```
import uuid
from collections import Counter
from datetime import datetime
from typing import Any, List, Optional

from pydantic import BaseModel, Field
from swarms.structs.agent import Agent
from swarms.utils.loguru_logger import initialize_logger
from swarms.utils.auto_download_check_packages import (
    auto_check_and_download_package,
)
from swarms.structs.conversation import Conversation


logger = initialize_logger(log_folder="tree_swarm")


# Pydantic Models for Logging
class AgentLogInput(BaseModel):
    log_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()), alias="id"
    )
    agent_name: str
    task: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)
```

```
class AgentLogOutput(BaseModel):
    log_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()), alias="id"
    )
    agent_name: str
    result: Any
    timestamp: datetime = Field(default_factory=datetime.utcnow)



class TreeLog(BaseModel):
    log_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()), alias="id"
    )
    tree_name: str
    task: str
    selected_agent: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)
    result: Any



def extract_keywords(prompt: str, top_n: int = 5) -> List[str]:
    """
    A simplified keyword extraction function using basic word splitting instead of NLTK tokenization.
    """
    words = prompt.lower().split()
```

```
    filtered_words = [word for word in words if word.isalnum()]
    word_counts = Counter(filtered_words)
    return [word for word, _ in word_counts.most_common(top_n)]



class TreeAgent(Agent):
    """
    A specialized Agent class that contains information about the system prompt's
    locality and allows for dynamic chaining of agents in trees.
    """

    def __init__(
        self,
        name: str = None,
        description: str = None,
        system_prompt: str = None,
        model_name: str = "gpt-4o",
        agent_name: Optional[str] = None,
        *args,
        **kwargs,
    ):
        agent_name = agent_name
        super().__init__(
            name=name,
            description=description,
            system_prompt=system_prompt,
```

```
    model_name=model_name,
    agent_name=agent_name,
    *args,
    **kwargs,
)

try:
    import sentence_transformers
except ImportError:
    auto_check_and_download_package(
        "sentence-transformers", package_manager="pip"
    )
    import sentence_transformers

self.sentence_transformers = sentence_transformers

# Pretrained model for embeddings
self.embedding_model = (
    sentence_transformers.SentenceTransformer(
        "all-MiniLM-L6-v2"
    )
)
self.system_prompt_embedding = self.embedding_model.encode(
    system_prompt, convert_to_tensor=True
)
```

```
    # Automatically extract keywords from system prompt
    self.relevant_keywords = extract_keywords(system_prompt)

    # Distance is now calculated based on similarity between agents' prompts
    self.distance = None  # Will be dynamically calculated later

def calculate_distance(self, other_agent: "TreeAgent") -> float:
    """
    Calculate the distance between this agent and another agent using embedding similarity.

    Args:
        other_agent (TreeAgent): Another agent in the tree.

    Returns:
        float: Distance score between 0 and 1, with 0 being close and 1 being far.
    """
    similarity = self.sentence_transformers.util.pytorch_cos_sim(
        self.system_prompt_embedding,
        other_agent.system_prompt_embedding,
    ).item()
    distance = (
        1 - similarity
    )  # Closer agents have a smaller distance
    return distance

def run_task(
```

```
    self, task: str, img: str = None, *args, **kwargs
) -> Any:
    input_log = AgentLogInput(
        agent_name=self.agent_name,
        task=task,
        timestamp=datetime.now(),
    )
    logger.info(f"Running task on {self.agent_name}: {task}")
    logger.debug(f"Input Log: {input_log.json()}")

    result = self.run(task=task, img=img, *args, **kwargs)

    output_log = AgentLogOutput(
        agent_name=self.agent_name,
        result=result,
        timestamp=datetime.now(),
    )
    logger.info(f"Task result from {self.agent_name}: {result}")
    logger.debug(f"Output Log: {output_log.json()}")

    return result

def is_relevant_for_task(
    self, task: str, threshold: float = 0.7
) -> bool:
    """
```

Checks if the agent is relevant for the given task using both keyword matching and embedding similarity.

Args:
task (str): The task to be executed.
threshold (float): The cosine similarity threshold for embedding-based matching.

Returns:
bool: True if the agent is relevant, False otherwise.
"""
# Check if any of the relevant keywords are present in the task (case-insensitive)
keyword_match = any(
keyword.lower() in task.lower()
for keyword in self.relevant_keywords
)

# Perform embedding similarity match if keyword match is not found
if not keyword_match:
task_embedding = self.embedding_model.encode(
task, convert_to_tensor=True
)
similarity = (
self.sentence_transformers.util.pytorch_cos_sim(
self.system_prompt_embedding, task_embedding
).item()
)

```
            logger.info(
                f"Semantic similarity between task and {self.agent_name}: {similarity:.2f}"
            )
            return similarity >= threshold

        return True  # Return True if keyword match is found



class Tree:
    def __init__(self, tree_name: str, agents: List[TreeAgent]):
        """
        Initializes a tree of agents.

        Args:
            tree_name (str): The name of the tree.
            agents (List[TreeAgent]): A list of agents in the tree.
        """
        self.tree_name = tree_name
        self.agents = agents
        self.calculate_agent_distances()

    def calculate_agent_distances(self):
        """
            Automatically calculate and assign distances between agents in the tree based on prompt similarity.
        """
```

```
    logger.info(
        f"Calculating distances between agents in tree '{self.tree_name}'"
    )
    for i, agent in enumerate(self.agents):
        if i > 0:
            agent.distance = agent.calculate_distance(
                self.agents[i - 1]
            )
        else:
            agent.distance = 0  # First agent is closest

    # Sort agents by distance after calculation
    self.agents.sort(key=lambda agent: agent.distance)

def find_relevant_agent(self, task: str) -> Optional[TreeAgent]:
    """
    Finds the most relevant agent in the tree for the given task based on its system prompt.
    Uses both keyword and semantic similarity matching.

    Args:
        task (str): The task or query for which we need to find a relevant agent.

    Returns:
        Optional[TreeAgent]: The most relevant agent, or None if no match found.
    """
    logger.info(
```

```
        f"Searching relevant agent in tree '{self.tree_name}' for task: {task}"
    )
    for agent in self.agents:
        if agent.is_relevant_for_task(task):
            return agent
    logger.warning(
        f"No relevant agent found in tree '{self.tree_name}' for task: {task}"
    )
    return None


def log_tree_execution(
    self, task: str, selected_agent: TreeAgent, result: Any
) -> None:
    """
    Logs the execution details of a tree, including selected agent and result.
    """
    tree_log = TreeLog(
        tree_name=self.tree_name,
        task=task,
        selected_agent=selected_agent.agent_name,
        timestamp=datetime.now(),
        result=result,
    )
    logger.info(
        f"Tree '{self.tree_name}' executed task with agent '{selected_agent.agent_name}'"
    )
```

```
        logger.debug(f"Tree Log: {tree_log.json()}")



class ForestSwarm:
    def __init__(
        self,
        name: str = "default-forest-swarm",
        description: str = "Standard forest swarm",
        trees: List[Tree] = [],
        shared_memory: Any = None,
        rules: str = None,
        *args,
        **kwargs,
    ):
        """
        Initializes the structure with multiple trees of agents.

        Args:
            trees (List[Tree]): A list of trees in the structure.
        """
        self.name = name
        self.description = description
        self.trees = trees
        self.shared_memory = shared_memory
        self.save_file_path = f"forest_swarm_{uuid.uuid4().hex}.json"
        self.conversation = Conversation(
```

```
        time_enabled=True,
        auto_save=True,
        save_filepath=self.save_file_path,
        rules=rules,
    )

def find_relevant_tree(self, task: str) -> Optional[Tree]:
    """
    Finds the most relevant tree based on the given task.

    Args:
        task (str): The task or query for which we need to find a relevant tree.

    Returns:
        Optional[Tree]: The most relevant tree, or None if no match found.
    """
    logger.info(
        f"Searching for the most relevant tree for task: {task}"
    )
    for tree in self.trees:
        if tree.find_relevant_agent(task):
            return tree
    logger.warning(f"No relevant tree found for task: {task}")
    return None

def run(self, task: str, img: str = None, *args, **kwargs) -> Any:
```

"""

Executes the given task by finding the most relevant tree and agent within that tree.

Args:

task (str): The task or query to be executed.

Returns:

Any: The result of the task after it has been processed by the agents.

"""

```
try:
    logger.info(
        f"Running task across MultiAgentTreeStructure: {task}"
    )
    relevant_tree = self.find_relevant_tree(task)
    if relevant_tree:
        agent = relevant_tree.find_relevant_agent(task)
        if agent:
            result = agent.run_task(
                task, img=img, *args, **kwargs
            )
            relevant_tree.log_tree_execution(
                task, agent, result
            )
            return result
    else:
        logger.error(
```

```
                "Task could not be completed: No relevant agent or tree found."
            )
            return "No relevant agent found to handle this task."
        except Exception as error:
            logger.error(
                f"Error detected in the ForestSwarm, check your inputs and try again ;) {error}"
            )



# # Example Usage:


# # Create agents with varying system prompts and dynamically generated distances/keywords
# agents_tree1 = [
#     TreeAgent(
#         system_prompt="Stock Analysis Agent",
#         agent_name="Stock Analysis Agent",
#     ),
#     TreeAgent(
#         system_prompt="Financial Planning Agent",
#         agent_name="Financial Planning Agent",
#     ),
#     TreeAgent(
#         agent_name="Retirement Strategy Agent",
#         system_prompt="Retirement Strategy Agent",
#     ),
# ]
```

```
# agents_tree2 = [
#     TreeAgent(
#         system_prompt="Tax Filing Agent",
#         agent_name="Tax Filing Agent",
#     ),
#     TreeAgent(
#         system_prompt="Investment Strategy Agent",
#         agent_name="Investment Strategy Agent",
#     ),
#     TreeAgent(
#         system_prompt="ROTH IRA Agent", agent_name="ROTH IRA Agent"
#     ),
# ]

# # Create trees
# tree1 = Tree(tree_name="Financial Tree", agents=agents_tree1)
# tree2 = Tree(tree_name="Investment Tree", agents=agents_tree2)

# # Create the ForestSwarm
# multi_agent_structure = ForestSwarm(trees=[tree1, tree2])

# # Run a task
# task = "Our company is incorporated in delaware, how do we do our taxes for free?"
# output = multi_agent_structure.run(task)
# print(output)
```