

Xây dựng một ứng dụng đơn giản (graph) quyết định xem người dùng nên uống cà phê hay trà

Mục tiêu của bài tập này là làm quen với các thành phần và bước chính của ứng dụng LangGraph cơ bản. **State**, **Schema**, **Node**, **Edge**, và **Compile**.

I. Cài đặt

- Cài đặt Poetry tại đây

- tại terminal:

- `cd project_name`
- `pyenv local 3.11.4`
- `poetry install`
- `poetry shell`
- `jupyter lab`

- Tạo file `.env`

- ```
OPENAI_API_KEY=your_openai_api_key
LANGCHAIN_TRACING_V2=true
LANGCHAIN_ENDPOINT=https://api.smith.langchain.com # Bạn sẽ cần đăng ký tài khoản tại smith.langchain.com để lấy API key.
LANGCHAIN_API_KEY=your_langchain_api_key
LANGCHAIN_PROJECT=your_project_name
```

- Kết nối với tệp `.env` nằm trong cùng thư mục vào notebook

- ```
#pip install python-dotenv
```

- ```
import os
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
```

```
openai_api_key = os.environ["OPENAI_API_KEY"]
```

- Cài đặt LangChain(Nếu bạn đang sử dụng poetry shell được tải sẵn, bạn không cần phải cài đặt gói sau vì nó đã được tải sẵn cho bạn:)

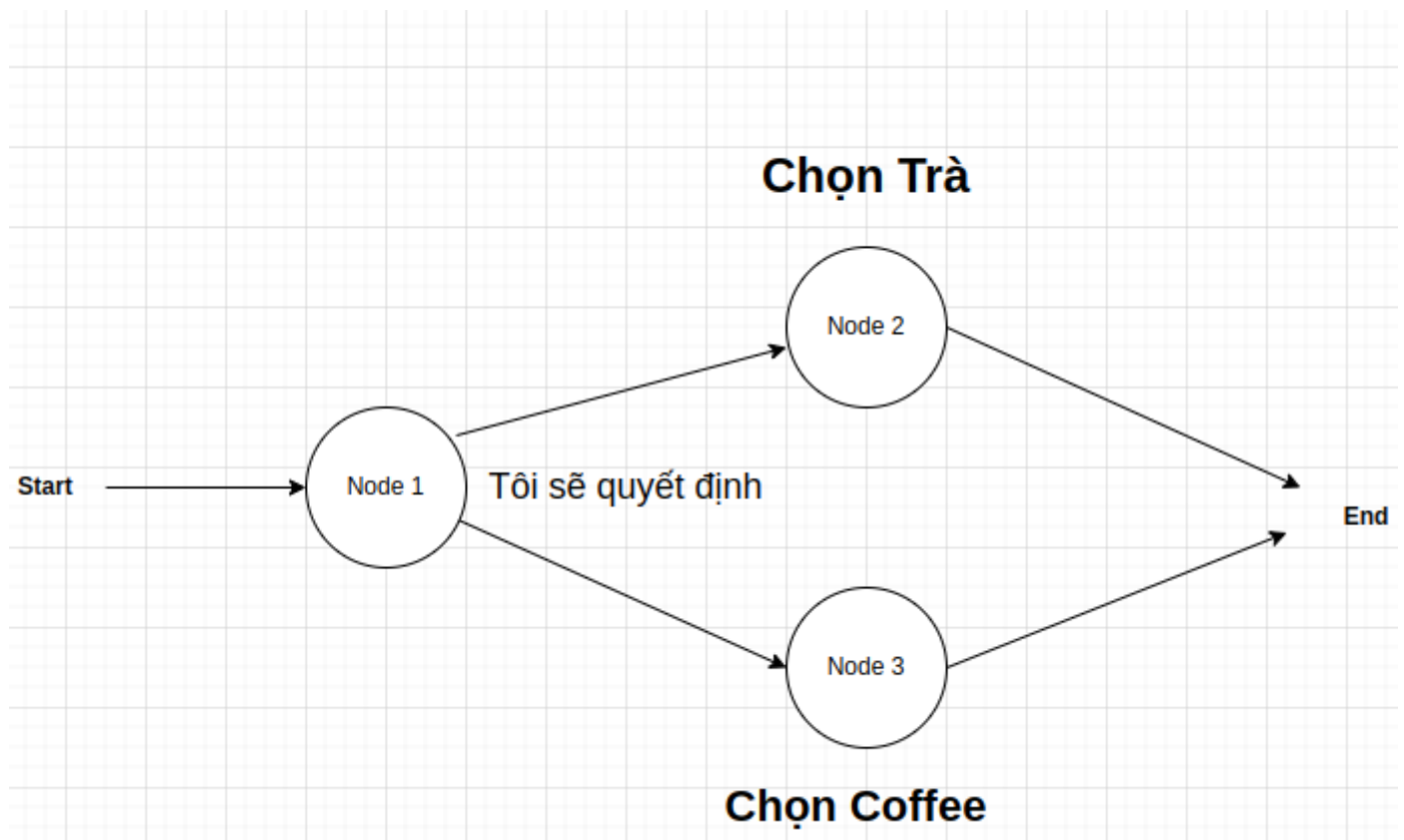
```
#!pip install langchain
```

- Kết nối với **LLM**

```
#!pip install langchain-openai
from langchain_openai import ChatOpenAI

chatModel35 = ChatOpenAI(model="gpt-3.5-turbo-0125")
chatModel4o = ChatOpenAI(model="gpt-4o")
```

Đây là hình ảnh đồ họa của ứng dụng chúng ta sẽ xây dựng



## 2. Xác định lược đồ trạng thái(state schema)

- Điều đầu tiên chúng ta sẽ làm là xác định State của ứng dụng.
- State schema cấu hình những gì, sẽ có trong state và với định dạng nào.

- State thường sẽ là TypedDict hoặc mô hình Pydantic.

Đối với ví dụ này, lớp State sẽ chỉ có một khóa, `graph_state`, với định dạng chuỗi

```
from typing_extensions import TypedDict

class State(TypedDict):
 graph_state: str
```

### TypedDict là gì?

- TypedDict cho phép bạn định nghĩa các từ điển có cấu trúc cụ thể. Nó cho phép bạn chỉ định các khóa chính xác và các loại giá trị liên quan của chúng, giúp mã của bạn rõ ràng hơn và an toàn hơn về mặt kiểu.

Ví dụ, thay vì một từ điển đơn giản như

```
{"name": "Ngoc Tu", "age": 40}
```

bạn có thể định nghĩa một TypedDict để thực thi rằng tên phải luôn là một chuỗi và tuổi phải luôn là một số nguyên.

Ở đây, một cấu trúc giống như từ điển mới có tên là State được định nghĩa.

State kế thừa từ TypedDict, nghĩa là nó hoạt động giống như một từ điển, nhưng có các quy tắc cụ thể về các khóa và giá trị mà nó phải có.

```
state: State = {"graph_state": "active"} # Đúng
```

```
state: State = {"graph_state": 123} # Sai, Điều này sẽ gây ra lỗi kiểm tra kiểu vì giá trị 123 không phải là chuỗi.
```

## 3. Xác định các Nodes của ứng dụng

Các Nodes của ứng dụng này được định nghĩa là các hàm python.

Đối số đầu tiên của hàm Node là state. Mỗi Node có thể truy cập khóa `graph_state`, với `state['graph_state']`.

Trong bài tập này, mỗi Node sẽ trả về một giá trị mới của khóa trạng thái `graph_state`. Giá trị mới do mỗi Node trả về sẽ ghi đè lên giá trị **state** trước đó.

Nếu luồng đồ thị là Node 1 đến Node 2, trạng thái cuối cùng sẽ là **"Chọn Trà"**.

Nếu luồng đồ thị là nút 1 đến nút 3, trạng thái cuối cùng sẽ là **"Chọn Coffee"**.

```
def node_1(state):
 print("---Node 1---")
 return {"graph_state": state['graph_state'] + " Tôi sẽ quyết định"}
```

```
def node_2(state):
 print("---Node 2---")
 return {"graph_state": state['graph_state'] + " Chọn Trà"}

def node_3(state):
 print("---Node 3---")
 return {"graph_state": state['graph_state'] + " Chọn Coffee"}
```

## 4. Xác định các Edges(cạnh) kết nối các Node

Cạnh thông thường đi từ Node này sang Node khác.

### 4.1 Edge không điều kiện giữa các node

```
builder = GraphBuilder()

Thêm các node
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

Tạo edge không điều kiện
builder.add_edge("node_1", "node_2") # Sau node_1 thì luôn đi đến node_2
builder.add_edge("node_2", "node_3") # Sau node_2 thì luôn đi đến node_3

Đặt node bắt đầu
builder.set_entry_point("node_1")

Tạo app và chạy thử
app = builder.compile()

initial_state = {"graph_state": ""}
final_state = app.invoke(initial_state)
print("Kết quả:", final_state["graph_state"])
```

- `add_edge("node_1", "node_2")`: Khi node\_1 chạy xong, **luôn luôn** chuyển sang node\_2.
- `add_edge("node_2", "node_3")`: Khi node\_2 chạy xong, **luôn luôn** chuyển sang node\_3.
- Không cần bất kỳ điều kiện nào – đó là lý do gọi là **unconditional edge**.

```
---Node 1---
---Node 2---
```

---Node 3---

Kết quả: Tôi sẽ quyết định. Chọn Trà. Chọn Coffee.

## 4.2 Edge Có điều kiện giữa các node

Cạnh có điều kiện được sử dụng khi bạn muốn tùy chọn định tuyến giữa các Node. Chúng là các hàm chọn nút tiếp theo dựa trên một số logic.

Trong ví dụ trên, để sao chép kết quả quyết định của người dùng, chúng ta sẽ xác định một hàm để mô phỏng xác suất **Chọn Trà** là 60%.

```
import random
from typing import Literal

def decide_drink(state) -> Literal["node_2", "node_3"]:
 """
 Quyết định nút tiếp theo dựa trên xác suất chia 60/40.
 """
 # Mô phỏng xác suất 60% cho "node_2"
 if random.random() < 0.6: # 60% chance
 return "node_2"

 # Còn lại 40% cơ hội
 return "node_3"
```

```
from IPython.display import Image, display
from langgraph.graph import StateGraph, START, END

Build graph
builder = StateGraph(State)

builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

Add the logic of the graph
builder.add_edge(START, "node_1")
builder.add_conditional_edges("node_1", decide_drink) #decide_drink trả về tên `node 2` hoặc `node 3`
builder.add_edge("node_2", END)
builder.add_edge("node_3", END)

Compile the graph
```

```
graph = builder.compile()

Visualize the graph
display(Image(graph.get_graph().draw_mermaid_png()))
```

Đầu tiên, chúng ta khởi tạo StateGraph với lớp State mà chúng ta đã định nghĩa ở trên.

Sau đó, chúng ta thêm các nút và cạnh.

START Node là một nút đặc biệt gửi dữ liệu đầu vào của người dùng đến **graph**, để chỉ ra nơi bắt đầu đồ thị của chúng ta.

```
builder.add_conditional_edges("node_1", decide_dring)
```

Mỗi lần hàm được gọi, nó đưa ra quyết định dựa trên cơ hội ngẫu nhiên:

60% cơ hội: Trả về "node\_2".

40% cơ hội: Trả về "node\_3".

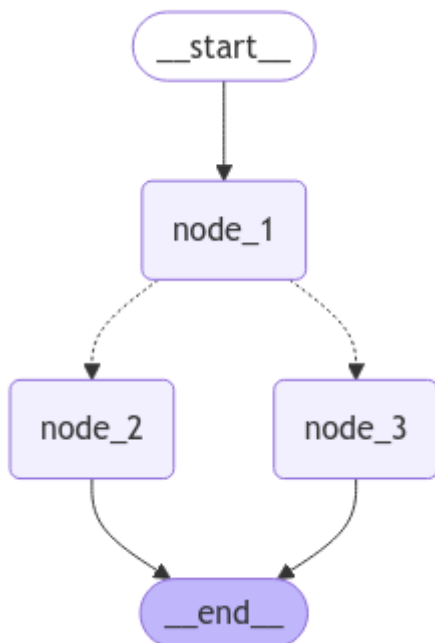
Tại sao sử dụng điều này?

Để ra quyết định: Nó mô phỏng một lựa chọn xác suất, có thể hữu ích trong các ứng dụng như mô phỏng, trò chơi hoặc học máy.

Kiểu trả về theo nghĩa đen đảm bảo rằng hàm sẽ luôn chỉ trả về "node\_2" hoặc "node\_3", giúp dễ dự đoán và gỡ lỗi hơn.

END Node là một nút đặc biệt biểu thị một nút đầu cuối.

Chúng ta biên dịch **graph** của mình để thực hiện một vài kiểm tra cơ bản trên cấu trúc **graph**.



## 5. Chạy ứng dụng

Graph đã biên dịch triển khai giao thức runnable, một cách chuẩn để thực thi các thành phần LangChain. Do đó, chúng ta có thể sử dụng invoke làm một trong những phương pháp chuẩn để chạy ứng dụng này.

Đầu vào ban đầu của chúng ta là từ điển {"graph\_state": "Xin chào, đây là VHTSoft."}, từ điển này đặt giá trị ban đầu cho từ điển trạng thái.

Khi invoke được gọi:

Biểu đồ bắt đầu thực thi từ nút START.

Nó tiến triển qua các nút đã xác định (node\_1, node\_2, node\_3) theo thứ tự.

Cạnh có điều kiện sẽ đi qua từ nút 1 đến nút 2 hoặc 3 bằng quy tắc quyết định 60/40.

Mỗi hàm nút nhận trạng thái hiện tại và trả về một giá trị mới, giá trị này sẽ ghi đè trạng thái biểu đồ.

Việc thực thi tiếp tục cho đến khi đạt đến nút END.

```
graph.invoke({"graph_state" : "Hi, Tôi là VHTSoft."})
```

```
{'graph_state': 'Hi, Tôi là VHTSoft. Tôi sẽ quyết định Chọn Trà'}
```

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**