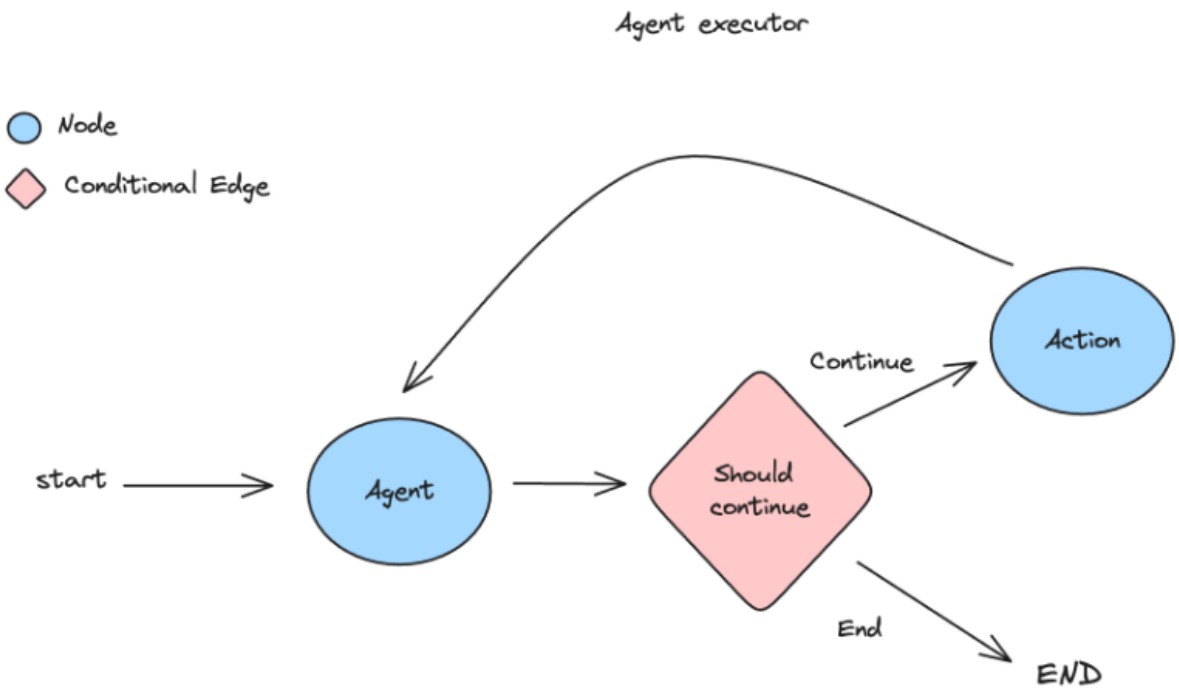
Report on LangGraph and its Integration with LangChain

Introduction

LangGraph represents a novel approach to running agents with LangChain, leveraging custom chains through the LangChain expression language. It adopts a graph-based methodology where nodes denote actions (such as running a chain or invoking a tool), and edges symbolize the pathways or conditions governing the transitions between these actions. Unlike a Directed Acyclic Graph (DAG), LangGraph permits nodes to make dynamic decisions regarding the subsequent node, resembling a state machine.



Key Components

1. State Graph
   - Purpose: Maintains and updates the state throughout the agent's lifecycle.
   - Functionality: Facilitates state persistence and modification as the agent operates.

2. Nodes
   - Definition: Components like chains or tools that serve as the fundamental building blocks of the agent.
   - Function: Represent various actions and processes within the workflow.

3. Edges
   - Types:

- Fixed Edges: Direct transitions between nodes.
- Conditional Edges: Transitions based on conditions determined by functions, often executed by a Large Language Model (LLM).

Graph Compilation Process

The process of compiling a LangGraph involves configuring and finalizing the structure of nodes and edges to define agent behavior. The steps include:

1. Setting Up Nodes
   - Task: Add and configure nodes to perform specific actions, such as running chains or invoking tools.

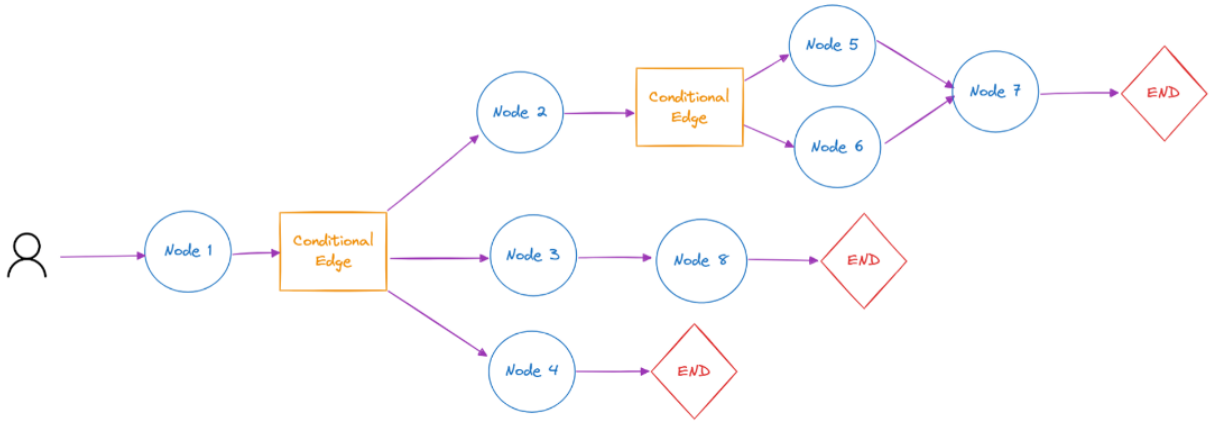
2. Configuring Edges
   - Task: Establish connections between nodes, setting up fixed or conditional edges based on desired pathways or decision criteria.

3. Defining Entry and Exit Points
   - Task: Specify the starting (entry) and ending (exit) points of the graph.

4. Ensuring State Management
   - Task: Configure the state graph component to manage and persist the agent's state across nodes.

5. Validation and Finalization
   - Task: Validate the entire structure to identify and rectify logical errors or misconfigurations, ensuring readiness for execution.

Once compiled, the graph is executable within the LangChain ecosystem, facilitating state transitions, decision-making, and tool invocations.



Example Use Case

Overview

The example provided demonstrates the setup and execution of a LangChain agent using LangGraph, integrating with a chat model and tools, and managing state and message history efficiently.

Code Breakdown

1. Install and Import Libraries
  - Commands: `!pip install` for necessary packages; `dotenv` for managing environment variables.

2. Load Environment Variables
  - Methods: `load_dotenv()` for loading API keys; `os.getenv()` for retrieval.

3. Set Up API Keys
  - Action: Configuring OpenAI and LangChain API keys as environment variables.

```
import os
from dotenv import load_dotenv

load_dotenv()

os.environ["OPENAI_API_KEY"] = os.getenv('OPENAI_API_KEY')
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = os.getenv('LANGCHAIN_API_KEY')
os.environ["LANGCHAIN_PROJECT"] = "LangGraph_02"
```

4. Initialize the Chat Model
  - Configuration: `ChatOpenAI` with specific settings (`temperature=0`, `streaming=True`).

5. Define Tools
  - Tools:
    - `to_lower_case`: Converts strings to lowercase.
    - `random_number_maker`: Generates random numbers between 0-100.

6. Format Tools for OpenAI
  - Method: `format_tool_to_openai_function` converts tools into a compatible format for OpenAI's function calling.

7. Bind Functions to the Model
  - Action: `model.bind_functions(functions)` associates tools with the chat model.

```
[41]  1 from langchain.tools import BaseTool, StructuredTool, Tool, tool
      2 import random
      3
      4 @tool("lower_case", return_direct=True)
      5 def to_lower_case(input:str) -> str:
      6   """Returns the input as all lower case."""
      7   return input.lower()
      8
      9 @tool("random_number", return_direct=True)
     10 def random_number_maker(input:str) -> str:
     11     """Returns a random number between 0-100. input the word 'random'"""
     12     return random.randint(0, 100)
     13
     14 tools = [to_lower_case,random_number_maker]
```

```
[42]  1 from langgraph.prebuilt.tool_executor import ToolExecutor
      2
      3 tool_executor = ToolExecutor(tools)
```

```
[43]  1 from langchain.tools.render import format_tool_to_openai_function
      2
      3 functions = [format_tool_to_openai_function(t) for t in tools]
      4 model = model.bind_functions(functions)
```

8. Define Agent State and Functions
  - Components:
    - AgentState: Tracks messages.
    - Functions:
      - `should_continue`: Determines if processing should continue.
      - `call_model`: Invokes the model and returns responses.
      - `call_tool`: Executes tools based on function calls and returns results.

```
1 from typing import TypedDict, Annotated, Sequence
2 import operator
3 from langchain_core.messages import BaseMessage
4
5
6 class AgentState(TypedDict):
7     messages: Annotated[Sequence[BaseMessage], operator.add]
```

9. Create and Configure StateGraph

- Nodes:
  - `agent` : Handles model invocation.
  - `action` : Executes tools.
- Edges:
  - Conditional edge: Based on `should_continue`.
  - Normal edge: Transitions from the action node back to the agent node.

Nodes

```python
from langchain_core.agents import AgentFinish
from langgraph.prebuilt import ToolInvocation
import json
from langchain_core.messages import FunctionMessage

# Define the function that determines whether to continue or not
def should_continue(state):
    messages = state['messages']
    last_message = messages[-1]
    # If there is no function call, then we finish
    if "function_call" not in last_message.additional_kwargs:
        return "end"
    # Otherwise if there is, we continue
    else:
        return "continue"

# Define the function that calls the model
def call_model(state):
    messages = state['messages']
    response = model.invoke(messages)
    # We return a list, because this will get added to the existing list
    return {"messages": [response]}

# Define the function to execute tools
def call_tool(state):
    messages = state['messages']
    # Based on the continue condition
    # we know the last message involves a function call
    last_message = messages[-1]
    # We construct an ToolInvocation from the function_call
    action = ToolInvocation(
        tool=last_message.additional_kwargs["function_call"]["name"],
        tool_input=json.loads(last_message.additional_kwargs["function_call"]["arguments"]),
    )
    print(f"The agent action is {action}")
    # We call the tool_executor and get back a response
    response = tool_executor.invoke(action)
    print(f"The tool result is: {response}")
    # We use the response to create a FunctionMessage
    function_message = FunctionMessage(content=str(response), name=action.tool)
    # We return a list, because this will get added to the existing list
    return {"messages": [function_message]}
```

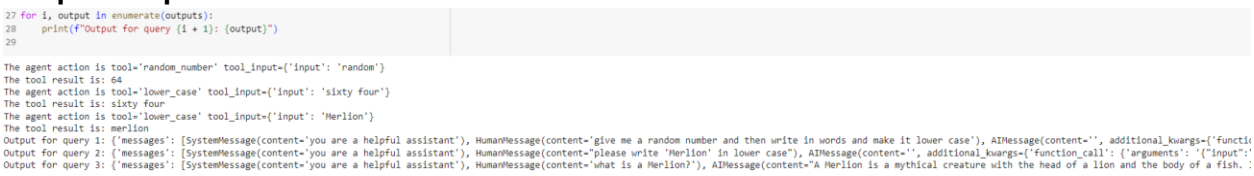
10. Compile the Graph
  - Method: `workflow.compile()` converts the graph into a runnable LangChain component.

```python
from langgraph.graph import StateGraph, END
# Define a new graph
workflow = StateGraph(AgentState)

# Define the two nodes we will cycle between
workflow.add_node("agent", call_model)
workflow.add_node("action", call_tool)

# Set the entrypoint as `agent` where we start
workflow.set_entry_point("agent")

# We now add a conditional edge
workflow.add_conditional_edges(
    # First, we define the start node. We use `agent`.
    # This means these are the edges taken after the `agent` node is called.
    "agent",
    # Next, we pass in the function that will determine which node is called next.
    should_continue,
    # Finally we pass in a mapping.
    # The keys are strings, and the values are other nodes.
    # END is a special node marking that the graph should finish.
    # What will happen is we will call `should_continue`, and then the output of that
    # will be matched against the keys in this mapping.
    # Based on which one it matches, that node will then be called.
    {
        # If `tools`, then we call the tool node.
        "continue": "action",
        # Otherwise we finish.
        "end": END
    }
)

# We now add a normal edge from `tools` to `agent`.
# This means that after `tools` is called, `agent` node is called next.
workflow.add_edge('action', 'agent')

# Finally, we compile it!
# This compiles it into a LangChain Runnable,
# meaning you can use it as you would any other runnable
app = workflow.compile()
```

11. Run the Agent
    - Inputs: Defines conversation context using `SystemMessage` and `HumanMessage`.
    - Execution: `app.invoke(inputs)` processes inputs through the compiled workflow.

```python
from langchain_core.messages import HumanMessage, SystemMessage

# Define the system message once
system_message = SystemMessage(content="you are a helpful assistant")

# Define the queries
queries = [
    "give me a random number and then write in words and make it lower case",
    "please write 'Merlion' in lower case",
    "what is a Merlion?"
]

# Collect outputs
outputs = []

for query in queries:
    user_message = HumanMessage(content=query)
    inputs = {"messages": [system_message, user_message]}

    # Invoke the app and store the result
    result = app.invoke(inputs)

    # Collect the result in outputs
    outputs.append(result)

# Print all outputs
for i, output in enumerate(outputs):
    print(f"Output for query {i + 1}: {output}")

```

Outputs Explanation



1. Random Number and Lowercase Conversion
   - Input: "Give me a random number and then write it in lowercase."
   - Output: Displays the random number and its lowercase representation.

2. Convert Text to Lowercase
   - Input: "Please write 'Merlion' in lowercase."
   - Output: Shows "merlion" as the lowercase text.

3. Provide Definition
   - Input: "What is a Merlion?"
   - Output: Provides a definition or description of "Merlion."

Each invocation tests various aspects of the workflow, including function calls, tool executions, message handling, and state management.

Conclusion

The LangGraph approach to running agents with LangChain offers a flexible and powerful mechanism for managing complex workflows. By incorporating a graph-based structure and dynamic decision-making capabilities, LangGraph facilitates efficient interaction with chat models and tools, enabling effective state management and message processing. The example code illustrates the practical application of LangGraph, showcasing its ability to handle diverse queries and maintain conversation history seamlessly..