mapM es una función de alto orden definida por *haskell* como:

```
1. mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

En esta función la utilizo para que me dé los resultados de ejecutar las funciones IO () que son las *OutputFunction*. Recibe una lista de outputters, y una función que dado un outputter lo ejecuta con el historial, y así *mapM* ejecuta a todos los outputters.

La función *movePlayers* devuelve una lista con todos los jugadores movidos, según la estrategia que estos posean.

```
movePlayers :: GameState ->  BoardSize -> Players
movePlayers (players, planktons) bS = playersAfterMoving
  where
    newVelocities = applyStrategies (players,planktons)
    playersAfterMoving = applyNewVelocities bS players newVelocities
```

```
applyStrategies :: GameState -> [Vector]
applyStrategies (players, planktons) = map (\x -> (getStrategy x) x
(players,planktons)) players
```

Strategy.hs

El archivo Strategy contiene las implementaciones de las estrategias de las que disponen los jugadores. Implementé tres para este trabajo:

- *Closest agent greedy:* Busca la célula comestible más cercana y se acerca a ella.
- *Plankton first greedy:* Busca el plankton más cercano y se acerca a él. En caso de no haber más plankton, busca al jugador comestible más cercano.
- *Player first greedy*: Busca al jugador más cercano y se acerca a él. En caso de no haber jugadores comestibles, busca el plankton más cercano.

Como vimos en la definición de las StrategyFunction, éstas reciben el estado actual del juego y devuelven un Vector (dos doubles). Este vector es un vector con módulo uno apuntando en la dirección que la estrategia haya decidido.

Outputter.hs

En este archivo se encuentran las funciones mediante las cuales el programa se comunica con el mundo exterior.
Dentro del archivo se encontrarán las funciones:

```
type GameStateToStringFunction = GameState -> String
writeFileOutputter :: String -> GameStateToStringFunction -> OutputFunction
writeFileOutputter filePath fn = (\gh -> writeFile filePath (concatenateWholeGameState fn gh))
```

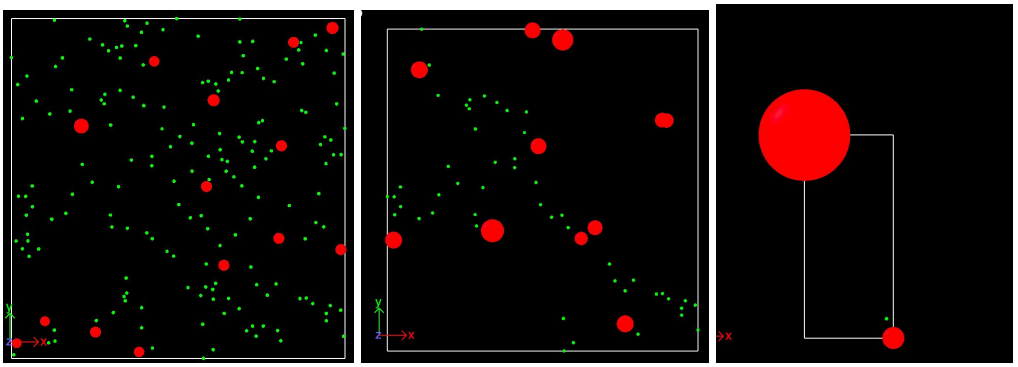
writeFileOutputter es una función que recibe un path y una función de gamestate a string y devuelve una *OutputFunction* que recibe un historial de partido y devuelve una acción de IO (), en este caso es *writeFile*, utilizada para escribir archivos en un path indicado. Entonces, en el inicio del programa puedo hacer:

```
outputters = [
  writeCsvOutputter "./runs/count.csv" countPlayersAndPlanktonInState countPlayersAndPlanktonInStateHeader,
  writeFileOutputter "./runs/run.xyz” stateToXyzFormat
  ]
```

Y con esto consigo que el main escriba en el path indicado el resultado del partido. Las funciones *stateToXyzFormat* y *countPlayersAndPlanktonInState* son implementaciones de *GameStateToStringFunction*.

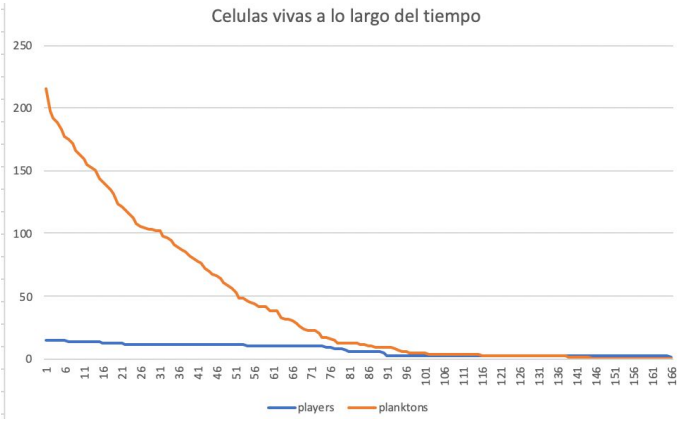
A continuación encontrarán capturas de los resultados de ambos *outputters*.

- *State to xyz:* Se utilizó para visualizar la simulación en *Ovito*, siendo los jugadores rojos y el plankton verde



Screenshots de Ovito

- *Count players and plankton in state:* Cuenta la cantidad de jugadores y planktons vivos en un instante determinado, se puede utilizar para graficar y comprobar el comportamiento de las estrategias.



Screenshot de gráfico hecho en Excel

Main.hs

El archivo main contiene la llamada al iniciador del juego, y es donde se configuran inicialmente todas las variables del partido (cuántos jugadores y planktons, radios inciales, tamaño del tablero, etc..)

```
s1 = 130 -- seed for players
s2 = 222 -- seed for planktons
boardSize = 100
noY = 15 -- number of players
rY = 1.0 -- initial radius of players
noK = 200 -- number of planktons
rK = 0.5 -- planktons radius
outputters = [
  writeCsvOutputter "./runs/count_2.csv" countPlayersAndPlanktonInState
countPlayersAndPlanktonInStateHeader,
  writeFileOutputter "./runs/r2.xyz" stateToXyzFormat
```

```
  ]

main :: MainReturnType
main = do
  startSimulation s1 s2 boardSize noY rY noK rK  outputters
closestAgentGreedy
```

Misceláneo

En el directorio del proyecto se pueden encontrar más archivos con código, como *EatingMechanics.hs*, *CellUtils.hs, Cells.hs y Random.hs*. Estos archivos contienen las funciones para crear la lista inicial de células, decidir cuándo se muere una célula o no, cómo generar aleatoriamente todos los estados iniciales, etc.

Conclusiones

La utilización del paradigma orientado a objetos facilitó unas abstracciones que no se habían podido hacer con facilidad en *Java*, además de permitirnos hacer iteraciones a través de todas las células con abstracciones y el uso de funciones de alto orden (*map, filter, any, entre otras)*.
Otra observación sobre una ventaja no prevista fue la diferencia de performance que hubo con el programa escrito en *Java*. Las simulaciones originales las hicimos en *Java,* y nos tomaban hasta cinco minutos correrlas mientras que este programa en Haskell prácticamente sólo tarda lo que tarda en escribir en disco los *outputters*.