Proyecto Final Simulación-Prgramación Declarativa.

Tema: Agentes.

Autor:

Abel Antonio Cruz Suárez C-411

Ejecución

Para ejecutar el proyecto se deberá ejecutar en una consola donde se encuentra el archivo $Tester.hs$ el comando $ghci$, luego se ejecuta : $l\ Tester.hs$ y finalmente se llama a la función $run\ 0\ 0$. Este método esta hecho para simular 10 ambientes distintos con los 4 tipos de agentes. Dentro de las función existen unas listas con las cantidades iniciales de cada elemento para el $i-ésimo$ ambiente. Por tanto si se desean modificar solo tiene que cambiar dichos valores. La lista $envs$ está formada por tuplas que indican $(filas, columnas)$ de un ambiente.

Los resultados son mostrados en una lista de tuplas formada por una tupla que contiene como primer elemento el promedio del porciento de suciedad luego de finalizar las n simulaciones en un ambiente, la cantidad de veces que se cumplió el objetivo de mantener el ambiente con un nivel de suciedad inferior a 60% y la cantidad de veces que se falló. El segundo elemento es el tipo del agente que obtuvo dichos resultados.

La función $runSingle$ funciona de igual forma a la anterior, solo que está hecha para simular un solo tipo de ambiente, para los 4 agentes. Recibe los mismos parámetros que run.

Módulos

ENV

Para plasmar la idea de lo que representa el ambiente se creo un nuevo tipo de datos llamado ENV. Su principal característica es que sus funciones constructoras son en su mayoría listas de tuplas de Int que permiten determinar las posiciones del agente u elemento del ambiente que representan. El tipo ENV tiene como funciones constructoras las siguientes:

- $rows :: Int$: Esta funcion permite conocer el numero de filas del ambiente
- $columns :: Int$: Esta función permite conocer el numero de columnas del ambiente
- $kids :: [(Int, Int)]$: Es una funcion utilizada para mantener las posiciones de los niños.
- $obstc :: [(Int, Int)]$: Se utiliza para mantener las posiciones de los obstáculos.
- $dirt :: [(Int, Int)]$: Devuelve una lista con las posiciones de las celdas del ambiente sucias.
- $playpen :: [(Int, Int)]$: Devuelve una lista con las posiciones que ocupa el corral.
- $robots :: [(Int, Int)]$:lista con las posiciones que ocupa cada agente de limpieza en el ambiente.
- $carryingKid :: [Bool]$:Devuelve una lista indicando que el i-esimo agente se encuentra cargando un niño.
- $playpenTaken :: [(Int, Int)]$: lista indicando las posiciones del corral que han sido tomadas.

Trabajar con un ambiente que se materializa a través del tipo Env nos fue muy beneficioso ya que cuando alguna función necesitaba modificar el ambiente recibía un ENV que podía cambiar agregándole nuevas listas con posiciones cambiadas o boleanos distintos, y este pasaba a ser el nuevo ambiente actual.
Por otra parte podemos agregar que resulta totalmente escalable esta solución ya que si en una

instancia futuro queremos agregar nuevos elementos al ambiente(díganse baches, mascotas, etc) solo debemos agregar otras funciones constructoras con las posiciones de dichos elementos.

Módulo Agent

Para la creación del agente nos basamos en la arquitectura de Brooks (de categorización o inclusión). Esta arquitectura nos permite realizar una acción exclusiva basándonos, únicamente, en las entradas perceptúales. Como varias de estas acciones serían propensas a realizarse según lo que percibe el robot (ej. se pudiera limpiar o buscar un niño pues hay suciedad y niños sueltos), es necesario ordenar las acciones que queremos realizar en una jerarquía por categorías, con los comportamientos ordenados por capas. Las capas inferiores de la jerarquía inhiben a las capas más altas: mientras más abajo este una capa, más alta será su prioridad.

Al utilizar las guardas que nos proporciona el lenguaje funcional haskell se hace más sencillo programar el proceder de los agentes. Esto se debe a que al cumplirse la condición de una guarda, se realizará la acción propuesta por esta y más ninguna. De esta forma se simula el comportamiento de cada agente en un turno, pues se asegura que solo una acción será realizada.

Con la función $action$ es como se define la toma de decisiones. Destacar que nos valemos de la característica de $pattern\ matching$ de haskell que en dependencia de que tipo de agente $(0, 1, 2)$ se base en una jerarquía u otra.

Las conductas establecidas para los agentes son las siguientes:

1. Si está sucia la celda sobre la que está parado el agente, se limpia
2. Si está cargando un niño y se encuentra en una celda disponible del corral para colocarlo, lo deja en dicha celda.
3. Si está cargando un niño, lo lleva hacia el corral.
4. Si detecta suciedad, se mueve hacia la posición de la celda sucia más cercana.
5. Si detecta un niño, se mueve hacia la posición de la celda con niño más cercana.
6. Quedarse quieto.

Dependiendo del tipo de agente se ordenan las conductas de la siguiente forma:

- Tipo 0 :

  $1 < 2 < 3 < 5 < 4 < 6$

- Tipo 1 :

  $1 < 2 < 3 < 4 < 5 < 6$

- Tipo 2 :

  $1 < 4 < 2 < 3 < 5 < 6$

- Tipo 3:

  Este es un agente especial que si el porciento de suciedad del ambiente es mayor de 30% se comporta como un agente de tipo 2, en caso contrario se comporta como tipo 0.

Como podemos apreciar nuestros agentes siempre tienen como su objetivo primordial el de limpiar el ambiente. Es por ello que como primera conducta siempre tiene la de limpiar si la celda en la que están está sucia. Luego los dos primeros tipos de agente en caso de cargar un niño intentaran colocarlo en el corral o llevarlo en caso de no estar en el lugar del corral posible. La diferencia radica en que el tipo 0 se enfoca en eliminar la fuente de suciedad, por ello escoge buscar un niño antes de buscar una celda sucia que es lo que prioriza el agente de tipo 1. Luego tenemos un agente de tipo 2 que a diferencia de los otros va a priorizar buscar una celda sucia antes que colocar un niño en el corral o llevarlo.

No es difícil notar que el comportamiento de estos agentes reactivos resuelve el problema, pues como principal conducta tiene la limpieza de una celda. Al darle un orden jerárquico a sus conductas estamos propiciando diferentes estrategias que pueden ser tomadas para llevar a cabo la limpieza y esto nos conducirá, a que dadas determinadas condiciones, se puedan saber que agentes son más eficientes, lo cual será analizado en la sección dedicada a las simulaciones.

Es necesario destacar aquí que como conducta más básica se pudiera tener la de evitar un obstáculo cambiando la dirección, pero como el ambiente es de información completa entonces es posible realizar un bfs que trace el mejor camino hacia un objetivo evitando los obstáculos.

Módulo Robot

En este módulo es donde recaen las funciones que son realizadas por el agente y como se desarrolla su interacción con el ambiente. Es importante aclarar que muchas de estas funciones representan acciones específicas que realiza el robot y por tanto tienen un impacto sobre el ambiente que puede implicar su modificación. Ejemplo de esto es la acción de limpiar. Al estar realizando acciones que pueden transformar el ambiente es necesario recibir en dichas funciones el ambiente actual y devolver uno con los cambios realizados.

Dentro de las funciones que se almacenan tenemos:

- bfs : Este método es un bfs tradicional que es el símil de lo que representa una función see de un agente. En otras palabras, representa la visión que tiene el agente sobre un objetivo en el ambiente. Se habla de un objetivo pues a esta función se le pasa una lista con las posiciones objetivo que se quieren alcanzar. De esta forma el método resulta más reutilizable, pues en el caso de que el objetivo a encontrar sean los niños, una posición específica o las basuras; solo es necesario pasar como parámetro la lista con dichas posiciones.

  A modo general lo que hacemos en este método es dado una posición buscamos todos sus adyacentes que no se encuentran en la lista de visitados, ni en la de posiciones que faltan por visitar. Luego con los adyacentes restantes analizamos si son posiciones libres para un robot y esos que resten se agregan a una lista de nodos por visitar. A parte vamos manteniendo una lista tuplas de tuplas de la forma $[(pos\_hijo, pos\_padre)]$, donde se conoce de cada celda visitada, quién la descubrió. Si se alcanzó uno de los objetivos se retorna una tupla conformada por una lista de los nodos visitados y sus padres junto con la posición del objetivo alcanzado. En caso contrario se devuelve una lista vacía junto con una posición fuera de la matriz.
- $updatePi$: Esta función devuelve una lista de tuplas indicando que el padre de cada una de las posiciones en la lista que recibe.
- $nextStep$: Con esta función es como se confecciona el camino desde el objetivo encontrado hacia la celda inicial donde se encuentra un robot. Destacar que siempre que reciba una lista ("$road$") con las tuplas de posición de un nodo y posición de su padre, indica que existe un camino entre la posiciones $start$ y $final$. Pues de lo contrario se recibe una lista vacía del bfs y se retorna como camino la posición actual del agente.

  Para construir el camino se hace un llamado a la función $findParent$ que dado una celda retorna quién es su padre. De esta forma si nos vamos moviendo de padre en padre, empezando por la celda objetivo final, confeccionaremos un camino hacia la celda inicial donde se encuentra el agente.
- $findParent$: Este es la función encargada de dado una celda y la lista de tuplas de tuplas $[(pos\_hijo, pos\_padre)]$, determinar la posición del padre.

- $getStep$: Esta función se apoya en las anteriores para determinar el siguiente paso que debe dar el agente. Se realiza el bfs buscando la primera posición del objetivo más cercano, luego construimos el camino y en caso de que estemos cargando un niño y el camino sea mayor que dos celdas, nos movemos a la segunda celda encontrada, en otro caso a la primera de la lista del camino.

La siguientes funciones están más enfocadas a las acciones del robot que pueden modificar el ambiente, algunas que estarían relacionadas con su función "see" y otras a conocer su estado actual:

- $isCarryingChild$: Permite conocer si el agente se encuentra cargando un niño.
- $detectDirty$: Función para conocer si existe suciedad en el ambiente.
- $detectKid$: Función para conocer si existen niños sueltos en el ambiente.
- $clean$: Función para limpiar una determinada posición
- $dropKid$: Con esta función se simula la acción de dejar un niño en el corral. Para ello se recibe una posición factible(más adelante se explica que sería una posición factible) donde dejar un niño en el corral por un agente. Luego procedemos a marcar dicha posición como que ha sido tomada del corral, indicamos que el agente que dejó al niño ya no se encuentra en un estado de cargando un niño(setear en False en la lista carryingKid, en el índice del agente que estaba cargando al niño y lo suelta) y quitamos de las posiciones que ocupa el corral la que fue tomada, para que los próximos agentes no la tengan como posible posición a colocar un niño.
- $carryKidToPlaypen$: Esta función actualiza la posición de un robot, con la siguiente que debe dar para ir hacia el corral.
- $moveTowardsObj$: Actualiza la posición de un robot con la siguiente que debe tomar para llegar a un objetivo. Si se encuentra con un niño simula la acción de cargarlo. Inicialmente esta función se encontraba dividida en dos funciones $moveTowardsKid$ y $moveTowardsDirty$, pero en aras de que esta función fuera más reescalable, se determina que paso dar en dependencia de la lista de objetivos. De esta forma dará el mejor paso hacia un niño o suciedad en dependencia de la lista de posiciones objetivo que se le pase.

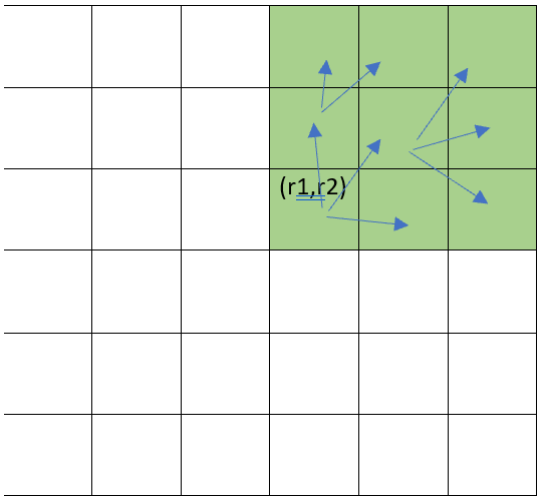
Módulo playpen

Para construir el $playpen$ se realiza un bfs como el del módulo robot, con la diferencia de que no es necesario llevar una lista con los nodos visitados y de quien proceden.
Una de las cuestiones principales es como establecer una especie de orden en las celdas ocupadas por el corral, para colocar a los niños. Esta necesidad surge de que se pueden dar ocasiones en que al colocar a los niños en una determinada disposición en el corral, estos impidan el paso hacia las celdas interiores de agentes que cargan más niños.

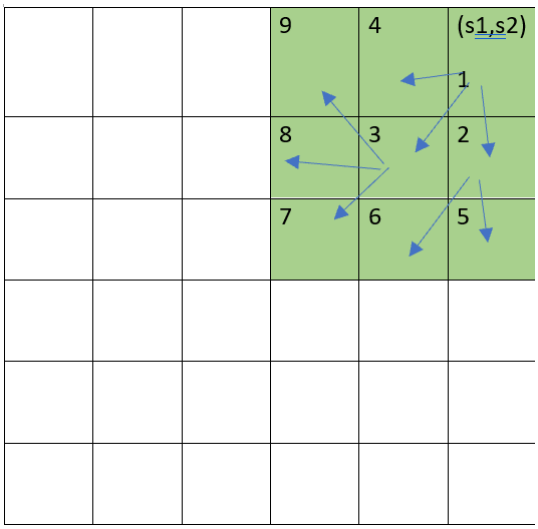
Ejemplo: Si la celda por la que empieza a expandirse el corral (mediante bfs) es la (1,1) entonces sus adyacentes serán las próximas celdas a analizar en el algoritmo de bfs y que, por tanto, pueden constituir parte del corral. Si comenzamos colocando niños en los adyacentes $(2,2),(2,1),(2,0),(1,2) y (0,2)$; entonces será imposible acceder a la celda (1,1) y el resto de sus adyacentes que también formarían parte del corral, pues los niños constituyen obstáculos en ese caso.
Después de analizarse varias ideas para dar solución a este problema, algunas fueron: asignarle un número especial indicando el orden por el que se debe comenzar a colocar en el corral; escoger las posiciones donde colocar mediante un criterio de la más cercana a la central utilizando la distancia euclidiana(por lo que la inicial sería la propia central, luego cualquiera de sus adyacentes, después los adyacentes de sus adyacentes, etc.); determinar el orden en que se debe construir el corral mediante el uso de bfs, pero cambiando la forma en que se escogen los adyacentes; entre otras. Todas estas ideas fallaban en dar una solución factible al problema de

que se trabara el corral, muchas de ellas por la misma razón, se colocan los niños en un borde exterior haciéndose imposible acceder a las inferiores.
Finalmente, la solución encontrada fue la siguiente:
En el método $buildPlaypen$ obtenemos las celdas que conformaran el corral. Esto se hace mediante la función $expandPlaypen$, que como se expone anteriormente, no es más que un algoritmo de bfs con algunas modificaciones. Este algoritmo recibe una posición a analizar y si no ha sido visitada se procede a determinar cuáles son sus adyacentes que no están en la lista ady que contiene los futuros nodos a visitar. Para ello se usa el método $disjoin$ entre los adyacentes del nodo y los nodos de la lista ady y esto nos da aquellos elementos que están en la primera lista y no en la segunda, luego procedemos agregarlos. Aquí importante señalar que como la matriz esta vacía no es necesario el análisis de sí los adyacentes están libres, pues siempre lo están. El segundo paso sería ir conformando la lista de celdas del corral, agregando el nodo visitado y construyendo la lista mediante llamados recursivos, que solo se detienen cuando la cantidad de niños es cero, indicando que hemos creado una lista de posiciones del corral de tamaño cantidad de niños.
En esta segunda parte es donde esta la idea que da solución al problema del corral. Para ello se determina de las celdas encontradas sus distancias a las esquinas de la matriz (se conoce la dimensión de la misma), y para cada una de ella nos quedamos con la menor, con esto estamos calculando la distancia de cada celda a su esquina más cercana. Una vez hecho este proceso se busca la celda con la menor distancia, que geométricamente hablando, es aquella celda del corral más próxima a su esquina más cercana. Esta nueva posición $((s_1, s_2))$ constituye el punto de partida a partir del cual se va a colocar el primer niño, y se va a comenzar a realizar el algoritmo bfs anterior para obtener en orden las celdas el las que se deben seguir colocando los niños para no traba el corral. Este bfs, que comienza en (s_1, s_2), nos vamos a asegurar que solo descubra celdas de las que ya constituyen el corral. La idea es construir una nueva disposición de celdas del corral tal que el primer elemento de la lista sea la primera primera posición donde colocar un niño sin trabar el corral, luego la próxima posición para no trabar y así hasta haber seleccionado todas las celdas.

Ilustración gráfica:



En esta imagen lo primero que se buscan las celdas que pertenecerán al corral, si la posición de inicio para expandirlo es la (r_1, r_2). Notar que si se llenan las dos celdas encima de la posición (r_1, r_2) y luego las dos a su derecha, entonces no se puede acceder a las interiores.



Luego de determinar que la celda con coordenadas (s_1, s_2) es la que más cerca está de su esquina $((0, 5))$, realizamos el bfs a partir de la posición (s_1, s_2) tomando como adyacentes a visitar los que formen parte del corral existente. Esto nos asegura que la nueva distribución del corral estará ordenada según la numeración establecida, y al tomarlas en ese orden se da solución al problema.

Anotación

Los adyacentes de una posición se selecciona en el sentido horario, por tanto se visitan en el siguiente orden: norte, noreste, este, sureste, sur, suroeste, oeste y noroeste.

Las otras funciones empleadas para realizar el algoritmo fueron:

- $cells2Corners$: Función para determinar que celda del corral es la más cercana a una de las cuatro esquinas. Para ello se reciben las celdas del corral, la celda más cercana actualmente, las dimensiones de la matriz y la mejor distancia actual. Luego recursivamente se computa la distancia de cada celda, de forma tal que al llegar al caso base devolvemos la mejor encontrada.
- $calculateDistance$: Esta función nos sirve para saber dado una posición y las cuatro esquinas, a cual de ellas está mas cerca y esa es la distancia devuelta.
- $manhatamDist$: Es la forma en la que calculamos la distancia entre dos posiciones de la matriz. Fue más conveniente usar esta, pues evitamos tener que trabajar con $float$ si utilizamos la distancia euclidiana e igualmente nos da un sentido de distancia entre celdas que se ajusta a nuestras necesidades.
- $setPriority$: Funciona de la misma forma que $expandForPlaypen$ solo que en este caso queremos analizar un nodo si previamente fue tomado como parte del corral.

Módulo Kids

Este es el módulo que almacena la lógica del comportamiento de los niños y como actúan sobre el ambiente, o sea, aquí es donde se simulan las acciones de un niño y el resultado de las misma en el ambiente. Ejemplo de ello es que si un niño se mueve genere la suciedad o empuje uno o varios obstáculos, bajo determinadas condiciones.

Las funciones desarrolladas son:

- $moveKids$: Esta función recibe el ambiente y un $stdGen$ utilizado para obtener números aleatorios. Su objetivo es determinar que niños se van a mover, para ello se crea una lista (potencialmente infinita) de números con valor 0 o 1 de forma aleatoria, indicando si en el índice i hay un 0, que el niño i no se mueve y en el caso de que sea 1 si lo hace. Luego se

selecciona los n primeros, siendo n la cantidad de niños sueltos y se procede a simular sus movimientos.

- $findMoves$: Dada una posición de un niño y el ambiente determina la lista con las posibles posiciones a las que se puede mover un niño. Para ello dicha posición debe estar vacía, por lo que buscamos cuales son la celdas que cumplen ser vacías para un niño, mediante la función $emptyCellForKid$.
- $scanKids$: Determina dada una posición y el ambiente la cantidad de niños que existen en sus celdas adyacentes.
- $carriedKids$: Función para saber cuantos niños están siendo cargados por robots.
- $computeKidsInGrid$: el objetivo de esta función es determinar cual de las nueve posibles cuadrículas de $3x3$ a las que pertenece el niño será ensuciada. Para ello escogemos de forma aleatoria uno de sus adyacentes o a su posición y contamos la cantidad de niños de la cuadrícula que contiene a la celda seleccionada como posición central. Una vez hecho esto se devuelve la cantidad de niños encontrada y la cuadrícula que se pudiera ensuciar o no.
- $makeMove$: Con esta función simulamos la acción de un niño de dar un paso.
- $simulateMoves$: Esta función es la encargada de realizar el movimiento del niño y de llamar a las funciones que generarían otros cambios en el ambiente como la aparición de suciedad o mover obstáculos. Aquí se recibe una lista binaria indicando si el niño en la i-ésima posición se moverá o no, la lista de niños en sus posiciones actuales un generador y un ambiente.

  Haciendo uso de $pattern\ matching$ determinamos que si valor del primer elemento de la lista binaria es 0 entonces el niño no se mueve y el ambiente no se modifica, solo se hace un llamado recursivo con los restos de las listas, para continuar a analizar otros niños.

  En caso de que el primer elemento de la lista binaria sea 1, entonces pasamos a determinar los posibles movimientos que puede tomar un niño dado un ambiente. Esto lo hacemos con la función $findMoves$, luego procedemos a seleccionar la cuadrícula de $3x3$ que se va a ensuciar en dependencia de la cantidad de niños encontrada en ella. Señalar que esta cuadrícula se hace de forma aleatoria y el paso que de el niño no tiene porque estar contenido en dicho $grid$. Ahora pasamos a determinar si la posición a la que se movió el niño tenía obstáculos, en dicho caso se mueven y se procede a tomar, de forma aleatoria, que celdas de la cuadrícula serán ensuciadas de acuerdo a la cantidad de niños encontrada y se agregan a las celdas sucias actuales. Esto se realiza sobre un ambiente modificado por el movimiento del niño sobre los obstáculos, en caso de no modificarlos solo se crea un nuevo ambiente con las nuevas celdas sucias y sus actuales condiciones.

  En el caso base de este algoritmo retornamos el ambiente con todos los cambios que sufrió tras mover a todos los niños que se determinaron para moverse.

  Como se puede apreciar en el código con cada llamado recursivo y pasamos un nuevo $random\ seed$ obtenido a partir de la función $randomR$. Con el objetivo de obtener generadores distintos se hacen varios usos de esta función con entradas distintas. Asegurando así que en las funciones donde es necesario realizar operaciones estocásticas no se han empleado o escogido los mismo valores.

Módulo Dirt

Este modulo alberga la función encargada de generar la suciedad tras el movimiento de un niño. La idea es que cuando un niño se mueve se analice la cuadricula de $3x3$ que tiene como centro su antigua posición. Luego nuestra función $generateDirt$ recibe los niños encontrados en la cuadricula, la lista con las posibles posiciones a ensuciar(celdas libres) de la cuadricula de 3x3 y

dos generadores. Esto nos sirve para, según la cantidad de niños en la cuadricula, escoger aleatoriamente la cantidad que se va a ensuciar, atendiendo los limites establecidos en el proyecto, y cuales de dichas posibles celdas a ensuciar serán tomadas de forma aleatoria.

Módulo Obstacle

Este módulo posee dos funciones encargadas de conocer si se puede mover cierto obstáculo y otra realiza el movimiento de los obstáculos.

Las funciones encargadas de dichas responsabilidades son:

- $canMoveObstcs$: Esta función recibe como parámetros una posición (x, y) a la que el niño se va a mover y otra posición (x_1, y_1) que no son más que la resta de las coordenadas destino menos las del origen. De esta forma si sumamos (x_1, y_1) a (x, y) obtenemos una coordenadas (x_2, y_2) que es la posición a la que se pudiera desplazar un objeto si está vacía la celda, si es otro el obstáculo hacemos un llamado recursivo partiendo como celda de destino (x_2, y_2) y le sumamos (x_1, y_1) para determinar que se encuentra en la siguiente dirección en que se está desplazando un obstáculo. Se retorna que no se puede mover un obstáculo en determinada dirección si se alcanza el límite del ambiente o se llega a una posición no vacía que no tiene un obstáculo.
- $moveObstc$: Este método, al igual que el anterior, recibe una posición donde se encuentra un obstáculo, las coordenadas que hay que sumarle a su posición para desplazarlo en la dirección que se movió un niño y la lista de obstáculos. Pasamos por cada uno de los obstáculos, si no es el que se movió entonces se guarda en la lista y llamamos recursivamente con las mismas condiciones pero con el resto de obstáculos. Si se encuentra el obstáculo que se quiere mover se guarda en la lista sumándole las nuevas coordenadas para desplazarlo en la dirección requerida $(p1 + x1, p2 + y1)$. En este instante en las coordenadas $(p1 + x1, p2 + y1)$ puede ser que hayan dos obstáculos, por lo que se hace un llamado recursivo para mover $(p1 + x1, p2 + y1)$ hacia $(p1 + 2x1, p2 + 2y1)$ y así sucesivamente hasta haber desplazado todos los obstáculos.

Módulo Utils

Este módulo almacena aquellas funciones de utilidad general y que por ende son usadas en varios módulos. Aquí tenemos:

- $inList$: Esta función recibe un elemento y una lista y devuelve $True$ si dicho elemento está en la lista.
- $randomNumbers$: Esta función es la encargada de dado un entero x devolvernos una lista, que puede ser infinita, de números entre 0 y $x - 1$. Con esto nos estamos aprovechando de que en haskell se realiza una evaluación perezosa, de esta forma nuestra lista infinita no se computa realmente, solo cuanto necesitemos de ella. En muchas ocasiones solo es necesario pedir una cierta cantidad de elementos de la lista y serán solo esos los computados, permitiéndose así la terminación del algoritmo y no cayendo en una evaluación infinita de la expresión.
- $getAdy$: Dada una posición y un ambiente, devuelve la lista con sus adyacentes dentro de los límites de la matriz.
- $setElement$ Función utilizada para crear las listas con las posiciones de los elementos. Para ello se reciben dos listas con valores aleatorios representando los posibles valores de x y de y que pueden haber en la matriz. Si al tomar las cabeceras y analizar si la posición (x, y) ha sido tomada por otro elemento, llamamos recursivamente con el resto de las listas, En otro

caso se agrega a la lista que se está construyendo y se llama recursivamente con un elemento de menos a colocar, los restos de las listas y se actualiza la lista de celdas ocupadas.
- $pickRandom$: Este es un método que nos permite seleccionar n elementos de una lista de forma aleatoria. Para ello se escoge un índice en el rango de la lista,de forma aleatoria se añade a la lista que estamos formando y se quita de la lista de la que se está escogiendo. Si se nos acaban los elementos de la lista o hemos tomado la cantidad pedida se retorna.
- $remove$: Función que elimina un elemento de una lista.
- $inMatriz$: Sirve para determinar si una posición está dentro de los límites de la matriz.
- $getDir$: Dados dos pares de coordenadas $(x1, y1)$ y $(x2, y2)$ nos retorna $(x1 - x2, y1 - y2)$ para así saber las coordenadas que se tienen que sumar a un conjunto de celdas y poder desplazarlas.
- $disjoin$: Esta función nos retorna todos los elementos de la primera lista que no están en la segunda.
- $filterAdy$: Nos retorna aquellos elementos de una lista que están dentro de la matriz.
- $updateElement$: Actualiza una posición de una lista con un determinado valor.

Módulo Statistics

Este módulo alberga una función $calculateDirtPercent$ que dado un ambiente nos retorna su porciento de suciedad. Para ello es necesario calcular que porciento representan las celdas sucias con respecto al total de celdas no ocupadas. Destacar aquí que si una celda está ocupada por un robot y sucia no se cuenta como sucia, pues solo se analizan aquellas no ocupadas para determinar el porciento de suciedad.

$percent$ es una función para realizar el calculo del porcentual.

$finalState$ es una función que determina si estamos en un estado final y por tanto se detiene la simulación. Los estados finales serían: que todos los niños estén en el corral y no haya suciedad o que se haya sobrepasado el 60% de suciedad.

$victOrLoss$: Esta es una función auxiliar que utilizamos para contar si dado un porciento se obtuvo una victoria (1 en el primer elemento de la tupla devuelta) o una derrota (0 en el segundo elemento de la tupla devuelta).

$mean$ nos permite calcular la media de los porcientos de limpieza de cada simulación.

Módulo Environment

En este módulo tenemos la funciones que más tienen que ver con el ambiente. Aquí tenemos $generateEnv$ que es donde se crean los ambientes, conocidos varios elementos como la cantidad de niños, las dimensiones, el tamaño del corral, los obstáculos, los robots de limpieza, la suciedad y la lista indicando que robots se encuentran cargando niños. La idea principal es crear una lista inicial con todas las celdas de la matriz (que al ser al inicio son todas libres), luego pasamos a asignarles posiciones a los elementos tomando celdas de nuestra lista de posiciones vacía. De esta forma aseguramos que siempre tomamos celdas disponibles y cuando se vayan a seleccionar posiciones para otros elementos nos aseguramos de quitar las tomadas de la lista de celdas vacías, esto se hace con la función $disjoin$ entre las celdas vacías actuales y las nuevas tomadas, obteniendo aquellas que están en las vacías y no en las dadas a los elementos bajo análisis. Esa nueva lista constituirá ahora nuestra lista de posiciones válidas a tomar.

La función $emptyCellForKid$ determina dada una posición de un niño y una lista de celdas , cual de ellas están libres para moverse. Para ello se chequean las condiciones que la hacen libre en este caso especial y se va construyendo una lista con las que cumplen dichas condiciones . Esto se usa como una especie de función filtro a la hora de determinar las posibles celdas a las que puede moverse un niño. Señalar que se recibe la posición $(p1, p2)$ donde está el niño, para poder determinar si un obstáculo se puede desplazar o no, en caso de que se este desplazando hacia una celda ocupada por uno.

El método $emptyCellForRobot$ funciona de igual forma que el anterior, solo que aquí se quieren determinar las celdas libres para un robot. Por ejemplo no se analiza si una cierta posición se encuentra sucia o no pues el robot puede acceder a ella y limpiarla o pasar por encima y continuar su camino. Destacar que se recibe un argumento $withChld$, tal que si es verdadero y la posición a la que se quiere mover el robot está ocupada por un niño, entonces no se considera libre, en caso de ser falso si se considera libre y el robot cargaría al niño.

Con $emptyCellToMess$ estamos realizando los dos análisis anteriores, pero ahora se quiere crear una lista con aquellas celdas libres que pueden ser ensuciadas, dado un conjunto de ellas. La diferencia con los otros métodos es que no existen condiciones por las que una celda pueda ser libre o no. Por ejemplo en el caso del niño sabemos que una celda con obstáculo es libre si y solo si este puede ser desplazado. En este método las condiciones de ser libre indican que no puede existir absolutamente nada en la posición a analizar.

La función $countKids$ nos sirve para determinar cuantas celdas de una lista se encuentran ocupadas por niños. Esto es necesario para contar en una cuadrícula cuantos niños hay.

Simulación

Para poder conocer como se desenvuelven los agentes en diferentes ambientes se crea una algoritmo que realice las simulaciones y muestre los resultados. Esto se lleva a cabo en el módulo App, para ello se crea la función $main$ encargada de recibir las condiciones iniciales del ambiente y realizar una determinada cantidad de simulaciones indicada por el usuario. Luego de $noSim$ se obtienen el promedio de los porcietos de suciedad de cada ambiente, contando también las veces que cumplió su objetivo y las que no.

En la función $startSimulation$ es donde se lleva a cabo toda la simulación de acciones tanto de los agentes, como de los niños y la variación aleatoria. Si nos encontramos en un esatdo final retornamos el porciento de suciedad que se tiene hasta el momento. Recordemos que los estados finales son cuando todos los niños están en el corral y no hay suciedad o cuando se sobrepasa el 60% de suciedad en el ambiente. Si se completaron $100 * t$ turnos entonces se detiene la simulación sin haber alcanzado un estado final, retornando el porciento de suciedad. Si se han completado t turnos entonces toca realizar una variación aleatoria del ambiente. En nuestro caso esto representa que serán modificados todos los elementos del ambiente. En otras palabras se cambia la posición del corral, se redistribuye la suciedad, se sacan los niños que estén en el corral y se dan nuevas posiciones a ellos y a los sueltos también, se cambian los robots de posición y los obstáculos. Lo único que se mantiene igual es que si un robot está cargando un niño, lo seguirá cargando. De esta forma recreamos las peores condiciones en a las que se puede enfrentar un agente, pues todo cambia de lugar. Luego continuamos la simulación realizando un llamado recursivo.

Si ninguna de las premisas anteriores se cumplen entonces se realizan los movimientos de los agentes y luego los cambios naturales del ambiente, que pueden ser que se muevan los niños y a su vez obstáculos y/o que aparezca suciedad. Cabe destacar que si falta solo una unidad de tiempo para que se realice la variación aleatoria entonces se hace un llamado recursivo sin aumentar la cantidad de turnos. Esto se hace con el objetivo de que en un mismo turno ocurra el cambio natural y luego la variación aleatoria. En caso de que no se este en el turno $t - 1$ se hace

un llamado aumentando la cantidad de turnos. Destacar que siempre, luego de un cambio natural o variación, se pasa el nuevo ambiente obtenido.

Resultados obtenidos

Para comprobar el desempeño de nuestros agentes se crearon 10, algunos con las mismas dimensiones, con distintos parámetros de inicialización. Luego con cada uno de los tipos de agentes se realizan 30 simulaciones con diferentes valores de t, que en los peores casos cunado $t = 18$ se pueden llegar a los 1800 turnos, con cada una de las simulaciones, pues de no encontrarse en estados finales entonces se realizan $t * 100$ turnos, esto sumado también a los algoritmos implementados, el tamaño del ambiente y la estrategia del agente resultó en que correr la función $run\ 0\ 0$ con los valores establecidos tardara cerca de las $2h$ en concluir todas laas simulaciones

Para llevar a cabo las pruebas tenemos el módulo $Tester$ y dentro una función llamada run que ejecuta las simulaciones de los 4 agentes. En esta función se encuentran las condiciones iniciales de 10 entornos y se va creando una lista con los resultados de simular el $i - esimo$ ambiente con sus condiciones para cada agente. En nuestro caso que tenemos 4 agentes, 10 entornos y 30 simulaciones, implica que se analizaron 1200 simulaciones en total, obteniéndose los siguientes resultados.

(Filas,Cols) envs	rbts	obst	niños	suciedad	t	resultados
$(6, 6)$	2	1	1	0	2	$((0.0, 30, 0), 0)$
$(6, 6)$	2	2	3	1	4	$((62.15263, 0, 30), 0)$
$(10, 14)$	3	5	4	3	5	$((48.02248, 26, 4), 0)$
$(10, 14)$	3	5	4	4	10	$((40.68061, 30, 0), 0)$
$(13, 12)$	4	5	4	5	14	$((3.8157942, 30, 0), 0)$
$(13, 12)$	4	3	2	1	7	$((0.0, 30, 0), 0)$
$(12, 12)$	2	5	5	4	18	$((60.544186, 0, 30), 0)$
$(12, 12)$	3	12	5	10	10	$((60.369278, 0, 30), 0)$
$(14, 9)$	2	3	3	2	4	$((56.01939, 16, 14), 0)$
$(14, 9)$	4	10	3	6	12	$((0.0, 30, 0), 0)$
$(6, 6)$	2	1	1	0	2	$((0.0, 30, 0), 1)$
$(6, 6)$	2	2	3	1	4	$((62.22697, 0, 30), 1)$
$(10, 14)$	3	5	4	3	5	$((57.152668, 11, 19), 1)$
$(10, 14)$	3	5	4	4	10	$((36.16413, 30, 0), 1)$
$(13, 12)$	4	5	4	5	14	$((11.163927, 30, 0), 1)$
$(13, 12)$	4	3	2	1	7	$((4.56621e - 2, 30, 0), 1)$
$(12, 12)$	2	5	5	4	18	$((60.902836, 0, 30), 1)$
$(12, 12)$	3	12	5	10	10	$((61.18766, 0, 30), 1)$
$(14, 9)$	2	3	3	2	4	$((50.84058, 22, 8), 1)$
$(14, 9)$	4	10	3	6	12	$((0.0, 30, 0), 1)$
$(6, 6)$	2	1	1	0	2	$((0.0, 30, 0), 2)$
$(6, 6)$	2	2	3	1	4	$((28.411005, 16, 14), 2)$
$(10, 14)$	3	5	4	3	5	$((3.9158506, 29, 1), 2)$
$(10, 14)$	3	5	4	4	10	$((3.8538907, 30, 0), 2)$
$(13, 12)$	4	5	4	5	14	$((0.0, 30, 0), 2)$
$(13, 12)$	4	3	2	1	7	$((0.0, 30, 0), 2)$
$(12, 12)$	2	5	5	4	18	$((57.898495, 9, 21), 2)$
$(12, 12)$	3	12	5	10	10	$((7.715743, 27, 3), 2)$
$(14, 9)$	2	3	3	2	4	$((13.504672, 29, 1), 2)$
$(14, 9)$	4	10	3	6	12	$((0.0, 30, 0), 2)$
$(6, 6)$	2	1	1	0	2	$((0.0, 30, 0), 3)$
$(6, 6)$	2	2	3	1	4	$((59.53059, 1, 29), 3)$

(Filas,Cols) envs	rbts	obst	niños	suciedad	t	resultados
$(10, 14)$	3	5	4	3	5	$((39.177876, 24, 6), 3)$
$(10, 14)$	3	5	4	4	10	$((30.496601, 30, 0), 3)$
$(13, 12)$	4	5	4	5	14	$((2.331157, 30, 0), 3)$
$(13, 12)$	4	3	2	1	7	$((0.0, 30, 0), 3)$
$(12, 12)$	2	5	5	4	18	$((58.86897, 5, 25), 3)$
$(12, 12)$	3	12	5	10	10	$((42.47382, 23, 7), 3)$
$(14, 9)$	2	3	3	2	4	$((44.606594, 28, 2), 3)$
$(14, 9)$	4	10	3	6	12	$((0.0, 30, 0), 3)$

Los primeros 10 resultados de esta tabla corresponden al agente de tipo 0, el cual tiene como prioridad la de buscar niños antes que buscar suciedad. Como se puede apreciar en los resultados obtenidos este ambiente no se desempeña tan bien cuando el valor de t no es muy grande y existen varios niños en el ambiente. Por ejemplo en el caso del ambiente $(14, 9)$ con un $t = 12$, 6 celdas sucias iniciales, 3 niños y 10 obstáculos se obtiene la mejor efectividad posible, siempre se cumple el objetivo y se tiene un 0% de suciedad, pero disminuir el valor de t 8 unidades, 2 robots y las cantidades de obstáculos y de celdas sucias iniciales, nos arroja resultados poco alentadores, concluyendo con un 56% de suciedad. Esto se debe a que dicho tipo de agente no reacciona bien a los ambientes que están en constante cambio, pues siempre prioriza buscar un niño antes que suciedad y estos además de que se pueden mover naturalmente también lo hacen por la variación del ambiente, lo que no ocurre con la suciedad, que no cambia naturalmente, solo se crea por los niños o se redistribuye en caso de variación. Debemos destacar que el promedio de los porcientos de suciedad alcanzados en cada ambiente por este agente es de $33, 16043682\%$, con 192 ocasiones en que cumplió con el objetivo.

El próximo caso es el del agente que prioriza buscar suciedad antes que niños. Como se aprecia sus resultados no son muy disparejos del los del agente de tipo 0 . Teniendo incluso un menor número de veces que cumplió el objetivo, 183. Como se observa tampoco reacciona muy bien a los ambiente que varían mucho y aunque en ocasiones logra mantener el promedio del $\%$ de suciedad por debajo de 60, en varias ocasiones es con valores muy elevados. El promedio de los porcientos de suciedad alcanzados en cada ambiente por este agente es de $33, 96844331\%$ algo superior al anterior

Destacar de los agentes anteriores que se esperaba un mejor comportamiento cuando el valor de t es mayor, pero sin embargo observamos que con los ambientes de $12x12$ el porciento de limpieza se mantienen sobre el 60% y nunca logran cumplir el objetivo. Esto se debe en parte a la dimensión del ambiente, la poca cantidad de robots y que existen más niños y por tanto generan mayor suciedad. También tener en cuenta que toma un tiempo alcanzar un niño y una vez con este cargado el ambiente puede sufrir los cambios imprevistos, desubicando los robots.

Sin lugar a dudas el siguiente agente es el que mejor desempeño tuvo. Cumpliendo su objetivo 260 veces y solo fallando 40. No obstante como promedio obtenido en cada una de las 30 simulaciones de los ambientes, nunca sobrepasa el 60% de suciedad. Su peor desempeño fue en el ambiente de $12x12$ con 5 niños y $t = 18$, donde se obtiene un promedio del 57.898495%. Pero sabemos del trabajo que pasan los agentes con este ambiente por la cantidad de niños y la generación de basura de estos al moverse. Destacar que este agente a diferencia de los otros dos si carga un niño no prioriza llevarlo al corral antes que buscar más suciedad, por tanto cargará un

niño y limpiará con este encima, por lo que se pudiera esperar que con más robots se tenga un mejor desempeño. El promedio de los porcientos de suciedad alcanzados en cada ambiente por este agente es de $11,52996563\%$ muy inferior al de los anteriores.

Finalmente tenemos el caso del agente que según el nivel de suciedad se comporta como el tipo 2 (mayor de 30%) o como el de tipo 0. Esto fue programado con la idea de que fuera balanceado y que basara su comportamiento según convenga más. Como era de esperar se obtuvo un balance entre los dos agentes que puede interpretar, con un 204 ocasiones en que cumple su objetivo se encuentra en un punto medio entre las que cumple el agente de tipo 0 (192) y las del tipo 2 (260). Algo más cercano al comportamiento del agente de tipo 0, por lo que la suciedad puede que haya permanecido por debajo de 30% en más turnos. Respecto al peor entorno recreado, mantiene su promedio de suciedad ligeramente por encima del agente de tipo 2, pero no muy distante. El promedio de los porcientos de suciedad alcanzados en cada ambiente por este agente es de $27,7485608\%$, por lo cual se desempeña mejor que los agentes de tipo 0 y 1, pero no mejor que el agente de tipo 2. También aclarar que se comportó más como el de tipo 2 cuando la cantidad de niños era mayor, pues estos generan mayor suciedad.

Por tanto podemos concluir que el agente de tipo 2 fue el que mejor se enfrentó a los entornos pasados y que por tanto tienen una mejor jerarquía de comportamientos. Como tal intenta cumplir el objetivo de limpiar lo más posible, siendo reactivo a los cambios de entornos que varían mucho y que por tanto es preferible limpiar más y desarrolla un comportamiento más proactivo, dedicado a eliminar la fuente de suciedad cuando carga a los niños y limpia con estos encima, impidiendo así que ensucien.