Tecnológico de Monterrey

Actividad 01

Alejandro Hernández Álvarez	A01638684
Axel Iván Magallón Páramo	A01636714
Eduardo Venegas Hernández	A01638188
Pablo Emilio Blanco Celis	A01637761
Diego Alberto Ortiz Mariscal	A01552000

18 de noviembre de 2021

Modelación de Sistemas Multiagentes

01

1 Juego de disparejo

Considere un juego de disparejo entre cuatro jugadores, en el que cada jugador adopta la siguiente estrategia: - El jugador 1 siempre escoge al azar entre las dos opciones. - El jugador 2 escoge siempre hacia abajo sin importar lo que haya ocurrido anteriormente. - El jugador 3 escoge la última opción ganadora de las partidas anteriores. En la primera jugada, escoge arriba. - El jugador 4 escoge aquello opuesto al jugador 1 en la última partida. En la primera jugada, escoge al azar. Utilice `AgentPy` para generar una simulación del juego, y simule 1000 juegos consecutivos. Muestre los resultados obtenidos para cada uno de los jugadores al final de la simulación.

[1]:
```
import agentpy as ap
import numpy as np
from collections import Counter
```

Definir un objeto donde se va a almacenar la información del juego

[2]:
```
HISTORY = {
    "last_win": None,
    "plays": [],
    "winners": []
}
```

Definir la clase del jugador, donde se tendrán las reglas que debe de seguir cada uno de ellos

[3]:
```
class Player(ap.Agent):

    def setup(self):
        """
        Método que configura al agente
        """
        self.id = 0
        self.last_play = 0

    def play(self):
        """
        Método que define la acción a tomar
            1 -> arriba
            2 -> abajo
```

```
        """
        global HISTORY

        # Escoge un número aleatorio entre 1 y 2
        if self.id == 1:
            self.last_play = np.random.randint(0,2) + 1

        # Siempre escoge 2 (abajo)
        elif self.id == 2:
            self.last_play = 2

        # Siempre escogerá última opción que ganó, en caso de que no haya␣
↪ganado ninguna todavía entonces escogerá 1
        elif self.id == 3:
            self.last_play = HISTORY["last_win"] if HISTORY["last_win"] else 1

        # Siempre escogerá la opción contraria a la que escojió el jugador 1 en␣
↪la partida anterior. Al principio es al azar
        elif self.id == 4:
            if len(HISTORY["plays"]) == 0:
                self.last_play = np.random.randint(0,2) + 1
            elif HISTORY["plays"][-1][0] == 1:
                self.last_play = 2
            elif HISTORY["plays"][-1][0] == 2:
                self.last_play = 1
```

Definir la información del juego, donde se va a inicializar cada uno de los jugadores además de definir como se va a hacer cada una de las jugadas

[4]:
```
class Game(ap.Model):

    def setup(self):
        """
        Configuración inicial del modelo (sobrecarga)
        """
        self.agents = ap.AgentList(self, self.p.agents, Player)

        # A cada uno de los agentes asignarles un id diferente
        self.agents.id = ap.AttrIter([1, 2, 3, 4])


    def step(self):
        """
        Define el comportamiento del sistema paso a paso (sobrecarga)
        """
        global HISTORY
```

```
        # Llamar al método play de cada agente
        self.agents.play()

        # Comparar los resultados y definir al ganador
        results = [agent.last_play for agent in self.agents]
        HISTORY["plays"].append(results)

        # Definir si hay un ganador
        results = np.array(results) - 1
        if np.count_nonzero(results) == 1:
            HISTORY["last_win"] = 2
            for i in range(4):
                if HISTORY["plays"][-1][i] == 2:
                    HISTORY["winners"].append(i + 1)
                    break
        elif np.count_nonzero(results) == 3:
            HISTORY["last_win"] = 1
            for i in range(4):
                if HISTORY["plays"][-1][i] == 1:
                    HISTORY["winners"].append(i + 1)
                    break
```

Definir los parámetros del juego

[5]:
```
parameters = {
    'agents': 4,
    'steps': 1000,
    'seed': 13
}
```

Correr la simulación del juego

[6]:
```
model = Game(parameters)
model.run()
```

```
Completed: 1000 steps
Run time: 0:00:00.142021
Simulation finished
```

[6]:
```
DataDict {
'info': Dictionary with 9 keys
'parameters':
    'constants': Dictionary with 3 keys
'reporters': DataFrame with 1 variable and 1 row
}
```

Leaderboard

[7]:
```
ordered_players = Counter(HISTORY["winners"]).most_common()
wins = 0
for index, result in enumerate(ordered_players):
    print(f"#{index + 1}. Player 0{result[0]} (winner {result[1]} times)")
    wins += result[1]

print(f"-> {parameters['steps'] - wins} times were played without a winner")
```

```
#1. Player 02 (winner 190 times)
#2. Player 01 (winner 137 times)
#3. Player 03 (winner 136 times)
#4. Player 04 (winner 53 times)
-> 484 times were played without a winner
```

Ver 20 de los juegos que se hicieron para confirmar que se respeten las reglas del juego

[8]:
```
HISTORY["plays"][:20]
```

[8]:
```
[[1, 2, 1, 1],
 [1, 2, 2, 2],
 [1, 2, 1, 2],
 [2, 2, 1, 2],
 [2, 2, 1, 1],
 [2, 2, 1, 1],
 [1, 2, 1, 1],
 [1, 2, 2, 2],
 [2, 2, 1, 2],
 [1, 2, 1, 1],
 [2, 2, 2, 2],
 [2, 2, 2, 1],
 [1, 2, 1, 1],
 [1, 2, 2, 2],
 [1, 2, 1, 2],
 [1, 2, 1, 2],
 [1, 2, 1, 2],
 [2, 2, 1, 2],
 [2, 2, 1, 1],
 [1, 2, 1, 1]]
```

1.1 Preguntas

¿Qué jugador obtuvo mejores resultados? - Los mejores resultados, por un amplio margen, los obtuvo el jugador 02. Este jugador siempre tomaba la misma posición sin importar los resultados anteriores o lo que pusieran los demás jugadores. Esto lo podemos ver en la penúltima celda, donde vemos las veces que ganó cada uno de los jugadores de las 1000 veces que se realizó el juego.

¿Alguna de las estrategias fue mejor que las otras? - La mejor estrategia fue claramente siempre poner la mano para abajo (la misma posición en todos los juegos). Repetimos el experimento más de 5 veces y siempre el jugador 02 fue el vencedor, además de que por un amplio margen.

02

1 Forest Fire

1.1 "This model simulates the spread of a fire through a forest. It shows that the fire's chance of reaching the right edge of the forest depends critically on the density of trees."

1.1.1 First, we import the libraries

[37]:
```
import agentpy as ap

#Others
import random
# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
import IPython
```

1.1.2 Then, we create the class wich contains the setup method.

1.1.3 In this method we create the agents, the forest and define the posible conditions in which a tree could be

1.1.4 There are 5 possible conditions as state for the trees

1.2 0. Alive, 1: burn-low, 2: burn-mid, 3: burn-high and 4: totally burned

1.3 In what condition a tree stays depends on how many neighbours are on fire around it

1.3.1 On top of that, a randomizer start position was added so that in every iteration of the model we could start from a different position

1.3.2 this positions are top, down, left and right

[38]:
```
class ForestModel(ap.Model):

    def setup(self):

        # Create agents (trees)
        n_trees = int(self.p['Tree density'] * (self.p.size**2))
        trees = self.agents = ap.AgentList(self, n_trees)
```

```
        # Create grid (forest)
        self.forest = ap.Grid(self, [self.p.size]*2, track_empty=True)
        self.forest.add_agents(trees, random=True, empty=True)

        # Initiate a dynamic variable for all trees
        # Condition 0: Alive, 1: burn-low, 2: burn-mid, 3: burn-high, 4:␣
↪almost-burned 5: totally-burned
        self.agents.condition = 0

        self.random = random.randint(0,3)

        # Start a fire from the left side of the grid

        #Start from left side
        if self.random == 0:
            unfortunate_trees = self.forest.agents[0:self.p.size, 0:3]
        #Start from top side
        elif self.random == 1:
            unfortunate_trees = self.forest.agents[0:2, 0:self.p.size]
        #Start from right side
        elif self.random == 2:
            unfortunate_trees = self.forest.agents[0:self.p.size, 47:self.p.
↪size]
        #Start from down side
        elif self.random == 3:
            unfortunate_trees = self.forest.agents[47:self.p.size, 0:self.p.
↪size]

        unfortunate_trees.condition = 1

    def step(self):

        # Select burning trees
        burning_trees = self.agents.select(self.agents.condition == 1 or self.
↪agents.condition == 2 or self.agents.condition == 3)

        # Spread fire
        for tree in burning_trees:
            for neighbor in self.forest.neighbors(tree):
                if neighbor.condition < 5:
                    neighbor.condition = neighbor.condition + 1 # Neighbor␣
↪starts burning

            tree.condition = 5 # Tree burns out
```

```
        # Stop simulation if no fire is left
        if len(burning_trees) == 0:
            self.stop()

    def end(self):

        # Document a measure at the end of the simulation
        burned_trees = len(self.agents.select(self.agents.condition == 4 or␣
↪self.agents.condition == 5))
        self.report('Percentage of burned trees',
                    burned_trees / len(self.agents))
```

1.4 Here we defined the parameters of the model as of

1.4.1 tree density, size and steps

[39]:
```
# Define parameters

parameters = {
    'Tree density': 0.6, # Percentage of grid covered by trees
    'size': 50, # Height and length of the grid
    'steps': 100,
}
```

1.4.2 Now. we make a plot with the simulation of how all the trees were burned

1.4.3 In this simulation we defined different color to represent each state of the tree

1.5 The more red a tree is the more burned it is, if the tree is black it means is totally burned

[40]:
```
# Create single-run animation with custom colors

def animation_plot(model, ax):
    attr_grid = model.forest.attr_grid('condition')
    color_dict = {0:'#7FC97F', 1:'#B32A2A', 2:'#ffff00',3:'#DEBA09' ,4:␣
↪'#DE8709', 5: '#ffffff', None:'#d5e5d5'}

    ap.gridplot(attr_grid, ax=ax, color_dict=color_dict, convert=True)
    ax.set_title(f"Simulation of a forest fire\n"
                 f"Time-step: {model.t}, Trees left: "
                 f"{len(model.agents.select(model.agents.condition == 0))}")

fig, ax = plt.subplots()
model = ForestModel(parameters)
```

```
animation = ap.animate(model, fig, ax, animation_plot)
IPython.display.HTML(animation.to_jshtml(fps=15))
```

[40]: <IPython.core.display.HTML object>

[41]:
```
# Prepare parameter sample
parameters = {
    'Tree density': ap.Range(0.2, 0.6),
    'size': 100
}
sample = ap.Sample(parameters, n=30)
```

1.5.1 Then we perform an experiment to test many simulations

[42]:
```
# Perform experiment
exp = ap.Experiment(ForestModel, sample, iterations=40)
results = exp.run()
```

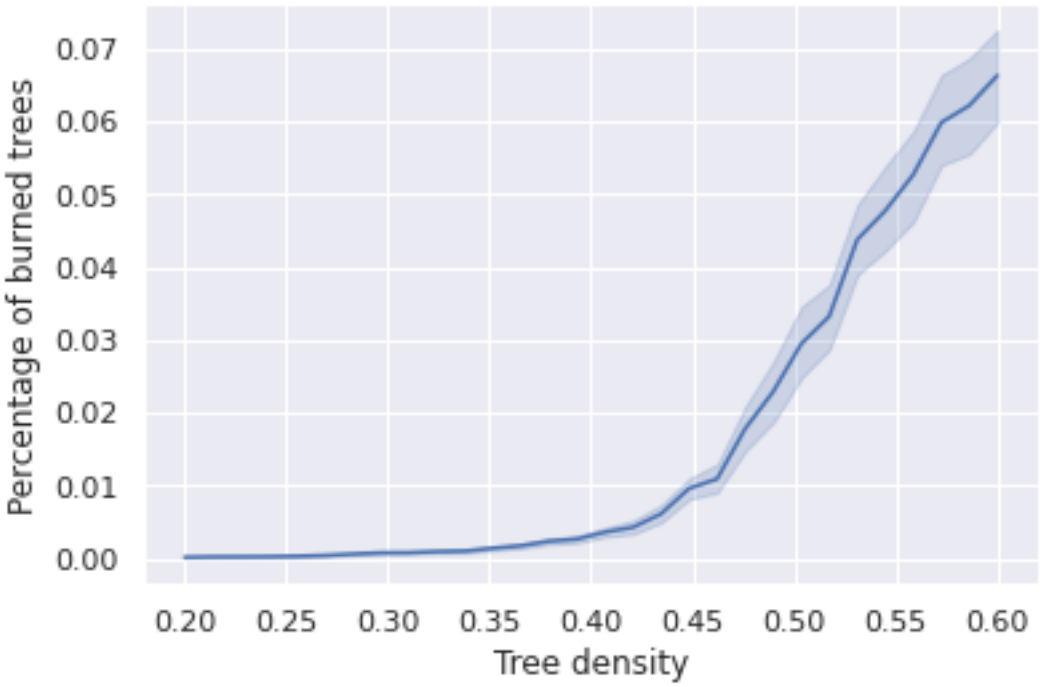
```
Scheduled runs: 1200
Completed: 1200, estimated time remaining: 0:00:00
Experiment finished
Run time: 0:06:57.161091
```

[43]:
```
# Save and load data
results.save()
results = ap.DataDict.load('ForestModel')
```

```
Data saved to ap_output/ForestModel_1
Loading from directory ap_output/ForestModel_1/
Loading reporters.csv - Successful
Loading info.json - Successful
Loading parameters_constants.json - Successful
Loading parameters_log.json - Successful
Loading parameters_sample.csv - Successful
```

1.6 Finnaly, we plot the results of the simulation

[44]:
```
# Plot sensitivity
sns.set_theme()
sns.lineplot(
    data=results.arrange_reporters(),
    x='Tree density',
    y='Percentage of burned trees'
);
```


Percentage of burned trees
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0.20
0.25
0.30
0.35
0.40
0.45
0.50
0.55
0.60
Tree density

03

0.1 # Robots Limpiadores

0.2 Instrucciones

Considere que en una habitación con MxN espacios, hay P robots limpiadores reactivos. Cada uno de los robot limpiadores se comporta de la siguiente manera:

Si la celda en la que se encuentra está sucia, entonces aspira durante 10 segundos. Si la celda está limpia, el robot elije una dirección aleatoria para moverse (unas de las 8 celdas vecinas que esté sin otro robot) y elije la acción de movimiento (si no puede moverse allí, permanecerá en la misma celda). Este movimiento dura 2 segundos. Si varios robots coinciden en alguna de las celdas, uno se queda en dicha celda y los otros se tiene que mover a alguna otra celda. Quien se queda o quien se tiene que mover se elige aleatoriamente. Al inicio de la simulación, las posiciones de los robots son elegidas al azar, y de igual forma las posiciones están limpias o sucias aleatoriamente.

Realiza al menos 5 simulaciones en este entorno con diferente número de robots, y reporta lo siguiente:

- Tiempo necesario hasta que todas las celdas estén limpias.
- Porcentaje de celdas limpias después del termino de la simulación.
- Número de movimientos realizados por todos los agentes.
- Analiza cómo la cantidad de agentes impacta el tiempo dedicado, así como la cantidad de movimientos realizados. ¿Qué comportamiento agregarías a los robots para que limpiaran con mayor velocidad?

0.3 Solución

0.3.1 Importar Librerías

Se importan las librerías necesarias para el entorno y el agente.

[253]:
```
# Librerías para el agente y modelo
import agentpy as ap
import numpy as np

# Librearías de Visualización y otros
import matplotlib.pyplot as plt
import IPython, copy
plt.style.use('ggplot')
```

0.3.2 Definición de las clases

Se define la clase agente `CleaningRobot`, la cuál se encarga de simular a los robots limpiadores, y la clase de modelo `CleaningRobotsModel`, la cuál se encarga de simular el entorno.

[254]:
```
class CleaningRobot(ap.Agent):

    # Inicialización del agente
    def setup(self):
        self.moves = 0  # Numbero de movimientos
        self.state = -1  # -1: buscando, 0: limpiado, 1,2,3,4,5: limpiando

    # Función de acción del agente
    def next(self):
        currentPos = self.model.room.positions[self]

        # Si la celda está limpia, seguir buscando
        if self.model.room_state[currentPos] == 0:
            self.state = -1  # Estado de buscando
            self.move()  # Mover a posición aleatoria
            self.moves += 1  # Aumentar el número de movimientos;
            return

        if self.state == -1:  # Si se enceuntra una celda se comienza a limpiar
            self.state = 5
        elif self.state > 0:  # Si se está limpiando, seguir limpiando
            self.state -= 1
        else:                 # Si termina de limpiar, volver al estado de␣
↪busqueda
            self.model.room_state[currentPos] = 0  # Se limpia la celda
            self.state -= -1

    # Mover a posición aleatoria
    def move(self):
        pos = self.getRandomPosition()
        self.model.room.positions[self] = pos

    # Obtener posición aleatoria
    def getRandomPosition(self):
        pos = False
        while pos == False:
            posM = self.model.random.randint(0, self.p.M - 1)
            posN = self.model.random.randint(0, self.p.N - 1)
            pos = (posM, posN)
            # Revisar si la posición está ocupada
            for robot in self.model.room.positions:
                if self.model.room.positions[robot] == pos:
                    pos = False
```

```
                    break
        return pos
```

[255]:
```
class CleaningRobotsModel(ap.Model):
    def setup(self):
        # Inicialización de los robots
        self.robots = ap.AgentList(self, self.p.robots, CleaningRobot)

        # Inicialización de la habitación
        self.room = ap.Grid(self, [self.p.M, self.p.N], track_empty=True)
        self.room.add_agents(self.robots, random=True, empty=True)  # Añade los␣
↪robots en posiciones aleatorias
        self.room_state = np.zeros([self.p.M, self.p.N]) # Estado de la␣
↪habitación: limpia

        # Añadir celdas sucias aleatoriamente
        n_dirty_cells = int(self.p.M * self.p.M * self.p.dirty_cells) # Obtener␣
↪el número de celdas sucias
        for _ in range(n_dirty_cells):
            row = self.random.randint(0, self.p.M - 1)
            col = self.random.randint(0, self.p.N - 1)
            # Si la celda está limpia, se pone sucia
            if self.room_state[row, col] == 0:
                self.room_state[row, col] = 1

    # Función paso
    def step(self):
        # Llamar acción de los robots
        self.robots.next()

        # Si la habitación está limpia, terminar
        if np.sum(self.room_state.flatten()) == 0:
            self.stop()

    # Función de Actualizado
    def update(self):
        # Guarda los movimientos de los robots
        n_moves = np.sum(np.array(self.robots.moves))
        self.record("Moves", n_moves)

        # Guarda la cantidad de celdas sucias
        n_dirty = np.sum(self.room_state.flatten())
        self.record("Dirty cells", n_dirty)

    # Función de fin
    def end(self):
        # Guarda los movimientos y el tiempo
```

```
        n_moves = np.sum(np.array(self.robots.moves))
        n_dirty = np.sum(self.room_state.flatten())
        self.report("Moves", n_moves)
        self.report("Time", 2 * self.t)
        self.report("Dirty cells", n_dirty)
```

0.3.3 Función de animación

Se define la función `animation_plot`, la cuál se encarga de animar el entorno.

[256]:
```
def animation_plot(model, ax):
    # Crea una copia del estado de la habitación
    grid = copy.deepcopy(model.room_state)

    # Identifica a los robots
    for robot in model.robots:
        grid[model.room.positions[robot]] = 2

    color_dict = {
        0: "#E5E5E5",
        1: "#D62C2C",
        2: "#207567",
    }

    # Dibuja la habitación
    ap.gridplot(grid, ax=ax, color_dict=color_dict, convert=True)
    ax.set_title(
        f"Cleaning robots\n"
        f"Time-step: {model.t * 2}, Cels left: "
        f"{np.sum(model.room_state).flatten()}"
    )
```

0.3.4 Simulación

Se definen los parámetros, se crea un objeto de la clase `CleaningRobotsModel`, y se ejecuta la simulación.

[257]:
```
parameters = {
  'M': 20,            # Number of rows
  'N': 10,            # Number of columns
  'robots': 5,        # Number of robots
  'dirty_cells': 0.2, # Percentage of dirty cells
  'steps': 100,       # Number of steps
}
```

[258]:
```
fig, ax = plt.subplots()

# Crea un modelo y guarda la animación
```

```
model = CleaningRobotsModel(parameters)
animation = ap.animate(model, fig, ax, animation_plot)

# Inicia la simulación
IPython.display.HTML(animation.to_jshtml(fps=15))
```

[258]: <IPython.core.display.HTML object>

0.4 Resultados

Resultados obtenidos en la simulación.

Nueva lista de parámetros:

[259]:
```
parametersList = [
  {
    'M': 20,             # Number of rows
    'N': 10,             # Number of columns
    'robots': 2,         # Number of robots
    'dirty_cells': 0.2, # Percentage of dirty cells
    'steps': 1000,        # Number of steps
  },
  {
    'M': 20,             # Number of rows
    'N': 10,             # Number of columns
    'robots': 5,         # Number of robots
    'dirty_cells': 0.2, # Percentage of dirty cells
    'steps': 1000,        # Number of steps
  },
  {
    'M': 20,             # Number of rows
    'N': 10,             # Number of columns
    'robots': 7,         # Number of robots
    'dirty_cells': 0.2, # Percentage of dirty cells
    'steps': 1000,        # Number of steps
  },
  {
    'M': 20,             # Number of rows
    'N': 10,             # Number of columns
    'robots': 10,         # Number of robots
    'dirty_cells': 0.2, # Percentage of dirty cells
    'steps': 1000,        # Number of steps
  },
  {
    'M': 20,             # Number of rows
    'N': 10,             # Number of columns
    'robots': 20,         # Number of robots
    'dirty_cells': 0.2, # Percentage of dirty cells
```

```
    'steps': 1000,       # Number of steps
  },
]
```

0.4.1 Simulacion

Se simularon 5 escenarios con diferentes números de robots. Se obtuvieron los siguientes resultados:

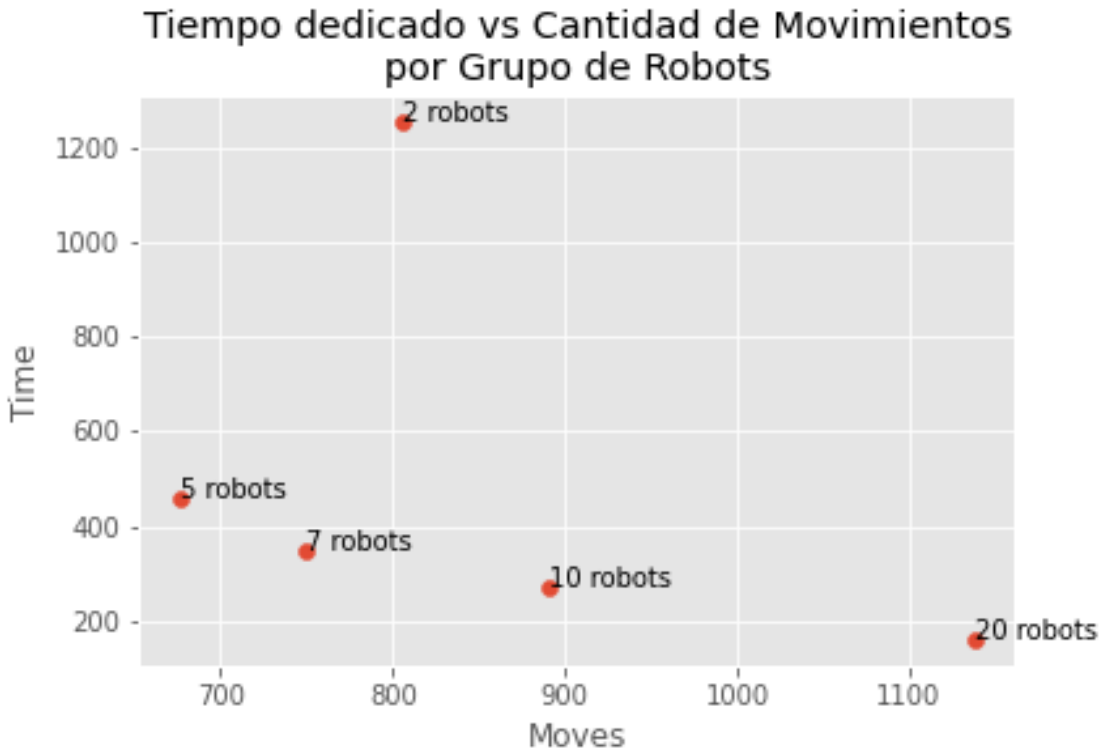
[260]:
```
data = {
  'Moves': [],
  'Dirty cells': [],
  'Time': [],
}
for parameter in parametersList:
    model = CleaningRobotsModel(parameter)
    result = model.run()
    data['Moves'].append(result.reporters['Moves'])
    data['Time'].append(result.reporters['Time'])
    data['Dirty cells'].append(result.reporters['Dirty cells'])
```

```
Completed: 627 steps
Run time: 0:00:00.325315
Simulation finished
Completed: 229 steps
Run time: 0:00:00.126025
Simulation finished
Completed: 175 steps
Run time: 0:00:00.107428
Simulation finished
Completed: 136 steps
Run time: 0:00:00.068207
Simulation finished
Completed: 80 steps
Run time: 0:00:00.061508
Simulation finished
```

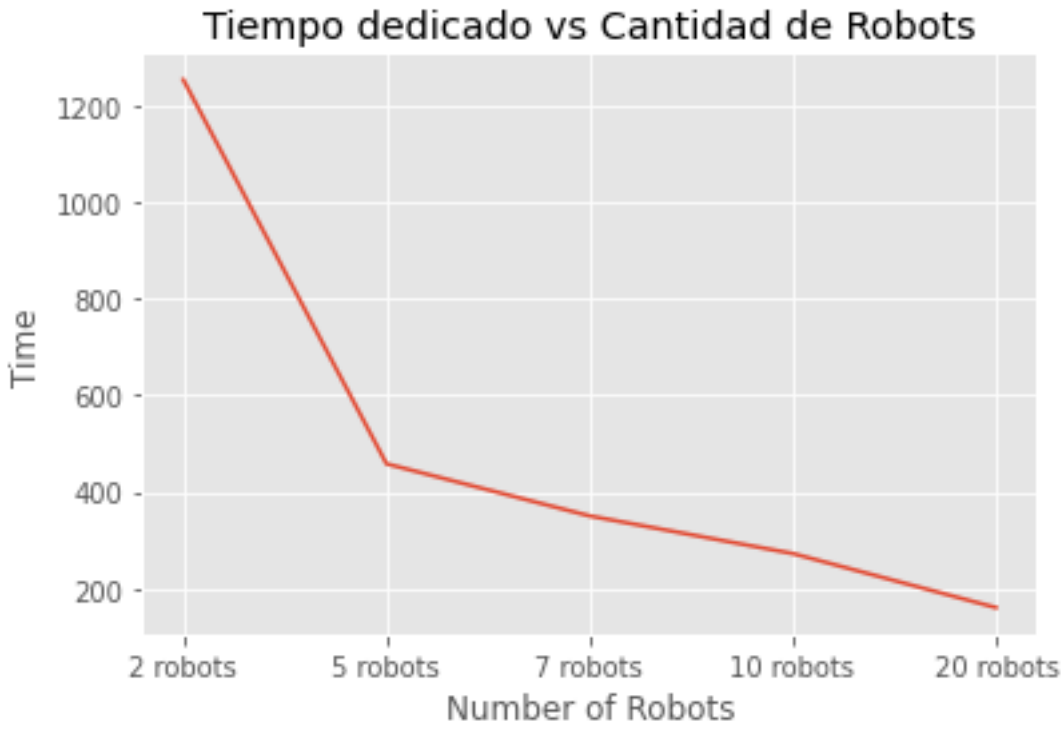
0.4.2 Gráficas

[261]:
```
# Data
x = data['Moves']
y = data['Time']
names = ['2 robots', '5 robots', '7 robots', '10 robots', '20 robots']

# Plot
fig, ax = plt.subplots()
ax.scatter(x, y)
```

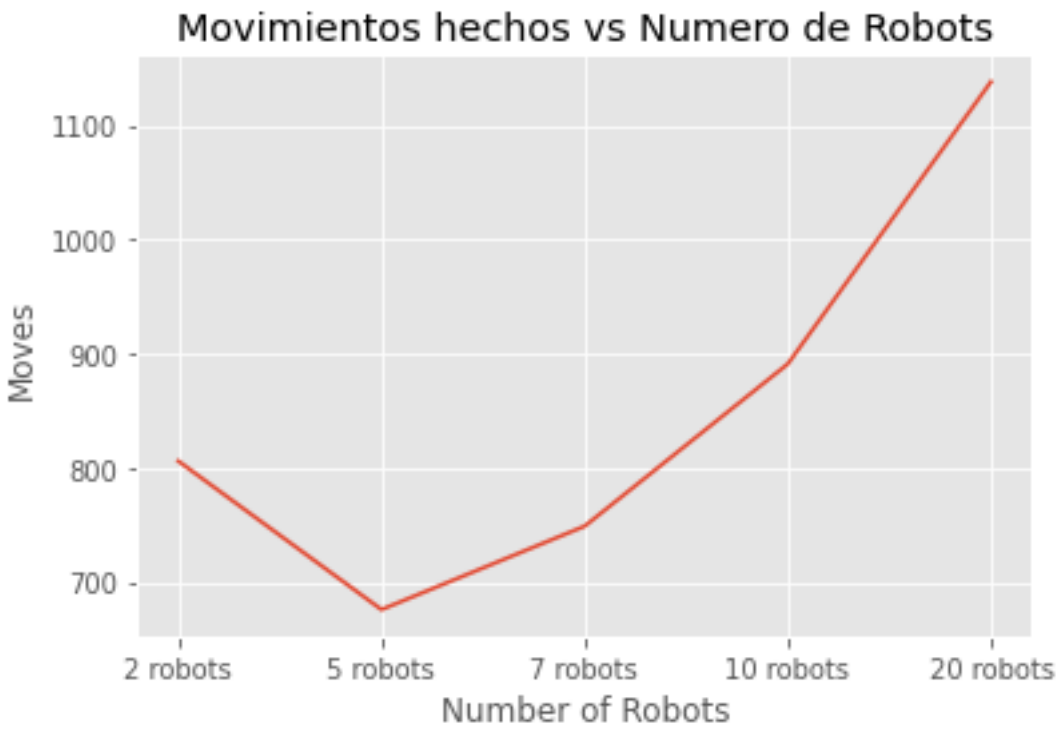
```
# Set labels
plt.xlabel('Moves')
plt.ylabel('Time')
plt.title('Tiempo dedicado vs Cantidad de Movimientos\npor Grupo de Robots')

for i, txt in enumerate(names):
    ax.annotate(txt, (x[i], y[i]))
plt.show()
```



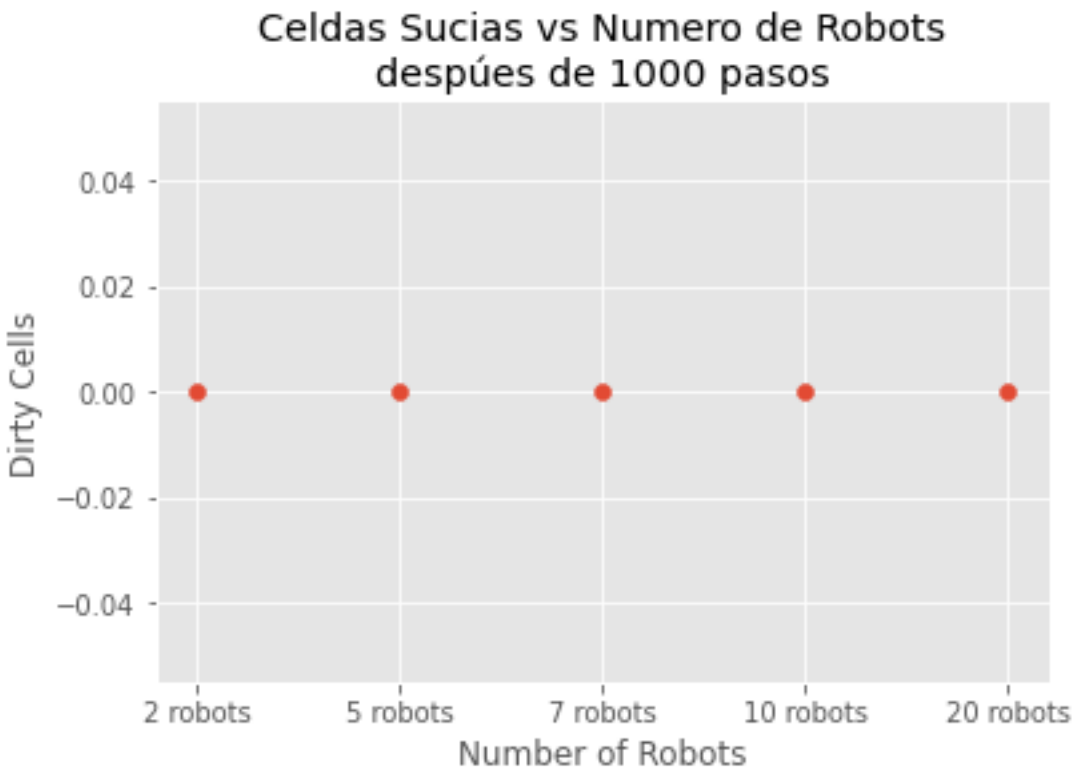
[262]:
```
# Data
y = data['Time']
x = ['2 robots', '5 robots', '7 robots', '10 robots', '20 robots']

# Plot
fig, ax = plt.subplots()
ax.plot(x, y)

# Set labels
plt.xlabel('Number of Robots')
plt.ylabel('Time')
plt.title('Tiempo dedicado vs Cantidad de Robots')
plt.show()
```



[263]:
```
# Data
y = data['Moves']
x = ['2 robots', '5 robots', '7 robots', '10 robots', '20 robots']

# Plot
fig, ax = plt.subplots()
ax.plot(x, y)

# Set labels
plt.xlabel('Number of Robots')
plt.ylabel('Moves')
plt.title('Movimientos hechos vs Numero de Robots')
plt.show()
```



[264]:
```
# Data
y = data['Dirty cells']
x = ['2 robots', '5 robots', '7 robots', '10 robots', '20 robots']

# Plot
fig, ax = plt.subplots()
ax.scatter(x, y)

# Set labels
plt.xlabel('Number of Robots')
plt.ylabel('Dirty Cells')
plt.title('Celdas Sucias vs Numero de Robots\ndespúes de 1000 pasos')
plt.show()
```



Preguntas

- Tiempo necesario hasta que todas las celdas estén limpias.
  - Ver la gráfica `Tiempo dedicado vs Cantidad de Robots`
- Porcentaje de celdas limpias después del termino de la simulación.
  - Ver la gráfica `Celdas Sucias vs Numero de Robots despúes de 1000 pasos`
- Número de movimientos realizados por todos los agentes.
  - Ver la gráfica `Movimientos hechos vs Numero de Robots`
- Analiza cómo la cantidad de agentes impacta el tiempo dedicado, así como la cantidad de movimientos realizados. ¿Qué comportamiento agregarías a los robots para que limpiaran con mayor velocidad?
  - Actualmente los robots solo se mueven a celdas libres, sin embargo, se podría añadir un "sensor" de tal forma que solo avance a celdas sucias. Esto se podría hacer repitiendo el randomInt hasta que de como resultado una celda sucia y no ocupada.