Knowledge-based Solutions to Text-based Worlds

Morgan Wajda-Levie

May 18, 2022

Abstract

Text-based games, also called text adventures or interactive fiction, are a form of games meant to evoke reading a work of fiction, where the player controls the main character, popularized in the 1980s by games like Zork I[2]. Players interact with the environment through text commands and the game gives feedback in the form of natural language descriptions. I developed the beginnings of a knowledge and logic-based agent to leverage logical reasoning in solving rudimentary text-based games.

Deliverables checklist

- ☑ Project report:
  `https://github.com/morganwl/csci740-text-based-worlds/raw/main/text-based-worlds.pdf`
- ☑ Github repository:
  `https://github.com/morganwl/csci740-text-based-worlds`
- ☐ Poster

1 Description

Text-based games, also called text adventures or interactive fiction, are a variety of computer games originally appearing in 1976 with the game Adventure[5]. The gameplay experience is meant to evoke reading a work of fiction, where the player controls a character, and through that character, the outcome of the story. Early text-based games tended to be a mixture of puzzles and exploration, with a simple goal, while more recent games have further explored the potential of narrative control with diverging storylines and interpretations. The games can be quite challenging and require identifying and solving puzzles with common sense reasoning to achieve a final goal. This challenge has recently made them an intriguing area of research in Artificial Intelligence, though a generalized agent capable of solving a wide range of text-based games remains elusive[4].

Players in text-based games control a *player character* in an environment which is rendered to the player entirely through prose descriptions. The environment, which is unique from one game to the next, generally models a lifelike world, governed by common sense rules. The player character (which we will, by the genre's convention, simply call *player*), moves through discrete locations, or *rooms* in this world and interacts with objects contained within. All of these actions are communicated to the game through natural language commands, typed at a prompt. The commands are interpreted by the game's parser, and the game responds, in natural language, based on the outcome of the player's action.

While the prompt will accept any return-terminated string, the parser understands only a small subset of English vocabulary, with simple grammatical rules.[1]Commands generally take the form of: `verb [noun phrase [adverb phrase]]`, where `noun phrase` often denotes an object in the environment and `adverb phrase` often indicates a second object. Sample commands might be, "`go` `north`", "`look under bed`" or "`unlock wooden chest with bronze key.`"[4] All recognized words have been explicitly coded, either by the authors of the parser, or by the authors of the individual game, but these words are not necessarily given to the player in an organized manner. Most games are written using one of a few widely available frameworks, which include a parser with a large vocabulary, but most games will make changes to that vocabulary or the parser itself. The parser will reject commands that it does not understand, though individual games might communicate this rejection in different ways. The game will also often inform players of commands that are understood but have no effect, or are deemed irrelevant to a game.

As a problem for an artificial intelligence agent, text-based games present a number of clear challenges. The state space is finite, but potentially large, and only partially observable. The transition model is unknown, and the action-space is large, sparse and unknown. A verb might only have meaning when applied to a single object in the game, and it might only affect that object under specific conditions. Human players navigate this action space through a mixture of common sense, generalized and genre-specific knowledge, and ingenuity. Some games require the use of made-up words. All information is given to the player through natural language, from which meaningful state information and action outcomes must be extracted. Some information is given to the character unbidden, but other information must be asked explicitly by examining objects in the environment. Furthermore, each game has its own unique goal (or goals), which may or may not be clearly communicated to the player.

The challenges of text-based games also make them an interesting environment for developing AI agents. An AI agent must be capable of extracting information from natural language processing, recognizing objects of interest, changes in the game environment, and other potential clues. The agent must be able to employ, or simulate, common sense reasoning, recognizing which actions might impact which actions, and combining those actions to reach goals.

```
 Alley
This narrow aperture between two
buildings is nearly blocked with piles of
rotting cardboard boxes and overstuffed
garbage cans.  Ugly, half- crumbling
brick walls to either side totter
oppressively over you.  The alley ends
here at a tall, wooden fence.

High up on the wall of the northern
building there is a narrow, transom-style
window.

>examine window
It's about eighteen inches wide and a
foot tall, with hinges along the top that
allow it to swing out.  It's currently
closed.

>open window
The window is too high.

>move garbage can under window
Grunting and holding your breath, you
manhandle one of the filthy cans under
the window.
```

Figure 1: Object interaction from ANCHORHEAD[7]. Player commands are after the > prompt.

While generalized text-based games remain unsolved by AI agents, agents have achieved partial successes in those environments, and have been able to solve constrained games created as benchmarks. This project aims to create an AI agent capable of solving games that have been constrained in the goal and the kind of puzzles that must be solved to achieve it. Namely, the agent will be asked to solve mazes broken into four difficulty levels.

1. Trivial mazes (interconnected rooms without obstacles)
2. Familiar obstacles requiring one action, such as doors
3. Composite obstacles requiring multiple actions, such as locked doors
4. Complex, novel obstacles, such as laying a plank across a chasm

[1] Text-based games have been published in many natural languages, including Dutch, French, German, and Russian, but this paper will focus exclusively on those available in English.

The agent's performance will be compared to existing agents, and, hopefully, it will exceed them under some if not all circumstances.

2 Related work

2.1 Frameworks for text-based games

The excellent TextWorld paper lays out a formalization of text-based games as Partially Observable Markov Decision Processes. The challenges of text-based games in this context are explored, such as partial observability, large state space, large and sparse action space, exploration vs exploitation and long-term credit assignment. The challenge of common sense reasoning and *affordance extraction*, or knowing which verbs are applicable to which objects, is explored.

The TextWorld framework aids in the study of text-based games by providing a Python interface for connecting AI agents to any text-based game published in the popular Z-machine format, and also provides tools for procedurally generating text-based games with various constraints and criteria[4].

2.2 Agents for playing text-based games

Many AI agents have been written to attempt text-based games, some of which were entered in the 2016, 2017 and 2018 *Text-Based Adventure AI Competition.* The first two competitions pitted agents against a single game developed specifically for the competition, whereas the 2018 competition evaluated agents on their performance on 20 previously released games[1]. Agents were given 1000 moves per run of a game, repeated 10 times, and their average score, measured as a fraction of the total possible score, was compared. Three agents have publicly available source code, BYUAgent 2016[13], Golovin[10] and NAIL[8]. This source code, along with published work, provides some interesting approaches, and also elucidates some of the thorny sub-problems that researchers are attempting to solve.

NAIL, the winner of the 2018 competition, combines many components and a number of specialized heuristics for text-based games. A 5-gram language model is used to estimate probable affordance extractions. Information about locations in the game, and all objects in it, is stored in a knowledge graph, but not before being confirmed as *valid* by a custom validity detector. Room identity is determined with a fuzzy string match. Decisions are made by a number of specialized decision modules which take control of the agent based on their eagerness under the current game state[8].

BYUAgent 2016 uses word2vec word embeddings to create an *affordance vector* between objects extracted from game text and a pre-defined list of verbs. Commands are generated from the most promising verbs.[1]

Golovin learns command patterns by studying a collection of published game solutions (called *walkthroughs*), tutorials and decompiled source code, and then uses a combination of LTSM neural networks trained on fantasy novels and a word2vec embedding to match those commands with objects extracted from scene descriptions. Locations are mapped using a simple room-direction graph. A MergNode algorithm is performed after every movement to merge nodes with identical pathways[10].

Dambekodi et al have explored augmenting agents with the COMET inference model and the BERT large language model, showing, in both cases, that the statistical models encoded common sense which could be leveraged in affordance extraction[6].

Haroush et al combined two deep learning algorithms to aid in action selection. One learns the likelihood of action acceptance, while the other learns a formal q-value.

3 Approach

Much of the existing work in text-based games focuses on deep learning, combined with natural language processing, to aid in affordance extraction. Some of the most successful agents rely on heavily tuned heuristics specific to the *genre* of text-based games as much as the format. Agents generally work in a reflexive fashion — looking for objects in the game world and pushing them with whichever action comes to mind, in the hope that accepted commands on objects will yield desirable results.

These are all valid approaches to an as-of-yet unsolvable problem, but I am interested in an approach that foregrounds interpretable common-sense reasoning, with the ability to not only recognize important objects and actions that change them, but ways in which those actions can be combined to reach explicit goals. For this reason, I chose to focus on a knowledge-driven approach, using a combination of logical reasoning and forward search to make inferences and decisions. This makes the reasoning of the agent as transparent as possible, allowing me to see what it is learning, what conclusions it is making, and why it is making them.

3.1 A logic-based agent

Why logic? First of all, knowledge, and logical inference upon it is a transparent and interpretable process, particularly when using First Order Logic with predicate names drawn from natural language vocabularies. Research has been done on using statistical methods such as reinforcement learning, neural networks and even large language models to shoulder the task of action affordance and common sense reasoning. While I am sure that these statistical methods must ultimately be incorporated, I am interested in seeing how much reasoning and intuition can be performed in a logical realm. Logical methods also have the benefit of learning from a much smaller number of observations, something that I consider meaningful when placed next to large language models which are trained on observations that would take several lifetimes for humans to evaluate.

Logic also is also well-suited to some of the particular problems of text-based games. For instance, logic provides an excellent tool for solving the object identity problem — establishing that two objects must be one and the same or, conversely, that two objects must be distinct is a form of logical inference. The puzzles in text-adventure games also tend to require novel combinations of common-sense reasoning which must be performed once before completing the puzzle and moving on to the next. Because we lack a statistically significant wealth of similar puzzles, an approach that does not rely on pattern-matching could be advantageous.

Text-based games also present favorable conditions for logical reasoning that many other applications do not. Action outcomes are generally, though not universally, deterministic, allowing definite rules to be successfully employed in most cased. (The right key will always open the right lock, which cannot be said of the door to my apartment building.) Good games are also generally designed to behave in a logical fashion, even though that logic is rarely formalized and might not be immediately apparent to the player. Nonetheless, a logical agent in the environment of a text-based game will have advantages that they would not have in the real-world environment that that game is meant to model — actions will almost always have a consistent outcome once the necessary conditions are met, and the information necessary to identify those conditions will almost always be provided.[2]

3.2 Linear logic

In choosing an appropriate logic for our agent, the demands of human interpretability favor a variant of First-order logic, which allows us to define multiple predicates about a single *constant* (or game-

[2]There are games where action outcomes are non-deterministic and games where the information necessary to solve a puzzle is not provided. While some of these might be cases of shoddy design, many puzzles do depend on upending conventional expectations. Solving these problems is a challenge for another day.

world object), and to make universal statements of the form:

$$\forall \text{L}, \text{DIR}, \text{DEST} \;\; \textit{connects}(\text{L}, \text{DIR}, \text{DEST}) \rightarrow \textit{exit}(\text{L}, \text{DIR})$$

Meaning that any location connected by a direction to a destination implies the existence of an exit from that location in that direction. Words in uppercase letters are treated as variables, whereas predicates and constants are written in lowercase letters. To simplify computation, I only consider universally qualified variables, allowing me to omit the $\forall$ preamble. To maintain the completeness of the agent's reasoning, I remove constants, meaning that reasoning is limited to universally qualified variables and explicitly declared constants. The values of functions in sentences such as $\textit{exit}(\textit{location}(\textit{player}), \textit{south})$ can be captured by adding a variable and a second conjunctive clause, such as $\textit{at}(\textit{player}, \text{LOCATION}) \wedge \textit{exit}(\text{LOCATION}, \textit{south})$.

Because the environment changes in response to player actions, a logic is needed that can reason about predicates changing over time. I chose linear logic because of its use in TextWorld to guarantee the satisfiability of procedurally generated games[4]. In my implementation of linear logic, I add the *linear implication* operator, which behaves much like traditional implication, with the difference that the right-hand side, or *consequent* of the implication does not become true until the following time state, and that some conditions on the left-hand side, or *premise*, of the operator, are *consumed*. Consider the following linear implication for traveling between rooms.

$$\textit{go}(\text{DIR}) :: \textit{at}(\textit{player}, \text{L}) \otimes \$\textit{connects}(\text{L}, \text{DIR}, \text{DEST}) \otimes \neg\$\textit{blocked}(\text{L}, \text{DEST}) \\ \multimap \textit{at}(\textit{player}, \text{DEST})$$

In this case, if the player action is, for instance, "`go south`", and an exit exists between the player's current location and some other destination, the player will be at that destination in the next turn. Furthermore, their current location will be consumed, which is to say, the predicate will be replaced with its complement in the knowledge base. The predicates *exit* and *connects* are marked with a \$ sign, meaning that their condition carries across the operator and is not consumed.

Because I am primarily concerned with changes in the environment triggered by player actions, I have chosen to model the player action as a special predicate that is required for all linear implication rules. If I were to model games where player actions were not the only actors in changing the environment (such as games with non-player characters), I would change this.

With a logically sound representation of the game environment, I can use a *forward chaining* algorithm to infer additional information about the environment and, hopefully, solve challenges.

3.3 Core components

In my approach, the agent can be seen as consisting of three fundamental components. The *parser*, which interprets descriptions and feedback from the game, the *knowledge base*, which stores observations made by the parser and attempts to infer additional knowledge, and the *decision maker*, which uses information stored in the knowledge base to find paths to goals and issue commands to the game. In practice, the decision maker and the knowledge base are tightly interwoven.

4 Experiments

4.1 Rover One

I started with a very simple knowledge-based agent that would be able to navigate a trivial maze based on observing labeled exits in the room description. This allowed me to work out some of the fundamental problems of connecting my agent to a text-based game and identify some of the pitfalls needing more attention. I called this simple agent *Rover One*.

4.1.1 The Rover One parser

Rover One's parser has two responsibilities. First, it needs to distinguish *accepted commands*, which either return information or produce some sort of effect on the game environment, from *rejected commands*, which produce no result either because they refer to actions not possible in the current state, or because they are not understood by the parser. This identification is aided by two heuristics. First of all, because Rover One's actions are limited to moving between locations, any successful action will result in a new player location. In almost all text-based games, a location change is announced by printing the new location's name in a recognizable format. Secondly, rejection messages are generally brief with commonly occurring keywords.

```
>go north
You can't go that way.
>hop like a madman
I only understood you as far as
wanting to hop.
>traverse steps
That's not a verb I recognize.
```

Figure 2: Rejected commands in CURSES[11].

```
>go west
Forest
This is a forest, with trees in all
directions.  To the east, there
appears to be sunlight.
```

Figure 3: Response to a command in ZORK I[2]. The room name is on its own line, with no period. The word *east* is recognized by the agent as a potential exit.

Once a room description is recognized, the parser needs to extract phrases from that description that might refer to meaningful objects in the game. Again, this process is greatly simplified for Rover One, because it is only concerned with objects that allow it to move between rooms, an action generally performed by the command "`go <direction>`." Using a list of pre-determined directions, Rover One is able to extract keywords from the scene description and convert those keywords to declarative observations in the form of $exit(\mathrm{L}, \mathrm{DIR})$ predicates. These observations are then passed on to the knowledge base via tell statements.

The Rover One parser also performs a simple form of assumption correction — when movement in a direction is rejected by the game, the parser reports an $\neg exit(\mathrm{L}, \mathrm{DIR})$ observation to the knowledge base.

4.1.2 The Rover One knowledge base

Rover One's knowledge base receives observations from the parser via a tell statement, and stores predicates in an OBJECT → PREDICATE → OTHER ARGUMENTS → VALUE hierarchy. As a further simplification (and departure from the original logical approach), destinations are stored explicitly as evaluated functions with the location object. (i.e. `objects{room} {'destination'}{direction}` contains the name of another room.)

When queried, the knowledge base reports the known value of predicates, or that they are unknown. The knowledge base can be queried for a path to a goal, which it supplies by using an iterative deepening search to find a valid path. The knowledge base can be queried for an exploration goal, for which it uses iterative deepening search to find the closest *exit* predicate without a matching *destination* value.

Rover One's knowledge base is, in many respects, a gross simplification, a two-dimensional knowledge graph masquerading as a first-order logic database. While it can store arbitrary predicates on arbitrary constants, the only reasoning is written directly into the python code, with no generalized proof or resolution mechanic.

4.1.3 The Rover One decision maker

First, the Rover One decision maker queries the knowledge maker for a path to any of its current goals (a symbolic endeavor, since it has no mechanism for extracting goals from its percepts.) If no path to a goal exists, it queries the knowledge base for an exploration goal, which is then added to an exploration-goals path. Finally, if no exploration goal is provided, the decision maker attempts any known direction keyword for which the exit predicate is unknown. (Exits are not always included in room descriptions, or a known direction keyword might be accepted as a synonym for a described exit that the parser did not recognize.)

Once some sort of goal has been found, the Rover One will move in that direction. Most importantly, because exploration goals are reported by the knowledge base from all explored rooms, the agent is able to follow an unexplored path in one direction, and then follow the shortest path back to the next unexplored exit, following a sort of depth-first exploration.

4.1.4 Evaluating Rover One

To evaluate Rover One's effectiveness, I procedurally generated 4 simple mazes using TextWorld. These are increasingly large environments containing only interconnected rooms, with no obstacles. A goal room is chosen at random; if a player reaches that room, they are awarded one point and the game ends.

Without too much confidence in Rover One's abilities, I matched it up against a random agent, TextWorld's Naive Agent, which randomly chooses from 15 pre-determined actions at every move.

In addition to the 4 generated mazes, I included 7 publicly released text-based games ranging from a re-released version of the original ADVENTURE to the 2007 LOST PIG[5, 12, 9, 7, 11, 2, 3]. While these games are not solvable by an agent capable only of movement, they provide an opportunity to see how an agent performs with room layouts and scene descriptions from a naturally created game.

Game	Random Agent			Rover One		
	Score	Moves	Locations	Score	Moves	Locations
maze 10	1	48	5	1	16	7
maze 20	1	19	4	1	3	2
maze 50	1	145	15	1	6	5
maze 100	0	465	42	1	22	16
Hunter, in Darkness	0	1000	1	0	1000	1
Lost Pig	1	1000	4	1	1000	4
Anchorhead	0	1000	24	0	1000	19
Curses	0	1000	4	0	1000	5
Adventure	36	1000	11	36	1000	8
Zork I	0	1000	18	0	1000	13
9:05	0	1000	5	0	1000	5

Figure 4: Comparison of agents on games 1,000 move limit

Each agent was given 10 playthroughs per game, stopping once the game was solved or once they had made 1000 moves. If a game ended early (from, say, being eaten by a terrible wumpus in HUNTER, IN DARKNESS), the agent was allowed to restart and play through with the moves remaining for that playthrough. The final score, number of moves made, and the number of unique locations visited, were averaged together over the 10 playthroughs and recorded.

As can be seen, Rover One, with its intentional exploration, dramatically outperformed the random agent in maze-like environments. On the other hand, in more natural environments, Rover One generally under-performed the random agent on a metric of unique locations visited. As a point of comparison, I evaluated the two agents on shorter games, which showed the two agents coming slightly closer in performance. There are two likely reasons for this discrepancy — firstly, given a

Game	Random Agent			Rover One		
	Score	Moves	Locations	Score	Moves	Locations
maze 10	0	74	4	1	4	3
maze 20	0	46	6	1	7	6
maze 50	0	74	12	1	42	23
maze 100	0	89	12	1	10	9
Hunter, in Darkness	0	100	1	0	100	1
Lost Pig	1	100	4	1	100	4
Anchorhead	0	100	15	0	100	13
Curses	0	100	4	0	100	4
Adventure	36	100	6	36	100	8
Zork I	0	100	11	0	100	15
9:05	0	100	5	0	100	5

Figure 5: Comparison of agents on games with 100 move limit

relatively small state space (fewer than 30 reachable locations in the natural games), and enough movement, the stochastic agent is able to hit most locations through dumb luck, eliminating the advantage of Rover One's intentionality. Secondly, it is clear that Rover One is making errors in perception or inference that are keeping it from discovering locations that the random agent is able to stumble upon.

One such shortcoming in Rover One's reasoning is the inability to distinguish two distinct objects with the same name. For instance, in ZORK I, several areas bear names like *Forest* or *Clearing*. The agent, thinking these areas the same, will try an exit that worked in the past, find that exit impassible, and mark *exit*(DIR) as false *for all rooms bearing the same name.* Because the agent will not waste its time on exits that it knows to be blocked, it will eventually convince itself that locations with a commonly occurring name have no possible exits. This suggests two areas of improvement for future Rovers. Clearly, the agent needs a more rigorous way of identifying objects, but, also, a successful agent might sometimes consider actions *even if it knows them to be impossible*, and have the capacity to adjust when proving itself wrong.

4.2 Rover Two

4.2.1 A formal logic-based knowledge base

Rover Two was written as a follow-up to Rover One, with a logically complete reasoning system at the center of its knowledge base. Predicates are stored in in two-dimensional dictionaries indexed by predicates and tuples of arguments. As with Rover One, care is taken to distinguish between predicates which are known to be false and predicates which are simply unknown.

Linear logic's linear implication operator allows us to change the next state of the truth model based on conditions in the current state. In practice, our knowledge base is encountering observations of this change after the player action, and its subsequent results, have occurred. To compare these *prior* and *posterior* states, predicates are stored in layered sparse models. With each move of the agent, the knowledge base is advanced, creating a new, empty dictionary of predicates. Entailment of literals is checked by successively querying layered models, starting with the most recent, until a matching predicate is found. In this way, the most recent change to a predicate will be found by a query, but predicates which have not been updated since earlier states retain their truth value in the latest model. Queries to the knowledge base can indicate that a $t-n$ state is desired, which is used for recognizing changed conditions resulting from player actions. Because reasoning is primarily focused on the immediate prior and posterior time states, the oldest models are merged after a threshold is reached (5 during Rover Two's development.) Hopefully, this layered sparse model will eventually prove useful for constraint satisfaction searches.

Whereas Rover One's knowledge base only allowed limited querying of variable objects, Rover Two implements a complete *unification* algorithm, allowing it to find substitutions that make sentences containing free variables equivalent to sentences contained in the knowledge base. (Or any other sentence provided.) In this way, a sentence containing free variables and many clauses can be *fetched* from the knowledge base, returning the list of all substitutions such that the sentence is entailed. This allows us to find specific literals matching implication rules.

After every successive tell to the knowledge base (containing a list of all observations made after single move), the knowledge base performs a *forward chaining* algorithm. Known facts are matched against stored implications, and inferred facts are added. For the time being, a brute force forward chaining algorithm is used — all known facts are checked against all known implications. A knowledge base equipped to handle large numbers of facts would improve this by limiting the checked implications to those which could be affected by the changed variables, avoiding a costly search of the entire knowledge base at every iteration.

Stored sentences:

$at(player, kitchen)[t-1]$

$at(player, hallway)$

$exit(kitchen, east)$

$action(go, east)$

Linear implication rule:

$[t-1](at(player, L) \wedge \$exit(L, DIR)$

$\wedge \$connects(L, DIR, DEST))$

$\multimap at(player, DEST)$

Becomes:

$[t-1](at(player, kitchen)\$exit(kitchen, east)$

$\wedge \$connects(kitchen, east, hallway))$

$\multimap at(player, hallway)$

Figure 6: A linear implication containing free variables is unified with sentences in the knowledge base, yielding a sentence which is neither entailed by nor contradicted by the knowledge base.

To draw conclusions from successful player actions, an unsound heuristic is used, which I have labeled the *Occam's razor* algorithm. After an action is performed which has changed the game state, known linear implication rules are searched to find any rules which are not *contradicted* by the prior state. The linear implication rule with the fewest unknown predicates, which *could* have caused the state change, is selected, and the missing values are stored in the knowledge base. This heuristic is not sound — just because certain conditions *could* cause an observed outcome doesn't mean that those conditions *did* cause it, or that those conditions must be true. A player that has gone north from a location believed to have a northern exit may have arrived at a new location because they traveled through that northern exit, or they may have fallen through a trap-door in the floor, revealing a previously unknown exit leading down. A more refined version of this algorithm would make more use of cues from the game text and parser to suggest whether an action had an expected or unexpected outcome. Guessed conditions could also be stored in a temporary model and subjected to forward chaining to look for potential contradictions before being added to the main model of the knowledge base.

4.2.2 Evaluating Rover Two

Rover Two is currently in an unfinished state. The knowledge base has been connected to the simple Rover One keyword parser (with necessary modifications), but the decision maker chooses actions at random, because the algorithms for finding paths to goals and directing exploration have not yet been written. As such, Rover One is capable of receiving observations and making inferences from them, but it is not capable of leveraging that knowledge in decision making.

No formal evaluations of Rover Two's performance have been made, but I have informally observed the growth of the knowledge base as it receives observations from random actions inside of ZORK I. The agent is able to accurately build a map of the environment, subject to the same object identification caveats as Rover One. Axioms of object equality, currently unwritten, should help resolve these problems.

Linear implication rules describing new actions, with their conditions and consequences, are easily added. As long as the parser reports predicates to the knowledge base, it can easily deduce conditions that result in successful and unsuccessful actions.

5 Results

Because Rover Two never became fully operational, this experiment failed to yield the results that I hoped in measuring the utility of logical inference in text-based games. I do not, however, consider the experiment a failure, nor do I consider the logic-based agent a dead-end, even if it proved to be a time-consuming agent to implement.

The success of Rover One on simple mazes is pleasing, though not surprising. Not all of the agents submitted to the Text-Based Adventure AI competition built maps of their environment or used goal-seeking exploration so, in this respect, Rover One outperforms some existing agents[1]. On the other hand, its strict limitation to navigation makes it useless for most text-based environments. Its tendency to become stuck in rooms with common names is also a severe limitation, even within the narrow domain of navigation.

Rover Two is currently unsuccessful as an AI agent, but, with its logic largely implemented, the ease of creating implication rules is encouraging. Coupled with existing techniques for extracting object information from room descriptions, Rover Two has promise at being able to solve puzzles in a text-based environment, given some common sense knowledge encoded as linear implication rules. While the current requires that these rules be added using Python objects, a parser was created using Tatsu for an interim version of Rover, which could be quickly updated to allow implication rules to be written in plain text (using an appropriate logical grammar) and imported into the Rover knowledge base at runtime.

6 Lessons learned

Most of the lessons learned in this project are within the domain of logic-based artificial intelligence. I learned how to implement tell, store, unify and fetch algorithms for a first-order logic knowledge base. I learned how to adapt those algorithms to accommodate temporal states, and to add a linear implication operator. I learned how to implement forward chaining to make inferences from observations and implications. I also learned a tiny bit about knowledge engineering and ways to frame logical sentences about cause and effect relationships in a way that would be relevant to an AI agent.

I learned how to work with the predominant framework in the study of AI agents for text-based games, TextWorld[4], and how to create my own procedural generation models in that framework. I read a number of papers about different techniques for approaching text-based games, techniques that I look forward to exploring more in the future, and potentially incorporating into my own work.

7 Challenges and next steps

I encountered a number of challenges over the course of this project. Some, such as object identification, are described in more detail within the experiments section. Other challenges inherent to the domain of text-based games were ignored entirely, such as parsing natural language text. Text-based games are an unsolved problem; this makes the potential for discoveries exciting, but it also leads to the inevitability of disappointing results.

I expected difficulties in applying logical methods to an AI agent, but I was not prepared for the challenges in simply implementing the back-end components necessary to make a logic-based agent function. Without functioning algorithms for unifying, fetching and evaluating logical sentences, no reasoning can be performed. Even once logical reasoning is implemented, this alone is not sufficient for decision making. This made it impossible to even connect the agent to the game until a significant amount of programming had been done.

I also made some missteps in auxiliary problems of implementing a logical knowledge-base. Originally, I was concerned about the best data structures for storing predicates; because I was dealing with arbitrary predicates and multiple arguments, I did not want to create a tangled graph of interlocking object references. I was also wary of leaving predicates unstructured and unindexed in any meaningful way. I spent a day implementing the knowledge base using an SQL database backend, because I knew that I ultimately wanted to be able to store observations between games, and because I thought that database tools might allow me to avoid the problem of predicate storage and focus on logical reasoning. (My prior professional experience includes some work with writing tools that interacted with databases through SQL queries.) Unfortunately, this proved a distraction, as filtering the textbooks logic algorithms through SQL queries added unneeded complexity. I ultimately chose a crude data structure of two-dimensional dictionaries, indexed on predicate names and argument tuples. This might prove unwieldy over time as thousands of predicates are accumulated, but is not a problem for my current experiments.

Frustrated with expressing logical sentences using nested lists, I found the Tatsu grammar parser, which creates Python parsers from EBNF grammars. This too proved a costly distraction, and, while the parser worked with an earlier version of Rover Two, it is not currently configured correctly to work with the current version.

I had initially hoped to use backward chaining as a goal-seeking algorithm. I abandoned this when I realized that linear logic makes one-way back-chaining unsound — premises are recursively found which prove the desired consequent, but, because linear implication results in state changes beyond its consequent, a sound back-chaining algorithm would need to move forwards and backwards to address the issue of resource consumption. I realized that a better and more sound approach would be constraint satisfaction using back-tracking search, but I had run out of time before I could implement it.

Logic, as a tool in AI presents many challenges. Namely, that it deals with absolute truths, while AI agents are generally in environments with a great deal of uncertainty. I did not get a chance to wrestle with many of these challenges, but I am excited about the possibility of mixing logic and uncertainty. How does a logical agent resolve the contradictions caused by learning facts that force it to abandon observations it once accepted as truth? Text-based games provide many opportunities for this, both because of the imperfections of observations made through natural language processing, and because of the possibility of unreliable narration in the game text.

I believe that there is untapped potential for logical approaches to text-based games and that these games, in turn, provide a meaningful environment for experimenting with logical reasoning in AI agents. There is no doubt that statistical methods and quantified uncertainty need to be incorporated for an agent to be truly successful, particularly in the parser component. The agent would also benefit from more finely tuned heuristics in the decision making component, even if those heuristics sometimes lead to actions that the knowledge base would deem inappropriate.

If I were to continue work on this project, I would, for the time being keep the focus on creating a functional, if limited, purely logical decision-making agent. I would start by implementing constraint satisfaction algorithms, which would, hopefully, give me enough of a tool for directing exploration and problem-solving. Back-chaining could also be implemented, unsound though it may be, as a heuristic, though forward search should probably be more than adequate for Rover's current capabilities of perception.

An interesting experiment might be taking a controlled set of games (including a test set and a validation set) and attempting to logically reason, at a higher level, through a complete solution of those games (with the help of published solutions) and then applying that information to designing rules for an agent to solve those games. What is the minimum number of rules required to solve a given text-based game? Hopefully not a single rule for each action in the published solution. Can rules be written that generalize between games?

Ultimately, I am interested in developing an agent that is capable of learning new logical rules. This would require incorporating statistical methods and using natural language processing to help identify potential predicates. I would probably explore Bayesian networks.

References

References

[1] Timothy Atkinson et al. "The Text-Based Adventure AI Competition". In: *IEEE Transactions on Games* 11.3 (Sept. 2019), pp. 260–266. ISSN: 2475-1510. DOI: `10.1109/TG.2019.2896017`.

[2] Mark Blank et al. *Zork I*. 1980. URL: `https://ifdb.org/viewgame?id=0dbnusxunq7fw5ro`.

[3] Adam Cadre. *9:05*. 2000. URL: `https://ifdb.org/viewgame?id=qzftg3j8nh5f34i2`.

[4] Marc-Alexandre Côté et al. "TextWorld: A Learning Environment for Text-Based Games". In: *Computer Games*. Ed. by Tristan Cazenave, Abdallah Saffidine, and Nathan Sturtevant. Communications in Computer and Information Science. Cham: Springer International Publishing, 2019, pp. 41–75. ISBN: 978-3-030-24337-1. DOI: `10.1007/978-3-030-24337-1_3`. URL: `https://arxiv.org/pdf/1806.11532`.

[5] William Crowther and Donald Woods. *Adventure*. 1976. URL: `https://ifdb.org/viewgame?id=fft6pu91j85y4acv`.

[6] Sahith Dambekodi et al. "Playing Text-Based Games with Common Sense". In: *arXiv:2012.02757 [cs]* (Dec. 2020). arXiv: 2012.02757. URL: `http://arxiv.org/abs/2012.02757` (visited on 04/14/2022).

[7] Michael Gentry. *Anchorhead*. 1998. URL: `https://ifdb.org/viewgame?id=op0uw1gn1tjqmjt7`.

[8] Matthew Hausknecht et al. "NAIL: A General Interactive Fiction Agent". en. In: (Feb. 2019). DOI: `10.48550/arXiv.1902.04259`. URL: `https://arxiv.org/abs/1902.04259v2` (visited on 05/02/2022).

[9] Adam Jota. *Lost Pig*. 2007. URL: `https://ifdb.org/viewgame?id=mohwfk47yjzii14w`.

[10] Bartosz Kostka et al. "Text-based adventures of the golovin AI agent". In: *2017 IEEE conference on computational intelligence and games (CIG)*. tex.organization: IEEE. 2017, pp. 181–188.

[11] Graham Nelson. *Curses*. 1993. URL: `https://ifdb.org/viewgame?id=plvzam05bmz3enh8`.

[12] Andrew Plotkin. *Hunter, in Darkness*. 1999. URL: `https://ifdb.org/viewgame?id=mh1a6hizgwjdbeg7`.

[13] Daniel Ricks. *BYU-Agent-2016*. Aug. 2019. URL: `https://github.com/danielricks/BYU-Agent-2016` (visited on 05/02/2022).