Tecnológico de Monterrey

Evidencia 1: Actividad Integradora

Jessica Odette Guizado Hernández A01643724

Santos Alejandro Arellano Olarte A01643742

Luz Patricia Hernández Ramírez A01637277

Carlos Alberto Mentado Reyes A01276065

Ángel Eduardo Esquivel Vega A01276114

Modelación de sistemas multiagentes con gráficas computacionales (Gpo 101)

Profesores:

Iván Axel Dounce Nava

Luis Raúl Guerrero Aguilar,

Obed Nehemías Muñoz Reynoso,

Carlos Johnnatan Sandoval Arrayga

Martes 27 de agosto, 2024

Link del Repositorio en Github:

https://github.com/Santos-Arellano/Modeling-of-Multi-Agent-Systems-with-Computer-Graphics-PROJECT

Evidencia 1: Actividad Integradora

En un almacén de tamaño $m \times n$ se encuentran cinco robots encargados de buscar cajas dispersas en el suelo, recogerlas, y llevarlas a una estantería designada. Para simular este escenario, se ha desarrollado un modelo en Python utilizando las librerías **agentPy** y **owlready2**. La simulación permite evaluar cómo estos robots logran organizar el almacén de manera eficiente.

Agente Robot

El agente robot es la unidad fundamental de la simulación, diseñado para operar de manera autónoma en el entorno del almacén. Su comportamiento se basa en una combinación de reglas predefinidas y aprendizaje por refuerzo, permitiéndole adaptarse a diferentes situaciones y optimizar su desempeño en la organización de objetos.

Arquitectura del Agente

- **Percepción:**
    - **Sensores:** El robot utiliza los sensores implementados en **agentPy** para percibir el estado de las celdas adyacentes, en una distancia de 10 unidades. Estos sensores le permiten detectar si una celda está libre, ocupada por otro robot, si contiene una pared o un objeto, y la altura de la pila de objetos en esa celda.
    - **Información de la Ontología:** Consulta la ontología creada con **owlready2** para obtener información semántica sobre los objetos y el entorno. Esta ontología proporciona al robot un marco de referencia para clasificar los objetos y entender sus posibles interacciones con el entorno.
- **Toma de Decisiones:**
    - **Motor de Inferencia:** El robot emplea un motor de inferencia basado en reglas definidas en la ontología para realizar razonamiento practico. Esto le permite tomar decisiones informadas basadas en el conocimiento del mundo y en las reglas del entorno.
    - **Aprendizaje por Refuerzo:** Utiliza un algoritmo de aprendizaje por refuerzo, como **Q-learning**, para aprender de la interacción con el entorno. Esto le permite mejorar su desempeño a lo largo del tiempo, ajustando sus estrategias para optimizar la eficiencia en la tarea.
- **Acción:**
    - **Actuadores:** El robot está equipado con actuadores que le permiten realizar acciones como moverse, girar, recoger objetos y apilarlos en las estanterías. Estas acciones están definidas como métodos en la clase del agente dentro de **agentPy**.

Comportamiento del Agente

- **Navegación:**

    - Emplea un algoritmo de búsqueda **A*** para encontrar la ruta más corta hacia su objetivo, evitando obstáculos y optimizando el recorrido. Esto es crucial en un entorno dinámico donde las posiciones de los otros robots y los objetos pueden cambiar constantemente.
- **Manipulación:**
    - Realiza tareas de **pick-and-place** (recogida y colocación) utilizando los mecanismos de acción proporcionados por **agentPy**. El robot debe ser capaz de recoger un objeto, transportarlo a la estantería designada y colocarlo en la pila adecuada.
- **Aprendizaje:**
    - El agente aprende a través de la experiencia, ajustando sus políticas de acción para maximizar una recompensa definida. Por ejemplo, puede buscar minimizar el tiempo necesario para apilar todas las cajas o maximizar la cantidad de cajas correctamente apiladas en el menor número de movimientos.

Evaluación del Desempeño

El desempeño del agente se evalúa utilizando las métricas definidas en la actividad, así como algunas métricas adicionales:

- **Tasa de Éxito:** Porcentaje de tareas completadas con éxito, es decir, cuántas veces los robots logran apilar todas las cajas en las estanterías designadas sin errores.
- **Flexibilidad:** Capacidad del robot para adaptarse a cambios en el entorno, como la reubicación de objetos o la aparición de nuevos obstáculos.
- **Robustez:** Capacidad del agente para operar en condiciones adversas, como fallos de sensores o la presencia de obstáculos inesperados que no fueron previstos en la programación inicial.

Integración con agentPy y owlready2

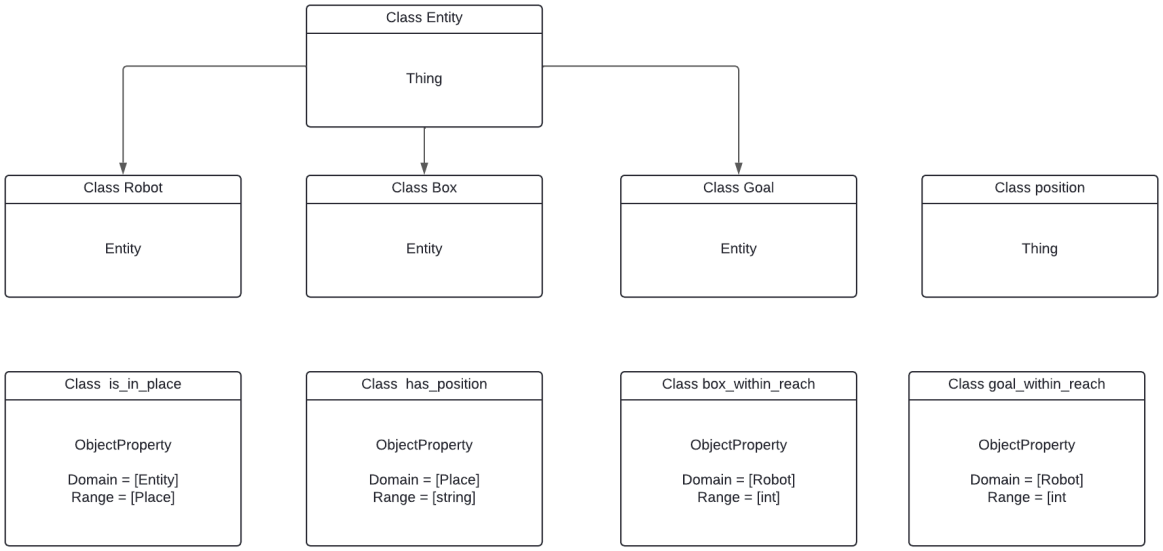
- **agentPy:**
    - **Modelado del Entorno:** Se utiliza para modelar el entorno del almacén, los agentes (robots), y las interacciones entre ellos. Las acciones de los robots, como moverse y recoger objetos, se definen como métodos en la clase del agente.
- **owlready2:**
    - **Creación de Ontología:** Se utiliza para crear una ontología que representa el conocimiento del mundo, incluyendo la clasificación de objetos, las relaciones entre ellos, y las posibles acciones que los robots pueden realizar. Esta ontología guía el razonamiento deductivo del robot, asegurando que las decisiones tomadas sean coherentes con las reglas y objetivos establecidos.

Diagramas de clases de los agentes y ontologías utilizadas

RobotAgent
agentType : int firstStep : bool currentPlan : List [Tuple [int, int]] RobotStorage : int RobotProcedure : int this_robot : Robot intentionSucceded : bool
setup(self) : None step(self) : None update(self) : None end(self) : None
brf(self, p : List[Agent]) : None brf_second(self, g : List[Agent]) : None options(self) : Dict[BoxAgent, float] options_second(self) : Dict[GoalAgent : float] filter(self) : Box or None filter_second(self) : Goal or None plan(self) : List [Tuple[int, int]] plan_second(self) : LIst [Tuple[int, int]] BDI(self, p : List[Agent]) : None BDI_second(self, g : List[Agent]) : None execute(self) : None initBeliefs(self, initPos : Tuple[int, int]) : None initIntentions(self) : None
see(self, e : Environment) : List[BoxAgent] see_second(self, e : Environment) : List[GoalAgent]
Capacidad para observar, actualizar creencias, filtrar deseos, planear acciones y ejecutar un ciclo BDI.
El agente puede observar su entorno, actualizar sus creencias basadas en una ontología, filtrar y seleccionar objetivos, planificar y ejecutar planes en un entorno de simulación.
Robots

BoxAgent
agentType : int
setup(self) : None step(self) : None update(self) : None end(self) : None
El agente es una entidad pasiva en la simulación, cuyo propósito principal es representar una caja. No tiene comportamiento activo definido más allá de su configuración inicial.
Boxes

Diagrama de ontología



Conclusión (descripción de soluciones alternativas para mejorar la eficiencia de los agentes)

La simulación desarrollada ha demostrado el potencial de los agentes robóticos para optimizar la gestión de almacenes. Sin embargo, se identifican diversas estrategias que pueden implementarse para mejorar aún más su eficiencia y adaptabilidad.

Soluciones Alternativas y Mejoras

1. **Optimización de Algoritmos de Búsqueda**:
    - **Heurísticas personalizadas**: Desarrollar heurísticas específicas para el entorno del almacén, como considerar la altura de las pilas o la proximidad a las estanterías, para guiar de manera más eficiente la búsqueda de rutas.
    - **Algoritmos de búsqueda probabilísticos**: Utilizar algoritmos como RRT* para encontrar rápidamente rutas en entornos dinámicos y con obstáculos impredecibles.
2. **Aprendizaje por Refuerzo Profundo**:
    - **Modelos de aprendizaje profundo**: Explorar arquitecturas como las redes neuronales convolucionales para procesar información visual y tomar decisiones más precisas.
    - **Aprendizaje por transferencia**: Utilizar el conocimiento adquirido en tareas similares para acelerar el proceso de aprendizaje en nuevas situaciones.
3. **Coordinación Multi-Agente**:

    - **Comunicación eficiente**: Implementar protocolos de comunicación robustos y de bajo latencia para garantizar una coordinación efectiva entre los agentes.
    - **Planificación centralizada y distribuida**: Combinar la planificación centralizada para tareas a largo plazo con la planificación distribuida para tareas a corto plazo.
4. **Optimización de Recursos**:
    - **Gestión de energía**: Implementar estrategias de gestión de energía para maximizar la duración de la batería y minimizar el tiempo de inactividad.
    - **Asignación de tareas dinámica**: Asignar tareas a los agentes en función de sus capacidades y de la situación actual del almacén.
5. **Simulación Física Realista**:
    - **Simulación de dinámica**: Incorporar un simulador físico más detallado para modelar la dinámica de los objetos y los robots, lo que permitiría una mayor precisión en la planificación de movimientos y la detección de colisiones.
6. **Aprendizaje Continuo**:
    - **Aprendizaje activo**: Permitir a los agentes seleccionar de forma activa las experiencias de aprendizaje más informativas.
    - **Meta-aprendizaje**: Desarrollar algoritmos que permitan a los agentes aprender a aprender, adaptándose a cambios en el entorno y a nuevas tareas de manera más eficiente.

Evaluación y Métricas

Para evaluar la efectividad de estas soluciones, se pueden utilizar las siguientes métricas:

- **Eficiencia**: Tiempo de completación de tareas, distancia recorrida, consumo de energía.
- **Robustez**: Capacidad de operar en condiciones adversas, como fallos de sensores o cambios en el entorno.
- **Flexibilidad**: Capacidad de adaptarse a nuevas tareas y entornos.
- **Aprendizaje**: Velocidad de aprendizaje, generalización a nuevas situaciones.
- **Colaboración**: Eficacia en la colaboración con otros agentes.

Conclusiones Finales

La simulación del almacén proporciona una plataforma valiosa para investigar y desarrollar nuevas técnicas de inteligencia artificial y robótica. Al explorar las soluciones alternativas propuestas, se pueden obtener valiosos conocimientos para el diseño de sistemas robóticos más eficientes y adaptables en entornos industriales y logísticos.

Código utilizado para la simulación del reto:

En esta primera parte se importan las librerías que usará el programa

```
!pip install agentpy pathfinding owlready2
import agentpy as ap
import pathfinding as pf
import matplotlib.pyplot as plt
from owlready2 import *
import itertools
import random
import IPython
import math
```

Después se obtiene la ontología definida, para evitar una sobrecarga de información que puede llevar a errores, si se tiene que correr varias veces el codigo se destruye la ontología anteriormente creada

```
onto = get_ontology("file://onto.owl")

onto.destroy(update_relation = True, update_is_a = True)
```

Después se define la ontología y todas las relaciones

```
with onto:
    #Se define que hay cosas (Thing) que van a ser agentes
    class Entity(Thing):
      pass

    class Robot(Entity):
      pass

    class Box(Entity):
      pass

    class Goal(Entity):
      pass

    class Place(Thing):
      pass

    class Position(Thing):
      pass

    #Se definen las relaciones entre los agentes
    class is_in_place(ObjectProperty):
      domain = [Entity]
      range = [Place]
      pass

    class has_position(ObjectProperty, FunctionalProperty):
      domain = [Place]
      range = [str]
      pass

    class box_within_reach(ObjectProperty):
      domain = [Robot]
      range = [int]

    class goal_within_reach(ObjectProperty):
      domain = [Robot]
      range = [int]
```

Después que se definieron las ontologías se empieza a definir la clase del Agente del Robot

- **`class RobotAgent(ap.Agent):`**: Define la clase `RobotAgent`, que hereda de `ap.Agent`, indicando que esta clase representa un agente en el sistema de simulación.

```
class RobotAgent(ap.Agent):

    def see(self, e):
        seeRange = 10
        P = [a for a in e.neighbors(self, distance=seeRange) if a.agentType == 1]
        return P
```

- **`see(self, e):`**: Observa a los agentes en un rango específico (`seeRange`). Devuelve una lista de agentes que son del tipo 1 (supuestamente cajas o algún otro tipo de objeto).

```
    def see_second(self,e):
      seeRange = 10
      G = [a for a in e.neighbors(self,distance=seeRange) if a.agentType==2 and
a.pile<=5]
      return G
```

`see_second(self,e):`: Similar al método anterior, pero se enfoca en agentes del tipo 2 (supuestamente objetivos) que tienen un atributo `pile` menor o igual a 5.

```
    def brf(self, p):

        for box in self.this_robot.box_within_reach:
            destroy_entity(box.is_in_place[0])
            destroy_entity(box)
        destroy_entity(self.this_robot.is_in_place[0])

        currentPos = self.model.Store.positions[self]
        self.this_robot.is_in_place = [Place(at_position=str(currentPos))]
```

```
for b in p:
            theBox = Box(is_in_place=[Place()])
            theBox.is_in_place[0].at_position = str(self.model.Store.positions[b])
            self.this_robot.box_within_reach.append(theBox)
```

`brf(self, p):`: Actualiza el sistema de creencias del robot. Primero destruye las creencias anteriores (cajas dentro del alcance y la posición actual). Luego, actualiza la posición del robot y las cajas dentro del alcance.

```
  def brf_second(self, g):

    for goal in self.this_robot.goal_within_reach:
      destroy_entity(goal.is_in_place[0])
      destroy_entity(goal)
    destroy_entity(self.this_robot.is_in_place[0])

    currentPos = self.model.Store.positions[self]
    self.this_robot.is_in_place = [Place(at_position=str(currentPos))]

    for b in g:
      theGoal = Goal(is_in_place=[Place()])
      theGoal.is_in_place[0].at_position = str(self.model.Store.positions[b])
      self.this_robot.goal_within_reach.append(theGoal)
```

- **`brf_second(self, g):`**: Similar a `brf`, pero actualiza el sistema de creencias del robot para los objetivos en lugar de las cajas.

```
  def options(self):

      distances = {}

      for onto_box in self.this_robot.box_within_reach:
          box_pos = eval(onto_box.is_in_place[0].at_position)
          robot_pos = eval(self.this_robot.is_in_place[0].at_position)
          d = math.sqrt((box_pos[0] - robot_pos[0]) ** 2 + (box_pos[1] -
robot_pos[1]) ** 2)
          distances[onto_box] = d

      return distances
```

`options(self):`: Calcula y devuelve las distancias a las cajas dentro del alcance del robot. La distancia se calcula en 2D usando la fórmula de la distancia euclidiana.

```
  def options_second(self):

    distances_to_goals = {}

    for onto_goal in self.this_robot.goal_within_reach:
      goal_pos = eval(onto_goal.is_in_place[0].at_position)
      robot_pos = eval(self.this_robot.is_in_place[0].at_position)
      d = math.sqrt((goal_pos[0] - robot_pos[0]) ** 2 + (goal_pos[1] -
robot_pos[1]) ** 2)
      distances_to_goals[onto_goal] = d

    return distances_to_goals
```

- **`options_second(self):`**: Similar a `options`, pero calcula las distancias a los objetivos dentro del alcance del robot.

```
  def filter(self):
      desires = {x: y for x, y in sorted(self.D.items(), key=lambda item:
item[1])}
      return list(desires.items())[0][0] if desires else None
```

- **`filter(self):`**: Ordena los deseos en función de sus valores y devuelve el deseo con el menor valor. Si no hay deseos, devuelve `None`.

```
  def filter_second(self):
    desires = {x: y for x, y in sorted(self.D.items(), key=lambda item: item[1])}
    return list(desires.items())[0][0] if desires else None
```

- **`filter_second(self):`**: Igual que `filter`, pero se utiliza para el contexto de los objetivos.

```
  def plan(self):
      if self.I is None:
          if random.randint(0, 1) == 0:
```

```
            return [(random.choice([-1, 1]), 0)]
        else:
            return [(0, random.choice([-1, 1]))]

    thePlanX = []
    thePlanY = []

    boxPos = eval(self.I.is_in_place[0].at_position)
    robotPos = eval(self.this_robot.is_in_place[0].at_position)
    distance2D = (boxPos[0] - robotPos[0], boxPos[1] - robotPos[1])

    for i in range(abs(distance2D[0])):
        thePlanX.append(1 if distance2D[0] >= 0 else -1)

    for j in range(abs(distance2D[1])):
        thePlanY.append(1 if distance2D[1] >= 0 else -1)

    thePlanX = list(zip(thePlanX, [0] * len(thePlanX)))
    thePlanY = list(zip([0] * len(thePlanY), thePlanY))

    thePlan = thePlanX + thePlanY

    return thePlan
```

- **`plan(self):`**: Crea un plan para moverse hacia la intención actual. Si no hay intención, el robot se mueve aleatoriamente. De lo contrario, el plan se crea en función de la distancia a la caja.

```
def plan_second(self):

  if self.I is None:
    if random.randint(0, 1) == 0:
      return [(random.choice([-1, 1]), 0)]
    else:
      return [(0, random.choice([-1, 1]))]

  thePlanX = []
  thePlanY = []

  goalPos = eval(self.I.is_in_place[0].at_position)
  robotPos = eval(self.this_robot.is_in_place[0].at_position)
  distance2D = (goalPos[0] - robotPos[0], goalPos[1] - robotPos[1])
```

```
  for i in range(abs(distance2D[0])):
      thePlanX.append(1 if distance2D[0] >= 0 else -1)

  for j in range(abs(distance2D[1])):
      thePlanY.append(1 if distance2D[1] >= 0 else -1)

  thePlanX = list(zip(thePlanX, [0] * len(thePlanX)))
  thePlanY = list(zip([0] * len(thePlanY), thePlanY))

  thePlantoGoal = thePlanX + thePlanY
  print(thePlantoGoal)

  return thePlantoGoal
```

- **`plan_second(self):`**: Similar a `plan`, pero para los objetivos.

```
def BDI(self, p):
    """Calls all functions from the BDI architecture."""

    self.brf(p)
    if self.intentionSucceded:
        self.intentionSucceded = False
        self.D = self.options()
        self.I = self.filter()
        self.currentPlan = self.plan()
```

- **`BDI(self, p):`**: Implementa la arquitectura BDI (Belief-Desire-Intention). Actualiza las creencias, determina las opciones, filtra las intenciones y crea un plan si la intención tiene éxito.

```
def BDI_second(self, g):
  self.brf_second(g)
  if self.intentionSucceded:
    self.intentionSucceded = False
    self.D = self.options_second()
    self.I = self.filter_second()
    self.currentPlan = self.plan_second().pop()
```

- **`BDI_second(self, g):`**: Igual que `BDI`, pero para el contexto de los objetivos.

```
  def execute(self):
      """Executes the plan, action by action."""
      if len(self.currentPlan) > 0:
          currentAction = self.currentPlan.pop(0)
          new_position = (self.model.Store.positions[self][0] +  currentAction[0],
                          self.model.Store.positions[self][1] + currentAction[1])


          if new_position not in [self.model.Store.positions[robot] for robot in
self.model.robots if robot != self]:
              self.model.Store.move_by(self, currentAction)

          else:
              print(f"Robot {self.id} encontró una colisión y no se movio")
          currentAction = (0, 0)

      else:
          self.intentionSucceded = True
          currentAction = (0, 0)

      if currentAction != (0, 0):
          self.model.Store.move_by(self, currentAction)
```

- **`execute(self):`**: Ejecuta el plan acción por acción. Si hay un plan, el robot intenta moverse y evita colisiones.

```
  def initBeliefs(self, initPos):
      place = Place(at_position=str(initPos))
      self.this_robot = Robot(is_in_place=[place])
```

- **`initBeliefs(self, initPos):`**: Inicializa el sistema de creencias del robot con su posición inicial.

```
  def initIntentions(self):"
      self.intentionSucceded = True
      self.I = None
```

- **`initIntentions(self):`**: Inicializa la intención del robot como vacía.

```
    def setup(self):
        self.agentType = 0
        self.firstStep = True
        self.currentPlan = []
        self.RobotStorage = 0
        self.RobotProcedure = 1
```

- **`setup(self):`**: Configura el agente inicialmente, incluyendo el tipo de agente y otros parámetros.

```
    def step(self):
        if self.firstStep:
            initPos = self.model.Store.positions[self]
            self.initBeliefs(initPos)
            self.initIntentions()
            self.firstStep = False

        if self.RobotStorage>0:
          self.firstStep = True
          self.BDI(self.see_second(self.model.Store))
        else:
          self.BDI(self.see(self.model.Store))

        self.execute()
```

- **`step(self):`**: Realiza un paso en la simulación. Inicializa creencias e intenciones en el primer paso y ejecuta el plan basado en la observación actual.

```
    def update(self):
        pass

    def end(self):
        pass
```

- **`update(self):`** y **`end(self):`**: Métodos vacíos para actualizaciones y finalización, respectivamente.

```
class BoxAgent(ap.Agent):
```

```
""" Representa un agente de tipo caja en el sistema de simulación. Hereda de
ap.Agent, lo que significa que es un agente en el entorno de simulación. """
    #Setup
    def setup(self):
""" Método de configuración inicial del agente. Aquí se establece el tipo del
agente como 1. """
        self.agentType = 1
```

`setup(self):`: Este método se llama una vez al inicio para configurar el agente. Aquí, se asigna el valor 1 al atributo `agentType` del agente, indicando que este agente es una caja (o un tipo específico de agente) en el sistema.

```
    #Step
    def step(self):
        pass
```

`step(self):`: Este método se llama en cada paso de la simulación. Está vacío (`pass`), lo que significa que el agente no realiza ninguna acción en cada paso. Este método se puede personalizar para definir el comportamiento del agente durante la simulación.

```
    #Update
    def update(self):
        pass
```

- **`update(self):`**: Este método se llama para actualizar el estado del agente. Al igual que `step`, está vacío (`pass`), lo que indica que no se realizan actualizaciones adicionales en el estado del agente en esta versión.

```
    #End
    def end(self):
        pass
```

Clase `GoalAgent`:

- **`setup`**: Inicializa el agente como tipo 2 y establece `pile` a 0.
- **`step`**: Método para definir el comportamiento en cada paso de la simulación (actualmente vacío).
- **`update`**: Actualiza el estado del agente durante la simulación (actualmente vacío).
- **`end`**: Realiza tareas de limpieza al final de la simulación (actualmente vacío).

```
class GoalAgent(ap.Agent):

 def setup(self):

   self.agentType = 2
   self.pile = 0

 def step(self):
   pass

 def update(self):
   pass

 def end(self):
   pass
```

```
class StoreModel(ap.Model):

 def setup(self):

   self.robots = ap.AgentList(self,self.p.robots,RobotAgent)
   self.boxes = ap.AgentList(self,self.p.box,BoxAgent)
   self.goals = ap.AgentList(self,self.p.goals,GoalAgent)

   self.Store = ap.Grid(self,self.p.storeSize,track_empty=True)

   self.Store.add_agents(self.robots,random = True,empty = True)
   self.Store.add_agents(self.boxes,random = True,empty = True)
   self.Store.add_agents(self.goals,random = True,empty = True)
```

`setup`:

- **Descripción**: Configura el entorno de la simulación.
- **Acciones**: Inicializa listas de agentes (robots, cajas, objetivos), crea un grid para los agentes, y añade los agentes al grid.

```
 def step(self):
   self.robots.step()
   self.boxes.step()
   self.goals.step()

   for robot in self.robots:
     if robot.RobotStorage<1:
       for box in self.boxes:
         if box in self.Store.positions and self.Store.positions[box] ==
self.Store.positions[robot]:
           robot.RobotStorage += 1
           self.Store.remove_agents(box)
           self.boxes.remove(box)
           print(f"Robot {robot.id} ha recogido una caja. Total:
{robot.RobotStorage}")
           break
     else:
       for goal in self.goals:
         if goal in self.Store.positions and self.Store.positions[goal] ==
self.Store.positions[robot]:
           if goal.pile < 5:
             goal.pile += 1
             robot.RobotStorage -= 1
             print(f"Robot {robot.id} ha subido una caja. Total:
{robot.RobotStorage}")
             print(f"Goal {goal.id} ha recibido una caja. Pila: {goal.pile}")
           else:
             print(f"Goal {goal.id} ha llegado a su límite de cajas.")
             self.goals.remove(goal)
             self.Store.remove_agents(goal)
             break




   if len(self.boxes) == 0 and all(robot.RobotStorage == 0 for robot in
self.robots):
     print("Fin de la simulación")
     self.stop()

   #if len(self.boxes == self.robots.RobotStorage):
     #self.stop()
   for robot in self.robots:
      print(f"Robot {robot.id} tiene {robot.RobotStorage} cajas.")
```

`step`:

- **Descripción**: Actualiza el estado de los agentes y gestiona sus interacciones.
- **Acciones**:
  - Llama al método `step` para cada tipo de agente.
  - Gestiona la recogida de cajas por parte de los robots.
  - Gestiona la entrega de cajas a los objetivos.
  - Finaliza la simulación si no quedan cajas y todos los robots están vacíos.
  - Imprime el estado actual de los robots.

```
 def update(self):
   pass

 def end(self):
   pass
```

```
#A FUNCTION TO ANIMATE THEE SIMULATION

def animation_plot(model, ax):
```

`agent_type_grid = model.Store.attr_grid('agentType')`: Obtiene la cuadrícula que representa los tipos de agentes desde el modelo de simulación.

```
   agent_type_grid = model.Store.attr_grid('agentType')
```

`ap.gridplot(agent_type_grid, cmap='Accent', ax=ax)`: Dibuja la cuadrícula en el gráfico utilizando el mapa de colores `'Accent'`.

```
   ap.gridplot(agent_type_grid, cmap='Accent', ax=ax)
```

- **`ax.set_title(f"Robot en almacen \n Time-step: {model.t}, ")`**: Establece el título del gráfico, mostrando el paso de tiempo actual de la simulación.

```
   ax.set_title(f"Robot en almacen \n Time-step: {model.t}, ")
```

```
#SIMULATION PARAMETERS


#a random variables (0,1)
r = random.random()
```

`r = random.random()`: Genera una variable aleatoria `r` entre 0 y 1.

```
#parameters dict
parameters = {
   "robots" : 5,      #Amount of Hunters
```

```
   "box" : 15,       #Amount of coins
   "goals" : 5,       #Amount of goals
"storeSize" : (15,15),       #Grid size
   "steps" : 100,           #Max steps
   "seed" : 13*r            #seed for random variables (that is random by itself)
}
```

`parameters = {...}`: Define un diccionario con los parámetros de la simulación, incluyendo la cantidad de robots, cajas, metas, el tamaño de la cuadrícula, el número máximo de pasos y una semilla aleatoria.

```
#============================================================================0


#SIMULATION:


#Create figure (from matplotlib)
fig, ax = plt.subplots()
```

`fig, ax = plt.subplots()`: Crea una figura y ejes para visualizar la animación usando Matplotlib.

```
#Create model
model = StoreModel(parameters)
```

`model = StoreModel(parameters)`: Crea una instancia del modelo `StoreModel` utilizando los parámetros definidos.

```

#Run with animation
#If you want to run it without animation then use instead:
#model.run()
animation = ap.animate(model, fig, ax, animation_plot)
#This step may take a while before you can see anything
```

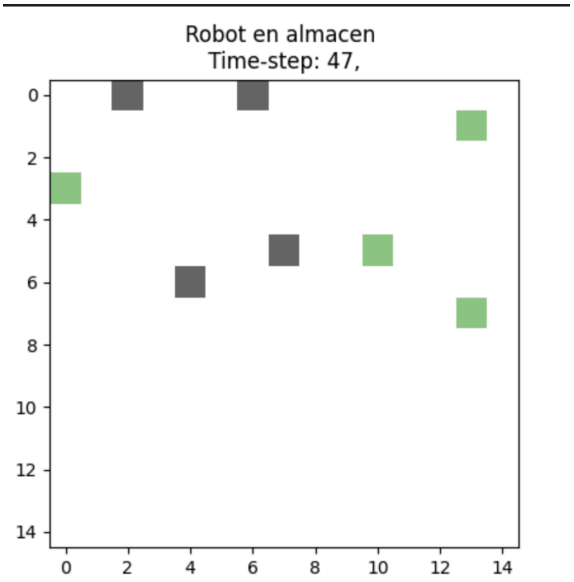
`animation = ap.animate(model, fig, ax, animation_plot)`: Ejecuta la simulación con animación. Reemplaza `some_animation_library` con la librería de animación que estás utilizando.

```
#Print the final animation
IPython.display.HTML(animation.to_jshtml())

```

- **`IPython.display.HTML(animation.to_jshtml())`**: Muestra la animación final en formato HTML en un notebook de Jupyter.

```
Robot 2 ha recogido una caja. Total: 1
Robot 1 tiene 0 cajas.
Robot 2 tiene 1 cajas.
Robot 3 tiene 0 cajas.
Robot 4 tiene 0 cajas.
Robot 5 tiene 0 cajas.
Robot 3 ha recogido una caja. Total: 1
Robot 4 ha recogido una caja. Total: 1
Robot 5 ha recogido una caja. Total: 1
Robot 1 tiene 0 cajas.
Robot 2 tiene 1 cajas.
Robot 3 tiene 1 cajas.
Robot 4 tiene 1 cajas.
Robot 5 tiene 1 cajas.
Robot 1 tiene 0 cajas.
Robot 2 tiene 1 cajas.
Robot 3 tiene 1 cajas.
Robot 4 tiene 1 cajas.
Robot 5 tiene 1 cajas.
Robot 2 ha subido una caja. Total: 0
Goal 21 ha recibido una caja. Pila: 1
Robot 4 ha subido una caja. Total: 0
Goal 24 ha recibido una caja. Pila: 1
Robot 1 tiene 0 cajas.
Robot 2 tiene 0 cajas.
...
Robot 2 tiene 0 cajas.
Robot 3 tiene 0 cajas.
Robot 4 tiene 0 cajas.
Robot 5 tiene 0 cajas.
```



• Para la Parte 2, el archivo .unitypackage con todo lo necesario para ejecutar la solución.