

# Proyecto de Agentes Simulación

Dalianys Pérez Perera  
C-411

# Índice

<b>1. Principales Ideas para la solución</b>	<b>3</b>
<b>2. Modelos de Agentes</b>	<b>4</b>
2.1. Agente ProtectRobot . . . . .	5
2.2. Agente CleanerRobot . . . . .	6
<b>3. Ideas seguidas para la implementación</b>	<b>7</b>
3.1. Representación del ambiente . . . . .	7
3.2. Creación del ambiente inicial . . . . .	8
3.3. Variación aleatoria . . . . .	8
3.4. Selección de la dirección a moverse . . . . .	8
3.5. Simulador . . . . .	9
<b>4. Simulaciones</b>	<b>10</b>
<b>5. Ejecución</b>	<b>10</b>
<b>6. Conclusiones</b>	<b>10</b>

# 1. Principales Ideas para la solución

Para la simulación del problema planteado fue necesario representar cada una de sus componentes, tratando de que esta modelación fuese lo más cercano posible a la realidad del mismo. Por tanto, se definieron tres módulos principales: **Environment**, **Agent** y **Simulator**.

Respondiendo a las especificaciones del proyecto, se satisface que el ambiente sea discreto, de información completa y dinámico pues está sujeto a los cambios realizados por los agentes además de la variación aleatoria que ocurre cada  $t$  unidades de tiempo. También la propiedad de accesibilidad del ambiente se cumple sin la necesidad de que los agentes contengan como parte de su definición a un ambiente y tampoco este último tenga a los agentes internamente. Por lo que ambos conceptos son totalmente independientes, es entonces el simulador el encargado de relacionarlos.

Tanto el robot de casa como los niños constituyen agentes mostrando su capacidad ejecutiva al poder modificar el medio en que habitan. Cada uno de ellos se especializa con su conjunto de acciones particular, quedando conformada la siguiente jerarquía:

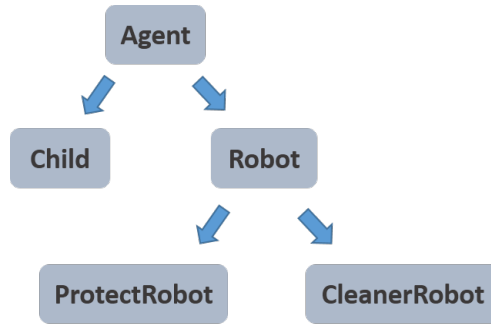


Figura 1: Jerarquía de clases de agentes

Se implementaron dos tipos de robot de casa: **ProtectRobot** el cual brinda más prioridad a guardar los niños en el corral y el robot **CleanerRobot** quien mantiene la casa lo más limpia posible y solo lleva a un niño al corral en caso de encontrarse con él. No obstante, el agente **Robot** de por sí, constituye un robot de casa que actúa aleatoriamente, seleccionando una acción a ejecutar entre todas las posibles. Como este último no se basa en una estrategia o política a seguir, los resultados del mismo no aportarán ningún tipo de información sobre su efectividad como robot de casa. Por tal motivo, solo se tendrán en cuenta los experimentos realizados con el *ProtectRobot* y el *CleanerRobot*.

La simulación parte de una configuración inicial del ambiente:

- $i$ : cantidad de iteraciones de la simulación.
- $t$ : intervalo de las variaciones aleatorias.
- $N$ : cantidad de filas del ambiente.
- $M$ : cantidad de columnas del ambiente.

- `dirty_porcent`: porciento de casillas sucias.
- `obst_porcent`: porciento de obstáculos a colocar.
- `num_childs` : cantidad de niños en el ambiente.
- `bot_type`: tipo de robot de casa(`Robot`, `ProtectRobot`, `CleanerRobot`).

Con esta información se genera un ambiente a través de la función `restart_map` la cual además de los datos anteriores recibe al robot ya con una posición aleatoria. En un inicio solo está ubicado el robot, luego se pasa a ubicar el corral garantizando siempre que sus casillas estén dispuestas consecutivamente, posteriormente se van seleccionando las casillas sucias, los obstáculos y los niños, descartando en cada paso, las casillas seleccionadas en la repartición anterior. Se asegura además que los niños no caigan dentro del corral inicialmente y que los robot no comiencen cargando a un niño.

## 2. Modelos de Agentes

Al estar en presencia de un ambiente dinámico, se decidió que los agentes fueran reactivos, ya que los eventos que ocurren en el ambiente pueden afectar los objetivos del agente o las suposiciones en las que se basa el proceso que el agente está ejecutando para lograr su objetivo. Por tanto el agente debe ser sensitivo a estos cambios. Por otro lado, ambos robots de casa en dependencia de su estado y el del ambiente pueden determinar cumplir un objetivo( encontrar un niño, llevar un niño al corral,...), es en este punto donde se manifiestan pequeños rasgos de pro-actividad.

Los niños son agentes de tipo *Child* y la única acción que realizan es moverse aleatoriamente a una de las direcciones posibles durante el turno del ambiente, una casilla a lo sumo, pues estos pueden decidir empujar a un obstáculo y que la acción no tenga efecto. Cada actuar de un niño trae consigo la generación de basura en la cuadrícula del ambiente donde está contenido dicho niño.

Los robots constituyen agentes con estados, pues su proceso de toma de decisión está basado en la percepción que necesitan captar del ambiente de acuerdo al estado interno actual del robot. El comportamiento de un robot es construido a partir de un número de conductas que él mismo puede asumir en dependencia de su estado. Los posibles estados de un robot son:

- **CLEAN**: el robot se encuentra limpiando y siempre se mueve por el ambiente hacia la casilla sucia más cercana.
- **SAVE**: el robot se encuentra cargando un niño y tiene el objetivo de llevarlo hacia el corral.
- **FIND**: el robot se encuentra de camino al niño más cercano que no esté en un corral.

Tanto *ProtectRobot* como *CleanerRobot* tienden a priorizar más un objetivo por encima de otro, por tanto tendrán lugar conductas con mayor prioridad que otras. Importante añadir que todos los agentes tienen las siguientes funcionalidades:

- `select_direction`: selecciona la dirección a donde se moverá posteriormente para alcanzar su objetivo actual.
- `move`: se mueve en dependencia de la dirección dada. En caso de ser un robot y está cargando un niño puede moverse hasta dos casillas.
- `do_action`: actualiza el estado del robot según la percepción que captó del ambiente y del estado en que se encontraba el propio robot. Luego con esta información es que decide cuál acción ejecutar. En el caso de los niños, no hay estados y la única acción es moverse.

En el caso de los robots, además de la acción de moverse tienen otras dos acciones posibles:

- `drop_child`: suelta el niño que lleva cargando en la casilla donde está parado.
- `clean_cell`: limpia la casilla donde está situado.

En las siguientes secciones se ejemplifica lo anterior con los modelos de agentes implementados.

## 2.1. Agente ProtectRobot

Como ya se mencionó, este robot prioriza llevar todos los niños al corral desde un inicio y cuando lo logre es que se mantiene limpiando la casa. Por defecto el estado de este robot es `FIND`, pues cuando comienza la simulación su primer objetivo es encontrar al niño más cercano. En el siguiente esquema se muestran cada una de las conductas de este tipo de agente.

```
def do_action(self, env):
    """
    env: environment
    """
    if env.all_childs_in_guard(): #ya guardó todos los niños
        self.state = Robot.CLEAN
    elif self.has_child(): #si tiene un niño se mueve hacia el corral
        self.state = Robot.SAVE
    else: #si no ha terminado de guardar los niños se mantiene buscando otro
        self.state = Robot.FIND
    bot_cell = env.get_position(self.position)
    posible_action = []

    if self.state == Robot.SAVE and bot_cell.is_guard(): #conducta 1
        return self.drop_child(env)
    if (not self.state == Robot.SAVE) and bot_cell.is_dirty(): #conducta 2
        posible_action.append(self.clean_cell)
    if not (self.state == Robot.CLEAN and bot_cell.is_dirty()): #conducta 3
        posible_action.append(self.move)
    action = rnd.choice(posible_action)
    return action(env)
```

**Conducta 1:** el robot se encuentra en el estado SAVE(cargando a un niño) y parado sobre un corral por tanto la única posible acción a ejecutar es `drop_child`.

**Conducta 2:** el robot no está cargando niño y está sobre una casilla sucia, entonces la acción `clean_cell` será una de las candidatas. Notar que si no está en estado SAVE, puede estar en estado CLEAN en el cual solo se limpian casillas o en estado FIND. En caso de estar en este último la acción a ejecutar puede ser moverse para acercarse al niño objetivo o limpiar la casilla actual. Ambas tendrían 50 % de probabilidades.

**Conducta 3:** el robot no está en estado CLEAN ni está parado sobre una casilla sucia, por tanto lo que hace es moverse. La dirección para la cual se moverá está determinada por el estado actual del robot, pues si es SAVE, se inclina hacia el corral más cercano, si es FIND se inclina hacia el niño más cercano y si es CLEAN se inclina hacia la casilla sucia más cercana.

Finalmente se selecciona aleatoriamente la acción a ejecutar entre todas las posibles y en el único caso que podrían haber dos opciones es cuando el robot está en estado FIND parado sobre una casilla sucia, por lo que puede avanzar para alcanzar al niño o gastar la acción deteniéndose a limpiar. Nunca se decide limpiar una basura mientras el robot carga un niño, claramente le da más prioridad a guardar el niño.

## 2.2. Agente CleanerRobot

Este robot se preocupa más por la limpieza que por lograr tener todos los niños en el corral. La idea puede traer el inconveniente de que al tener la gran mayoría de los niños afuera, estos estén generando mayor cantidad de suciedad, la cual se irá acumulando y provocará rápidamente el despido del robot. En este punto influye también la cantidad de niños que haya en el ambiente, pues en caso de ser pocos, por lo general el robot no es despedido, satisfaciendo así su objetivo principal. Por mucho que limpie, en raras ocasiones podrá alcanzar el estado final teniendo todos los niños ubicados en el corral, por tanto la simulación será interrumpida en la iteración 100.

Los robots de este tipo no estarán nunca en estado FIND, pues solo llevan a un niño al corral en caso de haber coincidido con ellos en la misma casilla. La función `do_action` de este agente muestra cómo se actualizan sus estados y se decide la acción a ejecutar.

```
def do_action(self, env):
    """
    env: environment
    """
    if self.has_child(): #si tiene un niño se mueve hacia el corral
        self.state = Robot.SAVE
    else: #se mantiene en estado CLEAN
        self.state = Robot.CLEAN
    bot_cell = env.get_position(self.position)
    action = None
```

```

if self.state == Robot.SAVE and bot_cell.is_guard(): #conducta 1
    action = self.drop_child
if self.state == Robot.CLEAN and bot_cell.is_dirty(): #conducta 2
    action = self.clean_cell
else: #conducta 3
    action = self.move

return action(env)

```

**Conducta 1:** Igual que en el *ProtectRobot* se encuentra en el estado SAVE(cargando a un niño) y parado sobre un corral por tanto la acción a ejecutar es `drop_child`.

**Conducta 2:** el robot está en estado CLEAN y situado sobre una casilla sucia por tanto su acción será limpiar.

**Conducta 3:** Si no se presentan ninguna de las dos situaciones anteriores, las cuales son conducta con mayor prioridad, el robot se moverá hacia la dirección que se determine según su estado actual.

### 3. Ideas seguidas para la implementación

A continuación se puntualizará cómo se tuvieron en cuenta e implementaron algunos aspectos importantes del problema a resolver.

#### 3.1. Representación del ambiente

La clase **Environment** representa el ambiente del problema. El mismo está conformado por una matriz de objetos de tipo **Cell**, los cuales encapsulan la información de una casilla del mapa del ambiente. Cada casilla de estas tiene los siguientes atributos:

```

class Cell:
    def __init__(self, i, j, floor):
        self.floor = floor
        self.p = (i, j)
        self.obj = None

```

`floor`: representa el tipo de suelo de la casilla. El mismo puede ser EMPTY, DIRTY o GUARD, este último significa que es una casilla del corral.

`p`: coordenadas de la casilla en el mapa

`obj`: elemento del ambiente situado sobre la casilla. Puede ser de tipo *Child* o *Robot*.

### 3.2. Creación del ambiente inicial

Para inicializar un ambiente se necesita una configuración inicial del mismo con los parámetros vistos en la primera sección. La función encargada de crearlo se adaptó de forma tal que pudiese ser reutilizada en la variación aleatoria. Por tanto recibe otros parámetros adicionales como `bot_has_child` para indicar si el robot se encuentra cargando un niño, `bot_floor` el tipo de casilla donde se pondrá el robot y `childs_in_guards` la cantidad de niños a ubicar sobre el corral. Durante la inicialización del ambiente estos argumentos toman los siguientes valores respectivamente: `False`, `EMPTY` y `0`, pues en un inicio el robot no contiene ningún niño, estará sobre una casilla vacía y no hay niños en el corral.

### 3.3. Variación aleatoria

Este proceso de variación aleatoria del ambiente consiste en reordenar todos los elementos del ambiente sin cambiar de estado ni posición al robot, por tanto, sería como reiniciar al ambiente pero a otro estado. Como se dijo anteriormente, se utiliza la misma función que crea el ambiente: `restart_map`, y en este caso, los parámetros adicionales ya cobran significado.

El encargado de ejecutar esta operación es el *Simulator* y se lanzará cada  $t$  iteraciones de la simulación. El simulador previamente recopila la información que necesita del robot, para que una vez se realice este proceso, se mantenga invariante su estado. Dicha información serían los parámetros adicionales antes mencionados.

### 3.4. Selección de la dirección a moverse

En la definición del problema se plantea que los agentes solo podrán moverse en cuatro direcciones: `NORTH ()`, `SOUTH ()`, `EAST ()` y `WEST ()`.

Los agentes tienen una función `select_direction` la cual es invocada cada vez que la acción a ejecutar sea moverse. La dirección retornada estará en correspondencia con el estado u objetivo en el que se encuentre el agente. En caso de ser un niño, la dirección será seleccionada aleatoriamente entre todos los posibles movimientos que pueda realizar el niño desde su posición actual, si no hay ninguno, entonces la dirección será `(0, 0)` indicando quedarse en el lugar.

Esta función en los robots, se auxilia de un dfs el cual recibe un predicado especificando el tipo de casilla que cumple con su objetivo. El dfs determina cual es el camino más corto para alcanzar dicha meta y retorna la dirección de la próxima posición a donde moverse.

A continuación se ejemplifican los posibles predicados mediante la función `select_direction` del agente *PotectRobot*. Notar que el mismo es determinado en dependencia del estado de robot, y que además, si no hay forma de alcanzar ese objetivo entonces retornará cualquier dirección válida para moverse.

```
def select_direction(self, env):
    print ('select_direction_from_protect_robot')
    f = None
    if self.state == Robot.CLEAN:
        print ('cleaning')
        f = lambda x: x.is_dirty()
```



```

elif self.state == Robot.SAVE:
    print('saving_a_boy')
    f = lambda x: x.is_guard() and (not x.is_full())
else:
    print('finding_a_boy')
    f = lambda x: x.is_full() and isinstance(x.obj, Child) and
        not x.is_guard()
d = bfs(env, self.position[0], self.position[1], self, f)
if d == None:
    possible_choices = self.possible_movements(env)
    if len(possible_choices) == 0: #state in place
        possible_choices = [(0, 0)]
    d = rnd.choice(possible_choices)
return d

```

### 3.5. Simulador

```

class Simulator:
    def __init__(self):
        self.t = None
        self.iter = 0
        self.env = None
        self.bot = None
        self.childs = {}
        self.statistics = {"STOP": 0, "FIRE": 0, "DONE": 0, "DIRTY": 0}

    def init_world(self, t, N, M, dirty_porcent, obstacle_porcent,
        num_childs, bot_type):

    def random_variation_world(self):

    def end_simulation(self):

    def run(self):

def simulate(iterations, t, N, M, dirty_porcent, obst_porcent, num_childs, bot_type):
    s = Simulator()
    for i in range(iterations):
        s.init_world(t, N, M, dirty_porcent, obst_porcent, num_childs, bot_type)
        print(s.env)
        print("START_SIMULATION:", i)
        s.run()
        "-----SIMULATION_RESULTS-----"
        print("number_of_layoffs:", s.statistics["FIRE"])

```

```
print("number_of_stop_in_iteration_100:", s.statistics["STOP"])
print("number_of_goal_accomplished:", s.statistics["DONE"])
print("average_percentage_of_dirt:", s.statistics["DIRTY"] / iterations)
```

#### 4. Simulaciones

#### 5. Ejecución

#### 6. Conclusiones