Maria Manina
IM WS 2025/26
LangChain experiments description

1 Introduction

LangChain agents work in the ReAct (Reasoning + Acting) loop, alternating between brief reasoning steps with tool calls and feeding the resulting observations into subsequent decisions. This iterative process continues until the agent determines that it is "confident" enough to produce the final answer.

In addition to the underlying **language model** and available **tool sets**, several components or controls can enhance LangChain agents' functionality. The following highlights focus specifically on features most suitable (in my opinion) for the integration within the **clembench** framework:

- System prompt(s)
- Structured output format
- Error/exception handling
- Model/tool call limits
- File search

In my opinion, tools hold significant potential for achieving higher performance levels since they enable (sometimes a simplified) simulation of various advanced strategies typically employed in game playing LLMs.

2 Experiments

2.1 (More or less) Successful experiments:

1. Baseline

   The baseline consists of a vanilla LLM with a tag extraction tool and a short-term memory component, integrated into a LangChain agent framework. The tag extraction tool aims to minimize the number of episodes where the game aborts due to incorrect response formatting or non-parsable answers. This feature becomes critical given that out of 14 clembench v2.0 games, only 4 allow self correction — i.e., do not abort immediately after a wrong answer. Excluding such cases will allow to evaluate architectural efficiency free from biases. The tag extraction tool operates as a subagent prompted with one-shot (see prompts in the **Appendix A**).

   The short-term memory component is necessary because the basic clemcore/playpen ClemAgent's observation only includes the most recent iteration. Hence, earlier contextual details may be lost, in comparison with the earlier versions using the Player class. The implementation follows the pattern of InMemorySaver class, and the memory is wiped after each episode.

   Language model: gpt-oss-20b

Results folder: https://github.com/mariamaninna/clemcore/tree/agent_experiments/game_results/baseline

Table 1: Baseline metrics

Game	clemscore	% Played	Quality Score	Quality Score (Std)
taboo	30.00	80.0	37.5	48.27
referencegame	79.17	95.83	82.61	38.76
textmapworld	56.79	88.0	64.53	16.16

2. Agent + core tools set

   The idea was to make the game-playing process more transparent and to simulate certain capabilities that are the focus of research into LLM-based gameplay (e.g., structural planning and iterative processes; game-based strategic behavior and decision-making; context-guided reasoning and long-term planning). To achieve this, the following tools were proposed:

   - **observe** writes down something important that happened throughout the game (e.g., some place was visited, or a forbidden word was used for the description)
   - **get_observations** returns everything **observe** has recorded
   - **remember** writes down important rules (answer format, forbidden words etc.)
   - **recall** returns every available rule that is stored

   As the tool use is potentially infinite, the agent is explicitly asked to call tools only 1-2 times per invocation. A tool call limit (=20) and a reduced max_tokens (=150) were also tested, but this resulted in the episode not being played at all (the tool call limit was exceeded before the answer had been given).

   Language model: gpt-4o-mini

   Results folder: https://github.com/mariamaninna/clemcore/tree/agent_experiments/game_results/agent_core_tools

Table 2: Universal tool set agent metrics

Game	clemscore	% Played	Quality Score	Quality Score (Std)
taboo	34.53	92.86	37.18	46.23
referencegame	54.17	100.0	54.17	50.90
textmapworld	0.00	100.0	0.00	0.00

2.2 What has been partly implemented, but I doubt its adequacy given the current results:

Optional tool sets

What can also be added is a dynamic tool set that is integrated into the agent's core tools based on initial prompt analysis (e.g., keyword-based). Examples include: a navigation tool set (for

textmapworld, adventuregame), a multimodal tool set (cloudgame, multimodal referencegame), and a word-guessing tool set (taboo, wordle). However, tool calls should be controlled, and testing here is only appropriate for models trained specifically for tool use—others are expected to yield poor results.

2.3 What is planned to be done, taking the last meeting into account :

1. Game transcripts access (via the LangChain external tools and/or via the Long-term memory)
2. Controlled ablations on the existing tools
3. Re-run the existing experiments with the clp-hosted qwen models (trained for better tool use)
4. Integrate the agent creation into the general clemcore/playpen pipeline (from notebooks to game running from terminal)
5. Some additional experiments

Appendix A

Baseline Prompts

Main agent prompt

```
You're going to play a game. You're a professional agent game player,
   but you also have a helpful tool extract_tags, that can help you.
   First of all, call the extract_tags tool, and use the result.
```

Subagent prompt

```
You are given a small piece of text which contains gameplay rules.
   You need to extract the necessary tags (often written in CAPITAL
   LETTERS), so that the player can use them for the answer. Do not
   output any text apart from the tag(s). Example IO pair:

INPUT:
Let's play a guessing game! Your task is to answer the other player's
   questions. Based on your knowledge of the word: \$TARGET WORD\$,
   respond to the following questions or guesses. Limit your response
   to only 'yes' or 'no' with no explanation or other words. Never
   reveal the answer in your response.

You must reply using the format below and DO NOT ADD ANY TEXT OTHER
   THAN THIS:

ANSWER: <some text>

Target Word: $TARGET WORD$

OUTPUT: ANSWER:
```

```
If you identified no tags, please return NO TAG as an answer.
```

Agent + core tool set prompt

```
You're a professional game player with memory tools.

STRATEGY:
1. On FIRST turn: understand the rules, store key info with remember()
2. After 1-2 tool calls, you MUST give your final answer.

Do NOT keep calling tools.
```