Mögliche Lösungen zu den Aufgaben aus Block 4

Beide Aufgaben beziehen sich auf die Implementierung des Arthur Modells, das in den Video Lectures entwickelt wurde. Ihr könnt entweder eure eigene, oder die von mir vorbereitete Implementierung als Ausgangspunkt verwenden.

In den Hausaufgaben geht es vor allem darum, die Implementierung nachzuvollziehen. Dann sollt ihr die folgenden beiden Erweiterungen implementieren und die Effekte entsprechend visualisieren.

Dazu könnt ihr entweder neue subplots zu der vorgeschlagenen Visualisierung hinzufügen, eine neue Form der Visualisierung erstellen, oder einen der alten Fälle in der alten Visualisierung ersetzen.

Erweiterung des Arthur Modells I

> Erstellt eine dritte Art der Nachbarschaft: hier sollen für jeden Agenten vier andere Agenten zufällig als Nachbard ausgewählt werden.

Hinweis 1: die Auswahl der Agenten kann durch die `numpy.random.choice` *Funktion implementiert werden. Allerdings müsst ihr sicherstellen, dass kein Agent zweimal gezogen wird, und dass kein Agent in seiner eigenen Nachbarschaft ist.*

Hinweis 2: In dieser Implementierung soll für einen Agent i, der Nachbar von Agent j ist, nicht automatisch gelten, dass Nachbar j ein Nachbar von Agent i ist. Der Nachbarschaftsbegriff ist hier also weit: es kann auch bedeuten, dass sich Agent i an Agent j orientiert, aber nicht umgekehrt (z.B. weil j ein Star, und i ein follower ist).

Hinweise zur Lösung

Das Modell muss hierfür nicht fundamental geändert werden. Die folgenden Änderungen müssen durchgeführt werden:

- In der `__init__` Funktion der Klasse `Model` muss die Möglichkeit zugelassen werden, dass die Nachbarschaft `random` spezifiziert wird.

```
class Model:
    """...
    """

    def __init__(self, n_agents, neighborhood_str, identifier):
        """...
        """
        # No change
        assert neighborhood_str in ["ring", "full", "random"], "No correct neighborhood \
        structure specified. Currently allowed: 'full' or 'ring' or 'random', but not {}"\
        .format(neighborhood_str)
        self.neighborhood = neighborhood_str
```

- Die Funktion `create_neighborhood_structure` muss ergänzt werden. Folgender Code wäre möglich:

```
class Model:
    """...
    """

    def __init__(self, n_agents, neighborhood_str, identifier):
        """...
        """
    # No change

    def create_neighborhood_structure(self):
        """...
        """
        if self.neighborhood == "ring":
            print("Creating ring neighborhood...", end="")
            # No change

        elif self.neighborhood == "full":
            print("Creating full neighborhood...", end="")
            # No change
```

```
        elif self.neighborhood == "random":
            print("Creating random neighborhood...", end="")
            for a in range(len(self.agentlist)):
                neighborhood_choice_list = self.agentlist.remove(self.agentlist[a])
                self.agentlist[a].add_neighbors(np.random.choice(self.agentlist, size=4, rep
```

Hierbei sind zwei Punkte zu beachten: * Wir müssen verhindern, dass der Agent sich selbst zu seiner Nachbarschaft hinzufügt indem man einer Variable `neighborhood_choice_list` erstellt, die alle Agenten außer dem Agenten selbst enthält * Wir müssen verhindern, dass ein Agent doppelt für die Nachbarschaft gezogen wird, das heißt `np.random.choice` muss mit dem Argumten `replace=False` aufgerufen werden.

- Zuletzt muss noch die Klasse `Main` angepasst werden, indem auch Modellinstanzen mit einer zufälligen Nachbarschaft erstellt werden. Entsprechend muss das auch im `neighborhood_dict` berücksichtigt werden:

```
class Main:
    """...
    """

    def __init__(self, nb_agents, neighborhood_dict, outcome_filename):
        # No change
        self.nb_simulations = (self.neighborhood_dict["full"] +
                               self.neighborhood_dict["ring"] +
                               self.neighborhood_dict["random"])
        # No change
        for i in range(self.neighborhood_dict["random"]):
            current_model = model.Model(self.nb_agents, "random", self.current_id)
        # No change

neigh_dict = {"full" : 20, # Nb of simulation runs with complete neigborhood
              "ring" : 20, # Nb of simulation runs with ring neigborhood
              "random" : 20}
```

Erweiterung des Arthur Modells II

Verändert das Modell dahingehend, dass 50% der Agenten eine strikte Präferenz für Technologie 0 und 50% eine strikte Präferenz für Technologie 1 haben.

Hinweis: Eine Strikte Präferenz zeigt sich durch einen von Anfang an fixierten Wert für `agent.pref_tech_0`*: für die Hälfte der Agenten soll dieser Wert* 0.25 *sein, für die andere Hälfte* 0.75*.*

Hinweise zur Lösung

Hier muss zunächst eine Änderung in der Klasse `Agent` vorgenommen werden: wir müssen ein zusätzliches Argument in der `__init__` Funktion hinzufügen, das spezifiziert ob der Agent entweder Technologie 0 oder Technologie ein setzt.

Entsprechend muss dann die Funktion `tec_choice` leicht angepasst werden.

```
class Agent():
    """...
    """

    def __init__(self, pref_tech):
        self.neighborhood = set()
        self.tech_chosen = None
        if pref_tech == 0:
            self.pref_tech_0 = 0.25
        else:
            self.pref_tech_0 = 0.75

    # No change

    def tec_choice(self):
        """...
        """
        # No change

        # make technology choice
        if len(techs_neighborhood) > 0:
            if self.pref_tech_0 < share_t1:
                self.tech_chosen = 1
            else:
                self.tech_chosen = 0
        else:
            if self.pref_tech_0 < 0.5 :
                self.tech_chosen = 1
            else:
                self.tech_chosen = 0
```

Zuletzt muss dann noch sichergestellt werden, dass 50% der Agenten mit Präferenz für Tech 0 und 50% mit Präferenz für Tech 1 erstellt werden. Das bedarf einer Änderung in der Klasse `Model`:

```
class Model:
    """...
    """

    def __init__(self, n_agents, neighborhood_str, identifier):
        """...
        """
        self.identifier = identifier
        n_agents_pref_0 = int(n_agents / 2)
        n_agents_pref_0 = n_agents - n_agents_pref_0
        ag_pref_0 = [agent.Agent(0) for i in range(n_agents_pref_0)]
        ag_pref_1 = [agent.Agent(1) for i in range(n_agents_pref_1)]
        self.agentlist = ag_pref_0 + ag_pref_1
```