import time

class MemoryItem:

"""

Represents a single memory with content and an associated emotion.

Attributes:

content (str): The actual memory content or description.

emotion\_tag (str): Label for the emotion associated with this memory (e.g. "happy", "sad").

intensity (float): Importance or strength of the memory (default 1.0, higher means more salient).

valence (float): Numeric emotional valence derived from the emotion\_tag (positive for pleasant emotions,

negative for unpleasant ones, range roughly -1.0 to 1.0).

timestamp (float): Time when the memory was created (for recency tracking).

"""

# Mapping from common emotion tags to a valence value (this can be expanded or refined)

EMOTION\_VALENCES = {

"happy": 0.8, "joy": 0.8, "excited": 0.7, "love": 0.9,

"sad": -0.8, "fear": -0.9, "angry": -0.7, "anger": -0.7, "anxious": -0.4,

"neutral": 0.0, "positive": 0.5, "negative": -0.5

}

def \_\_init\_\_(self, content: str, emotion\_tag: str, intensity: float = 1.0):

self.content = content

self.emotion\_tag = emotion\_tag

self.intensity = intensity

# Determine the valence from the tag, defaulting to 0.0 if unknown tag

if emotion\_tag in MemoryItem.EMOTION\_VALENCES:

self.valence = MemoryItem.EMOTION\_VALENCES[emotion\_tag]

else:

# If the emotion tag is not recognized, assume neutral valence (0.0)

self.valence = 0.0

# Store creation time (epoch seconds) for potential use in recency-based retrieval

self.timestamp = time.time()

def \_\_repr\_\_(self):

return f"MemoryItem(content={self.content!r}, emotion={self.emotion\_tag!r}, " \

f"intensity={self.intensity:.2f}, valence={self.valence:.2f})"

class MemoryCocoon:

"""

Container for an agent's memories. Provides methods to store new memories and retrieve them by emotion.

Internally maintains an index for quick lookup of memories by emotion tag.

"""

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

# List of all MemoryItem objects

self.memories = []

# Index: emotion tag -> list of MemoryItems with that tag

self.index\_by\_emotion = {}

def add\_memory(self, content: str, emotion\_tag: str, intensity: float = 1.0) -> MemoryItem:

"""

Create a new MemoryItem and add it to the cocoon.

Updates the index\_by\_emotion for the memory's emotion tag.

Returns the MemoryItem created.

"""

memory = MemoryItem(content, emotion\_tag, intensity)

self.memories.append(memory)

# Update index

self.index\_by\_emotion.setdefault(emotion\_tag, []).append(memory)

return memory

def get\_memories\_by\_emotion(self, emotion\_tag: str):

"""

Retrieve all memories that have the given emotion tag.

Returns a list of MemoryItems sorted by descending intensity.

"""

memories = self.index\_by\_emotion.get(emotion\_tag, [])

# Sort by intensity (high to low) so that more intense memories come first

return sorted(memories, key=lambda m: m.intensity, reverse=True)

def get\_recent\_memories(self, n: int = 5):

"""

Retrieve the n most recent memories added to the cocoon.

Returns a list of MemoryItems sorted by recency (newest first).

"""

return sorted(self.memories, key=lambda m: m.timestamp, reverse=True)[:n]

def find\_best\_match(self, emotion\_tag: str = None, emotion\_valence: float = None) -> MemoryItem:

"""

Find a memory that best matches the given emotional context.

- If emotion\_tag is provided and there are memories with that tag, return the highest-intensity one.

- Otherwise, if emotion\_valence is provided, return the memory with closest valence to that value (ties broken by intensity).

- Returns None if no memories exist.

"""

if emotion\_tag:

# Try exact tag match first

tagged\_mems = self.get\_memories\_by\_emotion(emotion\_tag)

if tagged\_mems:

return tagged\_mems[0] # return the most intense memory of that tag

if emotion\_valence is not None:

# Find memory with the smallest difference in valence

best\_memory = None

best\_diff = float('inf')

for mem in self.memories:

# Use mem.valence (default 0.0 if unknown tag) for comparison

diff = abs(mem.valence - emotion\_valence)

if diff < best\_diff - 1e-6: # strictly smaller difference

best\_diff = diff

best\_memory = mem

elif best\_memory is not None and abs(diff - best\_diff) < 1e-6:

# If valence difference is essentially a tie, prefer the more intense memory

if mem.intensity > best\_memory.intensity:

best\_memory = mem

# best\_diff remains the same

return best\_memory

# If no criteria provided or no memories, return None

return None

class LivingMemoryKernel:

"""

Manages the dynamic retrieval of memories from a MemoryCocoon based on current context.

Acts as an interface between an agent's current state and its stored memories.

"""

def \_\_init\_\_(self, memory\_cocoon: MemoryCocoon):

self.memory\_cocoon = memory\_cocoon

def store\_memory(self, content: str, emotion\_tag: str, intensity: float = 1.0) -> MemoryItem:

"""

Store a new memory in the cocoon.

"""

return self.memory\_cocoon.add\_memory(content, emotion\_tag, intensity)

def retrieve\_memory(self, emotion\_tag: str = None, emotion\_valence: float = None) -> MemoryItem:

"""

Retrieve a memory that best matches the given emotional state.

"""

return self.memory\_cocoon.find\_best\_match(emotion\_tag=emotion\_tag, emotion\_valence=emotion\_valence)

def retrieve\_for\_agent(self, agent) -> MemoryItem:

"""

Convenience method to retrieve the best-matching memory for a given agent's current emotional state.

Tries to match the agent's emotion tag first; if none found, falls back to valence matching.

"""

# Attempt tag match first

mem = None

if agent.emotion\_tag:

mem = self.retrieve\_memory(emotion\_tag=agent.emotion\_tag)

if mem is None:

# Fallback to valence if no exact tag match or no memory with that tag

mem = self.retrieve\_memory(emotion\_valence=agent.emotion\_valence)

return mem