CMCS 24-25

SOCIAL NETWORKS AND INTRASPEAKER VARIATION DURING PERIODS OF LANGUAGE CHANGE IN MESA

SAUL URSO - 614886

THE PROBLEM

MAIN QUESTION

There are **occurring characteristics of language change** at both the level of linguistic communities as well as individual speakers.

Question: What are the properties of language users such that we can account for these characteristics?

SOLUTION

A **computational model** of a social network of language users

EXAMPLE

Periphrastic *do* (or do support) in English:

- Prior to about 1400:

1. ...whiche he perceiueth not.

- Then between 15th-18th century **both** older form and the modern form (periphrastic do) **were available**:

1. I question not your friendship...
2. She does not deserve it...

THE MAIN CHARACTERISTICS

S-shaped curve: The time course of the change follows an S-shaped curve- change happened slowly at first, then proceeded very rapidly before slowing down again.

Intraspeaker variation: As a new form spreads, there is a gradual change from using the older form to the new one.

Categorical norms: When two syntactic variants are in competition, speakers often move toward categorically using just one of the competing variants.

Multistability: Language change can have multiple stable outcomes .

Threshold problem: what bias does the rarer form need in order to surpass the more common form?

APPROACHES

INDIVIDUAL

- Discrete grammars
- Choose one grammar from a neighbor

THRESHOLD

- Discrete grammars
- Majority among neighbors

REWARD

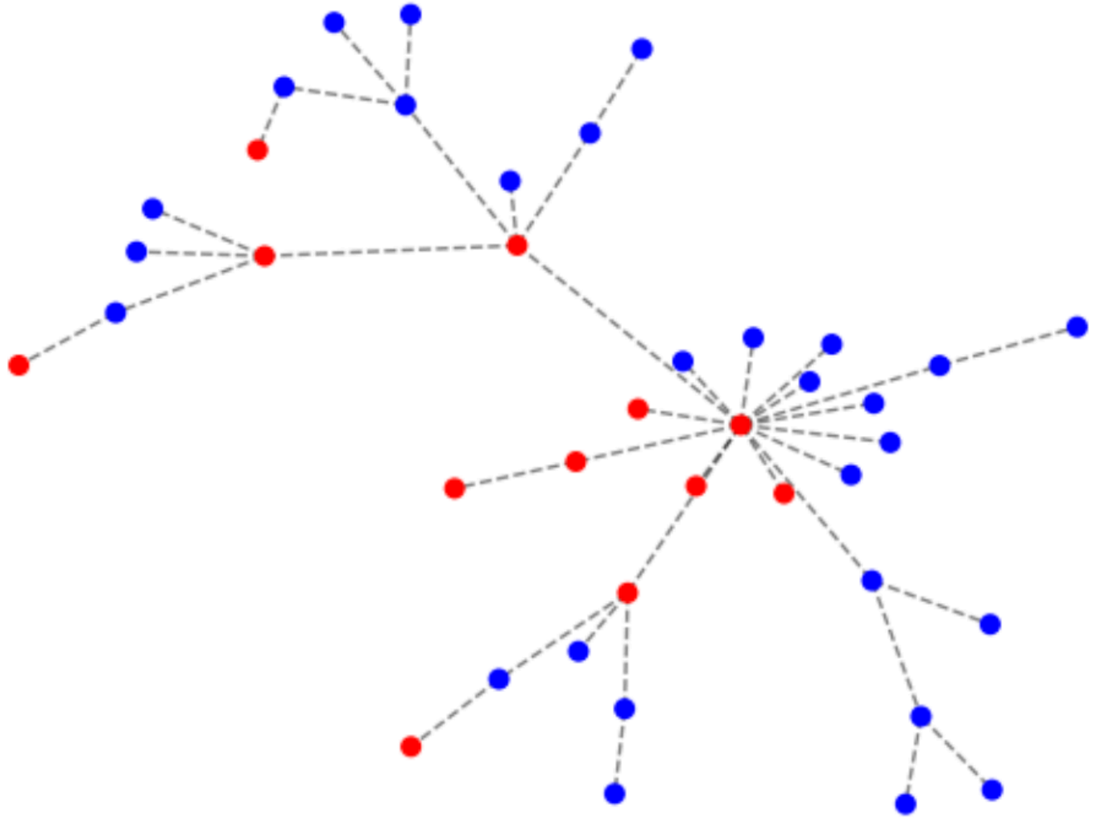
- Continuous states
- Use a grammar with certain probability

EXTENSIONS

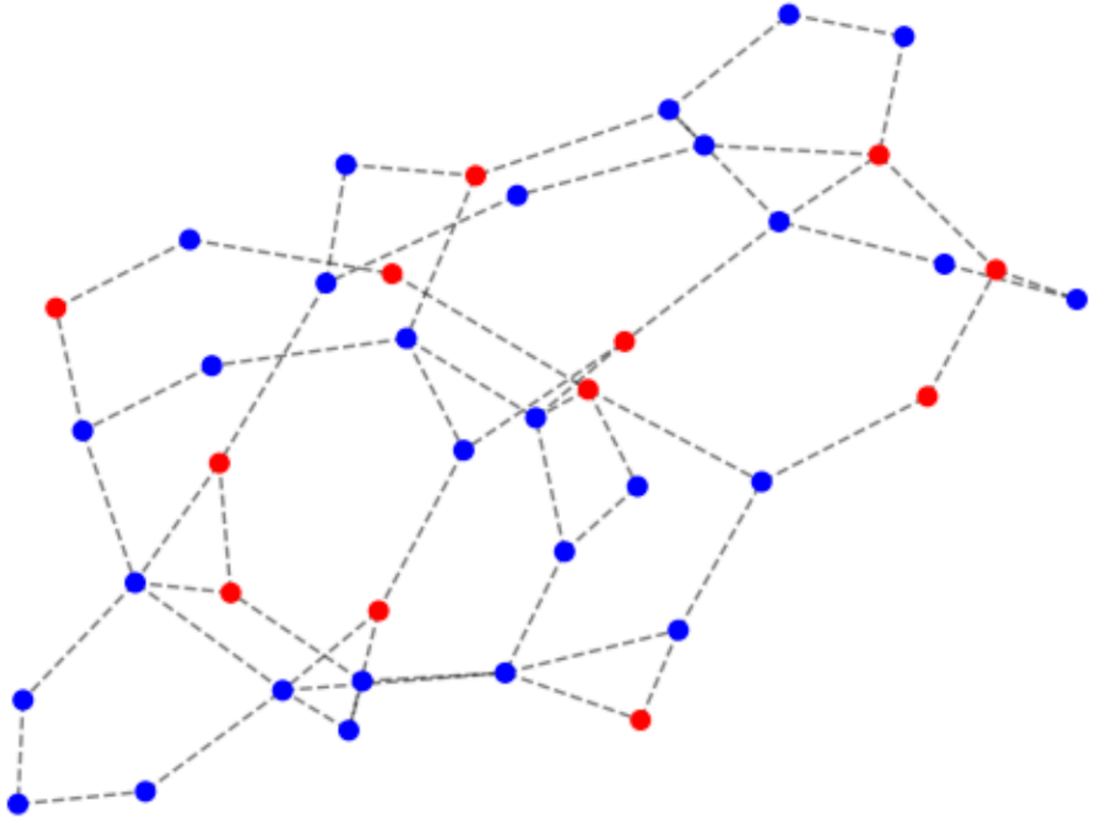
- Different topologies
- Multiple grammars

SCALE-FREE VS SMALL-WORLD GRAPHS

BARABASI-ALBERT



NEWMAN-WATTS-STROGATZ



LANGUAGE MODEL AND AGENT IN MESA

```python
super().__init__(seed=seed, rng=rng)

graph = make_graph(
    graph_type,
    num_nodes,
    seed=int(graph_seed),
    **kwargs)

self.grid = Network(
    graph,
    capacity=1,
    random=self.random)
```

```python
# data collection
self.datacollector = mesa.DataCollector(
    model_reporters={
        "Mean State": compute_average_state,
    },
    agent_reporters={"State": "state"},
)

# agent creation
LanguageAgent.create_agents(
    self,
    num_nodes,
    agent_states,
    update_algorithm,
    threshold_val,
    sink_state_1,
    logistic,
    alpha,
    list(self.grid.all_cells),
)
```

```python
def step(self):
    self.datacollector.collect(self)
    self.agents.shuffle_do("step")
```

```python
class LanguageAgent(FixedAgent):

    def __init__(
        self,
        model,
        initial_state,
        update_algorithm,
        threshold_val,
        sink_state_1,
        logistic,
        alpha,
        cell,
    ):
        super().__init__(model)

        self.cell = cell
        self.update_algorithm = update_algorithm
        self.state = initial_state

        # for threshold
        self.threshold_val = threshold_val
        self.sink_state_1 = sink_state_1

        # for reward
        self.logistic = logistic
        self.alpha = alpha
```

EXPERIMENTAL SETUP

Experiments executed on graphs of **20, 40 and 80 nodes,** to observe if varying the size would change the behavior in some manner

For Barabasi-Albert:

- m = 1 (amount of edges of incoming nodes)

For Newman-Watts-Strogatz:

- k = 2 (ring topology)
- p = 0.4 (probability of adding an edge for each existing edge)

All experiments were executed on **100 different graphs** for each # of nodes and each algorithm configuration

ABOUT THE GRAMMARS

We always assume a scenario with a **minority grammar** adopted by only 30% of the nodes

We compare scenarios where some agents are uncertain between which grammar are using

We start by observing the **behavior on two grammars**, and then proceed to observe what happens with **three grammars.**

EXPERIMENTS ON 2 GRAMMARS

RANDOM NEIGHBOR UPDATE ALGORITHM

IDEA: each agent adopts the grammar used by one of its neighbors randomly

Characteristics

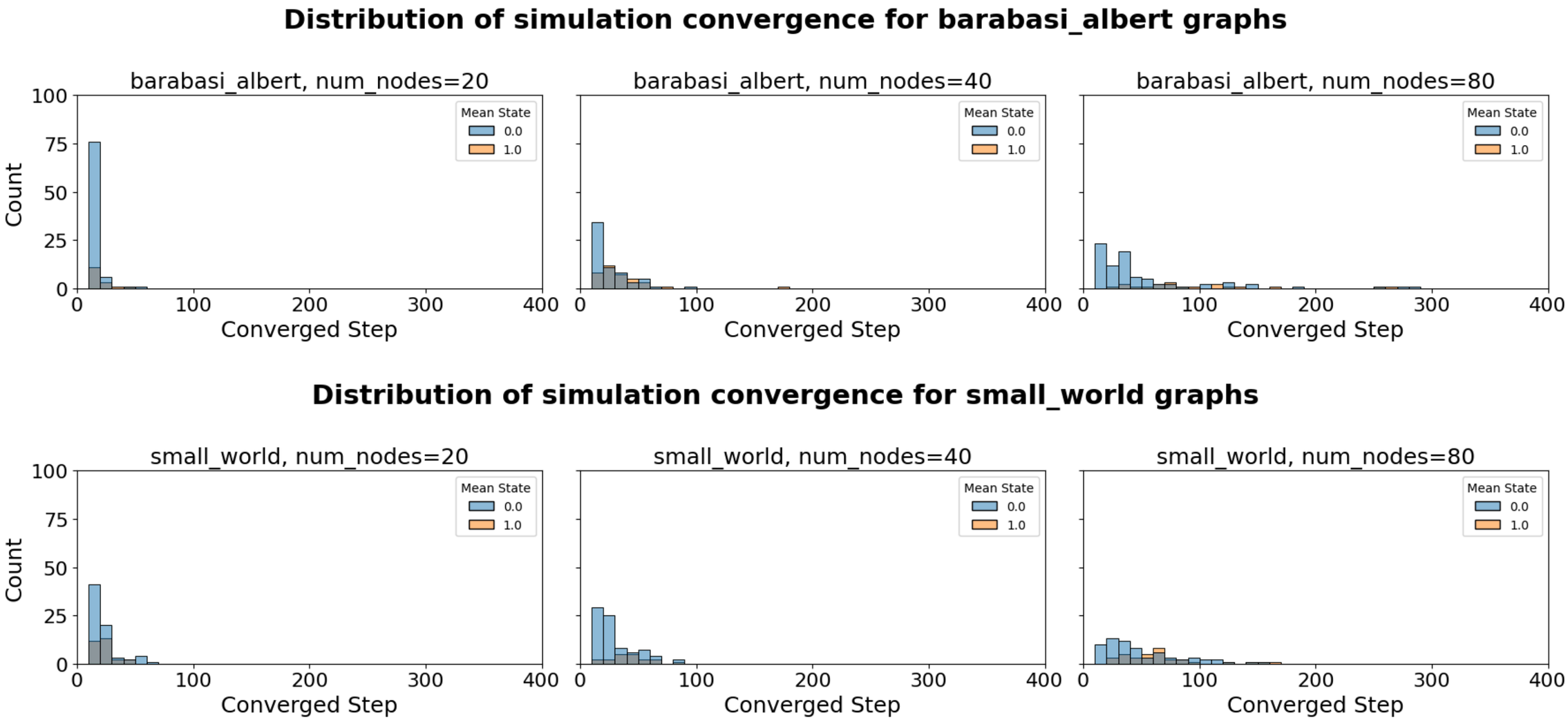
Straighforward idea and implementation

Uses discrete states

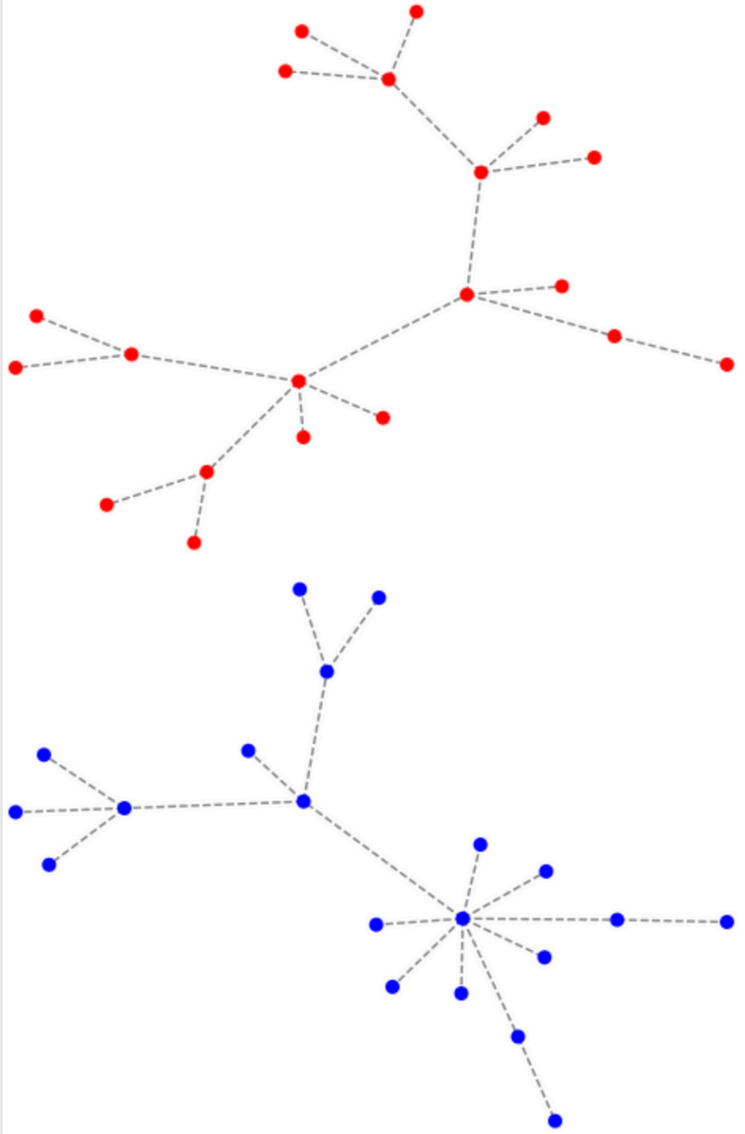
Lack of parameters

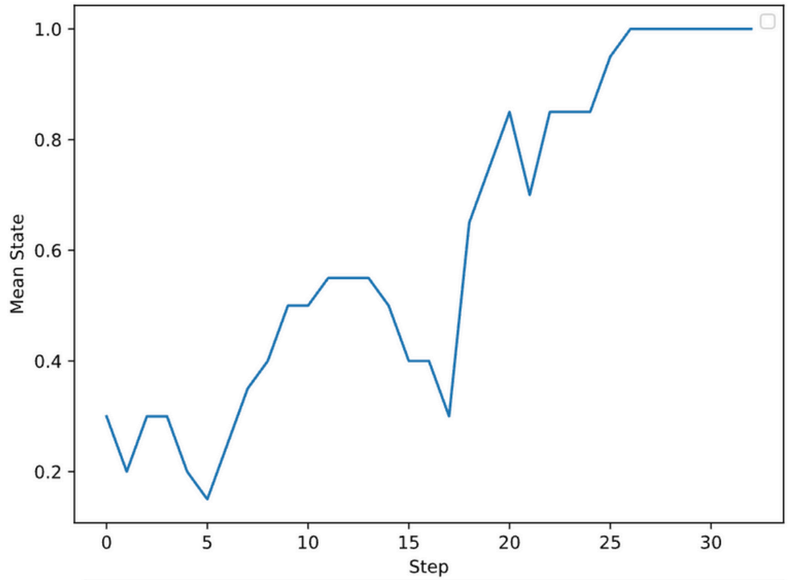
```
def adopt_rand_neighbor_grammar(self):
    neigh_cell = self.cell.neighborhood.select_random_cell()
    neigh = neigh_cell.agents[0]
    self.state = neigh.state
```

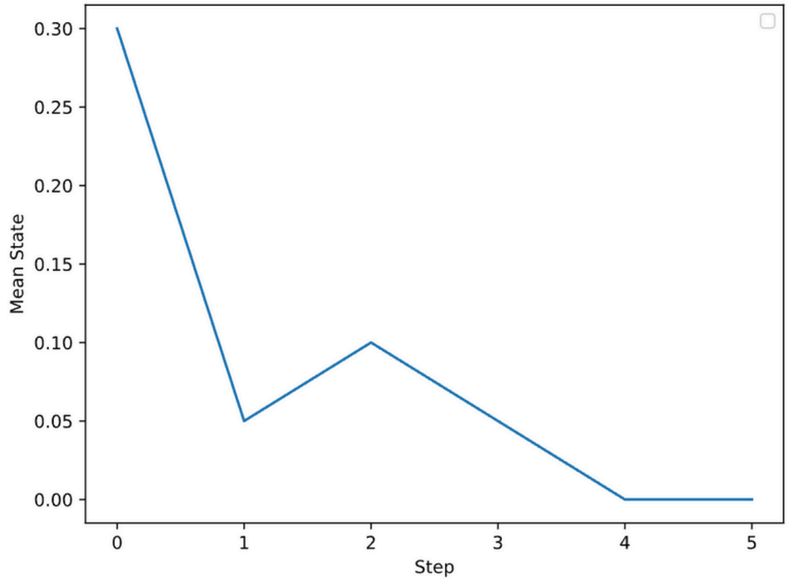
CONVERGENCE SPEED PER STATE (INDIVIDUAL)



TYPICAL RUN FOR INDIVIDUAL UPDATE ALGORITHM







THRESHOLD UPDATE ALGORITHM

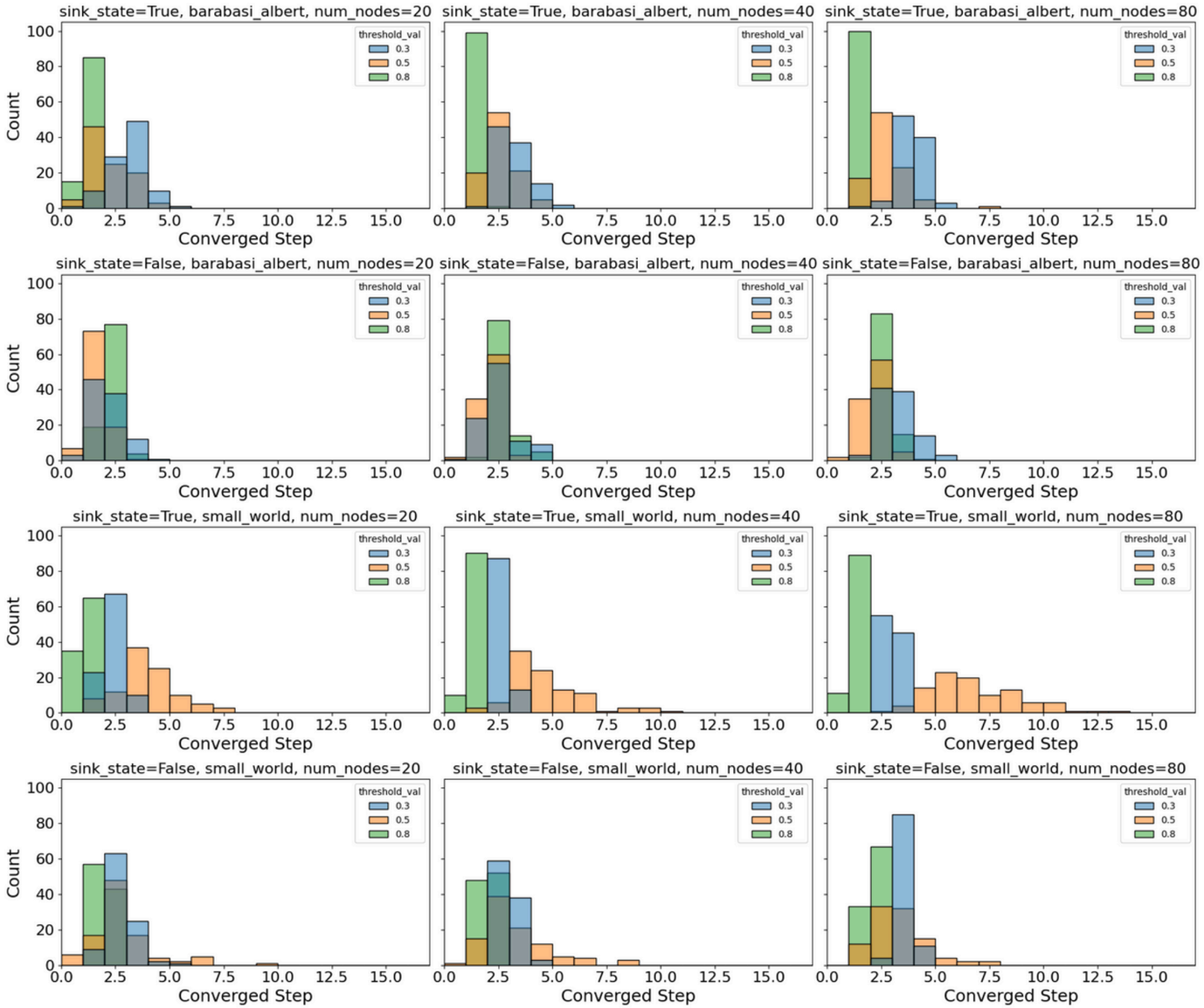
IDEA: Instead of a random neighbor, adopt a grammar if at least a certain percentage of your neighbors use it.

Characteristics

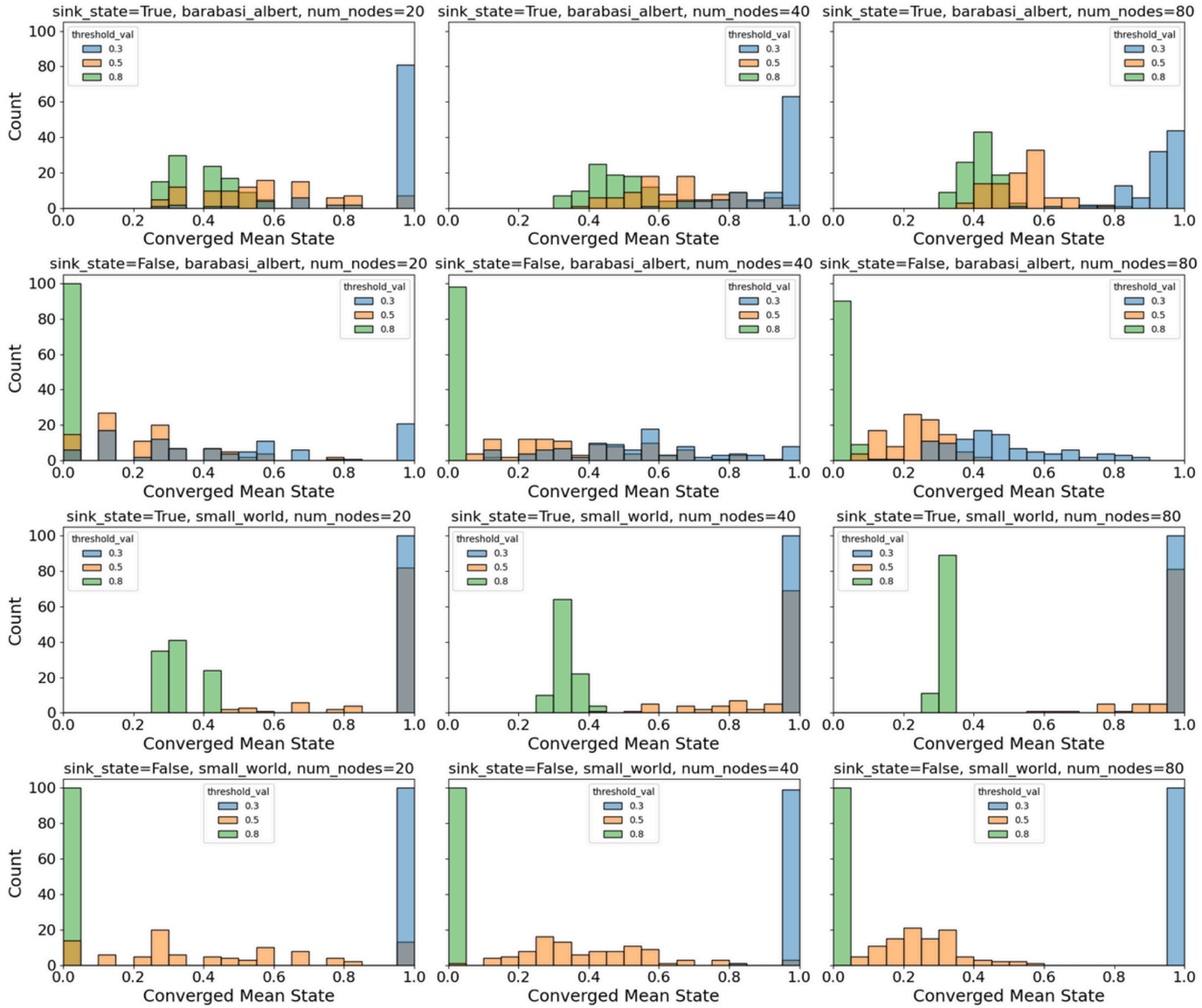
- Uses discrete states
- Two parameters:
  - The threshold of the minority state
  - The sink state

```
def adopt_threshold_grammar(self):
    neighbors = [n for n in self.cell.neighborhood.agents]
    grammar_1_neighs = [n for n in neighbors if n.state == 1]

    if len(grammar_1_neighs) / len(neighbors) >= self.threshold_val:
        self.state = 1
    else:
        if not self.sink_state_1:
            self.state = 0
```

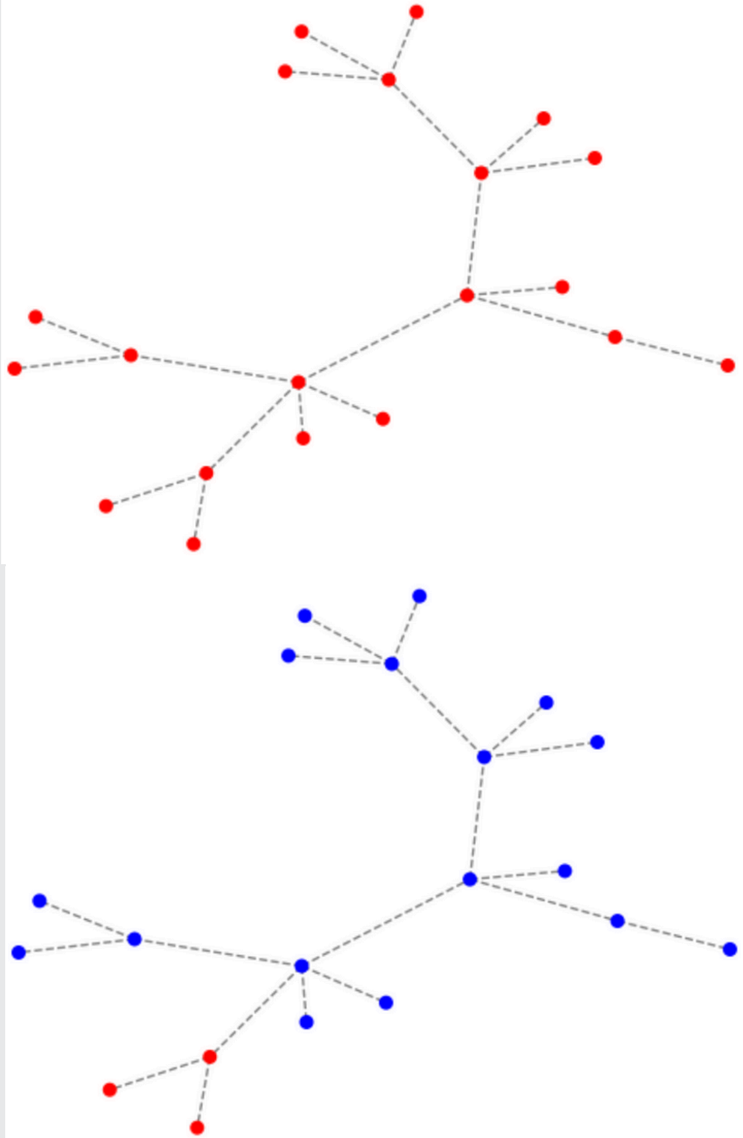
CONVERGENCE SPEED DISTRIBUTION PER THRESHOLD

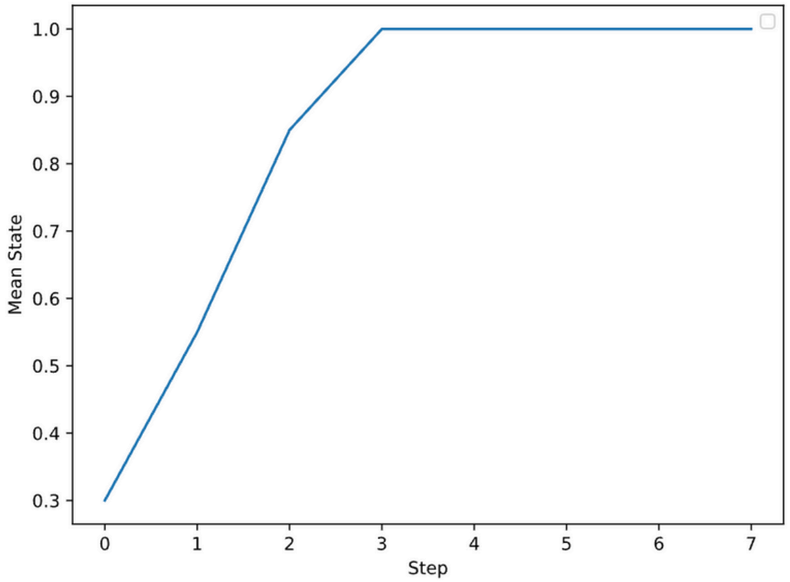


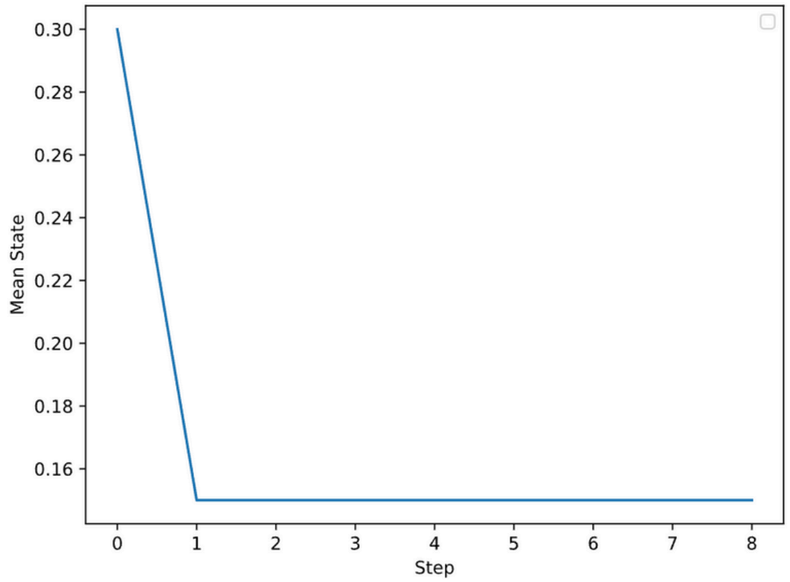
FINAL STATE DISTRIBUTION PER THRESHOLD



TYPICAL RUN FOR THRESHOLD UPDATE ALGORITHM







REWARD UPDATE ALGORITHM

IDEA: Instead of choosing one grammar, use one of the two grammars with a certain probability

Characteristics

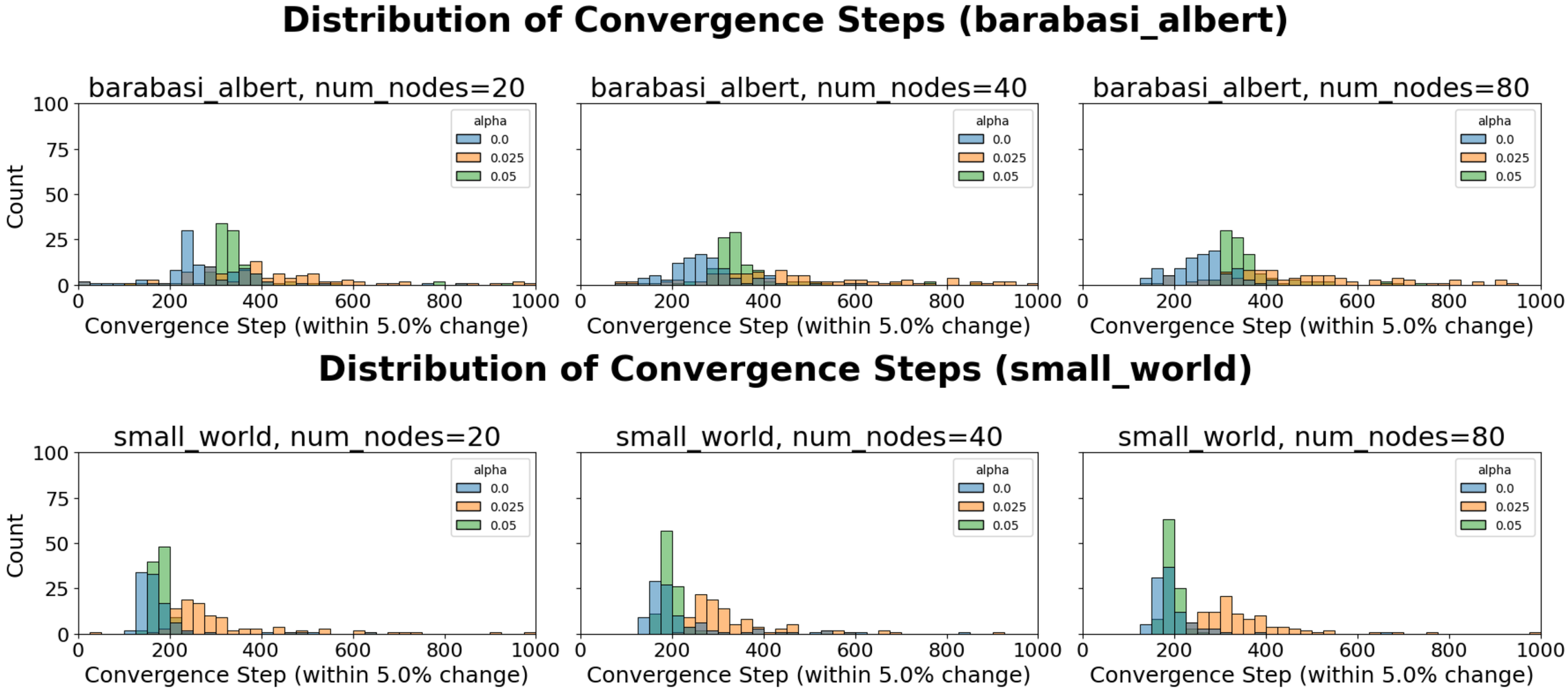
- Uses continuous states
- Parameters:
  - **Alpha**, which controls the bias of the minority state when speaking

```
elif self.update_algorithm == "reward":
    spoken_state = self.speak()
    for n in self.cell.neighborhood.agents:
        n.listen(spoken_state)
```

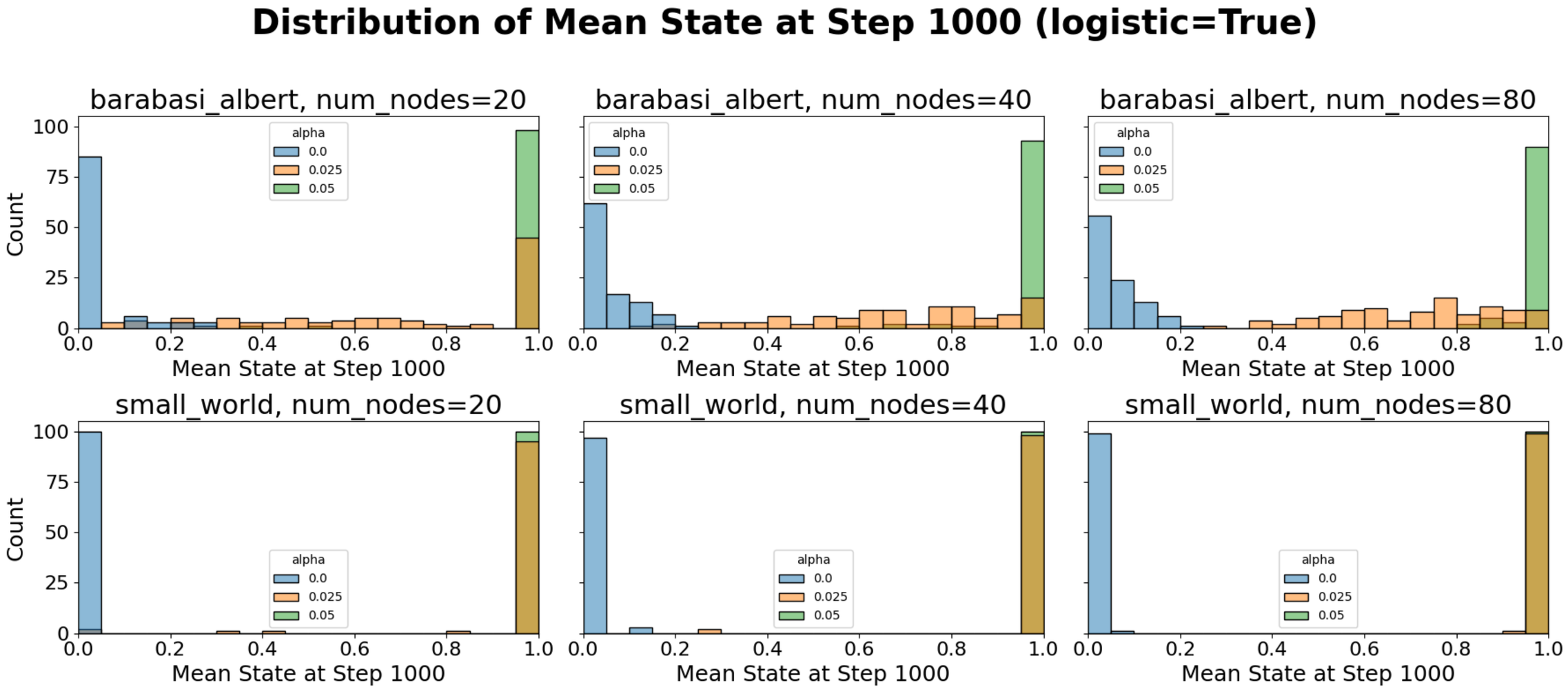
```
def speak(self):
    if self.logistic:
        gain = (self.alpha + 0.1) * 20
        filter_val = expit((gain * self.state - 1) * 5)
        if self.model.random.random() <= filter_val:
            spoken_state = 1
        else:
            spoken_state = 0

    else:
        biased_val = 1.5 * self.state
        if biased_val >= 1:
            biased_val = 1

        if self.model.random.random() <= biased_val:
            spoken_state = 1
        else:
            spoken_state = 0

    return spoken_state
```

```
def listen(self, heard_state):
    gamma = 0.01
    prob = self.model.random.random()

    if prob <= self.state:
        self.state += gamma * (heard_state - self.state)
    else:
        self.state = gamma * heard_state + (1 - gamma) * self.state
```

CONVERGENCE SPEED PER ALPHA (REWARD)

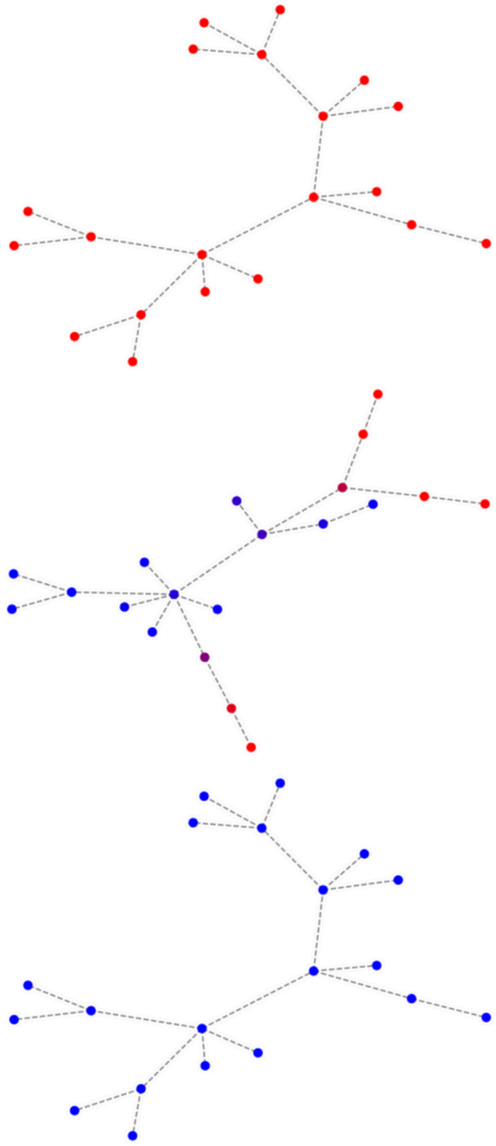


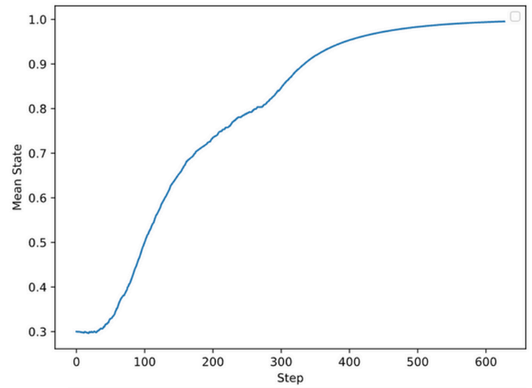
FINAL STATE PER ALPHA (REWARD)

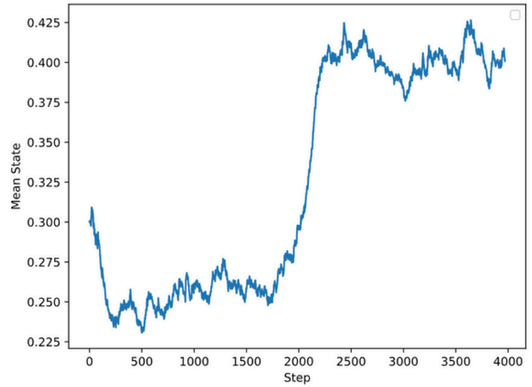


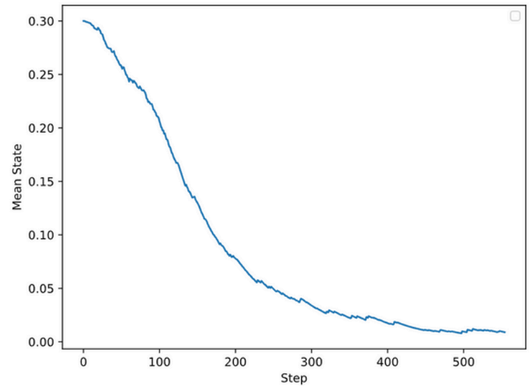
TYPICAL RUN FOR REWARD UPDATE ALGORITHM

In some cases, not all the agents converge to either 0 or 1: this signifies the **emergence of dialects**



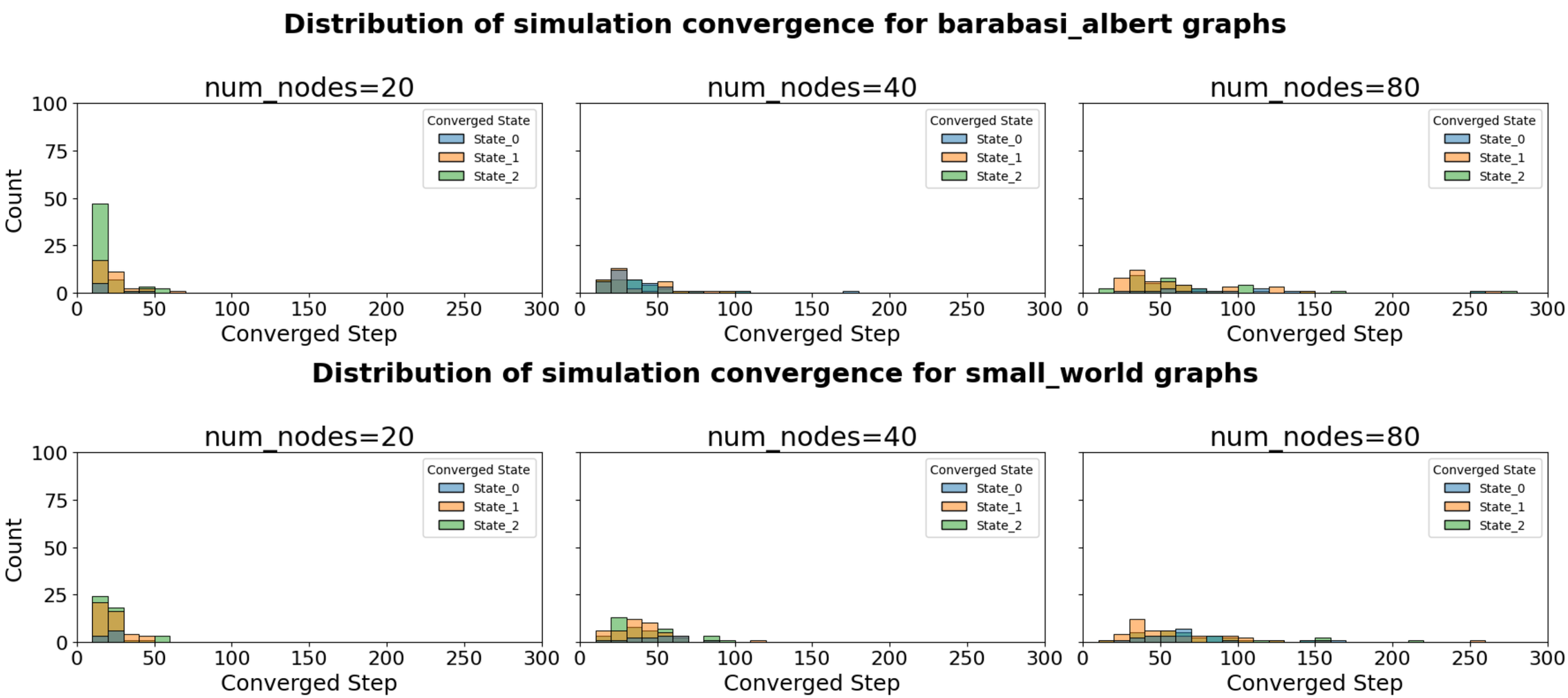




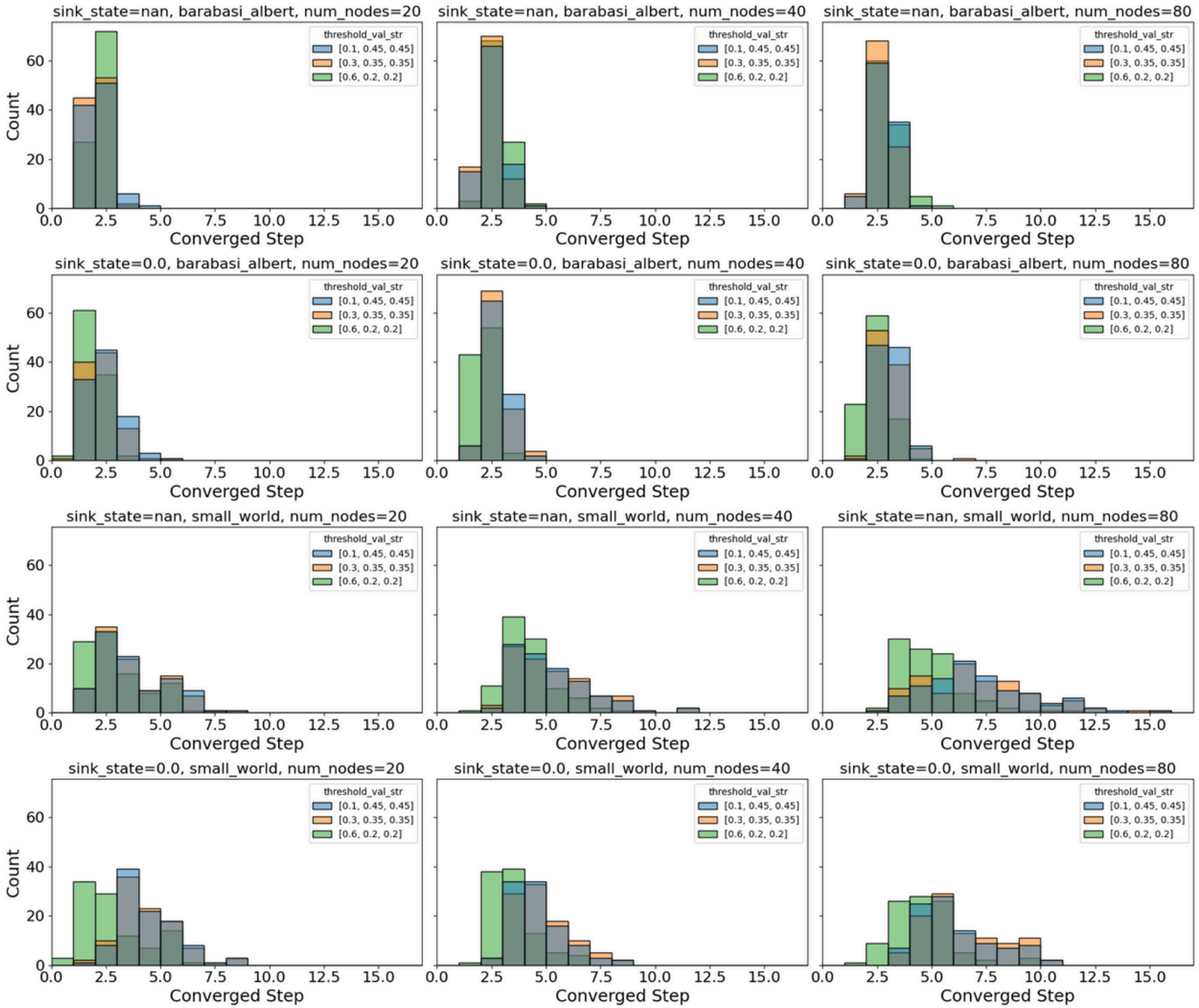


EXPERIMENTS ON 3 GRAMMARS

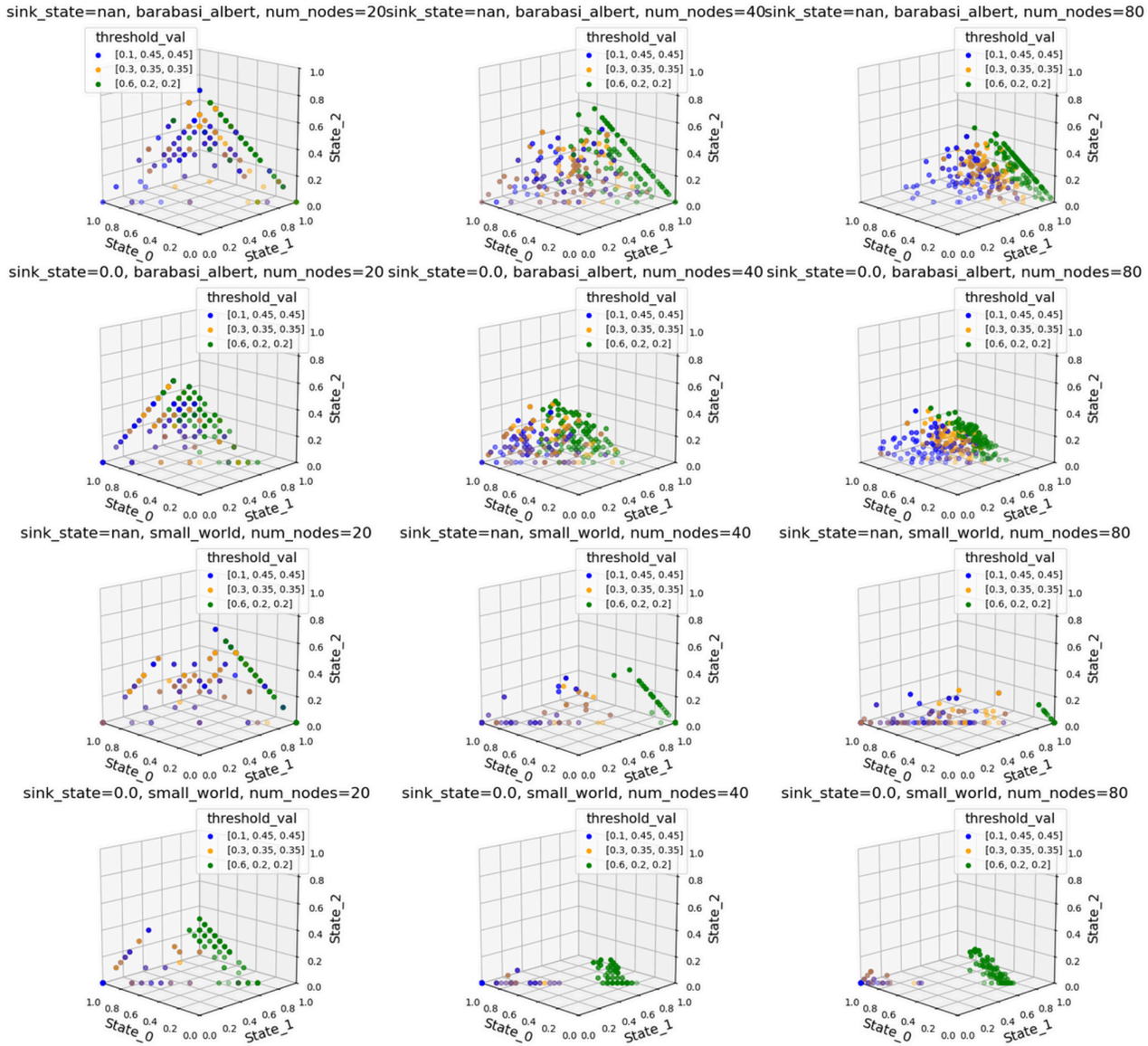
CONVERGENCE SPEED PER STATE (INDIVIDUAL)



CONVERGENCE SPEED DISTRIBUTION PER THRESHOLD



FINAL STATE DISTRIBUTION PER THRESHOLD



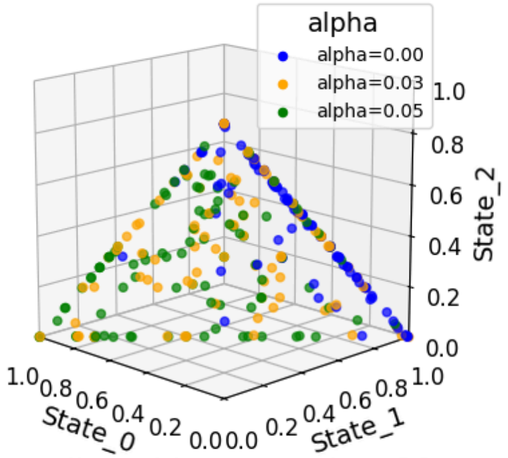
REWARD ALGORITHM (N GRAMMARS VER.)

```python
def speak(self):
    # Use softmax_with_bias, biasing toward sink_state_idx if set, else no bias
    if self.sink_state_idx is not None:
        preferred_index = self.sink_state_idx
    else:
        raise ValueError("sink_state_idx must be specified.")
    chosen_state, probabilities = softmax_with_bias(
        self.state, self.alpha, preferred_index
    )
    spoken_state = np.zeros(self.N)
    spoken_state[chosen_state] = 1
    return spoken_state

def listen(self, heard_state):
    gamma = 0.01
    extracted_idx = self.model.random.choices(range(self.N), weights=self.state)[0]
    heard_idx = np.argmax(heard_state)
    if extracted_idx == heard_idx:
        delta = gamma
    else:
        delta = -gamma

    self.state[extracted_idx] += delta
    for i in range(self.N):
        if i != extracted_idx:
            self.state[i] -= delta / (self.N - 1)

    self.state = np.clip(self.state, 0, 1)
    total = self.state.sum()
    self.state /= total
```

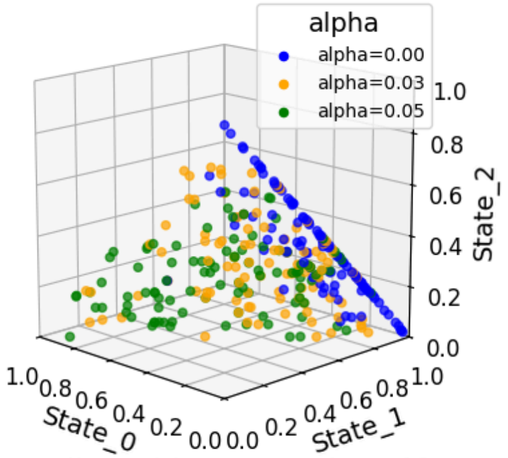
```python
def softmax_with_bias(state, alpha, preferred_index):
    epsilon = 0.1
    gain = (abs(alpha) + epsilon) * 30  # N * 10

    N = len(state)

    # Calculate scores with gain and bias
    scores = []
    for i, s in enumerate(state):
        bias = 1 / N - 1 / gain  # 1 / N
        bias *= 1 if i == preferred_index else (-1 / (N - 1))  # -1/(N-1)
        score = (s + bias) * (gain / N * 2) # * 5
        scores.append(score)

    # Compute softmax probabilities
    exp_scores = np.exp(scores)
    probabilities = exp_scores / np.sum(exp_scores)

    # Sample one category
    chosen_state = np.random.choice(3, p=probabilities)

    return chosen_state, probabilities.tolist()
```

FINAL STATE PER ALPHA (REWARD)

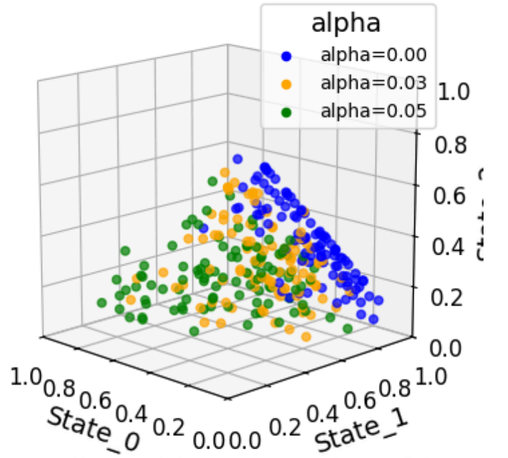
barabasi_albert, num_nodes=20



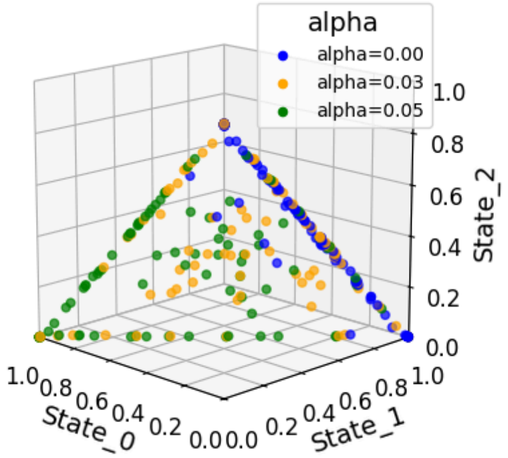
barabasi_albert, num_nodes=40



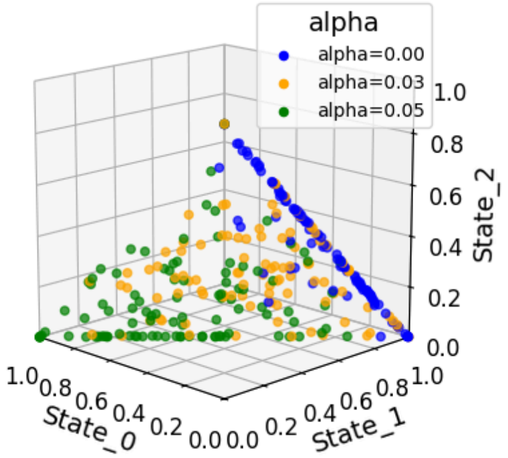
barabasi_albert, num_nodes=80



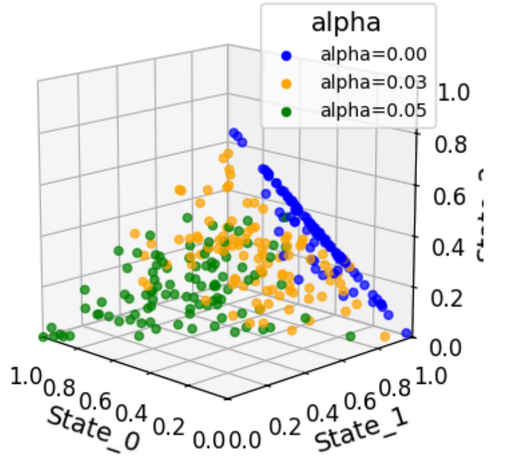
small_world, num_nodes=20



small_world, num_nodes=40



small_world, num_nodes=80



FUTURE WORKS

DIRECTED EDGES

Examine how directed edges influence the spreading of grammars in a graph

SYNCHRONOUS UPDATES

Examine the behaviour of grammar spreading when the speaking and listening are synchronized

THANK YOU