LangGraph

# 🦜🕸️LangGraph

⚡ Building language agents as graphs ⚡

## Overview

LangGraph is a library for building stateful, multi-actor applications with LLMs, built on top of (and intended to be used with) 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. It is inspired by Pregel and Apache Beam. The current interface exposed is one inspired by NetworkX.

The main use is for adding **cycles** to your LLM application. Crucially, this is NOT a **DAG** framework. If you want to build a DAG, you should use just use LangChain Expression Language.

Cycles are important for agent-like behaviors, where you call an LLM in a loop, asking it what action to take next.

## Installation

```
pip install langgraph
```

# Quick Start

Here we will go over an example of creating a simple agent that uses chat models and function calling. This agent will represent all state as a list of messages.

We will need to install some LangChain packages, as well as Tavily to use as an example tool.

```
pip install -U langchain langchain_openai
tavily-python
```

We also need to export some environment variables needed for our agent.

```
export OPENAI_API_KEY=sk-...
export TAVILY_API_KEY=tvly-...
```

Optionally, we can set up LangSmith for best-in-class observability.

```
export LANGCHAIN_TRACING_V2="true"
export LANGCHAIN_API_KEY=ls__...
```

## Set up the tools

We will first define the tools we want to use. For this simple example, we will use a built-in search tool via Tavily. However, it is really easy to create your own tools - see documentation here on how to do that.

```
from langchain_community.tools.tavily_search
import TavilySearchResults

tools = [TavilySearchResults(max_results=1)]
```

We can now wrap these tools in a simple ToolExecutor. This is a real simple class that takes in a ToolInvocation and calls that tool, returning the output. A ToolInvocation is any class with `tool` and `tool_input` attribute.

```
from langgraph.prebuilt import ToolExecutor

tool_executor = ToolExecutor(tools)
```

## Set up the model

Now we need to load the chat model we want to use. Importantly, this should satisfy two criteria:

1. It should work with messages. We will represent all agent state in the form of messages, so it needs to be able to work well with them.
2. It should work with OpenAI function calling. This means it should either be an OpenAI model or a model that exposes a similar interface.

Note: these model requirements are not requirements for using LangGraph - they are just requirements for this one example.

```
from langchain_openai import ChatOpenAI

# We will set streaming=True so that we can
stream tokens
# See the streaming section for more
information on this.
model = ChatOpenAI(temperature=0,
streaming=True)
```

After we've done this, we should make sure the model knows that it has these tools available to call. We can do this by converting the LangChain tools into the format for OpenAI function calling, and then bind them to the model class.

```
from langchain.tools.render import
format_tool_to_openai_function

functions =
[format_tool_to_openai_function(t) for t in
tools]
model = model.bind_functions(functions)
```

## Define the agent state

The main type of graph in `langgraph` is the `StatefulGraph`. This graph is parameterized by a state object that it passes around to each node. Each node then returns operations to update that state. These operations can either SET specific attributes on the state (e.g. overwrite the existing values) or ADD to the existing attribute. Whether to set or add is denoted by annotating the state object you construct the graph with.

For this example, the state we will track will just be a list of messages. We want each node to just add messages to that list. Therefore, we will use a `TypedDict` with one key (`messages`) and annotate it so that the `messages` attribute is always added to.

```
from typing import TypedDict, Annotated,
Sequence
import operator
```

```
from langchain_core.messages import
BaseMessage


class AgentState(TypedDict):
    messages:
Annotated[Sequence[BaseMessage],
operator.add]
```

## Define the nodes

We now need to define a few different nodes in our graph. In `langgraph`, a node can be either a function or a runnable. There are two main nodes we need for this:

1. The agent: responsible for deciding what (if any) actions to take.
2. A function to invoke tools: if the agent decides to take an action, this node will then execute that action.

We will also need to define some edges. Some of these edges may be conditional. The reason they are conditional is that based on the output of a node, one of several paths may be taken. The path that is taken is not known until that node is run (the LLM decides).

1. Conditional Edge: after the agent is called, we should either: a. If the agent said to take an action, then the

function to invoke tools should be called b. If the agent said that it was finished, then it should finish

2. Normal Edge: after the tools are invoked, it should always go back to the agent to decide what to do next

Let's define the nodes, as well as a function to decide how what conditional edge to take.

```
from langgraph.prebuilt import ToolInvocation
import json
from langchain_core.messages import FunctionMes

# Define the function that determines whether t
def should_continue(state):
    messages = state['messages']
    last_message = messages[-1]
    # If there is no function call, then we fin
    if "function_call" not in last_message.addi
        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 a
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 funct
    last_message = messages[-1]
    # We construct an ToolInvocation from the f
    action = ToolInvocation(
        tool=last_message.additional_kwargs["fu

tool_input=json.loads(last_message.additional_k
["arguments"]),
    )
    # We call the tool_executor and get back a
    response = tool_executor.invoke(action)
    # We use the response to create a FunctionM
    function_message = FunctionMessage(content=
name=action.tool)
    # We return a list, because this will get a
list
    return {"messages": [function_message]}
```

## Define the graph

We can now put it all together and define the graph!

```
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`
# This means that this node is the first one
called
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()
```

## Use it!

We can now use it! This now exposes the same interface as all other LangChain runnables. This runnable accepts a list of

messages.

```
from langchain_core.messages import
HumanMessage

inputs = {"messages":
[HumanMessage(content="what is the weather in
sf")]}
app.invoke(inputs)
```

This may take a little bit - it's making a few calls behind the scenes. In order to start seeing some intermediate results as they happen, we can use streaming - see below for more information on that.

# Streaming

LangGraph has support for several different types of streaming.

## Streaming Node Output

One of the benefits of using LangGraph is that it is easy to stream output as it's produced by each node.

```
inputs = {"messages":
[HumanMessage(content="what is the weather in
sf")]}
for output in app.stream(inputs):
    # stream() yields dictionaries with
output keyed by node name
    for key, value in output.items():
        print(f"Output from node '{key}':")
        print("---")
        print(value)
    print("\n---\n")
```

```
Output from node 'agent':
---
{'messages': [AIMessage(content='',
additional_kwargs={'function_call': {'arguments
'{\n  "query": "weather in San Francisco"\n}',
'name': 'tavily_search_results_json'}})]}

---

Output from node 'action':
---
{'messages': [FunctionMessage(content="[{'url':
'https://weatherspark.com/h/m/557/2024/1/Histor
Weather-in-January-2024-in-San-Francisco-Califo
United-States', 'content': 'January 2024 Weathe
History in San Francisco California, United Sta
Daily Precipitation in January 2024 in San Fran
```

```
Observed Weather in January 2024 in San Francis
San Francisco Temperature History January 2024
Hourly Temperature in January 2024 in San Franc
Hours of Daylight and Twilight in January 2024
San FranciscoThis report shows the past weather
San Francisco, providing a weather history for
January 2024. It features all historical weathe
data series we have available, including the Sa
Francisco temperature history for January 2024.
can drill down from year to month and even day
reports by clicking on the graphs.'}]",
name='tavily_search_results_json')]}

---

Output from node 'agent':
---
{'messages': [AIMessage(content="I couldn't fin
current weather in San Francisco. However, you
visit [WeatherSpark]
(https://weatherspark.com/h/m/557/2024/1/Histor
Weather-in-January-2024-in-San-Francisco-Califo
United-States) to check the historical weather
for January 2024 in San Francisco.")]}

---

Output from node '__end__':
---
{'messages': [HumanMessage(content='what is the
weather in sf'), AIMessage(content='',
```

```
additional_kwargs={'function_call': {'arguments
'{\n  "query": "weather in San Francisco"\n}',
'name': 'tavily_search_results_json'}}),
FunctionMessage(content="[{'url':
'https://weatherspark.com/h/m/557/2024/1/Histor
Weather-in-January-2024-in-San-Francisco-Califo
United-States', 'content': 'January 2024 Weathe
History in San Francisco California, United Sta
Daily Precipitation in January 2024 in San Fran
Observed Weather in January 2024 in San Francis
San Francisco Temperature History January 2024
Hourly Temperature in January 2024 in San Franc
Hours of Daylight and Twilight in January 2024
San FranciscoThis report shows the past weather
San Francisco, providing a weather history for
January 2024. It features all historical weathe
data series we have available, including the Sa
Francisco temperature history for January 2024.
can drill down from year to month and even day
reports by clicking on the graphs.'}]",
name='tavily_search_results_json'),
AIMessage(content="I couldn't find the current
weather in San Francisco. However, you can visi
[WeatherSpark]
(https://weatherspark.com/h/m/557/2024/1/Histor
Weather-in-January-2024-in-San-Francisco-Califo
United-States) to check the historical weather
for January 2024 in San Francisco.")]}

---
```

## Streaming LLM Tokens

You can also access the LLM tokens as they are produced by each node. In this case only the "agent" node produces LLM tokens. In order for this to work properly, you must be using an LLM that supports streaming as well as have set it when constructing the LLM (e.g. `ChatOpenAI(model="gpt-3.5-turbo-1106", streaming=True)`)

```
inputs = {"messages":
[HumanMessage(content="what is the weather in
sf")]}
async for output in app.astream_log(inputs,
include_types=["llm"]):
    # astream_log() yields the requested logs
(here LLMs) in JSONPatch format
    for op in output.ops:
        if op["path"] == "/streamed_output/-
":
            # this is the output from
.stream()
            ...
        elif op["path"].startswith("/logs/")
and op["path"].endswith(
            "/streamed_output/-"
        ):
            # because we chose to only
```

```
include LLMs, these are LLM tokens
            print(op["value"])
```

```
content='' additional_kwargs=
{'function_call': {'arguments': '', 'name':
'tavily_search_results_json'}}
content='' additional_kwargs=
{'function_call': {'arguments': '{\n',
'name': ''}}
content='' additional_kwargs=
{'function_call': {'arguments': ' ', 'name':
''}}
content='' additional_kwargs=
{'function_call': {'arguments': ' "', 'name':
''}}
content='' additional_kwargs=
{'function_call': {'arguments': 'query',
'name': ''}}
content='' additional_kwargs=
{'function_call': {'arguments': '":', 'name':
''}}
content='' additional_kwargs=
{'function_call': {'arguments': ' "', 'name':
''}}
content='' additional_kwargs=
{'function_call': {'arguments': 'weather',
'name': ''}}
content='' additional_kwargs=
{'function_call': {'arguments': ' in',
'name': ''}}
```

```
content='' additional_kwargs=
{'function_call': {'arguments': ' San',
'name': ''}}
content='' additional_kwargs=
{'function_call': {'arguments': ' Francisco',
'name': ''}}
content='' additional_kwargs=
{'function_call': {'arguments': '"\n',
'name': ''}}
content='' additional_kwargs=
{'function_call': {'arguments': '}', 'name':
''}}
content=''
content=''
content='I'
content="'m"
content=' sorry'
content=','
content=' but'
content=' I'
content=' couldn'
content="'t"
content=' find'
content=' the'
content=' current'
content=' weather'
content=' in'
content=' San'
content=' Francisco'
content='.'
content=' However'
```

```
content=','
content=' you'
content=' can'
content=' check'
content=' the'
content=' historical'
content=' weather'
content=' data'
content=' for'
content=' January'
content=' '
content='202'
content='4'
content=' in'
content=' San'
content=' Francisco'
content=' ['
content='here'
content='](('
content='https'
content='://'
content='we'
content='athers'
content='park'
content='.com'
content='/h'
content='/m'
content='/'
content='557'
content='/'
content='202'
```

```
content='4'
content='/'
content='1'
content='/H'
content='istorical'
content='-'
content='Weather'
content='-in'
content='-Jan'
content='uary'
content='-'
content='202'
content='4'
content='-in'
content='-S'
content='an'
content='-F'
content='r'
content='anc'
content='isco'
content='-Cal'
content='ifornia'
content='-'
content='United'
content='-'
content='States'
content=').'
content=''
```

# When to Use

When should you use this versus LangChain Expression Language?

If you need cycles.

Langchain Expression Language allows you to easily define chains (DAGs) but does not have a good mechanism for adding in cycles. `langgraph` adds that syntax.

# Examples

## ChatAgentExecutor: with function calling

This agent executor takes a list of messages as input and outputs a list of messages. All agent state is represented as a list of messages. This specifically uses OpenAI function calling. This is recommended agent executor for newer chat based models that support function calling.

- Getting Started Notebook: Walks through creating this type of executor from scratch
- High Level Entrypoint: Walks through how to use the high level entrypoint for the chat agent executor.

### Modifications

We also have a lot of examples highlighting how to slightly modify the base chat agent executor. These all build off the getting started notebook so it is recommended you start with that first.

- Human-in-the-loop: How to add a human-in-the-loop component
- Force calling a tool first: How to always call a specific tool first
- Respond in a specific format: How to force the agent to respond in a specific format
- Dynamically returning tool output directly: How to dynamically let the agent choose whether to return the result of a tool directly to the user
- Managing agent steps: How to more explicitly manage intermediate steps that an agent takes

## AgentExecutor

This agent executor uses existing LangChain agents.

- Getting Started Notebook: Walks through creating this type of executor from scratch

- High Level Entrypoint: Walks through how to use the high level entrypoint for the chat agent executor.

### Modifications

We also have a lot of examples highlighting how to slightly modify the base chat agent executor. These all build off the getting started notebook so it is recommended you start with that first.

- Human-in-the-loop: How to add a human-in-the-loop component
- Force calling a tool first: How to always call a specific tool first
- Managing agent steps: How to more explicitly manage intermediate steps that an agent takes

## Multi-agent Examples

- Multi-agent collaboration: how to create two agents that work together to accomplish a task
- Multi-agent with supervisor: how to orchestrate individual agents by using an LLM as a "supervisor" to distribute work
- Hierarchical agent teams: how to orchestrate "teams" of agents as nested graphs that can collaborate to solve a problem

### Chatbot Evaluation via Simulation

It can often be tough to evaluation chat bots in multi-turn situations. One way to do this is with simulations.

- Chat bot evaluation as multi-agent simulation: How to simulate a dialogue between a "virtual user" and your chat bot

### Async

If you are running LangGraph in async workflows, you may want to create the nodes to be async by default. In order for a walkthrough on how to do that, see this documentation

### Streaming Tokens

Sometimes language models take a while to respond and you may want to stream tokens to end users. For a guide on how to do this, see this documentation

## Documentation

There are only a few new APIs to use.

### StateGraph

The main entrypoint is `StateGraph`.

```
from langgraph.graph import StateGraph
```

This class is responsible for constructing the graph. It exposes an interface inspired by NetworkX. This graph is parameterized by a state object that it passes around to each node.

`__init__`

```
    def __init__(self, schema: Type[Any]) ->
None:
```

When constructing the graph, you need to pass in a schema for a state. Each node then returns operations to update that state. These operations can either SET specific attributes on the state (e.g. overwrite the existing values) or ADD to the existing attribute. Whether to set or add is denoted by annotating the state object you construct the graph with.

The recommended way to specify the schema is with a typed dictionary: `from typing import TypedDict`

You can then annotate the different attributes using `from typing imoport Annotated`. Currently, the only supported annotation is `import operator; operator.add`. This

annotation will make it so that any node that returns this attribute ADDS that new result to the existing value.

Let's take a look at an example:

```
from typing import TypedDict, Annotated,
Union
from langchain_core.agents import
AgentAction, AgentFinish
import operator


class AgentState(TypedDict):
    # The input string
    input: str
    # The outcome of a given call to the agent
    # Needs `None` as a valid type, since this
is what this will start as
    agent_outcome: Union[AgentAction,
AgentFinish, None]
    # List of actions and corresponding
observations
    # Here we annotate this with
`operator.add` to indicate that operations to
    # this state should be ADDED to the
existing values (not overwrite it)
    intermediate_steps:
Annotated[list[tuple[AgentAction, str]],
```

```
operator.add]
```

We can then use this like:

```
# Initialize the StateGraph with this state
graph = StateGraph(AgentState)
# Create nodes and edges
...
# Compile the graph
app = graph.compile()

# The inputs should be a dictionary, because
the state is a TypedDict
inputs = {
    # Let's assume this the input
    "input": "hi"
    # Let's assume agent_outcome is set by the
graph as some point
    # It doesn't need to be provided, and it
will be None by default
    # Let's assume `intermediate_steps` is
built up over time by the graph
    # It doesn't need to provided, and it will
be empty list by default
    # The reason `intermediate_steps` is an
empty list and not `None` is because
    # it's annotated with `operator.add`
}
```

### `.add_node`

```
    def add_node(self, key: str, action:
RunnableLike) -> None:
```

This method adds a node to the graph. It takes two arguments:

- `key`: A string representing the name of the node. This must be unique.
- `action`: The action to take when this node is called. This should either be a function or a runnable.

### `.add_edge`

```
    def add_edge(self, start_key: str,
end_key: str) -> None:
```

Creates an edge from one node to the next. This means that output of the first node will be passed to the next node. It takes two arguments.

- `start_key`: A string representing the name of the start node. This key must have already been registered in the graph.
- `end_key`: A string representing the name of the end node. This key must have already been registered in the graph.

## `.add_conditional_edges`

```
    def add_conditional_edges(
        self,
        start_key: str,
        condition: Callable[..., str],
        conditional_edge_mapping: Dict[str,
str],
    ) -> None:
```

This method adds conditional edges. What this means is that only one of the downstream edges will be taken, and which one that is depends on the results of the start node. This takes three arguments:

- `start_key`: A string representing the name of the start node. This key must have already been registered in the graph.
- `condition`: A function to call to decide what to do next. The input will be the output of the start node. It should return a string that is present in `conditional_edge_mapping` and represents the edge to take.
- `conditional_edge_mapping`: A mapping of string to string. The keys should be strings that may be returned by `condition`. The values should be the downstream node to call if that condition is returned.

### `.set_entry_point`

```
    def set_entry_point(self, key: str) ->
None:
```

The entrypoint to the graph. This is the node that is first called. It only takes one argument:

- `key`: The name of the node that should be called first.

### `.set_finish_point`

```
    def set_finish_point(self, key: str) ->
None:
```

This is the exit point of the graph. When this node is called, the results will be the final result from the graph. It only has one argument:

- `key`: The name of the node that, when called, will return the results of calling it as the final output

Note: This does not need to be called if at any point you previously created an edge (conditional or normal) to `END`

## Graph

```
from langgraph.graph import Graph

graph = Graph()
```

This has the same interface as `StateGraph` with the exception that it doesn't update a state object over time, and rather relies on passing around the full state from each step. This means that whatever is returned from one node is the input to the next as is.

**`END`**

```
from langgraph.graph import END
```

This is a special node representing the end of the graph. This means that anything passed to this node will be the final output of the graph. It can be used in two places:

- As the `end_key` in `add_edge`
- As a value in `conditional_edge_mapping` as passed to `add_conditional_edges`

# Prebuilt Examples

There are also a few methods we've added to make it easy to use common, prebuilt graphs and components.

## ToolExecutor

```
from langgraph.prebuilt import ToolExecutor
```

This is a simple helper class to help with calling tools. It is parameterized by a list of tools:

```
tools = [...]
tool_executor = ToolExecutor(tools)
```

It then exposes a runnable interface. It can be used to call tools: you can pass in an AgentAction and it will look up the relevant tool and call it with the appropriate input.

## chat_agent_executor.create_function_calling_executor

```
from langgraph.prebuilt import
chat_agent_executor
```

This is a helper function for creating a graph that works with a chat model that utilizes function calling. Can be created by

passing in a model and a list of tools. The model must be one that supports OpenAI function calling.

```
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search im
TavilySearchResults
from langgraph.prebuilt import chat_agent_execu
from langchain_core.messages import HumanMessag

tools = [TavilySearchResults(max_results=1)]
model = ChatOpenAI()

app =
chat_agent_executor.create_function_calling_exe
tools)

inputs = {"messages": [HumanMessage(content="wh
weather in sf")]}
for s in app.stream(inputs):
    print(list(s.values())[0])
    print("----")
```

## create_agent_executor

```
from langgraph.prebuilt import
create_agent_executor
```

This is a helper function for creating a graph that works with LangChain Agents. Can be created by passing in an agent and a list of tools.

```
from langgraph.prebuilt import
create_agent_executor
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain.agents import
create_openai_functions_agent
from langchain_community.tools.tavily_search
import TavilySearchResults

tools = [TavilySearchResults(max_results=1)]

# Get the prompt to use - you can modify
this!
prompt = hub.pull("hwchase17/openai-
functions-agent")

# Choose the LLM that will drive the agent
llm = ChatOpenAI(model="gpt-3.5-turbo-1106")

# Construct the OpenAI Functions agent
agent_runnable =
create_openai_functions_agent(llm, tools,
prompt)

app = create_agent_executor(agent_runnable,
tools)
```

```
inputs = {"input": "what is the weather in
sf", "chat_history": []}
for s in app.stream(inputs):
    print(list(s.values())[0])
    print("----")
```