Ashton Mayo-Beavers
April 15, 2025
MACS 40550 1 Agent-Based Modeling
Midterm Assignment 1

Assignment:

For full points, the attached paper should:

- Clearly state the research question your modification is designed to answer.
- Pose a tentative hypothesis.
- Explain how your modification changes how we should think about the base model (is it more realistic? Does it change what the agents represent?).
- Your modification should be clearly explained, such that it could be reproduced by someone with access to the base model code and your description (you do not have to explain the original model).
- Briefly outline speculative results based on your observations of model dynamics.

Model: **Axtell's SugarScape model**

The **SugarScape Model** is an agent-based model that is designed to simulate wealth distribution and inequality through modeling agents who gather, eat, and move toward sugar (the measure of wealth) until they eventually die. The original model activates/initalizes the group of agents uniformly and simultaneously, where no agent has priority but are equally random in their actions to move, gather, eat, and die. However, in order to simulate - still a simplification - of wealth inequality and access to resources, agents should have variability of type to emulate an agent class system and real-life applications of social-stratification of classes and wealth distribution. Class-Based Sequential activation, rather than simultaneous activation, better models systematic inequality where there is a hierarchy of access to resources/wealth (ie, sugar) varying/depending on class. This proposed model modification looks at how class-based priority in resource access influences wealth inequality and agent behavior in the **SugarScape Model**.

The agents still represent the same, but now have the additional attribute of *agent_type* – either "upper-class" (*agent_type* =1), "middle-class" (*agent_type* =2), or "lower-class" (*agent_type* =3) – that are initialized in three groups as such instead of one agent group. In the sequential order of resource (sugar) access, "upper-class" agents first initiate the move function, which determines currently empty cells within line of sight, followed by the "middle-class" and "lower-class" agents, respectively. The same order occurs as agents then initiate the *gather_and_eat function* – consume sugar in the current cell, deplete it, then calculate metabolism – in class order, and then the *see_if_die* function – sees if an agent has zero or negative sugar, if so, it then dies and is removed from the model.

The main modifications of this model were to the *agents.py* and *model.py* files. The *agents.py* file defines what each agent's attributes are and what they do, such as moving, eating, and dying. Within the *SugarAgent* class, in addition to the cell, sugar, metabolism, and vision attributes, the implemented modification for the Class-Based Sequential activation would be the addition of an *agent_type* attribute

that represents the 'socioeconomic class' of the agent (either 1,2,3). The additional change in this file is for edge case handling in the situation where agents, particularly "lower" or "middle-class" agents with less access than "high-class" agents, may have **only** empty cells within their line of *vision* due to the depletion of local resources/sugar in cells. Within the *move, gather_and_eat, and see_if_die* functions, there is a proposed modification implementation to check if the agent's current cell is empty before proceeding to the rest of the function.

The proposed model modification within the *model.py* file *SugarScapeModel* class is to change how the agents are initialized and activated from simultaneous to sequential. The creation and random assignment of properties for agents is proposed to be implemented iteratively by agent type, creating three batches of agents by type. The second proposed modification within this file is to the *step* function, where agents are sorted in ascending agent type and the sequential activation of, *move, gather_and_eat, and see_if_die* functions.

Finally, to edit the GUI, the proposed edit to the *app.py* file adds distinction to each *agent_type* by color to the Solara visualization where: 1 = "upper-class" = gold, 2 = "middle-class" = blue, and 3 = "lower-class" = red.

In the observed Solara visualization of the model simulation, as expected, the "upper-class" agents mostly stay sedentary in each iteration/step of the model because they have primary access. While the other agent groups continuously move around the grid looking for resources/sugar around the "upper-class" agents in gold. This pattern leads to greater long-term inequality, where the Gini Coefficient of time (ie, step iteration) has a step increase over time, hitting a plateau around *Step* 3.

Code:

agents.py

```
Unset
'''
PROPOSED MODEL MODIFICATION: Class-Based Sequential Activation
    - as opposed to simultaneous activation
    - Change in agents.py file is just adding atribute of type in individual
      agent activation
      - agents.py define what each agent's attributes are and what they do
(move,eat,die)

'''

import math
## Using experimental agent type with native "cell" property that saves its current
position in cellular grid
from mesa.experimental.cell_space import CellAgent # each agent knows what cell it is
in and how to interact with grid

## Helper function to get distance between two cells
def get_distance(cell_1, cell_2):
```

```
    x1, y1 = cell_1.coordinate
    x2, y2 = cell_2.coordinate
    dx = x1 - x2
    dy = y1 - y2
    return math.sqrt(dx**2 + dy**2)


class SugarAgent(CellAgent): # implement modification
    ## Initiate agent, inherit model property from parent class
    def __init__(self, model, cell, sugar=0, metabolism=0, vision=0, agent_type=0):
        super().__init__(model)
        ## Set variable traits based on model parameters
        self.cell = cell    # where the agent is on the grid
        self.sugar = sugar  # how much sugar/wealth the agent has
        self.metabolism = metabolism # how much sugar they burn per step (cost of
living)
        self.vision = vision    # how far they can look for sugar
        ## PROPOSED CHANGE ###
        self.agent_type = agent_type # their 'socioeconomic' class
(1=upper,2=middle,3=lower)
      ## Define movement action
      def move(self): # no implementation of modification
          ## Determine currently empty cells within line of sight
          if self.cell is None: # solves dead agent problem in the sequential
      appraoch
              return
          possibles = [
              cell
              for cell in self.cell.get_neighborhood(self.vision,
      include_center=True)
              if cell.is_empty
          ]
          ''' if all cells around are empty then agent should stay put'''
          if not possibles:
              return # no possible moves so stay in place
          ## compares how much sugar is in each possible movement target/cell
          sugar_values = [
              cell.sugar
              for cell in possibles
          ]
          ''' MOVE picked = picks the best sugary cell (closet+most sugar)'''
          ## Calculate the maximum possible sugar value in possible targets
          max_sugar = max(sugar_values)
          ## Get indices of cell(s) with maximum sugar potential within range
          candidates_index = [
              i for i in range(len(sugar_values)) if math.isclose(sugar_values[i],
      max_sugar)
```

```
            ]
            ## Indentify cell(s) with maximum possible sugar
            candidates = [
                possibles[i]
                for i in candidates_index
            ]
            ## Find the closest cells with maximum possible sugar
            min_dist = min(get_distance(self.cell, cell) for cell in candidates)
            final_candidates = [
                cell
                for cell in candidates
                if math.isclose(get_distance(self.cell, cell), min_dist, rel_tol=1e-02)
            ]
            ## Choose one of the closest cells with maximum sugar (randomly if more
        than one)
            self.cell = self.random.choice(final_candidates)

        ## consume sugar in current cell, depleting it, then consume metabolism
        def gather_and_eat(self):  # no implementation of modification
            if self.cell is None:
                return
            self.sugar += self.cell.sugar # adds sugar in cell to agent's sugar wealth
            self.cell.sugar = 0 # resets cells sugar to zero/removes sugar from cell
            self.sugar -= self.metabolism # subtracts the sugar needed to use energy

        ## If an agent has zero or negative suger, it dies and is removed from the
        model
        def see_if_die(self):  # no implementation of modification
            if self.cell is None:
                return
            if self.sugar <= 0:
                self.remove()
```

model.py

```
Unset
'''
PROPOSED MODEL MODIFICATION: Class-Based Sequential Activation
    - as opposed to simultaneous activation
    - Change in model.py file is changing priority order of sequential activation
      by type
      - model.py sets up world (grid,sugar, creation of agents) and controls the
simulation steps
      - time = one round = one step
```

```
'''
from pathlib import Path # allows access to sugar-map.txt files

import numpy as np

import mesa # agent based modeling framework
from agents import SugarAgent   # import mess agent class from agents.py

## Using experimental cell space for this model that enforces von Neumann
neighborhoods
''' Experimental Mesa Features - agents can only move up/down/left/right; no
diagonals'''
from mesa.experimental.cell_space import OrthogonalVonNeumannGrid

## Use experimental space feature that allows us to save sugar as a property of the
grid spaces
''' Activate cells to be able to hold things like sugar'''
from mesa.experimental.cell_space.property_layer import PropertyLayer

''' sub-class of mesa.Model (from Mesa base Model)'''
class SugarScapeModel(mesa.Model):
       def calc_gini(self):
           agent_sugars = [a.sugar for a in self.agents] # each agent's current sugar
       wealth
           sorted_sugars = sorted(agent_sugars)
           n = len(sorted_sugars) # stores how many agents are active

           # standard formula of the Gini coefficent
           x = sum(el * (n - ind) for ind, el in enumerate(sorted_sugars)) / (n *
       sum(sorted_sugars))
           return 1 + (1 / n) - 2 * x

       ## Define initiation, inherit seed property from parent class
       '''implement modification'''
       def __init__(
           self,
           width = 50,
           height = 50,
           initial_population=200,
           endowment_min=25,
           endowment_max=50,
           metabolism_min=1,
           metabolism_max=5,
           vision_min=1,
           vision_max=5,
           types = [1,2,3], # agent types
```

```
        seed = None
    ):
        super().__init__(seed=seed) # parent Model class
        ## Instantiate model parameters
        self.width = width
        self.height = height
        ## Set model to run continuously
        self.running = True
        ## Create grid

        ''' 2D Grid: Agents can't wrap from one side to the other==> torus = false
    '''
        self.grid = OrthogonalVonNeumannGrid(
            (self.width, self.height), torus=False, random=self.random
        )

        ''' STORE CLASS TYPES'''
        self.types = types

        ## Define datacollector, which calculates current Gini coefficient
        ''' tells mesa to calculate the Gini coefficent after each step'''
        self.datacollector = mesa.DataCollector(
            model_reporters = {"Gini": self.calc_gini},
        )
        ## Import sugar distribution from raster, define grid property
        ''' determines where sugar is located and how much from txt file '''
        self.sugar_distribution = np.genfromtxt(Path(__file__).parent /
    "sugar-map.txt")

        ''' adds sugar map to the grid '''
        self.grid.add_property_layer(
            PropertyLayer.from_data("sugar", self.sugar_distribution)
        )

        ''' Creation of Agents'''
        ''' a list to hold agent groups by type for agent organization'''
        self.sets = []

        ''' iterate by type - creates a batch of agents per class
                - assign random sugar, metabolism, vision, and positions'''
        ''' for each type create (1)upper (2)middle (3)lower class group of
    agents'''
        for type in types:
        ## Create agents, give them random properties, and place them randomly on
    the map
            self.sets.append(SugarAgent.create_agents(
                self,
```

```
            initial_population, # initial cell location

            # random attributes
            self.random.choices(self.grid.all_cells.cells,
k=initial_population),
            sugar=self.rng.integers(
                endowment_min, endowment_max, (initial_population,),
endpoint=True
            ),
            metabolism=self.rng.integers(
                metabolism_min, metabolism_max, (initial_population,),
endpoint=True
            ),
            vision=self.rng.integers(
                vision_min, vision_max, (initial_population,), endpoint=True
            ),
            agent_type = type ##  store agent/class type
        ))
    ## Initialize datacollector
    self.datacollector.collect(self)

## Define step: Sugar grows back at constant rate of 1, all agents move, then
all agents consume, then all see if they die. Then model calculated Gini
coefficient.
def step(self):
    for agents in self.sets:
        # the minimum of : sugar either grown by +1 OR defaults to original
limit
        self.grid.sugar.data = np.minimum(
            self.grid.sugar.data + 1, self.sugar_distribution
        )
        ''' UNIFORMLY AND SIMULTANEOUSLY
        No one has current priority; all agents are treated equal randomly
        self.agents.shuffle_do("move")
        self.agents.shuffle_do("gather_and_eat")
        self.agents.shuffle_do("see_if_die")

        '''
        all_agents = [agent for agent_set in self.sets for agent in agent_set]
# all agent groups (1,2,3) in one list

        all_agents.sort(key=lambda a: a.agent_type) # sort agents by class type
where 1 goes first

        ''' ALTERNATIVLEY
        for agent in all_agents:
            agent.move()
```

```
                agent.gather_and_eat()
                agent.see_if_die()
            '''
            # upper-class agents go first and grab sugar
            for agent in all_agents:
                agent.move()
            for agent in all_agents:
                agent.gather_and_eat()
            for agent in all_agents:
                agent.see_if_die()
        '''
        important to track inequality/gini coefficent over time, plot sugar
distribution and
        save current state of model to analyze trends over time

        '''
        self.datacollector.collect(self)
```