Reporte Actividad Integradora

Pablo Yamamoto A01022382, Gianluca Beltrán A01029098 y Javier Corona A01023063

Clases de los Agentes

Agente de organización

Para la solución de la actividad, nuestra clase de los agentes tiene varias propiedades y funciones. Contiene la posición en (x,y), la dirección que puede tomar (el robot no puede moverse en diagonales), box_count nos sirve para saber cuántas cajas movió una instancia de un agente y el move_count para saber cuantos movimientos (desplazamientos en el grid) realizó.

```
You, 9 minutes ago | 2 authors (GianlucaBeltran and others)
class OrganizingAgent(Agent):
    """
    Agent that moves, detects boxes, pickup boxes and puts them in a defined coordinate of the grid. (It can put 5 boxes in a stash)
    Attributes:
        unique_id: Agent´s ID
        direction: Randomly chose a direcction from the eight directions
    """

    def __init__(self, unique_id, model, pos):
        """
        Creates a new organizing agent.
        Args:
            unique_id: The agent's ID
            model: Model reference for the agent
        """
        super().__init__(unique_id, model)
        self.pos = pos
        self.direction = 0
        self.name = "OrganizingAgent"
        self.condition = "Searching"
        # How many moves the agent has made
        self.move_count = 0
        # How many boxes the agent has picked up
        self.box_count = 0
        self.stash_count = 0
        self.freeSpaces = []
        self.possible_steps = []
        self.carried_box = None

    def get_free_spaces(self):…

    def check_for_box(self):…

    def grab_box(self):…

    def head_home(self):…

    def move(self):…

    def step(self):…
```

La primera función que tiene nuestro robot es la de get_free_spaces, nos ayuda a determinar si el agente se puede mover en la dirección que escogió. Para esto creamos una variable con los

posibles pasos que puede dar y revisamos cada paso con un loop. En una lista asignamos valores del 0 al 3 para saber si hay una caja, obstáculo, otro robot o si la celda esta libre.

```
def get_free_spaces(self):
    """
    Determines if the agent can move in the direction that was chosen
    """
    self.possible_steps = self.model.grid.get_neighborhood(
        self.pos, moore=False, include_center=False)

    # Open a list of empty spaces
    self.freeSpaces = []
    for step in self.possible_steps:
        # If there is an Obstacle or Organazing Agent in the direction you are looking at, it will append a false to or free space list, meaning it wont be able to move there
        if self.model.grid.is_cell_empty(step):
            self.freeSpaces.append(2)
        else:
            for content in self.model.grid.iter_cell_list_contents(step):
                if content.condition == "Delivered":
                    self.freeSpaces.append(2)
                    break
                elif content.condition == "OrganizingAgent" or content.condition == "Obstacle":
                    self.freeSpaces.append(0)
                    break
                else:
                    self.freeSpaces.append(1)
                    break

    print(self.possible_steps, self.freeSpaces)
```

Checamos si hay cajas con nuestra función check_for_box que itera sobre todos los vecinos que tiene nuestro robot.

```
def check_for_box(self):
    for neighbor in self.model.grid.neighbor_iter(self.pos, moore=False):
        if neighbor.condition == "Box":
            return True
    return False
```

Después tenemos nuestra función grab_box, esta nos ayuda a actualizar el estado de nuestro robot, sabemos que ahora está cargando una caja y también le indicamos a la caja que seleccionó que se está movimiento con el agente.

```
def grab_box(self):
    self.condition = "Carrying"
    for neighbor in self.model.grid.neighbor_iter(self.pos):
        if neighbor.condition == "Box":
            self.carried_box = neighbor
            self.carried_box.condition = "Moving"
            break
    self.model.grid.move_agent(self.carried_box, self.pos)
```

Nuestra función head_home es de las más interesantes, esta función calcula la distancia más corta entre la posición actual de nuestro robot y la posición del stash (en nuestro caso es (1, 1)) sí es que esté ya esta cargando una caja, de esta manera el robot sigue el camino más corto siempre. Esta función mueve al robot y a la caja que está cargando.

```python
def head_home(self):
    if self.pos == (1, 1):
        self.condition = "Searching"
        self.carried_box.condition = "Delivered"
        return

    home_x = 1
    home_y = 1
    current_pos_x = self.pos[0]
    current_pos_y = self.pos[1]

    distances = []
    distances.append(sqrt(((current_pos_x - 1) - home_x)
                          ** 2 + (current_pos_y - home_y) ** 2))
    distances.append(sqrt((current_pos_x - home_x) **
                          2 + (current_pos_y - 1 - home_y) ** 2))
    distances.append(sqrt((current_pos_x - home_x) **
                          2 + (current_pos_y + 1 - home_y) ** 2))
    distances.append(sqrt(((current_pos_x + 1) - home_x)
                          ** 2 + (current_pos_y - home_y) ** 2))

    min_index = 0
    min_value = 1000
    for distance in range(len(distances)):
        print(distances)
        if distances[distance] < min_value and self.freeSpaces[distance] == 2:
            min_value = distances[distance]
            min_index = distance

    self.model.grid.move_agent(self, self.possible_steps[min_index])
    self.move_count += 1
    self.model.grid.move_agent(
        self.carried_box, self.possible_steps[min_index])
    self.pos = self.possible_steps[min_index]
```

Agente de Obstáculo

Nuestro agente de obstáculo se utiliza en el borde del grid, aunque también podría colocarse aleatoriamente para representar un cuarto con objetos que tienen que ser evitados

```python
class ObstacleAgent(Agent):
    """
    Obstacle agent
    """

    def __init__(self, model):
        """
            Creates a new obstacle agent.
            Args:
                unique_id: The agent's ID
                model: Model reference for the agent
            """
        super().__init__(self, model)
        self.condition = "Obstacle"

    def step(self):
        pass
```

Agente de Caja

El agente de la caja es lo mismo que el del obstáculo pero si la caja aparece en la posición (1, 1), esta ya no tiene que ser entregada por que ya está en la posición del stash,

```python
class BoxAgent(Agent):
    """
    Box agent
    """

    def __init__(self, unique_id, model, pos):
        """
        Creates a new box agent.
        Args:
            unique_id: The agent's ID
            model: Model reference for the agent
        """
        super().__init__(unique_id, model)
        self.pos = pos
        self.condition = "Box"
        if self.pos == (1, 1):
            self.condition = "Delivered"

    def step(self):
        pass
```

Tiempo y Movimiento de los Agentes para la solución del problema

Corrimos la simulación con distintos números de agentes y de cajas para sacar el tiempo promedio y el número de movimientos que realiza cada agente y analizar el comportamiento de estos. Cada simulación se corrió 3 veces.

Simulación con 1 agente

Número de agentes	Número de cajas	Tiempo	Movimientos	Tiempo promedio	Movimientos promedio
1	5	157	122	184	148
		293	240		
		100	81		
1	10	613	525	722	611
		812	681		
		741	627		
1	15	844	710	993	781
		1168	975		
		787	658		

Durante esta simulación pudimos notar que al incrementar el número de cajas el agente necesitaba mucho más movimientos totales para mover las cajas de las celdas al stash, esto es obvio pero el agente perdió demasiado tiempo y movimientos yendo y regresando al stash cuando no tenía cajas que acomodar ahí. Esto es un gran área de oportunidad ya que esta perdida de tiempo se hizó notorio cuando pasamos a acomodar 10 y 15 cajas.

También nos encontramos que al colocar 15 cajas hay configuraciones donde el agente al regresar al stash se atora porque toma en cuenta la distancia más corta para llegar, pero hay una caja impidiendo su regreso entonces entra en un loop infinito haciendo que la simulación nunca pueda completarse. Para resolver este problema tendríamos que darle memoria al robot para que no repita el mismo movimiento más de unas cuantas veces y evitar este loop infinito.

Mientras aumentamos el número de cajas también se aumenta el promedio de tiempo y movimientos que toma completarse la simulación.

Simulación con 5 agentes

Número de agentes	Número de cajas	Tiempo	Movimientos	Tiempo promedio	Movimientos promedio
5	5	83	59, 65, 65, 69, 71 **329 Total**	68	267
		57	36, 40, 43, 43, 46 **208 Total**		
		65	50, 52, 54, 54, 54 **264 Total**		
5	10	113	76, 88, 90, 92, 93 **439 Total**	125	468
		159	110, 111, 113, 115, 121 **570 Total**		
		103	70, 75, 79, 84, 86 **394 Total**		
5	15	304	227, 229, 223, 245, 246 **1170 Total**	204	768
		178	121, 123, 129, 138, 152 **663 Total**		
		131	83, 85, 90, 103, 115 **476 Total**		

Podemos ver que el Tiempo claramente disminuye en esta simulación ya que hay muchos más agentes organizando el almacén, aunque el número de movimientos aumenta por tener muchos más de estos mismos. En la primera simulación con 15 cajas encontramos que algunos de los agentes pasaron mucho tiempo con el mismo problema que en la simulación con 1 agente, que regresaba a la base sin estar cargando cajas y particularmente un agente quedó atrapado entre una caja sin poder regresar, pero no ocurrió un loop infinito ya que otro agente llego y movió la caja de su camino. Debido a todo esto el tiempo y número de

movimientos y tiempo en segundos se vió incrementado en comparación con las otras dos simulaciones.

Simulación con 10 agentes

Número de agentes	Número de cajas	Tiempo	Movimientos	Tiempo promedio	Movimientos promedio
10	5	37	12, 15, 16, 18, 22, 22, 24, 24, 26, 28 **207 Total**	37	257
		46	17, 33, 35, 36, 37, 38, 40, 40, 40, 42 **358 Total**		
		28	12, 14, 17, 21, 22, 22, 22, 23, 23, 27 **205 total**		
10	10	83	35, 38, 44, 46, 50, 55, 59, 59, 64, 68 **518 Total**	72	450
		47	20, 23, 25, 27, 28, 30, 31, 32, 35, 42 **293 Total**		
		85	44, 49, 50, 51, 51, 54, 55, 59, 62, 65 **540 Total**		
10	15	66	25, 29, 30, 35, 42, 44, 45, 47, 51, 53 **401 Total**	98	631
		97	41, 45, 45, 50, 57, 63, 71, 72, 74, 76 **594 Total**		

		131	81, 86, 89, 90, 90, 90, 90, 92, 93, 96 **897 Total**		

Con este número de agentes y encontramos otro problema, como la colocación de agentes y cajas es aleatoria, muchas veces la mayoría de los agentes encontraban cajas al inicio de la simulación haciendo que volvieran a la base, pero estos rodearon y bloquearon el perímetro completo del stash haciendo imposible que continuará el movimiento, esto ocasiona que un agente invadiera el espacio del obstáculo haciendo que la simulación terminé.

Solución cooperativa para el problema

Idealmente los agentes deberían comunicarse entre ellos para no estorbarse los unos a los otros, ya que en las simulaciones con 5 y 10 agentes, se desperdiciaba mucho tiempo y número de movimientos a la hora de esquivar a sus compañeros si es que se estaba llevando una caja de regreso. También sería necesario implementar algún sistema jerárquico para que los agentes no se amontonen y terminen la simulación sin cumplir el objetivo.

También se puede implementar memoria en los agentes para que recuerden dónde estuvieron y marcar donde encontraron una caja y hacérselo saber a sus compañeros, así no se perdería tiempo al buscar en los mismos lugares cuando esa área ya está limpia, aunque esto dependería de la situación ya que se pueden estar dejando cajas aleatoriamente y los agentes tendrían que regresar.