LangGraph Tutorial: Building Complex Agent Workflows

Table of Contents

Introduction to LangGraph

LangGraph is a library for building stateful, multi-actor applications with LLMs, built on top of LangChain. It extends the LangChain Expression Language with the ability to coordinate multiple chains (or actors) across multiple steps of computation in a cyclic manner.

Key Features:

- **Stateful**: Maintains state across multiple turns of conversation
- **Multi-actor**: Coordinate multiple LLM chains or agents
- **Cyclic**: Support for loops and conditional branching
- **Human-in-the-loop**: Easy integration of human feedback
- **Streaming**: Real-time streaming of intermediate results

Core Concepts

1. Nodes

Nodes represent individual processing units in your graph. Each node is a function that takes the current state and returns an updated state.

```
def my_node(state):
    # Process the state
    return {"messages": state["messages"] + [new_message]}
```

2. Edges

Edges define the flow between nodes. LangGraph supports:

- **Normal edges**: Direct connections between nodes
- **Conditional edges**: Branching based on state evaluation

- **Start/End edges**: Entry and exit points

3. State

State is the shared data structure that flows through your graph. It's typically a dictionary that gets updated by each node.

4. Checkpoints

Checkpoints allow you to save and restore the state at any point in the execution.

Installation and Setup

```
# Install LangGraph
pip install langgraph

# For development
pip install langgraph[dev]

# With additional integrations
pip install "langgraph[anthropic,openai]"
```

Environment Setup

```
import os
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolExecutor
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage

# Set your API keys
os.environ["OPENAI_API_KEY"] = "your-api-key-here"
```

Basic Graph Construction

Simple Linear Flow

```python
from langgraph.graph import StateGraph, END
from typing import TypedDict, List
from langchain_core.messages import BaseMessage

class GraphState(TypedDict):
    messages: List[BaseMessage]

def chatbot(state: GraphState):
    messages = state["messages"]
    llm = ChatOpenAI()
    response = llm.invoke(messages)
    return {"messages": [response]}

def create_simple_graph():
    workflow = StateGraph(GraphState)

    # Add nodes
    workflow.add_node("chatbot", chatbot)

    # Add edges
    workflow.set_entry_point("chatbot")
    workflow.add_edge("chatbot", END)

    return workflow.compile()

# Usage
app = create_simple_graph()
result = app.invoke({"messages": [HumanMessage(content="Hello!")]})
print(result["messages"][-1].content)
```

Conditional Branching

```python
def should_continue(state: GraphState) -> str:
    messages = state["messages"]
    last_message = messages[-1]

    if "FINAL ANSWER" in last_message.content:
        return "end"
    else:
        return "continue"

def researcher(state: GraphState):
    # Research logic here
    return {"messages": state["messages"] + [AIMessage(content="Research complete")]}

def writer(state: GraphState):
    # Writing logic here
    return {"messages": state["messages"] + [AIMessage(content="FINAL ANSWER: Here's the result")]}

def create_conditional_graph():
    workflow = StateGraph(GraphState)

    workflow.add_node("researcher", researcher)
    workflow.add_node("writer", writer)

    workflow.set_entry_point("researcher")
    workflow.add_conditional_edges(
        "researcher",
        should_continue,
        {
            "continue": "writer",
            "end": END
        }
    )
    workflow.add_edge("writer", END)

    return workflow.compile()
```

Advanced Patterns

Multi-Agent Collaboration

```
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import Tool

class MultiAgentState(TypedDict):
    messages: List[BaseMessage]
    next_agent: str

def create_multi_agent_system():
    # Create specialized agents
    researcher = create_react_agent(
        ChatOpenAI(),
        [search_tool, calculator_tool]
    )

    writer = create_react_agent(
        ChatOpenAI(),
        [writing_tool, fact_checker_tool]
    )

    def agent_node(state, agent, name):
        result = agent.invoke(state)
        return {
            "messages": [AIMessage(content=result["output"])],
            "next_agent": determine_next_agent(result)
        }

    workflow = StateGraph(MultiAgentState)
    workflow.add_node("researcher", lambda x: agent_node(x, researcher, "researcher"))
    workflow.add_node("writer", lambda x: agent_node(x, writer, "writer"))

    # Routing logic
    def route_next(state):
        return state.get("next_agent", "writer")

    workflow.add_conditional_edges("researcher", route_next)
    workflow.add_conditional_edges("writer", route_next)

    return workflow.compile()
```

Human-in-the-Loop

```
from langgraph.checkpoint.sqlite import SqliteSaver

def create_human_in_loop_graph():
    memory = SqliteSaver.from_conn_string(":memory:")

    def human_feedback(state):
        # This will pause execution and wait for human input
        pass

    workflow = StateGraph(GraphState)
    workflow.add_node("agent", chatbot)
    workflow.add_node("human", human_feedback)

    workflow.set_entry_point("agent")
    workflow.add_edge("agent", "human")
    workflow.add_edge("human", END)

    return workflow.compile(checkpointer=memory, interrupt_before=["human"])

# Usage with interruption
app = create_human_in_loop_graph()
config = {"configurable": {"thread_id": "1"}}

# Initial run - will stop at human node
result = app.invoke({"messages": [HumanMessage("Analyze this data")]}, config)

# Continue after human feedback
app.update_state(config, {"messages": [HumanMessage("User feedback here")]})
result = app.invoke(None, config)  # Resume from checkpoint
```

Parallel Processing

```
def create_parallel_processing_graph():
    def parallel_task_1(state):
        return {"task1_result": "Result from task 1"}

    def parallel_task_2(state):
        return {"task2_result": "Result from task 2"}

    def combine_results(state):
        combined = f"{state.get('task1_result', '')} + {state.get('task2_result', '')}"
        return {"final_result": combined}

    workflow = StateGraph(dict)
    workflow.add_node("task1", parallel_task_1)
    workflow.add_node("task2", parallel_task_2)
    workflow.add_node("combine", combine_results)

    # Both tasks run in parallel
    workflow.set_entry_point("task1")
    workflow.set_entry_point("task2")

    # Both feed into combine
    workflow.add_edge("task1", "combine")
    workflow.add_edge("task2", "combine")
    workflow.add_edge("combine", END)

    return workflow.compile()
```

Real-World Examples

Research Assistant

```
def create_research_assistant():
    class ResearchState(TypedDict):
        query: str
        research_results: List[str]
        summary: str
        citations: List[str]

    def search_step(state: ResearchState):
        # Implement web search
        results = web_search(state["query"])
        return {"research_results": results}

    def analyze_step(state: ResearchState):
        # Analyze search results
        analysis = llm_analyze(state["research_results"])
        return {"summary": analysis}

    def cite_step(state: ResearchState):
        # Generate citations
        citations = extract_citations(state["research_results"])
        return {"citations": citations}

    workflow = StateGraph(ResearchState)
    workflow.add_node("search", search_step)
    workflow.add_node("analyze", analyze_step)
    workflow.add_node("cite", cite_step)

    workflow.set_entry_point("search")
    workflow.add_edge("search", "analyze")
    workflow.add_edge("analyze", "cite")
    workflow.add_edge("cite", END)

    return workflow.compile()
```

Customer Service Agent

```python
def create_customer_service_agent():
    class ServiceState(TypedDict):
        customer_input: str
        intent: str
        customer_data: dict
        response: str
        escalate: bool

    def classify_intent(state: ServiceState):
        intent = classify_customer_intent(state["customer_input"])
        return {"intent": intent}

    def fetch_customer_data(state: ServiceState):
        data = get_customer_info(state.get("customer_id"))
        return {"customer_data": data}

    def handle_request(state: ServiceState):
        response = generate_response(
            state["intent"],
            state["customer_data"],
            state["customer_input"]
        )
        return {"response": response, "escalate": should_escalate(response)}

    def escalate_to_human(state: ServiceState):
        # Escalation logic
        return {"response": "Transferring to human agent..."}

    def should_escalate_decision(state: ServiceState):
        return "escalate" if state.get("escalate") else "respond"

    workflow = StateGraph(ServiceState)
    workflow.add_node("classify", classify_intent)
    workflow.add_node("fetch_data", fetch_customer_data)
    workflow.add_node("handle", handle_request)
    workflow.add_node("escalate", escalate_to_human)

    workflow.set_entry_point("classify")
    workflow.add_edge("classify", "fetch_data")
    workflow.add_edge("fetch_data", "handle")
    workflow.add_conditional_edges(
        "handle",
        should_escalate_decision,
        {"escalate": "escalate", "respond": END}
    )
    workflow.add_edge("escalate", END)

    return workflow.compile()
```

Best Practices

1. State Design

- Keep state minimal and focused
- Use TypedDict for type safety
- Avoid deeply nested structures
- Make state serializable for checkpoints

2. Error Handling

```
def robust_node(state):
    try:
        # Your logic here
        result = process_data(state)
        return {"result": result, "error": None}
    except Exception as e:
        return {"result": None, "error": str(e)}

def error_recovery(state):
    if state.get("error"):
        # Implement recovery logic
        return {"error": None, "retry_count": state.get("retry_count", 0) + 1}
    return state
```

3. Testing Strategies

```
def test_graph():
    app = create_your_graph()

    # Test individual nodes
    test_state = {"messages": [HumanMessage("test")]}
    result = app.get_node("your_node").invoke(test_state)
    assert "expected_key" in result

    # Test full flow
    final_result = app.invoke(test_state)
    assert final_result["messages"][-1].content is not None
```

4. Performance Optimization

- Use streaming for long-running operations
- Implement proper caching strategies
- Consider parallel execution where possible
- Monitor state size and complexity

5. Monitoring and Debugging

```
from langgraph.prebuilt import ToolExecutor
import logging

logging.basicConfig(level=logging.INFO)

def debug_node(state):
    logging.info(f"Node input: {state}")
    result = your_processing_function(state)
    logging.info(f"Node output: {result}")
    return result
```

Conclusion

LangGraph provides a powerful framework for building complex, stateful AI applications. By understanding its core concepts and patterns, you can create sophisticated multi-agent systems that handle real-world complexity.

Next Steps:

1. Experiment with the examples provided
2. Build your own custom nodes and edges
3. Explore the LangGraph documentation for advanced features
4. Join the LangChain community for support and updates

Additional Resources:

- LangGraph Documentation (https://python.langchain.com/docs/langgraph)
- LangGraph Examples Repository (https://github.com/langchain-ai/langgraph/tree/main/examples)
- LangChain Community (https://discord.gg/cU2adEyC7w)