Segundo Proyecto de las Asignaturas Simulación y Programación Declarativa (Informe)

Datos del alumno

`nombre: Daniel Orlando Ortiz Pacheco`
`grado: C-412`
`correo: daniel.ortiz@estudiante.matcom.uh.cu`
`github: https://github.com/danielorlando97`
`repo del proyecto: https://github.com/matcom-school/haskell-agent.git`

Orden del Problema

Realizar una simulación utilizando como lenguaje de programación `Haskell` y poniendo en practica lo aprendido sobre `Los Agentes`. El ambiente en el cual intervienen los agentes es discreto y tiene la forma de un rectángulo de N $\times$ M. El ambiente es de información completa. El ambiente puede variar aleatoriamente cada t unidades de tiempo. El valor de t es conocido. Las acciones que realizan los agentes ocurren por turnos. En un turno, los agentes realizan sus acciones, una sola por cada agente. En el siguiente, el ambiente puede variar. Si es el momento de cambio del ambiente, ocurre primero el cambio natural del ambiente y luego la variación aleatoria. En una unidad de tiempo ocurren el turno del agente y el turno de cambio del ambiente. Los elementos que pueden existir en el ambiente son obstáculos, suciedad, niños, el corral y los agentes, cada uno de estos miembros tiene distintas peculiaridades que se detallaron en la orden del proyecto.

Ideas Principales

Dadas las características del lenguaje y algunas condiciones del problema, inicialmente se considero que los agentes deberían ser proactivos, pues:

- `Haskell`: es un lenguaje funcional que aboga por la mayor pureza en sus implementaciones, y como tal una implementación basadas en estados internos que conserven los agentes para tomar decisiones basadas en una percepción anterior y solo modificar estas decisiones y estados basados en eventos, podría llegar a ser más complejo de modelar. Mientras que un agente proactivo, que paso a paso siempre debería computar todos los factores importantes que influyan en su eficacia al cumplir su tarea, se ve favorecido en `Haskell` frente a otros lenguajes por su característica de`lazy evaluation` y las optimizaciones sobre las funciones puras
- `Características del problema`: El ambiente es completo, característica que un agente puramente reactivo no explotaría en su máxima expresión al solo fijarse en determinados sucesos. Según la descripción del problema los agentes se mueven antes que el cambio natural del ambiente, que aunque se pudiera modelar un agente puramente reactivo que

responda al evento que sucedió en el tiempo t-1, en primeros momentos se consideró que esta condición provocaría que en los agentes tuvieran tomar desiciones que no serían influenciados por algún evento del ambiente. Y por últimos, en la documentación del problema se plantea un objetivo para el agente, "Mantener la suciedad del ambiente por debajo del 60%"

Partiendo de esa base, se modelos un ambiente donde se encuentras C niños y A roboses. Donde los niños representan la rama impura (funcionalmente hablando), pues toman muchas decisiones aleatoriamente siendo este un efecto segundario que convierte a toda function que depende de el en una función impropia. Y por otro lado los roboces son la rama propia, pues sus decisiones esta siempre basadas en los parámetros de entrada (el ambiente en el tiempo t), los agentes forma un sistema multi-agentes, donde en cada momento t de decisión inicialmente realizan una percepción sobre el ambiente de manera individual, los resultados los socializan entre todos para terminar tomando, entre todas las percepciones, una decision por cada agente que favorezca el objetivo final, común para todos.

Una en cada turno de los agentes, cada robot genera una percepción sobre el ambiente. Cada percepción consta de los niños a los que dicho robot puede alcanzar ordenados en función de cual sería mejor que este recogiera, para realizar esta evaluación y comparación se calcula el camino de costo máximo del robot hacia cada bebe, ponderando ligeramente las casillas con suciedad por encima de las demás y se comparan las longitudes de los caminos, siendo los mas cortos mas económicos para lograr la misión. Ademas contiene una ruta alternativa, para tener otro plan caso que al socializar las percepciones se decida que dicho agente no debe recoger a ningún infante o simplemente el mismo no tiene camino hacia ninguno, esta ruta se conforma con el camino de suciedad mas largo que es adyacente al robot.

Modelos de Agentes

En la moderación de dichos agentes proactivos se consideraron algunas heurísticas para optimizar las decisiones de estos. Los agentes presentan dos estados (con niño en cima o libre), los agentes que no tiene a niños cargados se pueden mover paso a paso y en cada turno debe decidir si limpia, se mueve o carga un niño adyacente al este. Y en el caso de un agente que carga a un bebe, se puede mover avanzando hasta dos casillas consecutivas o decidir descargar al infante. En búsqueda de la metodología más eficiente para los roboses se consideraron varias opciones:

- Robot de Limpieza (Sin Niño):
  - Siempre limpiar: Cuando el agente se encuentra en una casillas con suciedad, no realiza más percepción que la caracterización de su propia casilla y siempre decide limpiar antes que moverse. En caso de que la casilla este limpia investiga hacia que bebe moverse, en caso

de multi-agente solo se socializaría dicha investigación la decision de limpiar sería particular de cada robot

  - Siempre recoger: En el turno de los roboses, siempre realizan sus percepciones y las socializan (en caso de multi-agente) asignando un agente a cada niño y los mismo siempre decidirán moverse rumbo a su respectivo niño sin importa en la casilla en la que este se encuentre. En esta estrategia solo se limpia si la cantidad de roboses en el ambiente es mayor que la cantidad de niños fuera del corral

  - Recoge solo si esta cerca sino limpia: Considerando que los niños en cada turno decide entre tres opciones respecto a los agentes (alejarse, acercarse, o quedarse en el lugar) y esta decisión es una variable aleatoria con distinta función de probabilidad según la distancia que existe entre el niño y el robot en cuestión. Esta metodología busca aumentar la probabilidad de que el niño se quede en el mismo lugar y garantizar su captura. Para al modelar el problema se considero que los movimientos rendirán 8 posibilidades contando las diagonales, lo cual provoca las siguientes funciones de probabilidades:

    * si un robot y un niño tienen al menos una casilla de por medio:
      · probabilidad de alejarse del robot (1/3): cuando se mueve en dirección contraria a la aproximación del agente más las otras dos diagonales correspondientes
      · probabilidad de mantener la distancia (1/3): que el niño no se mueva, más las dos casillas transversales a la aproximación del robot
      · probabilidad de acercarse al robot (1/3): caso contrario a los movimientos en que el niño se aleja
    * si un robot es adyacente a un niño: En este caso el niño pierde una posibilidad de movimiento transformando la posibilidad de alejarse en 3/8 y el resto de los movimientos mantendrán la distancia 0 entre el niño y el robot (5/8)

    Siguiendo esta análisis bajo esta lógica los roboses solo limpian si esta a una distancia mayor de una casilla con algún niño, en caso contrario se mueven hacia esa casilla, para ser adyacente al niño antes que este genere su próxima decisión

- Robot de Limpieza (Con Niño):
  - Siempre caminar: El agente nunca considera la opción de descargar al niño, siempre intenta moverse rumbo al corral, solo recomputa el camino hacia el corral en caso de chocar con un obstáculo
  - Siempre busca un camino: En cada turno se analiza si se puede llegar al corral, en caso de que no encuentre camino deja al niño y comienza a limpiar. Para la implementación de esta variante el robot sin niño debe evaluar si existe camino entre el niño y el corral antes de decidir si limpiar o moverse hacia el.

Implementación

La implementación se separo entre tres grandes módulos **(Agent, Child, Ambient)** y otros auxiliares. Considerando la fortaleza de `Haskell` en el trabajo con listas se considero que la mejor manera de definir una matriz de NxM sería con una lista de tamaño N*M, definiendo las respectivas funciones para mapear los indices del espacio de matrices hacia los arrays y viceversa. Definiendo la estructura de un ambiente como:

```haskell
data SpaceOfAmbient = Clean | Dirt | Corral | Obstacles deriving (Show)
data MemberOfAmbient = Child | Robot | RobotWithChild deriving (Show)
type IndexOfAmbient = (Int, SpaceOfAmbient, Maybe MemberOfAmbient)

data Ambient = Ambient
  { getChildCount :: Int,
    getN :: Int,
    getM :: Int,
    getList :: [IndexOfAmbient] -- matriz
  }
  deriving (Show)
```

En este módulo se define varias funciones importantes para el desarrollo del resto de los módulos:

- *createAmbient* y *generateAmbient* : Estas funciones se utilizan para la simulación de cambio de ambiente y para crear los ambientes iniciales de cada simulación. El método createAmbient se apoya en generateAmbient, el cual genera ambientes aleatorios manteniéndo ciertas posiciones
- *changeAmbient* : Esta función recibe un ambiente y una lista de cambios y devuelve el nuevo ambiente efectuado los cambios
- *getAdjByDirection* y *getAdjList* : Funciones que dado un indice y un ambiente devuelve las posiciones adyacentes de ese indice (de la lista) en la matriz. Valiéndose siempre de la siguiente definición:

```haskell
data Direction = N | NE | E | SE | S | SO | O | NO

directionList :: [Direction]
directionList = [N, NE, E, SE, S, SO, O, NO]

directionToTuple :: Direction -> (Int, Int)
directionToTuple d = case d of
  N -> (-1, 0)
  NE -> (-1, 1)
  E -> (0, 1)
  SE -> (1, 1)
  S -> (1, 0)
  SO -> (1, -1)
```

```
O -> (0, -1)
NO -> (-1, -1)
```

- *getQuadricycle* : Función que devuelve una lista de la cuadricula de la que un cierto indice es centro

Apoyado en el módulo de **Ambient** (mayormente en las funciones puras) se desarrollo el módulo **Agent** con la definición de agentes proactivos que fueron respondiendo a las heurísticas antes señaladas para finalizar que las que resultaron más eficientes a lo largo del desarrollo y la experimentación, recoger solo si esta cerca (para agentes sin niños) y siempre buscar camino (para agentes con niños). El desarrollo de esta módulo se baso mayormente en la definición de lo que se entiendo como una percepción:

```
data Perception = Perception
  { getIndex :: Int, -- posición actual del agente
    getBestPathTo :: Int -> Maybe (Int, [Int]), -- función que dado el indice (de lista) de
                                                -- devuelve el mejor costo y camino del age
    selectionOrd :: [Int], -- los indices de los niños, alcanzables por el agente, ordenadas
    getAlternativeRoute :: Int -- indice inicial del camino alternativo
  }
```

En dicho módulo se define varias funciones que dan vida a los agentes:

- *see* : Función que dado un agente y el ambiente en el tiempo t genera la percepción correspondiente. Para la creación de dicha percepción se realiza un bfs y un dijkstra por cada niño alcanzable por el agente, estas búsquedas se resuelven a partir de las implementaciones del módulo `Algorithm.Search`, y además se realiza un dfs que solo se mueve por las casillas sucias, esta búsqueda si fue implementada por el autor con la función *adjDirtPath*

- *socialize*: Función que dada una lista de percepciones devuelve una lista de cambios al ambiente. En este método se realiza la comparación entre las distintas percepciones y decide para cada agente cual sera la acción que el mismo realizara en el turno en cuestión. A continuación muestra la función que define la comparación entre dos percepciones

  ```
  -- True if b perception is better of a
  comparePerceptions :: Int -> Perception -> Perception -> Bool
  comparePerceptions index a b = -- index posición de lista de un niño del ambiente
  case (getBestPathTo a index, getBestPathTo b index) of
    (Just aPerception, Nothing) -> False
    (Nothing, _) -> True
    (Just aPerception, Just bPerception) -> length (snd aPerception) > length (snd bPerce
  ```

- *robotWithChildMove* : Esta función es la encargada de gestionar el "Agente con Niño", no es mas que un bfs de `Algorithm.Search` en cada turno para encontrar el camino hasta el corral; en casos de no encontrar camino gestiona la descarga del niño, para la cual se tomo en consideración que

los niños no deben estar en una casilla con suciedad. En caso de que el agente se encuentra ya sobre el corral entonces descarga al niño.

De igual manera que el módulo de **Agent**, el módulo **Child** se desarrollo basado en las herramientas que ofrece el módulo **Ambient**. En el eje principal de este módulo se encuentra la definición de lo que representa una acción de un niño:

```
data Action = Action
  { getIndex :: Int, -- nuevo indice
    lastIndex :: Int,-- posición actual
    getDirection :: Direction -- dirección del movimiento
}
```

En dicho módulo se define varias funciones, mayormente impuras, que generan en cada paso una acción para cada niño. El módulo requiere de un constante trabajo con valores aleatorios, para la cual se uso la función *randomRIO* y el functor `MonadIO` de los módulos `System.Random` y `Control.Monad.IO.Class`:

- *getChildAction*: Función que dado un niño, genera o no una acción para el. La función comienza decidiendo si el niño se moverá o no, generando un entero aleatorio entre 0 y 1. En caso de que se decida mover el método filtra todas las posibles direcciones para las que este se pueda mover, según las restricciones de la orden. Entre las restantes se realiza una selección aleatoria y se crea la `Action` correspondiente
- *executeChildAction*: Función de dado un una lista de `Action` retorna una serie de cambios para el ambiente actual. Esta función se encarga de crear todos los cambios para reflejar el desorden que provoca una accion de los niños, tanto el desplazamiento de obstáculo, como la aparición de suciedad en cada una de las cuadriculas a las que pertenecía el niño antes de moverse

Finalizamos la sección de implementación señalando que todo lo anterior se complemente con los módulos `Simulation` y `Statistics`. El primero se encarga de orquesta toda la simulación, guardar los históricos y computar la suciedad media del ambiente al concluir cada simulación. Mientras que el segundo contiene las funciones *median* y *variance* que se utilizan para computar el resultado de cada simulación y controlar cuando se puede considerar que la simulación a arrojado un valor medio fiable y parar la ejecución del programa

Conclusiones

A lo largo del desarrollo fueron planteadas diversas heurísticas para resolver la auto regulación de los agentes. Dichas heurísticas son fruto de el ciclo (implementación -> experimentación -> optimización). Gracias a este proceder se observaron varias consideraciones sobre las heuristicas y los experimentos en cuestión. Sobre el experimentos se puede concluir que, este esta fuertemente influenciado por los valores iniciales (tamaño de la matriz, cantidad de agentes y niños, factor de cambio) provocado que para cada conjunto de datos iniciales se

observe que la varianza de la lista de resultados converge hacia distintos valores, lo cual imposibilita la definición de un valor de varianza (alpha) para detener el experimento al llegar a dicho valor. Por otro lado dicha influencia de los valores de entrada también tiene su efecto en la selección de que heuristica seleccionar:

- *Matriz pequeñas con factores de cambio grandes*: En estos casos se obtienen mejores resultados con la heuristica de *"Siempre recoger"* ,para "Agentes sin Niños", y *"Siempre buscar camino"* , para "Agentes con Niños". Y analizando los ambientes y escenarios que se pueden generar es un resultado bastante lógicos, al ser la matriz de tamaño pequeño los caminos (roboces - niños) y (niños - corral) no tienen dimensione excesivas, si simultáneamente el factor de cambio es muy grande con respecto al tamaño de la matriz entonces a los roboces les da tiempo a recoger a los niños, guardarlos y limpiar el ambiente antes que se vuelva a generar un nuevo ambiente. De igual manera; una matriz pequeña provoca que los obstáculos corten muchas rutas, con lo cual es muy probable que un "Agente con Niño" ,en el momento que se genera un ambiente nuevo, se quede en una posición en la que no pueda alcanzar el corral, con lo cual es mejor que dicho agente tenga la habilidad de detectar estos casos y pueda soltar al niño y ponerse a limpiar

- *Factores de cambio pequeños*: En estos casos se obtienen mejores resultados con una heuristica de *"Siempre limpiar"* y *"Siempre buscar camino"*. Como el factor de cambio es muy pequeño, cada pocos pasos del agente se generan ambientes nuevos, por tanto no tendría mucho sentido hacer grandes desplazamientos para recoger a un niño. Ademas de que con cambios frecuentes cambios de ambientes se generan frecuentes cambios de rutas hacia el coral

- *Más roboces que niños*: En estos casos se obtienen mejores resultados con una heuristica de *"Recoge solo si esta cerca sino limpia"* , para los "Agentes sin Niños". Como este método de tomar decisiones incluye un procesos de socialización de las percepciones entonces solos los agentes mas cercanos a los niños son los que se mueven rumbo a los bebes, mientras el resto optimiza la cantidad de casillas que puede limpiar

Para el caso más general se escogieron las heurística *"Recoge solo si esta cerca sino limpia"* y *"Siempre buscar camino"*, las cuales para los distintos casos se obtuvieron resultados aceptables y no muy lejanos de las mejores opciones para el caso en cuestión.