Proyecto de Simulación: Agentes

Gilberto González Rodríguez C-411
g.gonzalez@estudiantes.matcom.uh.cu
Ciencias de la Computación
Universidad de La Habana
Cuba

24 de noviembre de 2020

1. Principales ideas seguidas para la implementación

Siguiendo como guía principal el texto Temas de Simulación [1] la implementación se llevó a cabo en el lenguaje python. El hilo de pensamiento para dar solución al problema fue el siguiente: modelar los elementos de entorno, realizar lo más genérico y aleatoriamente posible un conjunto de métodos para la creación de los ambientes iniciales así como la variación aleatoria y por último añadir los comportamientos a los niños y los agentes. Para el paso de creación de los ambientes iniciales, se ubican primero en el ambiente las celdas del corral, luego el robot, después los obstáculos y por último en las celdas disponibles y alcanzables desde la posición del robot se ubica la suciedad inicial y los niños. Los detalles de la implementación se abordan más adelante.

2. Modelos de Agentes considerados

Siguiendo la definición de agente inteligente explicada en [1] la idea para implementar agente(s) fue la siguiente. En el proyecto se tiene en cuenta un prototipo de agente inteligente cuyo concepto se encapsula en la clase abstracta **MySmartAgent** (detalles en la próxima sección). Puesto que hay

un solo agente en cada simulación los esfuerzos de lograr la autonomía en el agente se ven enfocados en alcanzar un comportamiento híbrido entre proactividad y reactividad.

No obstante en las implementaciones finales de un **MySmartAgent** se tienen dos modelos que serán explicados en detalles en un momento, **ProactiveAgent** y **ReactiveAgent** los cuales según su nombre indica tienen un predominio de practividad o reactividad.

Ambos modelos son muy similares en implementación, sin embargo la característica que los diferencia y tal vez la más importante es la selección del siguiente objetivo a cumplir.

ProactiveAgent al ser más goal-direct prioriza siempre encontrar el niño más cercano a su posición actual para llevarlo a una celda del corral. Aquí cabe destacar que en cada turno el agente comprueba cuál es el niño más cercano, por lo que es una característica reactiva, sin embargo, este agente una vez fijado el niño más cercano a su posición actual ignora la limpieza de sucidad incluso estando en una celda sucia, su enfoque es avanzar hacia la posición del niño si no está cargando a ninguno o a la celda vacía del corral más cercana a su posición. Con respecto a lo anterior hay una excepción y una consecuencia negativa: la excepción a solo buscar un niño o una celda del corral ocurre cuando se da el caso de que en el camino seleccionado para la conclusión exitosa del objetivo hay un objeto no obstáculo pero que bloquea dicho camino. Este caso se da cuando el robot tiene a un niño cargado y en la siguiente celda del camino (encontrado con una estrategia BFS) hacia la conclusión del objetivo hay otro niño o suciedad (para estos casos el robot con prioridad va a mover al niño que está bloqueando su paso o a limpiar la suciedad en cuestión). La consecuencia negativa con este comportamiento del agente es que al llevar a los niños a la celda del corral más cercan, pudiera darse el caso que 4 o más casillas del corral se encuentran en una de las esquinas del entorno o bloqueadas por obtáculos de tal forma que el único camino hacia alguna de las casillas interiores del corral sea precisamente a través de otra casilla del corral; pero supongamos que hay 4 niños, el corral es una matriz de $2x2$ ubicada en la esquina superior izquierda del ambiente, entonces los primeros 3 niños serán llevados a las celdas menos esquinadas (porque están más cerca de cualquier posición posible del robot) y la celda $(0, 0)$ quedaría inaccesible para que el agente llegue con un niño cargado (aquí se está asumiendo que un robot con un niño cargado no puede pasar por una celda de corral que tenga a otro niño). La solución a este problema se explica en la siguiente sección sobre la implementación de la solución.

ReactiveAgent, como se ha mencionado con anterioridad esta implementación de agente es muy similiar en varios puntos al **ProactiveAgent**, sin embargo, mientras que el **ProactiveAgent** siempre intenta tener como siguiente objetivo el de llevar a un niño al corral, este agente casi siempre va hacia la dirección del objetivo más cercano, sea una celda con sucidad, una celda con un niño o una celda de corral en caso que tenga a un niño cargado. ¿Qué implicaciones tiene esto? Dada la variación natural del ambiente (cuando los niños se mueven y ensucian) así como el porcentaje de suciedad inicial, hace que esta implementación reactiva sea un tanto ineficiente. Por ejemplo, si el robot logra capturar a un niño la siguiente posición objetivo sería la casilla vacía de corral más cercana, no obstante por el volumen de suciedad que habría en el ambiente es muy probable que incluso antes de dar los primeros pasos a la celda del corral o durante el camino hacia la misma, el robot cambie de objetivo, dirigiéndose a una celda con suciedad.

Para ambos modelos en las siguientes secciones hablaremos más de sus etapas durante las implementación y tal vez de algunas mejoras (sobre todo para el más reactivo).

3. Ideas seguidas para la implementación

Para la implementación se tuvieron en cuenta varios módulos y el diseño de clases tales como:

ElementEnvironment que representa de forma genérica un elemento del entorno es una clase abstracta.

Agent que representa a un agente genérico, es una representación abstracta.

Void que representa una celda vacía del ambiente.

Obstacle que representa un obstaculo en el ambiente.

Dirt que representa una celda con suciedad.

Playpen que representa una de las celdas del corral.

Child que representa la entidad niño.

Las clases anteriores se encuentran definidas en el módulo **entities.py**, en dicho módulo además hay una clase estática **Directions** la cual contiene métodos con los nombres de 8 direcciones: norte, noreste, este, sureste, sur, suroeste, oeste y noroeste; estos métodos contienen el valor a decrementar, a aumentar o cero para cada uno de los ejes para lograr un movimiento correcto en la dirección deseada.

En **environment.py** se encuentran métodos relacionados con el entorno, entre los más destacables están **find paths** y **build path** , el primero a partir de una posición (normalmente la del robot) con una estrategia BFS retorna un array **pi** y la estructura matricial **visit**, **pi** es usado por **build path** para encontrar el camino si existe entre dos posiciones y **visit** contiene la distancia mínima con la que se puede visitar alguna posición si es que es posible. Los obstáculos del BFS son ajustables, aunque por lo general se consideran como obstáculos los elementos del ambiente **Obstacle** y **Playpen** si además hay un **Child** en la celda.

Una celda como tal del ambiente es una tupla de 3 elementos. Esto fue escogido así dado que la cantidad máxima de elementos que podrían convivir en una casilla según las consideraciones e interpretaciones realizadas del problema sería un robot que pasa por una celda del corral con un niño cargado. Este caso de una casilla con los 3 elementos anteriores es: (R, P, C). donde R es el robot, P indica que la posición es una celda del corral y C es el niño, estos elementos si hay alguno siempre toman la misma posición en una celda del corral. Para mantener más seguridad con el tipado y el control sobre la estructura ambiente todas la celdas se consideran que son una tupla de 3 **ElementEnvironment** con ciertos convenios (como el anterior con respecto al corral). El ambiente es una lista de listas simulando una matriz, a su vez cada elemento de las listas más internas es una casilla del ambiente o sea una tupla de 3 elementos. Para los casos en los que hay 2 elementos y sin contar el caso del corral, estos serían entonces: robot-suciedad y robot-niño, el robot siempre se ubica a la izquierda. Cada niño tiene una propiedad **num** usada para su representación en texto y mejor análisis del ambiente y simulación, principalmente en etapas de depuración de errores y pruebas. Todos los elementos de entorno tienen redefinida su representación en texto con una letra que los identifica y con tamaño estándar 3 para que sea más fácil y amigable su display en consola o en un fichero (esto fue indispensable en etapas de testing). Un ejemplo de un ambiente de prueba de $3x3$ se puede observar a continuación:

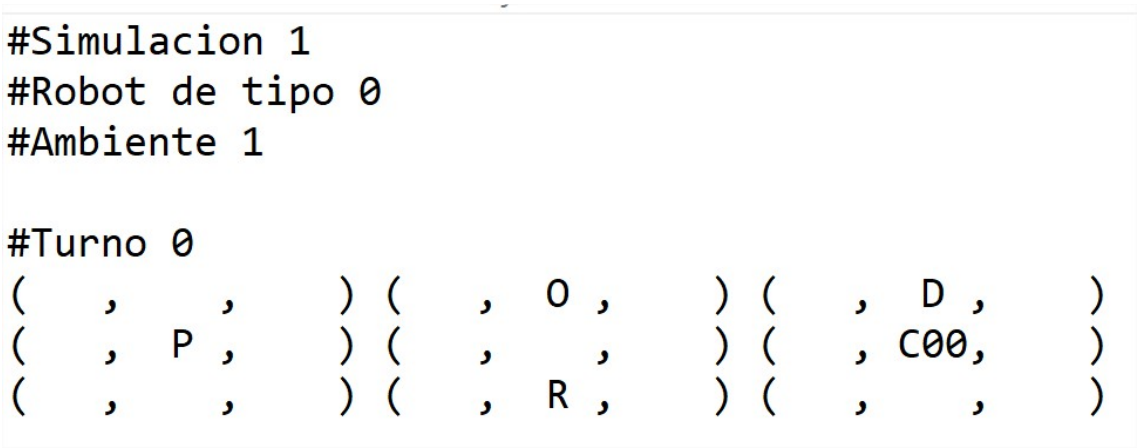

Figura 1. Ejemplo de Ambiente

Donde P es una celda del corral, O un obstáculo, R el robot de casa y C00 es el niño identificado por el índice 0, este valor es incremental.

Otros ejemplos de diferentes etapas y ambientes:

Ejemplo de ambiente que pudo ser limpiado

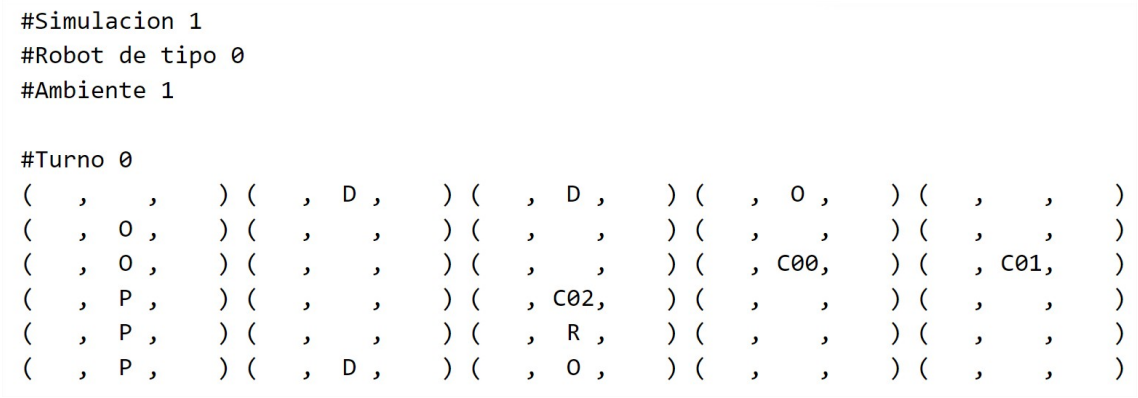

Figura 2. Ejemplo de un ambiente inicial para el cual el robot pudo limpiar la casa

```
#Turno 19
(   ,    ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  O ,    ) (   ,    ,    )
(   ,  O ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    )
(   ,  O ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    )
(   ,  P , C02) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    )
(   ,  P , C01) (   ,    ,    ) ( R , C00,    ) (   ,    ,    ) (   ,    ,    )
(   ,  P ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    ) (   ,    ,    )

#Turno 20 (de variación aleatoria)
(   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    ) (   ,  D ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,    ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    )
(   ,  P , C02) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    )
(   ,  P , C01) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    )
(   ,  P ,    ) ( R , C00,    ) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    )
```

Figura 3. Ambiente anterior en su primer cambio aleatorio con $t = 20$

```
#Turno 37
(   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    ) (   ,  R ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    )
(   ,  P , C02) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    )
(   ,  P , C01) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    )
(   ,  P , C00) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    )

La simulación terminó porque el robot logró poner a los niños en el corral y limpiar la casa
```

Figura 4. Estado final del ambiente que pudo ser limpiado

Ejemplo de ambiente que no pudo ser limpiado

```
#Simulacion 4
#Robot de tipo 0
#Ambiente 4

#Turno 0
(   ,  O ,    ) (   ,    ,    ) (   ,    ,    ) (   , C03,    ) (   ,  O ,    ) (   ,    ,    ) (   ,    ,    )
(   ,    ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  D ,    )
(   , C02,    ) (   ,  P ,    ) (   ,  P ,    ) (   ,    ,    ) (   ,    ,    ) (   , C01,    ) (   ,    ,    )
(   ,  P ,    ) (   ,  P ,    ) (   ,  O ,    ) (   ,  P ,    ) (   ,  O ,    ) (   ,    ,    ) (   ,    ,    )
(   , C00,    ) (   ,  P ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    )
(   , C06,    ) (   ,  P ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,  D ,    ) (   , C04,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  R ,    ) (   ,    ,    ) (   ,    ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  D ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   , C05,    ) (   ,    ,    )
(   ,    ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,    ,    ) (   ,    ,    )
```

Figura 5. Ejemplo de un ambiente inicial para el cual el robot no pudo limpiar la casa

```
#Turno 21
(   ,  D ,    ) (   ,  O ,    ) (   , C00,    ) (   ,  D ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,    ,    )
(   ,  D ,    ) (   , C01,    ) (   , C05,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    )
(   ,  D ,    ) (   ,  P , C02) (   ,  P , C03) (   ,    ,    ) (   ,    ,    ) (   ,  O ,    ) (   ,  O ,    )
(   ,  P ,    ) (   ,  P ,    ) (   ,  D ,    ) (   ,  P , C04) (   ,    ,    ) (   ,  D ,    ) (   ,  D ,    )
(   ,    ,    ) (   ,  P ,    ) (   ,  R ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,  O ,    )
(   ,  O ,    ) (   ,  P ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    )
(   , C06,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,    ,    )
(   ,  D ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,    ,    )
(   ,  O ,    ) (   ,  D ,    ) (   ,  O ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  D ,    )
(   ,    ,    ) (   ,    ,    ) (   ,    ,    ) (   ,  D ,    ) (   ,  D ,    ) (   ,    ,    ) (   ,  D ,    )

La simulación terminó porque la casa estaba sucia. El robot fue despedido
```

Figura 6. Estado final del ambiente que no pudo ser limpiado

Obstacle tiene un método llamado push, el cual recibe una dirección, siempre que se pueda se empuja al obstáculo en dicha dirección, si hay más obstáculos consecutivamente en la misma dirección se sigue llamando a push desde los nuevos obstáculos hasta que no se pueda empujar más o hasta que se encuentre una celda vacía, en este último caso todos los obstáculos se mueven si se desplazó la posición que mandaron a empujar. Un **Child** es el encargado de empujar al primer obstáculo, **Child** tiene un método llamado react, en este método se buscan las casillas adyacentes al niño que estén dentro del ambiente, y aleatoriamente se busca un número entre 0 y la cantidad de casillas dentro del ambiente (incluyendo el extremo), si el valor

es el **length** de las direcciones adyacentes entonces el niño no se mueve, si está en rango y está vacía la casilla adyacente el niño se mueve, si hay un obstáculo llama a push y si se hizo un espacio entonces se mueve, en otro caso el niño permanece en la misma posición.

Las cuadrículas de $3x3$ en las cuales aparece suciedad si hay niños sueltos se escogen de izquierda a derecha y de arriba hacia abajo, sin que hayan dos columnas que se superpongan. En un principio se partía desde la posición de cada niño y se buscaban todas las casillas de $3x3$ que entraran en rango y que contuvieran a la casilla del niño aunque hubiesen varias posiciones que se repitieran. Esto provocaba que la cantidad de suciedad que aparecía era demasiada y el robot incluso con un solo niño fallaba en la tarea en exceso.

La creación del ambiente inicial (explicada superficialmente con anterioridad) y el cambio aleatorio del ambiente tienen bastante código en común. En la creación del ambiente inicial se seleccionan las dimensiones del ambiente de forma aleatoria entre 5 y 10 (inclusivos) tanto para filas como para las columnas. La cantidad de suciedad inicial y los obstáculos se calculan como la parte entera por debajo de un valor aleatorio entre un 10 y 20 % del total de celdas, la cantidad de niños es igualmente un valor entre un 10 y 20 % pero del total de celdas que quedarían disponibles sin contar la suciedad y los obstáculos recién generados. Primero se ubican las celdas del corral, el robot y después los obstáculos garantizando que el robot llegue a todas las celdas del corral y que quedan suficientes celdas alcanzables desde la posición del robot para ubicar toda la suciedad inicial y los niños, si estas condiciones no se cumplen entonces se vuelve a generar desde cero otra configuración del ambiente, lo que siempre se va a generar un ambiente inicial factible. Con respecto a la variación aleatoria se tienen en cuenta las siguientes consideraciones, solo se redistribuyen de forma aleatoria las casillas con suciedad, con un niño y los obstáculos, en los primeros casos esto se hace siempre y cuando el robot no esté con el elemento en la misma casilla y el niño no esté en una casilla del corral. Teniendo en cuenta casos extremos y para garantizar la factibilidad del ambiente luego de la variación, se garantiza que el robot siempre llegue a todas las casillas del corral, a toda la suciedad y a todos los niños, mientras no se cumplan estas condiciones se intenta variar hasta 100 iteraciones, en casos extremos en los que tal vez el ambiente no es factible (porque algún niño moviendo obstáculos incomunicó alguna parte del ambiente con el robot) y tal vez hay pocas celdas disponibles con una configuración compleja de cantidad de obstáculos y suciedad, si tras 100 iteraciones no se ha obtenido una variación factible, entonces el ambiente se

deja sin variar (este caso no se encontró naturalmente durante la etapa de testing).

En la implementación de los agentes en el módulo **agents.py** interviene un elemento muy importante, los objetivos. El concepto encapsulado en la clase **Objective** tiene las siguientes características. Un Objective tiene 3 partes fundamentales un **find**, un **perfom** y una función **check if completed**, ambos 3 son funciones y reciben al propio objetivo, al ambiente, al robot y un diccionario llamado **env info** que contiene información del ambiente, como la cantidad de celdas sucias, la cantidad de celdas vacías o una posición que constituye un bloqueo para el agente. En la clase hay 5 métodos estáticos con la definición de los 5 objetivos usados por las 2 implementaciones específicas de **MySmartAgent**, estos son: **build dirty alert objective**, **build clean objective**, **build bring children to playpen objective**, **build clear block objective**, **build move in playpen objective**. Los **MySmartAgent** tienen una lista de objectivos, en este caso estos 5. **build dirty alert objective**, **build clear block objective** y **build move in playpen objective** son objetivos con prioridad por lo general cuando están activos los agentes los completan ignorando los otros 2 objetivos. Sin embargo **build clean objective** y **build bring children to playpen objective** son los objetivos para terminar la simulación en un estado exitoso, los otros son prioritarios puesto que en las condiciones donde se disparan el cumplimiento de estos ayuda con el de los otros dos. Cuando un agente, sea **ProactiveAgent** o **ReactiveAgent**, está en su primer turno o no tiene un objetivo activo (esto es una propiedad de los objective) entonces, en el caso de **ProactiveAgent** se busca el niño más cercano o si ya carga uno la celda del corral más cercana, en el caso de **ReactiveAgent** siempre se busca la celda objetivo más cercana y se cambia de objetivo en caso de que sea necesario. Los otros 3 objetivos se disparan en los siguientes escenarios: cuando un robot carga a un niño y en la siguiente celda en su camino se encuentra otro niño o una suciedad, en ese momento se dispara **clear block**, cuando un robot carga a un niño y la siguiente celda es una celda de corral con otro niño entonces se dispara **move in playpen** y al inicio del turno del robot si la cantidad de suciedad está a un 55 % o superior entonces se dispara **dirty alert** hasta que la suciedad esté a un 40 %. El método principal de los agentes es **perform action** que en dependencia de si tiene activo o no un objetivo activo (si es prioritario o no) y el modelo del agente entonces o se mueve en pos de cumpliar el objetivo actual o busca el siguiente. Disparar un objetivo no es más que en algún punto de la lógica del robot cambiar el objtivo activo

por el que toca lanzar en este momento y llamar a **perform action**, esto para el caso de **clear block** y **move in playpen** que se lanzan dentro del método **move** del agente cuando este no puede realizar el movimiento, en el caso de **dirty alert** esta se chequea y se dispara al comienzo del propio método **perform action**.

4. Consideraciones obtenidas a partir de la ejecución de las simulaciones del problema

Se realizaron simulaciones según la orden, al menos 10 ambientes iniciales y 30 iteraciones para cada uno. Ahora varias consideraciones aquí, esto se realizó varias veces para diferentes valores de t, se realizaron simulaciones para los valores de tiempos 5, 10, 15, 20, 30, 40 y 50. La simulación termina por tiempo en el turno $100t$ por lo que para $t = 10$ terminaría (si no se alcanza algún estado final antes) en el turno 1000 (iteración desde 1 hasta $100 * t$, simulando un tiempo lógico) y $t = 50$ en el turno 5000.

El entry point del programa es **main.py** donde está el método para crear los ambientes iniciales, un método para correr las simulaciones (llamando mientras no se esté en un estado final y no se deba acabar por tiempo) al performe action del robot, luego el turno del ambiente y la variación aleatoria (si es que toca) y mucha logica para imprimir y procesar los datos de las simulaciones. Se puede ejecutar de la siguiente forma:

python main.py [-t <time>] [-p True|False] [-i 30] [-s <seed>]

Donde $-t$ sería el tiempo para cada variación aleatoria, $-p$ sería para imprimir tanto en consola como a un fichero llamado **sim logs.txt** los turnos de todos los tableros en todas las iteraciones, este hay que usarlo con precaución puesto que para una sola iteración, con 10 ambientes iniciales, por cada uno de los 2 agentes y con tableros de $5x5$ el fichero llegaba a pesar más de 100 MB, para 30 iteraciones que sería el valor por defecto sería overkill, sin embargo esta característica resultó clave para el desarrollo del proyecto; $-i$ sería la cantidad de iteraciones por cada ambiente inicial y $-s$ para pasarle una semilla al random (los datos finales mostrados aquí se obtienen usando como semilla para el random el string **seed**). Solo se generan 10 ambientes iniciales, el agente se cambia en cada caso, por lo que cada agente se prueba en los mismos ambientes, igual pasa con los tiempos se varía el valor de t pero

sobre los mismo 10 ambientes iniciales. Para otras semillas y configuraciones los resultados son muy similares.

```
Para el tiempo t=5
ProactiveAgent
Fue despedido: 210 veces
Completó la tarea: 90 veces
Se detuvo por tiempo: 0 veces

ReactiveAgent
Fue despedido: 270 veces
Completó la tarea: 30 veces
Se detuvo por tiempo: 0 veces
```

```
Para el tiempo t=15
ProactiveAgent
Fue despedido: 180 veces
Completó la tarea: 120 veces
Se detuvo por tiempo: 0 veces

ReactiveAgent
Fue despedido: 210 veces
Completó la tarea: 90 veces
Se detuvo por tiempo: 0 veces
```

```
Para el tiempo t=20
ProactiveAgent
Fue despedido: 179 veces
Completó la tarea: 120 veces
Se detuvo por tiempo: 1 veces

ReactiveAgent
Fue despedido: 270 veces
Completó la tarea: 30 veces
Se detuvo por tiempo: 0 veces
```

```
Para el tiempo t=30
ProactiveAgent
Fue despedido: 150 veces
Completó la tarea: 150 veces
Se detuvo por tiempo: 0 veces

ReactiveAgent
Fue despedido: 240 veces
Completó la tarea: 60 veces
Se detuvo por tiempo: 0 veces
```

```
Para el tiempo t=40
ProactiveAgent
Fue despedido: 180 veces
Completó la tarea: 120 veces
Se detuvo por tiempo: 0 veces

ReactiveAgent
Fue despedido: 150 veces
Completó la tarea: 150 veces
Se detuvo por tiempo: 0 veces
```

```
Para el tiempo t=50
ProactiveAgent
Fue despedido: 120 veces
Completó la tarea: 180 veces
Se detuvo por tiempo: 0 veces

ReactiveAgent
Fue despedido: 210 veces
Completó la tarea: 90 veces
Se detuvo por tiempo: 0 veces
```

Como se puede observar en los resultados **ProactiveAgent** resultó ser superior que **ReactiveAgent** para la mayoría de los casos, excepto para $t = 40$ en el que **ReactiveAgent** tuvo un desempeño de 150 tareas completadas y 150 despidos, uno de los mejores, siendo solo superado por 180 tareas cumplidas y 120 despidos obtenidos por **ProactiveAgent** para $t = 50$. Solo se detuvo la simulación por tiempo una vez para $t = 20$ con **ProactiveAgent**. De forma general se puede concluir que esta implementación de modelo

más proactivo de agente resultó ser la más eficiente de las 2 propuestas, en general hubo más despidos que objetivos cumplidos, esto además de deberse a las implementaciones específicas de cada agente pudiera ser debido a las consideraciones e interpretaciones del marco general, así como la selección de los procentajes para las cantidades iniciales y las dimensiones de los ambientes. El principal problema del bajo desempeño de **ReactiveAgent** está dado como se explicó antes a que la casua de los despidos es la acumulación de nueva suciedad que aparece por acción de los niños y esta implementación específica hace que le sea muy difícil al agente reactivo completar la tarea de llevar a un niño a una celda del corral. Una mejoría para esta implementación pudiera ser no cambiar de objetivo una vez se tiene cargado a un niño, sería un poco más proactivo en este aspecto, sin embargo seguiría siendo un modelo al menos más reactivo que su contraparte ya que este seguiría optando por ir a la posición más cercana si no tiene ningún niño, mientras que **ProactiveAgent** sí toma la iniciativa de buscar primero a los niños que son la causa de la nueva suciedad, igual como **ProactiveAgent** seguría ignorando la suciedad su desempeño pudiera ser peor para algunas configuraciones que la nueva propuesta de **ReactiveAgent**

5. Bibliografía

- Temas de Simulación de Luciano García, Luis Pérez, Luis Martí [1]