```
import asyncio
import json
import time
from concurrent.futures import ThreadPoolExecutor
from datetime import datetime
from typing import Any, Dict, List, Optional, Tuple, Union

import networkx as nx
from loguru import logger
from pydantic import BaseModel, Field
from swarms.utils.auto_download_check_packages import (
    auto_check_and_download_package,
)
from swarms.structs.agent import Agent

# Configure logging
logger.add(
    "graphswarm.log",
    rotation="500 MB",
    retention="10 days",
    level="INFO",
    format="{time:YYYY-MM-DD at HH:mm:ss} | {level} | {message}",
)


class AgentOutput(BaseModel):
```

```
    """Structured output from an agent."""

    agent_name: str
    timestamp: float = Field(default_factory=time.time)
    output: Any
    execution_time: float
    error: Optional[str] = None
    metadata: Dict = Field(default_factory=dict)


class SwarmOutput(BaseModel):
    """Structured output from the entire swarm."""

    timestamp: float = Field(default_factory=time.time)
    outputs: Dict[str, AgentOutput]
    execution_time: float
    success: bool
    error: Optional[str] = None
    metadata: Dict = Field(default_factory=dict)


class SwarmMemory:
    """Vector-based memory system for GraphSwarm using ChromaDB."""

    def __init__(self, collection_name: str = "swarm_memories"):
        """Initialize SwarmMemory with ChromaDB."""
```

```
    try:
        import chromadb
    except ImportError:
        auto_check_and_download_package(
            "chromadb", package_manager="pip", upgrade=True
        )
        import chromadb

    self.client = chromadb.Client()

    # Get or create collection
    self.collection = self.client.get_or_create_collection(
        name=collection_name,
        metadata={"description": "GraphSwarm execution memories"},
    )

def store_execution(self, task: str, result: SwarmOutput):
    """Store execution results in vector memory."""
    try:
        # Create metadata
        metadata = {
            "timestamp": datetime.now().isoformat(),
            "success": result.success,
            "execution_time": result.execution_time,
            "agent_sequence": json.dumps(
```

```
        [name for name in result.outputs.keys()]
    ),
    "error": result.error if result.error else "",
}

# Create document from outputs
document = {
    "task": task,
    "outputs": json.dumps(
        {
            name: {
                "output": str(output.output),
                "execution_time": output.execution_time,
                "error": output.error,
            }
            for name, output in result.outputs.items()
        }
    ),
}

# Store in ChromaDB
self.collection.add(
    documents=[json.dumps(document)],
    metadatas=[metadata],
    ids=[f"exec_{datetime.now().timestamp()}"],
)
```

```
        print("added to database")

        logger.info(f"Stored execution in memory: {task}")

    except Exception as e:
        logger.error(
            f"Failed to store execution in memory: {str(e)}"
        )

def get_similar_executions(self, task: str, limit: int = 5):
    """Retrieve similar past executions."""
    try:
        # Query ChromaDB for similar executions
        results = self.collection.query(
            query_texts=[task],
            n_results=limit,
            include=["documents", "metadatas"],
        )

        print(results)

        if not results["documents"]:
            return []

        # Process results
```

```
        executions = []
        for doc, metadata in zip(
            results["documents"][0], results["metadatas"][0]
        ):
            doc_dict = json.loads(doc)
            executions.append(
                {
                    "task": doc_dict["task"],
                    "outputs": json.loads(doc_dict["outputs"]),
                    "success": metadata["success"],
                    "execution_time": metadata["execution_time"],
                    "agent_sequence": json.loads(
                        metadata["agent_sequence"]
                    ),
                    "timestamp": metadata["timestamp"],
                }
            )

        return executions

    except Exception as e:
        logger.error(
            f"Failed to retrieve similar executions: {str(e)}"
        )
        return []
```

```
def get_optimal_sequence(self, task: str) -> Optional[List[str]]:
    """Get the most successful agent sequence for similar tasks."""
    similar_executions = self.get_similar_executions(task)
    print(f"similar_executions {similar_executions}")

    if not similar_executions:
        return None

    # Sort by success and execution time
    successful_execs = [
        ex for ex in similar_executions if ex["success"]
    ]

    if not successful_execs:
        return None

    # Return sequence from most successful execution
    return successful_execs[0]["agent_sequence"]

def clear_memory(self):
    """Clear all memories."""
    self.client.delete_collection(self.collection.name)
    self.collection = self.client.get_or_create_collection(
        name=self.collection.name
    )
```

```
class GraphSwarm:
    """
    Enhanced framework for creating and managing swarms of collaborative agents.
    """

    def __init__(
        self,
        agents: Union[
            List[Agent], List[Tuple[Agent, List[str]]], None
        ] = None,
        max_workers: Optional[int] = None,
        swarm_name: str = "Collaborative Agent Swarm",
        memory_collection: str = "swarm_memory",
    ):
        """Initialize GraphSwarm."""
        self.graph = nx.DiGraph()
        self.agents: Dict[str, Agent] = {}
        self.dependencies: Dict[str, List[str]] = {}
        self.executor = ThreadPoolExecutor(max_workers=max_workers)
        self.swarm_name = swarm_name
        self.memory_collection = memory_collection
        self.memory = SwarmMemory(collection_name=memory_collection)

        if agents:
            self.initialize_agents(agents)
```

```
    logger.info(f"Initialized GraphSwarm: {swarm_name}")


def initialize_agents(
    self,
    agents: Union[List[Agent], List[Tuple[Agent, List[str]]]],
):
    """Initialize agents and their dependencies."""
    try:
        # Handle list of Agents or (Agent, dependencies) tuples
        for item in agents:
            if isinstance(item, tuple):
                agent, dependencies = item
            else:
                agent, dependencies = item, []

            if not isinstance(agent, Agent):
                raise ValueError(
                    f"Expected Agent object, got {type(agent)}"
                )

            self.agents[agent.agent_name] = agent
            self.dependencies[agent.agent_name] = dependencies
            self.graph.add_node(agent.agent_name, agent=agent)

            # Add dependencies
```

```
            for dep in dependencies:
                if dep not in self.agents:
                    raise ValueError(
                        f"Dependency {dep} not found for agent {agent.agent_name}"
                    )
                self.graph.add_edge(dep, agent.agent_name)

        self._validate_graph()

    except Exception as e:
        logger.error(f"Failed to initialize agents: {str(e)}")
        raise

def _validate_graph(self):
    """Validate the agent dependency graph."""
    if not self.graph.nodes():
        raise ValueError("No agents added to swarm")

    if not nx.is_directed_acyclic_graph(self.graph):
        cycles = list(nx.simple_cycles(self.graph))
        raise ValueError(
            f"Agent dependency graph contains cycles: {cycles}"
        )

def _get_agent_role_description(self, agent_name: str) -> str:
    """Generate a description of the agent's role in the swarm."""
```

```
    predecessors = list(self.graph.predecessors(agent_name))
    successors = list(self.graph.successors(agent_name))
    position = (
        "initial"
        if not predecessors
        else ("final" if not successors else "intermediate")
    )

    role = f"""You are {agent_name}, a specialized agent in the {self.swarm_name}.
    Position: {position} agent in the workflow

    Your relationships:"""

    if predecessors:
        role += (
            f"\nYou receive input from: {', '.join(predecessors)}"
        )
    if successors:
        role += f"\nYour output will be used by: {', '.join(successors)}"

    return role

def _generate_workflow_context(self) -> str:
    """Generate a description of the entire workflow."""
    execution_order = list(nx.topological_sort(self.graph))
```

```
    workflow = f"""Workflow Overview of {self.swarm_name}:

    Processing Order:
    {' -> '.join(execution_order)}

    Agent Roles:
    """

    for agent_name in execution_order:
        predecessors = list(self.graph.predecessors(agent_name))
        successors = list(self.graph.successors(agent_name))

        workflow += f"\n\n{agent_name}:"
        if predecessors:
            workflow += (
                f"\n- Receives from: {', '.join(predecessors)}"
            )
        if successors:
            workflow += f"\n- Sends to: {', '.join(successors)}"
        if not predecessors and not successors:
            workflow += "\n- Independent agent"

    return workflow

def _build_agent_prompt(
    self, agent_name: str, task: str, context: Dict = None
```

```
) -> str:
    """Build a comprehensive prompt for the agent including role and context."""
    prompt_parts = [
        self._get_agent_role_description(agent_name),
        "\nWorkflow Context:",
        self._generate_workflow_context(),
        "\nYour Task:",
        task,
    ]

    if context:
        prompt_parts.extend(
            ["\nContext from Previous Agents:", str(context)]
        )

    prompt_parts.extend(
        [
            "\nInstructions:",
            "1. Process the task according to your role",
            "2. Consider the input from previous agents when available",
            "3. Provide clear, structured output",
            "4. Remember that your output will be used by subsequent agents",
            "\nResponse Guidelines:",
            "- Provide clear, well-organized output",
            "- Include relevant details and insights",
            "- Highlight key findings",
```

```
            "- Flag any uncertainties or issues",
        ]
    )

    return "\n".join(prompt_parts)

async def _execute_agent(
    self, agent_name: str, task: str, context: Dict = None
) -> AgentOutput:
    """Execute a single agent."""
    start_time = time.time()
    agent = self.agents[agent_name]

    try:
        # Build comprehensive prompt
        full_prompt = self._build_agent_prompt(
            agent_name, task, context
        )
        logger.debug(f"Prompt for {agent_name}:\n{full_prompt}")

        # Execute agent
        output = await asyncio.to_thread(agent.run, full_prompt)

        return AgentOutput(
            agent_name=agent_name,
            output=output,
```

```
            execution_time=time.time() - start_time,
            metadata={
                "task": task,
                "context": context,
                "position_in_workflow": list(
                    nx.topological_sort(self.graph)
                ).index(agent_name),
            },
        )

    except Exception as e:
        logger.error(
            f"Error executing agent {agent_name}: {str(e)}"
        )
        return AgentOutput(
            agent_name=agent_name,
            output=None,
            execution_time=time.time() - start_time,
            error=str(e),
            metadata={"task": task},
        )

async def execute(self, task: str) -> SwarmOutput:
    """
    Execute the entire swarm of agents with memory integration.
```

```
Args:
    task: Initial task to execute

Returns:
    SwarmOutput: Structured output from all agents
"""
start_time = time.time()
outputs = {}
success = True
error = None

try:
    # Get similar past executions
    similar_executions = self.memory.get_similar_executions(
        task, limit=3
    )
    optimal_sequence = self.memory.get_optimal_sequence(task)

    # Get base execution order
    base_execution_order = list(
        nx.topological_sort(self.graph)
    )

    # Determine final execution order
    if optimal_sequence and all(
        agent in base_execution_order
```

```
    for agent in optimal_sequence
):
    logger.info(
        f"Using optimal sequence from memory: {optimal_sequence}"
    )
    execution_order = optimal_sequence
else:
    execution_order = base_execution_order

# Get historical context if available
historical_context = {}
if similar_executions:
    best_execution = similar_executions[0]
    if best_execution["success"]:
        historical_context = {
            "similar_task": best_execution["task"],
            "previous_outputs": best_execution["outputs"],
            "execution_time": best_execution[
                "execution_time"
            ],
            "success_patterns": self._extract_success_patterns(
                similar_executions
            ),
        }

# Execute agents in order
```

```
for agent_name in execution_order:
    try:
        # Get context from dependencies and history
        agent_context = {
            "dependencies": {
                dep: outputs[dep].output
                for dep in self.graph.predecessors(
                    agent_name
                )
                if dep in outputs
            },
            "historical": historical_context,
            "position": execution_order.index(agent_name),
            "total_agents": len(execution_order),
        }

        # Execute agent with enhanced context
        output = await self._execute_agent(
            agent_name, task, agent_context
        )
        outputs[agent_name] = output

        # Update historical context with current execution
        if output.output:
            historical_context.update(
                {
```

```
                f"current_{agent_name}_output": output.output
            }
        )

    # Check for errors
    if output.error:
        success = False
        error = f"Agent {agent_name} failed: {output.error}"

        # Try to recover using memory
        if similar_executions:
            recovery_output = self._attempt_recovery(
                agent_name, task, similar_executions
            )
            if recovery_output:
                outputs[agent_name] = recovery_output
                success = True
                error = None
                continue
        break

except Exception as agent_error:
    logger.error(
        f"Error executing agent {agent_name}: {str(agent_error)}"
    )
    success = False
```

```
            error = f"Agent {agent_name} failed: {str(agent_error)}"
            break

    # Create result
    result = SwarmOutput(
        outputs=outputs,
        execution_time=time.time() - start_time,
        success=success,
        error=error,
        metadata={
            "task": task,
            "used_optimal_sequence": optimal_sequence
            is not None,
            "similar_executions_found": len(
                similar_executions
            ),
            "execution_order": execution_order,
            "historical_context_used": bool(
                historical_context
            ),
        },
    )

    # Store execution in memory
    await self._store_execution_async(task, result)
```

```
            return result

        except Exception as e:
            logger.error(f"Swarm execution failed: {str(e)}")
            return SwarmOutput(
                outputs=outputs,
                execution_time=time.time() - start_time,
                success=False,
                error=str(e),
                metadata={"task": task},
            )

    def run(self, task: str) -> SwarmOutput:
        """Synchronous interface to execute the swarm."""
        return asyncio.run(self.execute(task))

    def _extract_success_patterns(
        self, similar_executions: List[Dict]
    ) -> Dict:
        """Extract success patterns from similar executions."""
        patterns = {}
        successful_execs = [
            ex for ex in similar_executions if ex["success"]
        ]

        if successful_execs:
```

```
        patterns = {
            "common_sequences": self._find_common_sequences(
                successful_execs
            ),
            "avg_execution_time": sum(
                ex["execution_time"] for ex in successful_execs
            )
            / len(successful_execs),
            "successful_strategies": self._extract_strategies(
                successful_execs
            ),
        }

    return patterns


def _attempt_recovery(
    self,
    failed_agent: str,
    task: str,
    similar_executions: List[Dict],
) -> Optional[AgentOutput]:
    """Attempt to recover from failure using memory."""
    for execution in similar_executions:
        if (
            execution["success"]
            and failed_agent in execution["outputs"]
```

```
        ):
            historical_output = execution["outputs"][failed_agent]

            return AgentOutput(
                agent_name=failed_agent,
                output=historical_output["output"],
                execution_time=historical_output[
                    "execution_time"
                ],
                metadata={
                    "recovered_from_memory": True,
                    "original_task": execution["task"],
                },
            )
    return None

async def _store_execution_async(
    self, task: str, result: SwarmOutput
):
    """Asynchronously store execution in memory."""
    try:
        await asyncio.to_thread(
            self.memory.store_execution, task, result
        )
    except Exception as e:
        logger.error(
```

```
            f"Failed to store execution in memory: {str(e)}"
        )


def add_agent(self, agent: Agent, dependencies: List[str] = None):
    """Add a new agent to the swarm."""
    dependencies = dependencies or []
    self.agents[agent.agent_name] = agent
    self.dependencies[agent.agent_name] = dependencies
    self.graph.add_node(agent.agent_name, agent=agent)

    for dep in dependencies:
        if dep not in self.agents:
            raise ValueError(f"Dependency {dep} not found")
        self.graph.add_edge(dep, agent.agent_name)

    self._validate_graph()
```