Learning Strategic Play with Language Agents in Text-Adventure Games

Stanford CS224N Custom Project

Nic Becker
Department of Computer Science
Stanford University
`nicbecker@stanford.edu`

Miranda Li
Department of Computer Science
Stanford University
`mirandal@stanford.edu`

Abstract

We explore the use of LLMs to power autonomous language agents in a text-adventure game environment. Text adventure games require flexible adaptation of playthrough strategy and action vocabulary based solely on natural language feedback. In this report, we present an implementation of two baselines, ReAct and Reflexion, in an interactive fiction environment designed for reinforcement learning. We then characterize what makes these games difficult for the baselines: 1) the limitations of keeping long game trajectories in-context and 2) the difficulty of inferring the set of available actions at any point in the game. We use these failure modes to motivate two extensions to Reflexion, namely a module for utilizing long-term memory and a module for self-verification of actions. We present a successful implementation of automatic action self-verification and candidate action generation. We implemented a memory storage and retrieval module which did not improve upon our baselines, and we hypothesize that the memory storage mechanism is to blame.

No external collaborators, no external mentor, not sharing project.

1 Introduction

Large language models (LLMs) have recently shown success in powering autonomous, embodied agents within a diverse range of external environments (Wang et al., 2023; Shinn et al., 2023; Yao et al., 2023b). In contrast to reinforcement learning (RL) and imitation learning approaches, current approaches of LLM-powered language agents do not employ gradient-based training or fine-tuning to adapt to a new domain. Rather, these agents use in-context examples to leverage knowledge learned by a pre-trained LLM to generate action plans, API calls, or code that can be executed in an environment Wang et al. (2023); Yao et al. (2022); Schick et al. (2023); Yao et al. (2023a). Since these agents do not change the weights of the underlying language model, designing agent architectures that support acquiring new knowledge, incorporating past experience, and adapting to online gameplay is a considerable challenge.

Interactive fiction presents several unique challenges compared to other interactive decision making benchmarks (Yao et al., 2022; Shridhar et al., 2020). In the games we test, agents must adapt to different notions of embodiment, gameplay conventions, reward structures, and goals. Moreover, a capable agent must be able to calibrate its strategy for exploration and planning to a particular game environment by only using natural language responses provided by the game. In this project, we explore the use of LLMs to power adaptive language agent architectures applied to text-adventure interactive fiction games. We first implement several existing language agents architectures Yao et al. (2023b); Shinn et al. (2023) in an interactive fiction environment. We then present an approach for long-term memory and self-validation of actions designed to overcome limitations of these approaches.

2 Related Work

2.1 Decision-making agents for interactive fiction

Interactive fiction games are a challenging test-bed for autonomous language-based agents. Historically, the predominant paradigm for this domain has been reinforcement learning. Microsoft's Jericho Hausknecht et al. (2020) provides a toolkit for reinforcement learning research with an accessible environment for human-made interactive fiction. To our knowledge, classical reinforcement learning approaches have only seen moderate success with games categorized in Jericho's easier 'Possible' category Jansen (2021). For instance, state-of-the-art performance on remains below 1% of the max score on *Anchorhead*, a title in the hard difficulty category Jansen (2021). Previous work that mirrors the self-validation module discussed in this paper: CALM uses a language model to generate a set of action candidates at each game state combined with a reinforcement learning agent which re-ranks the generated action candidates to maximize in-game rewards Yao et al. (2020). A notable non-reinforcement learning agent is NAIL Hausknecht et al. (2019) which learns from playing interactive fiction games by incorporating experience into a knowledge graph which is then used to reason about objects, locations, and actions using a set of manually-created heuristics.

2.2 Powering language agents with LLMs

Recently, there has been a substantial effort to utilize LLMs for smart planning and decision making in text agents. ReAct Yao et al. (2023b) uses Chain-of-Thought prompting Wei et al. (2023) to generate reasoning traces and actions with LLMs. Reflexion Shinn et al. (2023) extends this approach with a self-reflection step they conceptualize as verbal reinforcement learning. Generative Agents Park et al. (2023) uses ChatGPT to imitate human behavior by storing agents' experiences as memories and retrieving recent and relevant memories before acting. Voyager Wang et al. (2023) is a lifelong learning agent for Minecraft that uses an automatic curriculum for generating new actions, a skill library for storing complex behaviors, and iterative prompting mechanism to self-identify compilation and execution errors. Voyager is also able to generalize to novel tasks using the learned skill library. Interactive fiction typically places more constraints on a player than Minecraft. In this way, interactive fiction can be viewed as a semi-embodied, semi-open-ended environment. We take inspiration from these works to design improved reasoning and action generation for language agents in the domain of interactive fiction.

3 Approach

As interactive fiction is a less common domain for the current generation of LLM-based agents, we first adapt a number of representative algorithms as baselines. Several of these approaches were designed for NLP tasks rather than adaptive, semi-embodied agents, so we re-interpret to fit the structure of gameplay in text-adventure games. We then motivate two extensions, long-term memory and action self-verification, based on common failure modes of the baselines. We implement each agent using DSPy (Khattab et al., 2023) which begins each prompt with a high-level description of the task and a format to follow based on the agent architecture. Each experiment uses Mixtral-8x7B Team (2023) as the LLM backend and we play each game for a maximum of 50 steps or until victory.

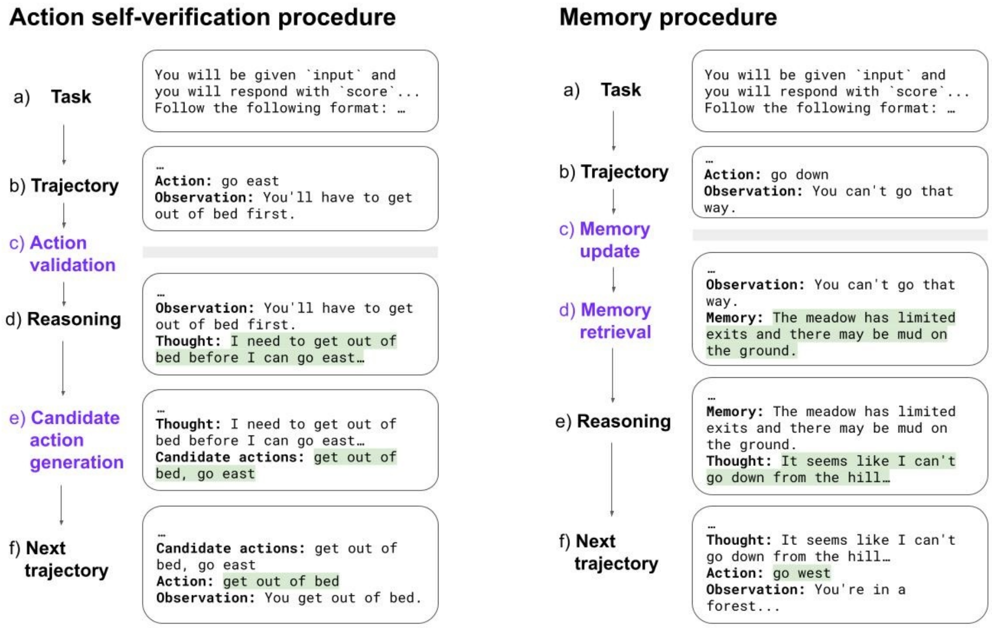
3.1 Baselines

ReAct (Yao et al., 2023b) uses chain-of-thought prompting to generate interleaved reasoning traces and action plans. We implemented a variant of ReAct based on its application to ALFWorld (Shridhar et al., 2020).

Reflexion (Shinn et al., 2023) adds a self-reflection step to ReAct to induce better decision-making across timesteps. We experimented with two variations: one where the agent is provided self-reflection if it repeatedly attempts improperly formatted actions, and one where it is provided self-reflection at regular 5 step intervals. Since we observed a marginal difference in performance between these procedures in our baselines, we solely refer to the latter variant as 'Reflexion' for the remainder of this report.

Our code for each implementation can be found at https://github.com/caenopy/if-agents/. Note that DSPy is an immature framework, and we had to make several internal changes not included in this repository which are needed to support the following experiments.

3.2 Learning strategic play from memories and action self-verification



You will be given `input` and you will respond with `score`... Follow the following format: …

…
Action: go east
Observation: You'll have to get out of bed first.

…
Observation: You'll have to get out of bed first.
Thought: I need to get out of bed before I can go east…

…
Thought: I need to get out of bed before I can go east…
Candidate actions: get out of bed, go east

…
Candidate actions: get out of bed, go east
Action: get out of bed
Observation: You get out of bed.

You will be given `input` and you will respond with `score`... Follow the following format: …

…
Action: go down
Observation: You can't go that way.

…
Observation: You can't go that way.
Memory: The meadow has limited exits and there may be mud on the ground.

…
Memory: The meadow has limited exits and there may be mud on the ground.
Thought: It seems like I can't go down from the hill…

…
Thought: It seems like I can't go down from the hill…
Action: go west
Observation: You're in a forest...

Figure 1: Overview of our extensions for (a) action self-verification and (b) long-term memory. We show the basic methods interleaved into a ReAct style trajectory with one 'Thought' step before action generation. These approaches may be similarly interleaved with Reflexion.

Figure 1 shows a conceptual overview of the procedures for action self-verification and memory agents. We exclude the self-reflection step used in Reflexion for clarity. In the experiments discussed below, we integrate these procedures into the Reflexion agent as it is the highest performing baseline. See the appendix for the prompts of each of the agents and example trajectories.

3.2.1 Giving language agents long-term memory

Motivation: While the baselines discussed in the previous section can in principle incorporate in-context experience to improve playing strategy within a single episode, in practice, relying on long-range in-context memories is not sufficient for playing interactive fiction games successfully. Moreover, each baseline approach is not well-suited for acquiring knowledge over multiple games. Since it is desirable for agents to learn from experience within an episode (e.g. if 10 timesteps ago, an item appeared that is relevant to the most recent observation, remember that item), we build a procedure for language agents to store long-term memories within an episode. While the same approach can be used to provide memory to agents across multiple games, we leave this experiment to future work.

Approach: Inspired by Park et al. (2023), we implement a long-term memory module to give agents the capability to flexibly store and retrieve experiences when reasoning about the next action. Our approach uses a persistent *memory stream* similar to Generative Agents.

After receiving an observation from the game environment, a **write-to-memory** module updates the memory stream provided that the observation contains *novel* information that is not already present in the memory stream (See Appendix section A.5). This can be information about the player's location, rewards the player has received, or any other observation derived from the current game state.

Following this, our architecture implements a retrieval function that takes the current observation and returns a short synthesis of the memory stream. We implement this function as a **fetch-from-memory** module that prompts an LLM to summarize a relevant subset of the memory stream based on how *relevant* the information is to the most recent observation (see Appendix section A.6). This memory is then provided to the primary actor LLM to inform the next action (Figure 1). We use Chain-of-Thought prompting Wei et al. (2023) for both the write and fetch module.

3.2.2 Action self-verification

Motivation: Given that the actions available to a player are not explicitly provided at any point in the game, learning the rules of the game while playing it is one of the central challenges of interactive fiction. This challenge corresponds to a common failure mode of the baselines, as we discuss in results. Observing this failure mode motivates the following self-verification procedure for learning valid actions.

Approach: For action self-verification we maintain two caches: one for invalid actions the agent has seen before and another for valid actions the agent has seen before. We initialize the invalid actions cache as empty and we initialize the valid actions cache with a small set of universal interactive fiction commands: "go north", "go south", "go east", "go west", "look", "examine x", "take x", "drop x", "inventory", "restart".

At each turn, we call an **update-action-cache** module which takes as input the current invalid and valid action caches as well as the most recent action taken by the agent and the observation that resulted from that action (the most recent observation). The module prompts an LLM to judge if the action was recognized as valid by the game engine's parser. Based on this prediction we add the action to either the invalid or valid actions cache. See Appendix section A.3 for an example of one call to this module.

Before taking an action, we then implement a hybrid generation/retrieval function that provides candidate actions to the primary actor agent. The **generate-action-candidates** module takes as input the current invalid and valid action caches as well as the most recent thought produced by the agent in the "Reasoning" step. The module instructs an LLM to translate the thought into a list of parser-recognized actions which reflect the goals of the thought while also respecting the action syntax. It outputs this list as a string of choices for the agent to choose from. See Appendix section A.4 for an example of one call to this module. The game engine executes the produced action to yield another observation and the next turn begins.

4 Experiments

4.1 Data

We evaluate each agent on games in the 'possible' difficulty category in the Jericho dataset (Hausknecht et al., 2020) for a maximum of 50 steps or until victory. At step of the game, an agent receives an observation from the game environment and must reply with an action (Figure 2).

```
The table has a fresh, white tablecloth.
On it you see a bunch of bananas...
> get bananas
Taken.
> rub lamp
Everything seems to be spinning round...
You gaze around at your new surroundings...
The lamp flies out of your hand and disappears from sight...
```

Figure 2: An excerpt from William Stott's Dragon.

4.2 Evaluation method

Given the diversity of reward structures in the games tested, there is no single metric that captures incremental performance of different agents across all games. In particular, the score achieved by an agent in a game is problematic as some games only provide a score in the event of winning the game. In addition to measuring the *normalized score* – the achieved score divided by the maximum score achievable in each game – provided by Jericho (Hausknecht et al., 2020), we define several metrics for quantitatively evaluating our agent's improvements across several failure modes.

4.2.1 Agents struggle to generate actions accepted by the game's parser

In our baseline experiments, we noticed that models frequently generated invalid actions which were not recognized by the interactive fiction game's parser (for example, “go north" is a valid command whereas “follow the path to the north" is not). This is a difficult task even for human players, since the player is not provided a list of available actions and each game has a different set of parser-recognized words and phrase structures. Highly unusual words and phrases (like the name of a spell, for example) may be valid while reasonable, more common phrases (like “walk down the road") might not. However, most parser-recognized actions tend to be simple phrases no longer than 2 or 3 words, and the player can only present one action in each turn.

To further complicate things, sometimes commands are parsed correctly but unsuccessful due to the constraints of the particular game. For example, ‘go east’ is always parser-valid but only successful when there is a location to your east, and ’take key’ is always parser-valid but unsuccessful if there is no key nearby.

To measure the ability of the agent to understand the action space we record the *total number of valid moves* in each game.

Further, our baseline language agents often got stuck presenting excessively long directives (e.g. "look for a heavy object to break the lock on the trunk", "Go to bedroom and open dresser to put on clean clothes") or multiple actions in one turn and failing to recognize that the game was not parsing the command. We frequently observed a kind of "fixation" behavior where the agent would repeatedly try slight rewordings or rephrasings of failed commands or the same failed command over and over.

In an attempt to quantitatively measure this failure mode, we create a heuristic which is the *count of repeated 2-word prefixes of invalid actions*. That is, for each game, for all actions which the game fails to recognize either due to parser failure or game mechanics, we count their 2-word prefixes. The heuristic score is the total count of all of these 2-word prefixes which occur more than once. See Appendix section A.1 for a more detailed explanation with an example. Intuitively, a higher score on this heuristic is worse (failure to explore the action space) and a lower score is better.

4.2.2 Agents struggle to use previous experience for improving exploration strategies

In our initial experiments we also observed that agents would get stuck in infinite loops between a few states, going from one to the other and back and performing the exact same sequence of actions. This is a failure to explore the state space of the game. To measure this we keep track of the *number of unique game states* visited by the agent, which we learn by querying the game engine (Hausknecht et al. (2020)); we hope the agent will explore a greater number of unique game states.

Finally we also examine the playthroughs for each trial to analyze reasoning traces and identify where agents struggle to make progress.

4.3 Experimental details

4.3.1 Learning strategic play from memories

A common failure mode exhibited by each of the baseline agent architectures is an inability to leverage previous experience for better decision-making. For instance, in games where the player can die and restart in the span of a single episode, we observe that baseline architectures were not sufficient for avoiding the situation that led to the player's demise in the next attempt. We hypothesize that this is due to the fact that ReAct and Reflexion force the agent to rely on in-context experience that may be many timesteps in the past. Therefore, bringing these memories closer to the reasoning trace of the next action may improve agents ability to learn from experience within a single episode. We use the number of unique states as proxy for evaluating the exploration abilities of agents with long-term memory compared to baselines.

4.3.2 Action self-verification

From our baseline experiments, we hypothesized that successfully learning the action space / syntax would improve agent performance. To test this hypothesis, we compared the Reflexion baseline to a Reflexion agent which also includes the action self-verification modules described above (no memory modules). We also included comparison with an architecture that extends the Reflexion baseline with an action space oracle where in each turn of the game, we provide the model with the list of valid actions from the game engine along with the observed state and instruct it to choose from the list of valid actions. See Appendix section A.2 for details. These lists are not exhaustive of all possible valid actions, but all actions listed are parser-recognized.

We also compared two different versions of the action self-verification module – one where the cache of valid and invalid actions was persistent across multiple games (so that for example in the second game, all of the valid / invalid actions from the first game were still present in the valid / invalid actions cache) and another where the caches were re-initialized at the beginning of each game. We compare these experiments to an 'action oracle', where a list of valid actions for the given game state is provided in the observation.

4.4 Results

	Avg. normalized score	Avg. # valid moves (of 50)	Avg. prefix heuristic	Avg. # unique game states
ReAct	0.01	16.89	15.67	5.28
Reflexion	0.01	17.67	18.06	6.22
Refl. + ASV-I	0.01	25.78	11.56	7.17
Refl. + ASV-P	0.02	27.22	13.0	7.5
Refl. + action oracle	0.06	31.5	9.56	15.89
Refl. + memory	0.01	11.89	19.28	3.11
Refl. + ASV-P	0.0	19.44	18.72	4.83

Figure 3: Zero-shot performance of agent architectures for 50 steps on each game. The prefix heuristic is the count of repeated 2-word prefixes of invalid actions, described in detail in Appendix A.1.

4.4.1 Baselines

ReAct and Reflexion were very close on all of our metrics (see figure 3), with Reflexion always slightly outperforming ReAct. Thus we implemented our memory and action modules on top of a Reflexion architecture.

4.4.2 Long-term Memory

The long-term memory module hindered performance both when used in isolation with Reflexion, and in tandem with self-verification of actions. Agents with long-term memory were prone to getting stuck in parsing failures and loops at a much higher rate than baseline. This is reflected across all metrics reported in Figure 3. Across games, agents with memory took a diversity of approaches in deciding what to write to memory. For instance, the verbosity of episodic memories from in-game observation differed between games, some agents decided to include score information at each timestep, and the format for storing these entries (i.e. short phrases, sentences, paragraphs) varied widely. We discuss possible reasons for this in the analysis.

4.4.3 Action Self-Verification

We found that our action self-verification modules outperformed the baseline in terms of the number of parser-recognized moves, and performed nearly as well as the action oracle. Agents with persistent

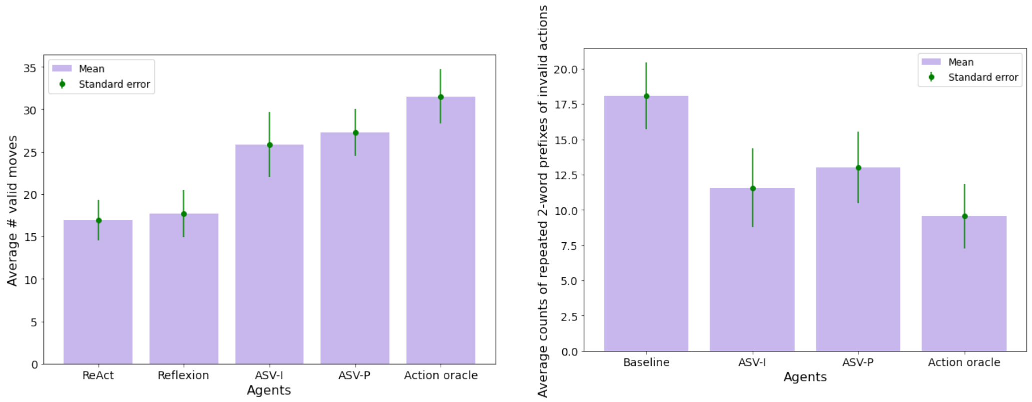


Figure 4: Performance of action oracle, action self-verification module with persistent cache (ASV-P), and action self-verification module with impersistent cache (ASV-I) compared to Reflexion baseline

action caches seemed to do slightly better in making valid moves than the action self-verification with impersistent action caches, potentially indicating that learning persistent knowledge across games can improve agent performance. We also saw that our action self-verification modules decreased the counts of repeated 2-word prefixes of invalid actions, suggesting an improved exploration of the action space. We also found that the action oracle improved the number of unique game states reached by the agent over the baseline, but our action self-verification modules did not (see figure 3).

These results indicate that the action self-verification modules were successfully able to reduce the frequency of invalid commands executed by the agent and improve exploration of the action space, but did not help the agent strategically. This is what we expect since the action self-verification modules take a generated thought (where most of the strategic "reasoning" has already occurred) as input and are responsible purely for its translation into valid action space. The action oracle likely encodes some strategy in the valid actions it outputs (which are generated by the game engine itself, which has full knowledge of the game state space), which causes the increased exploration of game state space.

4.4.4 Memory + Action Self-Verification

Combining the above modules resulted in worse performance than agents given solely action self-verification. For the experiments reported in Figure 3, we used the action self-verification module with persistent cache (ASV-P) along with our memory module. See Appendix section A.7 for a sample of the LM trace for this agent architecture. As shown in Figure 3, we saw improvement in the number of valid moves taken compared to the agent with only memory, but not compared to the baseline. We also did not observe positive results compared to the agents which had action self-verification modules.

5 Analysis

We designed both the long-term memory and action self-verification modules based on specific failure modes elicited by our baseline experiments. Giving language agents better tools for determining valid action confirmed our intuitions about the importance of this kind of strategic reasoning in interactive fiction games, but our method of storing and retrieving long-term memories seems to fail spectacularly. We consider a few reasons that could account for this result.

The utility of the memory stream depends on the information contained within it. The write-to-memory module used in our long-term memory approach relies on an LLM to both determine the appropriate information to add to the stream and the format in which to store it. Prompt engineering experiments revealed this module to be very sensitive to small changes in instructions. This could be do to the fact that it takes an observation as input, and the format of this observation depends entirely on the choices of a game designer. We found that several memory streams had drastically misformatted entries, repeated memories, and verbatim copies of observations. It is plausible that with further prompt engineering, or a more powerful LLM, the write-to-memory module would be more consistent and effective at storing meaningful information in the memory stream.

Examining the valid and invalid action caches after gameplay reveals that the model seems to correctly identify valid actions; however, it frequently misidentifies invalid actions. That is, while there are few invalid actions in the valid action cache, there are many validly formatted actions in the invalid action cache. This overly-conservative action validation scheme clearly improves the agents' reasoning abilities over baseline, however there are several opportunities for improving the result. For instance, one behavior we observe in agents with action self-verification is that they will still fixate on retrying rephrasings of commands which they understand as not recognized by the parser ("examine book" and "examine book carefully" will both fail if there is no book in the room). Similar to the comment before, it is plausible that a more powerful classifier (i.e. LLM) would improve on these issues.

6 Conclusion

In this project we implemented two baseline LLM-powered autonomous language agent architectures, ReAct and Reflexion, for successfully playing open-ended text-adventure games, a nontrivial task even for humans because it necessitates long-term memory and reasoning and dynamic learning and exploration of action space / vocabulary. We successfully improved upon our baseline architecture through the implementation of LLM-powered modules aiming to automatically improve the agent's ability to generate valid actions and assess whether a past action was valid with minimal guidance. We also implemented modules for utilizing long term memory which unfortunately did not improve upon our baseline.

Future work should address the limitations of our long-term memory and action self-verification modules. For example, a significant limitation in many agents tested is a limited ability to employ spatial reasoning, for example misunderstanding where the agent is currently located or being unable to identify that if one arrived at the current location by going north, one should return to their previous location by going south. We have begun implementation of a tool which encodes a map of the game as seen so far that is automatically updated (by an LLM-powered module, that is) and can be called by the agent for better spatial reasoning. We also hope to explore in more depth how to best enable the agent to learn persistent insights across multiple games. Our promising results from keeping a persistent cache of valid / invalid actions give us optimism that multi-game learning can improve agent performance.

References

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2020. Interactive Fiction Games: A Colossal Adventure. ArXiv:1909.05398 [cs] version: 3.

Matthew J. Hausknecht, Ricky Loynd, Greg Yang, Adith Swaminathan, and Jason D. Williams. 2019. NAIL: A general interactive fiction agent. *CoRR*, abs/1902.04259.

Peter A. Jansen. 2021. A Systematic Survey of Text Worlds as Embodied Natural Language Environments. ArXiv:2107.04132 [cs].

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. ArXiv:2304.03442 [cs].

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. ArXiv:2302.04761 [cs].

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning.

Mistral AI Team. 2023. Mixtral of experts. `https://mistral.ai/news/mixtral-of-experts/`. Available from Mistral AI `https://mistral.ai/news/mixtral-of-experts/`.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. ArXiv:2305.16291 [cs].

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. ArXiv:2201.11903 [cs].

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents.

Shunyu Yao, Rohan Rao, Matthew Hausknecht, and Karthik Narasimhan. 2020. Keep CALM and Explore: Language Models for Action Generation in Text-based Games. ArXiv:2010.02903 [cs].

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. ArXiv:2305.10601 [cs].

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. ArXiv:2210.03629 [cs].

A Appendix

A.1 Prefix heuristic

For each game we keep track of a dictionary of all of the 2-word prefixes of actions which are deemed invalid by the game engine and the count of how frequently they occur, like so:

```
"invalid_prefixes": {
        "enter nikolai's": 1,
        "look north": 2,
        "use notepad": 3,
        "examine placard": 1,
        "examine the": 9,
        "examine area": 1,
        "look down": 3,
        "examine lock": 2,
        "look for": 10,
        "try to": 1,
        "take tool": 1,
        "find a": 1,
        "find key": 1
    },
```

Our heuristic, the *count of repeated 2-word prefixes of invalid actions*, is computed by summing all of the counts of the 2-word prefixes which occur more than once. In the dictionary above, the heuristic score would be $2 + 3 + 9 + 3 + 2 + 10 = 29$. Since we only run each game for 50 steps, this score attempts to measure the number of moves which we "waste" trying the same invalid actions over and over. Of course, it is only a heuristic and it is imprecise, but intuitively, a higher score on this heuristic is worse and a lower score is better.

A.2 Action space oracle

Here is an edited example of game playback after action space oracle prompt engineering. Note how we change the guidelines to instruct the model to choose from the valid actions provided in the observation and how we provide the valid actions in each observation.

```
You will be given ‘input‘ and you will respond with ‘score‘.

To do this, you will interleave Thought, Action, and Observation steps.

Thought can reason about the current situation, and Action can be the following
types:

(1) InteractiveFictionGame[a simple action consisting of a few words like ’go
north’, ’check inventory’ or ’take the key’], which takes a step in a text-based
interactive fiction game.
(2) Finish[score], which returns the final ‘score‘ and finishes the task
Additionally, every 5 steps you will be instructed to reflect on your progress
and the strategy taken thus far.

---

Follow the following format.

Observation: observations based on previous action and a list of valid actions
for the next action

Reflect: self reflection on your progress and strategy taken thus far

Thought: next steps to take based on last observation

Action: always either one of the valid actions provided in the observation or,
when done, Finish[answer]

---
... {playback}
Observation:
You pick up the phone. "Hadley!" a shrill voice cries. "Hadley, haven’t you even
left yet?? You knew that our presentation was at nine o’clock sharp! First the
thing with the printers, now this - there won’t even be enough left of you for
Bowman to fire once he’s done with you. Now get the hell down here!!"

Click. Valid actions: InteractiveFictionGame[get up],
InteractiveFictionGame[take off watch], InteractiveFictionGame[take off clothing],
InteractiveFictionGame[take off all], InteractiveFictionGame[take wallet],
InteractiveFictionGame[take keys], InteractiveFictionGame[close door],
InteractiveFictionGame[put watch down], InteractiveFictionGame[put clothing down],
InteractiveFictionGame[look under bed], InteractiveFictionGame[open wallet],
InteractiveFictionGame[take all from end], InteractiveFictionGame[put watch on end],
InteractiveFictionGame[put watch on telephone], InteractiveFictionGame[put clothing on end],
InteractiveFictionGame[put clothing on telephone]

Thought: It seems that I have overslept and missed an important presentation at
work. I should get ready and leave as soon as possible. I will take the action
’InteractiveFictionGame[get up]’ to get out of bed.

Action: InteractiveFictionGame[get up]

Observation:
```

```
You get out of bed.

... {more playback}
```

A.3 Action cache update

Below is an example of a prompt to the LM from the action cache update module. The generated section is surrounded by [[double brackets]].

```
    Given the fields ‘prev_action‘, ‘observation‘, ‘invalid_actions‘, ‘valid_actions‘, produce th

---

Follow the following format.

Prev Action: the previous action taken

Observation: the game’s text response to the previous action which reflects the effects of the pr

Invalid Actions: a list of actions that are known to be invalid

Valid Actions: a list of actions that are known to be valid

Is Valid: A single word, either True or False. True if the action is valid and was successfully i

---

Prev Action: InteractiveFictionGame[Call the Ghostbuster Central BBS]

Observation: That’s not a verb I recognise.

Invalid Actions: []

Valid Actions: [InteractiveFiction[go north], InteractiveFiction[go south], InteractiveFiction[go

Is Valid: [[False]]
```

A.4 Candidate action generation

Below is an example of a prompt to the LM from the candidate action generation module. The generated section is surrounded by [[double brackets]].

```
Given the fields ‘thought‘, ‘invalid_actions‘, ‘valid_actions‘, produce the fields
‘candidate_actions‘.

---

Follow the following format.

Thought: next steps to take which need to be translated into actions

Invalid Actions: a list of actions that are known to be invalid

Valid Actions: a list of actions that are known to be valid

Candidate Actions: A bracketed list of valid candidate actions to take based on the
last observation and thought, for example ’[InteractiveFictionGame[go north],
InteractiveFictionGame[go south]]’. Each action is a few words using only
simple verbs and nouns present in the environment from previous observations
formatted as ’InteractiveFictionGame[<action>]’
```

```
---

Thought: Based on the observation, it seems that the game does not recognize the ’dial’
verb for connecting to the BBS even when I specify the number. I will try using a
different approach and examine the paper given by the Chief again to see if
there is any additional information that could help me access the BBS or gather
more information about the case.

Invalid Actions: [InteractiveFictionGame[check inventory],
InteractiveFictionGame[access the BBS],
InteractiveFictionGame[connect to the BBS], InteractiveFictionGame[dial the BBS number],
InteractiveFictionGame[call the BBS number],
InteractiveFictionGame[dial 510-208-5657 to connect to the BBS],
InteractiveFictionGame[examine the paper in detail]]

Valid Actions: [InteractiveFiction[go north], InteractiveFiction[go south],
InteractiveFiction[go east], InteractiveFiction[go west],
InteractiveFiction[look], InteractiveFiction[examine x],
InteractiveFiction[take x], InteractiveFiction[drop x],
InteractiveFiction[inventory], InteractiveFiction[restart],
InteractiveFictionGame[read the paper], InteractiveFictionGame[go north]]

Candidate Actions:[[ ["InteractiveFiction[go north]", "InteractiveFiction[go south]",
"InteractiveFiction[go east]", "InteractiveFiction[go west]", "InteractiveFiction[look]",
"InteractiveFiction[examine the paper]", "InteractiveFiction[read the paper]"]]]
```

A.5 Memory update module prompt

```
Given the fields ‘observation‘, ‘memory_stream‘, produce the fields ‘memory‘.

---

Follow the following format.

Observation: the game\’s text response to the last action

Memory Stream: your memory stream

Reasoning: Let\’s think step by step in order to {produce the memory}. We ...

Memory: a single sentence containing the most relevant memory to the current observation

---

Observation: [INSERT OBSERVATION]

Memory Stream: [INSERT MEMORY STREAM]

Reasoning: Let\’s think step by step in order to
```

A.6 Memory retrieval module prompt

```
Given the fields ‘observation‘, ‘memory_stream‘, produce the fields ‘new_memory‘.

---
```

```
Follow the following format.

Observation: the game\'s text response to the last action

Memory Stream: your memory stream

Reasoning: Let\'s think step by step in order to ${produce the new_memory}. We ...

New Memory: a new memory if the observation contains new information that should be
remembered in addition to the memory stream, otherwise \'None\'

---

Observation: [INSERT OBSERVATION]

Memory Stream: [INSERT MEMORY STREAM]

Reasoning: Let\'s think step by step in order to
```

A.7 Reflexion + action self-verification + memory playback

```
...
Observation:
You pick up the phone. "Hadley!" a shrill voice cries. "Hadley, haven't you even left yet?? You k

Click.

Memory: Hadley's boss is angry because Hadley is late for a presentation.

Thought: It seems that Hadley's boss is upset because Hadley is late for a presentation. I should

Candidate actions: [InteractiveFictionGame[Go to the bathroom]]

Action: InteractiveFictionGame[Go to the bathroom]

Observation: You'll have to get out of bed first.
...
```