import math, random

from agent import Agent

class QuantumHarmonicDynamics:

"""

Simulates the multi-agent synchronization with quantum-inspired behavior.

Manages entanglement links between agents and updates their states over time applying:

- Memory influence and intent modulation for each agent.

- Coupling (entanglement) between agents to induce synchronization (resonance).

- Quantum tunneling effects for sudden state changes.

- Decoherence to occasionally remove entanglements.

"""

def \_\_init\_\_(self):

self.agents = [] # List of Agent objects in the simulation

self.entangled\_pairs = [] # List of tuples (agentA, agentB) representing entangled agent pairs

# Parameters to control the dynamics:

self.coupling\_strength = 0.1 # How strongly entangled agents pull each other (0 = no coupling, 1 = immediate sync)

self.tunneling\_prob\_factor = 5.0 # Factor for tunneling probability decay; higher = lower tunneling chance for same diff

self.decoherence\_rate = 0.02 # Probability per update step that an entangled pair decoheres (link breaks)

def add\_agent(self, agent: Agent):

"""

Add an Agent to the simulation.

"""

self.agents.append(agent)

def entangle(self, agent1: Agent, agent2: Agent):

"""

Entangle two agents, establishing a coupling between their states.

"""

if agent1 is agent2:

return # cannot entangle an agent with itself

# Avoid duplicate entanglement entries

for (a, b) in self.entangled\_pairs:

if (a is agent1 and b is agent2) or (a is agent2 and b is agent1):

return # already entangled

self.entangled\_pairs.append((agent1, agent2))

agent1.entangled\_with.append(agent2)

agent2.entangled\_with.append(agent1)

def disentangle(self, agent1: Agent, agent2: Agent):

"""

Remove entanglement between two agents, if it exists.

"""

# Remove from entangled\_pairs list

self.entangled\_pairs = [pair for pair in self.entangled\_pairs

if not ((pair[0] is agent1 and pair[1] is agent2) or

(pair[0] is agent2 and pair[1] is agent1))]

# Remove references in each agent's entangled\_with list

if agent2 in agent1.entangled\_with:

agent1.entangled\_with.remove(agent2)

if agent1 in agent2.entangled\_with:

agent2.entangled\_with.remove(agent1)

def update(self):

"""

Advance the simulation by one time step:

1. Each agent retrieves a resonant memory and modulates its own intent/emotion.

2. All entangled pairs are then harmonized (coupled) toward each other (synchronization).

3. Apply quantum tunneling: some agents may make sudden jumps in intent if large differences remain.

4. Apply decoherence: randomly break some entanglements (and slightly reset those agents toward baseline).

"""

# 1. Memory influence on each agent

for agent in self.agents:

mem = agent.recall\_memory()

if mem:

# Modulate intent based on memory's emotional valence and intensity.

# A positive memory increases intent, a negative memory decreases it.

delta = mem.valence \* 0.1 \* mem.intensity

agent.modulate\_intent(delta)

# Adjust agent's current emotion toward the memory's emotion.

agent.adjust\_emotion(mem.valence)

# 2. Coupling for entangled agents (synchronization/resonance)

for (a, b) in list(self.entangled\_pairs): # use list(...) to avoid issues if we modify entangled\_pairs during loop

# Calculate differences in state

intent\_diff = a.intent - b.intent

emo\_diff = a.emotion\_valence - b.emotion\_valence

# Pull both intents toward each other (diffusive coupling)

a.intent -= intent\_diff \* self.coupling\_strength

b.intent += intent\_diff \* self.coupling\_strength

# Pull both emotional valences toward each other

a.emotion\_valence -= emo\_diff \* self.coupling\_strength

b.emotion\_valence += emo\_diff \* self.coupling\_strength

# After coupling, update qualitative emotion tags based on new valences

if a.emotion\_valence > 0.1:

a.emotion\_tag = "positive" if a.emotion\_valence >= 0.5 else "slightly positive"

elif a.emotion\_valence < -0.1:

a.emotion\_tag = "negative" if a.emotion\_valence <= -0.5 else "slightly negative"

else:

a.emotion\_tag = "neutral"

if b.emotion\_valence > 0.1:

b.emotion\_tag = "positive" if b.emotion\_valence >= 0.5 else "slightly positive"

elif b.emotion\_valence < -0.1:

b.emotion\_tag = "negative" if b.emotion\_valence <= -0.5 else "slightly negative"

else:

b.emotion\_tag = "neutral"

# 3. Quantum tunneling effect: allow sudden jump if large disparity remains

# Recalculate difference magnitude after coupling adjustment

intent\_diff = abs(a.intent - b.intent)

if intent\_diff > 1e-6: # any non-zero difference

# Probability decays exponentially with difference

# (Small differences => prob ~1, large differences => prob ~0)

prob = math.exp(-self.tunneling\_prob\_factor \* intent\_diff)

if random.random() < prob:

# We trigger a tunneling event: the agent with lower intent jumps closer to the higher's intent

if a.intent > b.intent:

# b is behind a; bring b up by closing half the gap

b.intent += intent\_diff \* 0.5

else:

# a is behind b

a.intent += intent\_diff \* 0.5

# 4. Decoherence: randomly break entanglements

for (a, b) in list(self.entangled\_pairs):

if random.random() < self.decoherence\_rate:

# Break the entanglement link

self.disentangle(a, b)

# After decoherence, gently steer each agent back toward its baseline intent and neutral emotion

if hasattr(a, 'base\_intent'):

a.intent = (a.intent + a.base\_intent) / 2.0

if hasattr(b, 'base\_intent'):

b.intent = (b.intent + b.base\_intent) / 2.0

# Dampen emotions (assuming environment measurement causes loss of emotional intensity)

a.emotion\_valence \*= 0.5

b.emotion\_valence \*= 0.5

# Update emotion tags post-decoherence

if a.emotion\_valence > 0.1:

a.emotion\_tag = "positive" if a.emotion\_valence >= 0.5 else "slightly positive"

elif a.emotion\_valence < -0.1:

a.emotion\_tag = "negative" if a.emotion\_valence <= -0.5 else "slightly negative"

else:

a.emotion\_tag = "neutral"

if b.emotion\_valence > 0.1:

b.emotion\_tag = "positive" if b.emotion\_valence >= 0.5 else "slightly positive"

elif b.emotion\_valence < -0.1:

b.emotion\_tag = "negative" if b.emotion\_valence <= -0.5 else "slightly negative"

else:

b.emotion\_tag = "neutral"