Homework 4 - Final Report

Sarah Morin

Results

Accuracy Number: 4/20 = 20%
Full Results: final_results.json

Discussion

My initial implementation was simple:

- Add the `run_bash_cmd`, `replace_in_file`, and `show_file` tools
- Create a minimal system prompt
- Run the basic ReAct loop

Unfortunately, this version only produced 1 or 2 successful instances. I saw the following errors frequently:

- Empty Patches → Most of the instances (15+/20) were empty (meaning the agent did not actually apply its changes to the original source code files.)
- Malformed Responses → Usually an empty message of the form:

```

----END_FUNCTION_CALL----
```

- Calls to non-existent tools or interpreting the response template too literally and trying to call `function_name`.

I tried the following:

1. Improving the system prompt → I added more detailed instructions to the system prompt with specific instructions about response format, to only use tools available, and a basic process to follow to implement code changes.
2. Adding `check_patch` logic → I added a function similar to generate patch that just returns `true` if the patch is non-empty and `false` otherwise. In the loop, if the agent calls run, but the patch is empty, we add a message indicating the patch is empty and that changes must be written in place and tell the agent to continue iterating. *Note: this does not override the max iterations.*

After these changes, I saw fewer empty patches, but I got the following results:

- Only 1 empty patch
- More complete instances (6)
- Slightly more passing instances (4/6)
- Mostly error instances (13)

So although the agent was no longer producing empty patches (yay!) it was still struggling to write correct code. I still saw many malformed responses and calls to nonexistent functions. In response:

1. Improved system prompt again → I was very explicit about response format. I also included notes about paying attention to Python syntax and indentation. Many of the errors I saw during the evaluation were related to incorrectly formatted Python.
2. Introduce feedback on malformed responses and missing functions → When we see a malformed response or missing function, I add a message to the context with the result and a message explaining that the response does not follow the expected format or the tool did not come from the list of available tools. Hopefully, including this feedback will prevent the problem from snowballing.
3. Added more tools for common operations → I wanted to reduce the chance of 'incorrect bash command' situations for common operations and hopefully streamline the reasoning.

All of these changes combined actually hurt performance quite a bit, so my final strategy was this:

1. Simplify the system prompt → Reduce redundant instructions, use a more rigid and consistent format, simplify language.
2. Remove some of the tools → Some of the tools I tried adding actually caused the agent to much worse. Details below.
3. Eliminate the use of the "uesr" role everywhere except for the initial task prompt → In an earlier change, I added "feedback" messages from the user about malformed responses. I believe this simulated conversation caused the agent to produce results that were proposed solutions (waiting on approval form the user) rather than actually implementing them.

Even though this version still didn't solve any more instances, (and I still got empty patches and some malformed responses), the number of malformed responses and calls to fake functions decreased on the whole.

Custom Tools

In addition to the required `run_bash_cmd` and optional `replace_in_file` and `show_file`, I implemented the following tools. Most of these tools could be run with bash commands, but providing tools for common actions rather than relying on the agent to properly devise bash commands at every step seemed practical.

- `append_to_file()` → Append contents to the end of an existing file.
  - Good for adding unit tests to a main method call.
- `grep_in_file()` → Find lines with a given regex pattern in a file.
  - Identify what lines to use when calling `replace_in_file()`
- `check_python_syntax()` → Check if Python syntax is valid.
  - Avoid calling finish with solutions that don't run.
- `check_patch()` → Check if a patch has been written, i.e. return `true` if the patch is non-empty.
  - Avoid generating an empty patch by calling finish without actually writing the solution in the correct files.

Failed Tools

Here are the tools I added, but later removed from to the agent:

- `create_file()` → Create a new file with the given content.
  - **Seemed** convenient for generating a test script file

    - It actually made the agent populate the repo with a bunch of trash files instead of writing a patch.
- `list_dir()` → List contents of current directory.
    - **Seemed**: Good to determine where the source files are.
    - It actually just called this too much looking for things.
- `list_python_files` → List all python files in the current directory.
    - Same as above.