Nakul Khanna
Gonville & Caius
nk420

Intelligent Methods for Enhanced Expressiveness in Interactive Fiction

Computer Science Tripos, Part II

9th May 2016

Proforma

Name: Nakul Khanna

College: Gonville & Caius

Title: Intelligent Methods for Enhanced Expressiveness in Interactive Fiction

Examination: Computer Science Tripos, Part II

Year: 2016

Approximate Word Count: 11,355

Originator: Dr Sean Holden

Supervisor: Dr Sean Holden

Original Aims

To design an interactive fiction engine based around knowledge representation and natural language processing rather than predefined inputs and outputs as was traditional in classic interactive fiction, and to create a small example game using this engine.

To subsequently test the effectiveness of this type of engine by means of a user study in which the small example game designed using the new engine is compared blindly against a more traditional interactive fiction experience, and determining which game was thought to accept more input, to give better feedback and to entertain the player more.

Work Completed

A game engine was designed in Java, using a Prolog-based knowledgebase (via the TuProlog Java interface) and input processing based on Stanford CoreNLP. A brief example game was designed using this engine.

In a user study, the game built using the new engine was preferred to the classic interactive fiction experience *Colossal Cave Adventure* on all three measures. It was perceived as being of slightly below the global average of usability, but this was within the margin of error.

Special Difficulties

None.

Declaration

I, Nakul Khanna of Gonville & Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: 

Date: 9th May 2016

Contents

1 Introduction

Interactive fiction (IF), or text adventure, is a genre of computer game in which, alternately, a user is presented with a paragraph of text and must enter a command to advance the game.

In the following example, the first paragraph describes the player's location, and is followed by > to prompt the user to enter an instruction; here, the user opts to take the spyglass, and the game confirms that the action was successful.

```
You stand in a dimly lit room. On a table in front of you, you see a
spyglass, a quill and a faded tome. To the west is a carved wooden door,
and to the east is a dusty mirror.

>take the spyglass

You took the spyglass.
```

The platonic ideal of such a game would be able to accept any possible human input and advance the game in an intelligent way. Naturally, such a task is AI-hard and far beyond the reaches of current technology – though the player character could now conceivably throw the spyglass at the mirror, the game designer is unlikely to have accounted for this instruction and thus the engine will not know how to respond. Nevertheless, the ideal gives a sensible indication of how to improve the genre: by creating a game that accepts and responds to a wider range of input, in a realistic and convincing way.

Any approach directly mapping inputs to outputs is highly unlikely to bring us anywhere near this goal. There are tens of thousands of verbs in the English language, each of which would need to have a different effect in each state of the game. There is at least one state per game location and many locations will have multiple states, because user actions will often change the location in some way.

A useful thought experiment whenever reasoning about a problem of artificial intelligence is to consider how a human would approach the problem. Imagine a human game master (GM) controlling the engine, receiving input from the player and using it to update the game world (in a similar fashion to tabletop role-playing games such as *Dungeons and Dragons*). The GM would intuitively understand the instruction given by the player, consider the knowledge they have about the game world, and use these two together to perform the update.

Another problem that emerges in any incomplete interactive fiction system is that of giving feedback to the player: although the hypothetical ideal intelligent agent will understand any coherent sentence, there will inevitably be times when any realistic IF system needs to indicate that the most recent input does not make sense given the engine's understanding of the world and human language. Thus some kind of error

message needs to be provided, and this too can be a direction of improvement for an IF engine.

When in 1972 Will Crowther wrote what is regarded as the first text adventure game, *Adventure* (now known as *Colossal Cave Adventure*), he was writing in FORTRAN to run on a PDP-10. The natural language processing solutions of the time would now be seen as highly primitive: it was only in the late 1980s that NLP researchers moved from systems of handwritten rules towards the machine-learning-based approaches that have been so successful in recent years. These were computationally intensive, but by Moore's Law a computer in the late 1980s would have on the order of a thousand times the processing power of a computer of comparable size in 1972.

The genre began to enter into a decline in the late 1980s, and during its heyday not much improvement was made on the technologies that would underpin successful interactive fiction. As such, it was seen more as an artistic pursuit than a scientific one, and so many of the papers written about it focus on its relation to literature rather than its technological foundations. However, there has been one academic paper of note.

Toward a Theory of Interactive Fiction [1] is focused on the structure of an interactive fiction experience, and highlights a few key recurring concepts that could inform the development of a game engine. The fundamental unit is the *session*, a single playthrough of an interactive fiction experience by a single player. Other important concepts are also highlighted. To handle the problem of unbounded growth in action effects due to the infinite number of states in the real world, Montfort defines a *room* as one of a set of discrete positions the player can occupy (potentially removing the need for logic to explain how a player moves around within a single location). Two rooms are *adjacent* if a player can move between them.

He also highlights the importance of interactive fiction having mechanisms for *winning* and *losing*, and containing *puzzles* for the player to solve (these typically being the primary interactive element of the experience) and potentially *characters* with whom the player can interact.

There have also been some major developments in the last few decades, mostly making use of domain-specific languages. Text Adventure Development System (TADS) [2] was released in 1988, designed by Michael J. Roberts. The language here is strongly typed and partially based on C and Pascal. It should be noted that this was far from user-friendly for a designer: although certain useful features were added, such as support for HTML to provide text formatting, graphics and sound, it is still a traditional programming language.

Inform [3] was designed by Dr Graham Nelson in 1993. Its first publicly available version, version 6, created an object-oriented, procedural language for the development of interactive fiction. The provided *Inform library* aided with the processing of user input from the player, and the representation of knowledge about the world. The latest

version, Inform 7, defines a domain-specific declarative programming language for use by the designer. This language is rule-based and so lends itself well to rule-based knowledge representation. It includes several predefined relations, such as *wearing*, performed by a person upon a wearable object: the statement *John wears a hat* initialises both John and the hat, recognising that John must be a person and the hat must be wearable. Further relations can be defined by the game designer. Thus the engine is performing a kind of natural language processing upon input provided by the game designer.

These systems, as well as the original intuition and the academic work, highlight some considerations for the development of a new engine, which will be fleshed out in the next section.

2 Preparation

Both the intuitive human-based understanding of the nature of an interactive fiction engine and the successful example of Inform led me towards a vision of interactive fiction centred on knowledge representation and natural language processing. It was necessary to more precisely define these concepts and decide upon the technology to be used to implement them, as well as the techniques that would allow me to properly evaluate my success.

2.1 Backups and Version Control

I set up a private GitHub repository for my project, and synchronised after major changes or after the end of a day of development, whichever came sooner. The entire code repository was backed up onto Google Drive at the end of each day of development. Few reverts were required, and I did not need to restore from the backup.

2.2 Development Methodology

The nature of the dissertation made the Waterfall model – with a few caveats – the sensible choice. There were no clients involved, and so there was no concern about the misunderstanding of requirements which lends Spiral development one of its main advantages.

It was also infeasible to hold more than one user study: attracting enough participants for a single study to appraise a finished product would be quite difficult; finding different participants willing to endure and provide feedback on a work-in-progress would be significantly more so.

Naturally, the project was tested as I developed it. The system was fairly modular, with tight coupling between the game and the editor but a reasonable level of independence between these, the knowledgebase and the input processing. This meant that each could be tested independently.

Thus I began with the Specification, in the form of the Project Proposal. I then proceeded with Design, determining the form of the knowledge representation and input processing and the majority of functions that would need to be used. Once these had been fully written in the Implementation phase, and tested in a brief Verification phase, I returned to Design to extend the range of permitted player actions, before implementing these and testing them. Finally, I carried out the full Verification phase which included exhaustive testing in preparation for the user study, and the study itself.

2.3 Classic IF – Colossal Cave Adventure

A full treatment of the design and code of Will Crowther's *Colossal Cave Adventure* could take up the length of this entire dissertation; one thorough example was published by Digital Humanities Quarterly in 2007 [10]. What follows is a brief summary to provide some context for the development of my own engine and highlight the difference between the original approach and the approach used in my system.

2.3.1 Description

The source code is split into two files, one for code and one for data, as is typical for a FORTRAN program. The data file consists of six tables.

Tables 1 and 2 contain location text, printed when the player moves into a different area.

Table 4 holds all recognised vocabulary keywords, of which there are only 193. Only the first five characters are stored, and these are checked against the player's entry (for example, the instruction `DOWNSTREAM` is truncated to `DOWNS`).

Table 3 associates map locations with tuples of:

- A location (entry in Table 3) which may be travelled to;
- The group from Table 4 of keywords which would be accepted to travel to that location
- Text entries in Table 1 and 2 which should be printed if one of the above keywords is entered.

Table 5 holds persistent changes to game states (for example, `THE GRATE IS LOCKED` and `THE GRATE IS OPEN` are both rows in this table; initially the first applies, but once the gate has been opened it is always open).

Table 6 holds descriptions of game events and hints.

Tables 3, 4 and 6 are of especial interest; the others simply hold descriptive strings associated with game concepts, and little improvement on these is possible, especially within the scope of this project.

2.3.2 Problems

The average adult English speaker has a vocabulary of around 20,000 words; WordNet contains over 150,000. By contrast, Table 4 in *Colossal Cave Adventure* recognises only 193, and even then only the first five characters of each. Naturally, the player would not think of using the vast majority of words in their commands to an interactive fiction game; nevertheless, CCA seems comparatively limited.

The accepted range of input is further constrained by the nature of Table 3: every association of keywords to results is associated with a state. Thus there can be no abstraction by which, for example, striking a match has the same result in any location

in the game. The only way to achieve this is by brute force repetition: encoding the same input-output pattern into every location.

The results of actions, stored in Table 6, are pre-written by the game designer and similarly cannot be abstracted to cover varying situations. Thus, every time an action is added it is necessary to write a full set of responses to a player's attempt to use that item.

Although *Colossal Cave Adventure* was a remarkable technological (and indeed artistic) achievement for its time, the technological limitations of the era limited its ability to accomplish the objectives described in the Introduction. Its range of input consists of 193 words, each of which is only applicable in a specific set of situations, and each situation must be individually coded. Its feedback to the player is pre-written and tailored to each individual action, which hampers the game's scalability.

Thus, for constant programmer time and effort, either the completeness of the existing game (that is, the range of input and feedback programmed in) or the length of the game must give way. *Colossal Cave Adventure* took several months to develop, and to create a new game would require starting again from scratch. Fortunately, with modern technology and libraries it is possible to do much better.

2.4 Requirements Analysis

I discovered that functional requirements generally apply to the implementation of the system, and non-functional requirements to its evaluation procedure. Here, each is detailed in turn.

2.4.1 Functional

Consideration both of interactive fiction experiences that I have played and the sources described in the Introduction led to some natural functional requirements for the engine.

Firstly, it must provide some kind of editing interface with which to design a game. In TADS and Inform this took the form of a domain-specific language, but I judged that a simpler interface would better fit within the time constraints of the project. By virtue of having an editor, much of the effort involved in the design of an IF experience could be reused, unlike in Crowther's structure for *Colossal Cave Adventure.*

Secondly, it must provide some processing of the natural language input provided by the player, to get it into a form whereby it can be compared with the system's knowledge and used to update the game state and return feedback to the player.

Thirdly, it must include some way of storing knowledge about the world. At its most fundamental, this includes Montfort's ideas of locations and their descriptions, and items within them. A more thorough knowledge representation describes how the

player can interact with these items and locations, and is tied in with the input processing.

The above will be discussed in this chapter and the chapter on Implementation.

2.4.2 Non-Functional

Some non-functional requirements became apparent when I considered that the purpose of a game is to entertain, and thus the most important measure of its success is its perception by those who play it.

Simply considering the enjoyability of the game, however, was insufficient. This would be affected by a range of factors outside of my control, such as the tastes of the players and their mood on the day, as well as my own ability to design an entertaining game, which is independent of the quality of the engine. In addition, this is an instinctive and emotional response by the player, and one that they might not fully be able to justify.

Further thought highlighted two main criteria with a greater level of objectivity: that the game in the new engine be perceived to accept a wider range of input than a traditional IF game (such as CCA, with its 193 understood words), and that the user be happier with the feedback they received. To achieve any kind of statistical significance, it was necessary to have at least ten participants.

Ideally these participants would include have a range of levels of experience with interactive fiction, so that both my engine's appeal to existing fans of the genre and its accessibility to novices could be tested. However, I was mindful of putting any hard constraints on the makeup of the testing base; since funding was not available for a cash incentive, I would have to rely on goodwill and small gifts. The sell would be easiest when the required investment on the part of the user was low, so I decided to focus on advertising the experiment within my College.

The setup and results of the experiment are described further in the Chapter 4 (Evaluation).

Thus the three main components of the system must be the game editor, the knowledge representation and the natural language processing, and it must be evaluated through some sort of user study looking at range of input, quality of feedback and enjoyment.

2.5 Game Editor

I decided early on that it was infeasible to spend too much time making the game editor usable – in the context of this project, I would be the only one using it to design a game, and as the designer of the interface I could be expected to have knowledge of it that a lay user might not.

The editor had to be able to change the state of the world. This would involve creating and editing locations and items, designing the topography of the world, and placing items in locations. It also needed some interaction with the knowledge representation, so the game designer could create inference rules and define properties on items.

A graphical user interface being too time-consuming for little gain, and editor usability being outside the scope of the project, I decided to proceed with a text-based interface. The game designer moves through a menu system by selecting the options provided; options are chosen by typing in the numbers that label them, or the name of the entity to be edited (for example, the name of an item).

This had the advantages of being simple to design and implement and fairly intuitive and easy to use. The textual nature of the menu would also allow for easy debugging by simply printing certain values to the terminal along with the menu. Java was chosen for the main editor language, for reasons detailed below.

2.6 Knowledge Representation

First, it was necessary to think more precisely about what had to be represented. Trivially, this included the list of locations in the game, the items, the mapping between them, and the positioning of different locations with respect to one another. This naturally lent itself to an object-oriented approach: each item and location would be an object, and properties within those objects would determine how they interacted.

Java was the obvious choice: it is the world's most widely used object-oriented language, and I was intimately familiar with it both from university and from independent projects.

More in-depth knowledge representation would involve knowledge of the properties of items, a list of actions that could be taken as well as their preconditions and effects, inference rules about properties, and other details necessary to inform the engine of the commands that the player could reasonably give.

My first instinct here was to consider graph-based knowledge representation, and indeed this was listed in my proposal as the approach with which I intended to proceed. In particular, the knowledge would be implemented as a semantic network. Items, locations and actions would be encoded as nodes, and relations between them would determine the actions that could be carried out.

It emerged during my initial research on the topic that semantic network libraries are either tailored towards heavy-duty, corporate usage, or are hobbyist open-source projects, which are often deprecated and rife with bugs.

TinkerPop3 (in the former category) was the library that merited the most consideration, but it was clearly not designed for this kind of application. The graph would run on a server, and the designer would have to use a client to interact with it.

The client was a command-line interface using a proprietary domain-specific JDK language; a programmatic Java interface was available, but poorly documented and needlessly complicated. It became clear that this would not interact well with my chosen model for the game editor – one that would abstract away the underlying code and provide a text-based interface that allowed for pure design of an interactive fiction game.

I thus began to consider other options for knowledge representation, and Prolog struck me as especially appropriate. The knowledge I wished to encode was relational: between actions and properties; between properties and items; between properties and the properties they implied. Prolog's inference capabilities naturally lent themselves towards this. Much use has been made of Prolog in knowledge representation, including in IBM's highly successful Watson computer [4]. In addition, the chapter on Expert Systems in Ivan Bratko's *Prolog Programming for Artificial Intelligence* [5], provided inspiration for a layer of inference that could be built on top of Prolog to allow it to return more useful feedback.

It was then necessary to choose a library providing an interface between Java (in which the editor would run) and Prolog (which would hold the knowledgebase and inference capabilities). Two strong options presented themselves: JPL [6] and TuProlog [7]. Both offered quite similar interfaces and were in fairly widespread usage; I finally settled on TuProlog because it provided a JDK implementation of Prolog itself, whereas JPL relied upon an installation of SWI-Prolog already on the system, which could lead to compatibility issues.

I thus resolved to design a knowledgebase in Prolog. This would include an expert system built on top of the core Prolog inference; store actions, properties, property rules and other relations; and provide an interface to the text-based Java menu and the game itself via TuProlog's Java API.

2.7 Natural Language Processing

Again, it was useful to define the task more precisely.

The engine could provide some instructions to the user about the kinds of commands accepted, but would have to be as flexible as possible within the templates it provided (and slightly outside them, as humans rarely follow instructions to the letter).

A few commands are integral enough to interactive fiction that they should be hard-coded: **inventory** to check the items currently in the player's possession; **look** to get a description of the current location; **move** to move to an adjacent location. However, hard-coding the response to every action simply brings us back to the naïve, intractable method used by Crowther and others of his time.

Other than the abovementioned single-word commands (and a few others: **save**/**restore**/etc) it can reasonably be assumed that the commands provided by the

player constitute a grammatical phrase in the imperative form: `take the bottle` or `rub the lamp` or similar. As such, a general-purpose natural language processing library could potentially be used to great success.

When I was researching libraries, one clearly stood out as being easy to use and well-respected within the development community: Stanford CoreNLP [8], designed by the Stanford Natural Language Processing Group. The usual API provides a set of annotators that can be used to carry out certain analyses on the text; tokenization, open information extraction and part of speech tagging could all be very useful here.

Tokenization splits the input into words, discarding punctuation marks that may be entered. Open information extraction can sometimes be an easy way of extracting the meaning of a sentence; when this fails, part of speech tagging along with the somewhat constrained nature of the accepted input (as described above: a grammatical phrase in the imperative form) can allow for quite effective extraction of the meaning of the entered command.

A few other processing steps can help to draw out the intended meaning of the command.

The verb in the sentence should not have to exactly match the label of the action specified by the game designer. Even the subset of English words in common use is rife with synonyms. Because there is a finite (and small) set of verbs with specified behaviour within the game, even an exhaustive lookup is feasible: enumerate a list of synonyms of the verb, and compare them with the set of specified verbs to see whether there is any overlap. WordNet, as the most exhaustive listing of synonyms and hypernyms, was a sensible choice as a dataset.

Because the nouns used generally correspond to items in the game world, and these are named in the output the player sees, it is highly unlikely that the player will attempt to use synonyms instead of the original item names. However, they may well refer to an item by only part of its name – e.g. **dustbin** rather than **small dustbin** – and this method of referencing an item needs to be handled in some way.

Humans often make errors in typing in words, especially when engrossed in a task that requires them to look at something other than what they are typing, in this case previous replies given by the game. It is thus sensible to run some kind of spelling checker over what they have entered, and automatically correct (if possible) or give a list of suggestions of what the player may have meant.

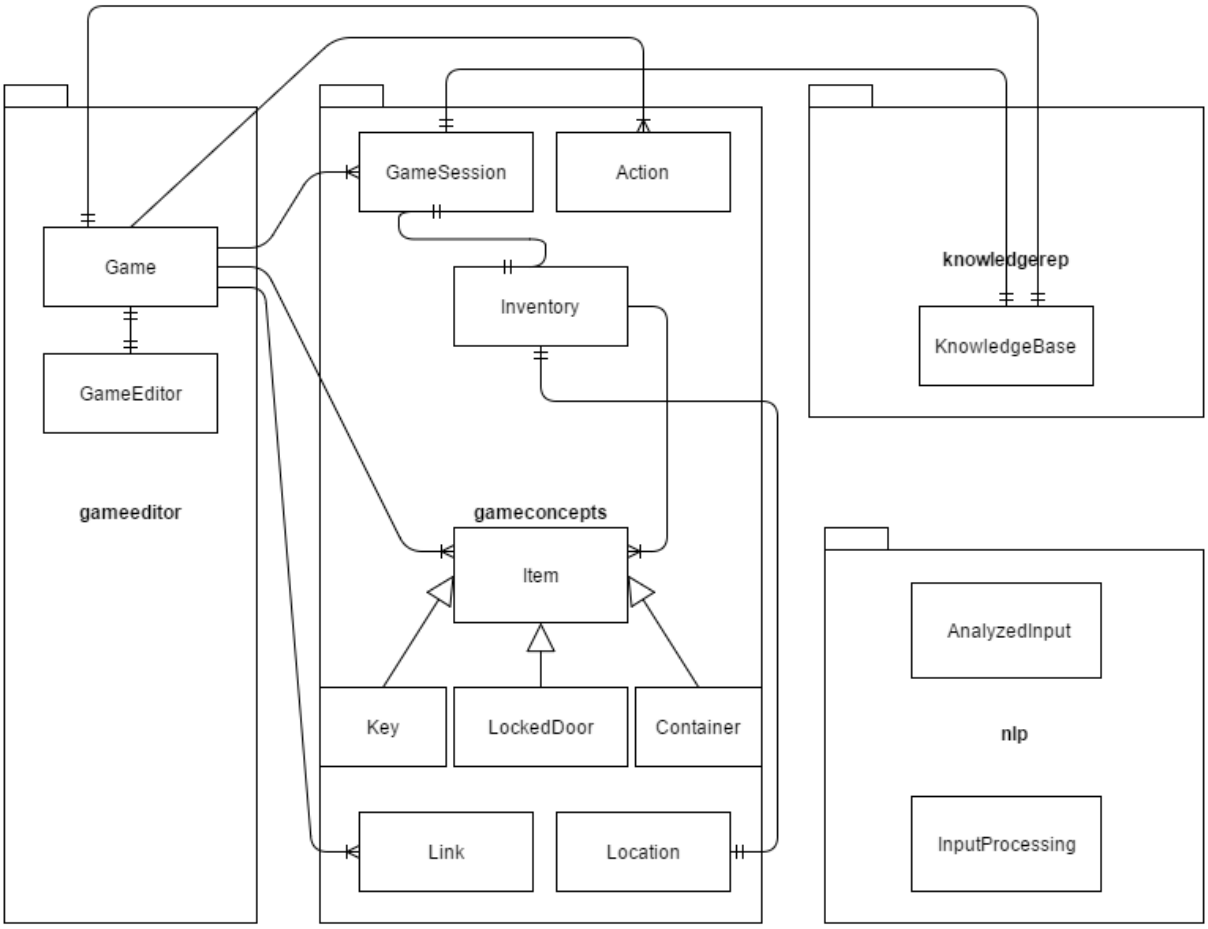
In total, the system should take in a user's input and:

- Check whether it matches any of the predefined instructions;
- Pass it through CoreNLP to have information extracted and words tagged with their part of speech, and WordNet to enumerate verb synonyms;
- Correct any detected spelling errors and relate the processed input to the knowledgebase.

To summarise: the system needed an editor, a knowledgebase and an input processing module. The editor was to be written in Java and consist of a text-based menu; the knowledgebase would be implemented in Prolog with an interface to the Java editor; the input processing would make use of Stanford CoreNLP to get the input into a form whereby it could be compared with the knowledgebase.

3 Implementation

The implementation phase can be split into the development of the core system, the game editor, the object classes, the knowledge representation and the input processing.



A class diagram showing the relationship between the core system and the object classes

3.1 Core System

It was first necessary to implement a core framework around which the main components could be built. This was contained in the Game.java class, which initialises the game editor, allows for the creation of sessions, and handles the saving and loading of the game.

It is very important that *deep copying* is performed when creating a new game session. The game editor edits fields within the Game object to hold the values specified by the designer; the session should be initialised to these values but should not be able to change them. For example, if the designer places a **book** in the **bedroom** and the player starts in the bedroom and says `take the book`, the book should be removed from the bedroom in that session, but not in the main game.

The use of Java's **clone** operator is widely discouraged in favour of the employment of copy constructors, and so each of the storage classes has such a constructor. Almost all of the storage classes have their created objects stored in a HashMap, wherein the key is the name of the object (a string), and the value is a pointer to the object itself. Deep copying this is a matter of initialising a new HashMap, iterating over the keys in the original and mapping them to newly copy-constructed versions of the original values.

Sessions are stored in a similar way: one field of the Game object is a HashMap of GameSession objects, keyed by names specified by the player.

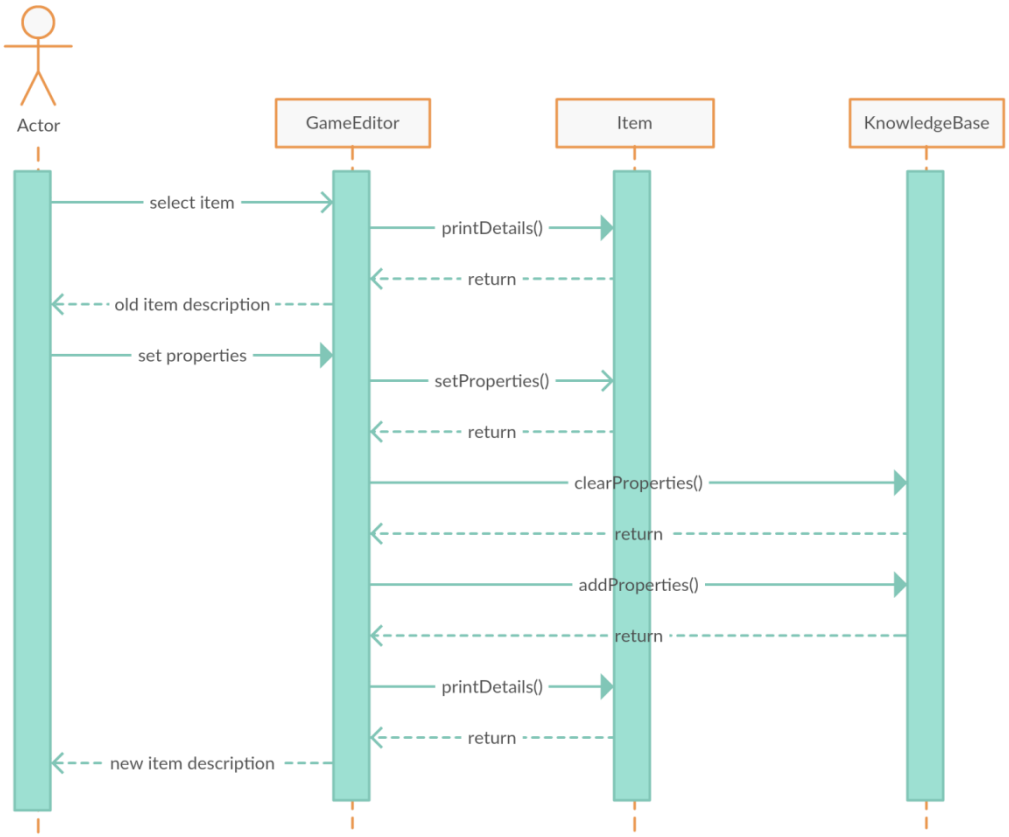
On the main screen, there are three options: `play`, `edit` and `quit`.

The `quit` command terminates the program, saving the current game and the list of sessions by serializing them to a file (all storage classes implement the Serializable interface for this purpose).

The `play` command prompts the player to select or create a session (initialized with the latest game state from the editor) and accordingly places them in the last known location or the designated start location of the game, provides them with the instruction menu and the description of their location, and begins accepting input.

The `edit` command launches the game editor, which is described in the next section.

The Game.java class thus provides a base on which to build the more substantive elements of the system.



A sequence diagram showing the process of setting properties on an item

3.2 Game Editor

The editor is launched from the core system and allows the editing of a game through a text-based menu, which appears as follows:

```
Enter a number to continue
(1) Manage locations
(2) Manage items
(3) Set start location
(4) Set victory location
(5) Set defeat locations
(6) Set victory item
(7) Manage actions
(8) Manage property rules
(9) Print knowledgebase
(10) Manage links
(11) Create locked door
(12) Create container
(q) Exit
```

Each of these options will now be described.

(1): provides a list of locations created so far, and allows the designer to modify existing locations or create a new location. Locations have a name (set at time of creation), flavour text (a description that appears when the player enters or looks around the location), a list of items, and a list of exits.

(2): provides a list of created items, allowing for modification of any item, or initialization of a normal item. All items have a name, description and list of properties. Containers, Locked Doors and Keys have extra properties of their own.

(3) – (6): Designate locations and items as specified.

(7): Allows for the creation of a new action, which has a base property and a set of effects.

(8): Allows the creation of rules for inference on item properties.

(9): Prints the entire knowledgebase as it currently exists. Useful for debugging, especially with respect to minor errors of Prolog syntax, such as missed full stops and improper spacing.

(10): Allows the designation of linked objects, which can be combined into a single object.

(11): Allows the creation of locked doors, and keys to go along with them.

(12): Allows the creation of containers and the initialization of their contents.

(q): A persistent command that quits and returns to the main game menu.

Options **1, 2** and **7** are detailed further in **Object Classes.** Options **8–12** are detailed further in **Knowledge Representation**.

3.3 Storage Classes

Several classes of object are used to hold the object-oriented model of the game world. Some of these have ramifications for the knowledge representation of the system, and so are described further in the next section.

3.3.1 Location

A location corresponds to a single *room* as described by Montfort [1]. It is an atomic position that the player can occupy at any point in time. It is initialised with the name of the room and its flavour text, which is printed when the player enters the room; subsequently, the designer can edit this flavour text, add and remove items or add and remove exits. The `exits` field is a HashMap mapping a Direction (an enum holding one of the eight cardinal directions) to a String specifying the name of the destination room.

Items are stored in an Inventory assigned to the room, detailed in the next section.

3.3.2 Inventory

Inventory is a wrapper class around a String-Item HashMap. It allows items to be added or removed, and has a method to check whether an item is present. It also has a method to return a set of item names, which is useful for error correction.

Each location in the game has an Inventory, and the GameSession object has its own Inventory, corresponding to the items in the player's possession at any point in time.

3.3.3 Item

An item is initialized with a name and description; from this, it generates a set of tags by splitting the item name on white space (for example, the item **red ball** would have **red** and **ball** as tags, so that the command `take the ball` would be interpreted as taking the red ball if this is the only item present, and would prompt the player to be more specific if another matching item (such as a **green ball**) is also available. The designer must also specify whether or not the item can be taken – the player should not be able to take a heavy chest into their inventory.

The designer can then set the item's properties via a pop-up window.

3.3.4 LockedDoor

A LockedDoor is a subclass of Item that also has an origin location (to which it is automatically added as an item), a destination location and a direction. When the LockedDoor is unlocked, a door opens in the origin location, leading to the destination location via the specified direction. The LockedDoor also takes the name of a key, which the editor generates at the same time as the locked door.

The LockedDoor is automatically set to be non-takable.

3.3.5 Key

Key is a special subclass of item, an instance of which is automatically created whenever the designer creates a LockedDoor. It cannot be created independently of a LockedDoor. This key is the only object that can open its door.

3.3.6 Container

The Container is another special class of item, which allows other items to be put into it. When creating a Container, the designer can initialise a set of items that are inside it. The player can either take items from the container or place items inside it.

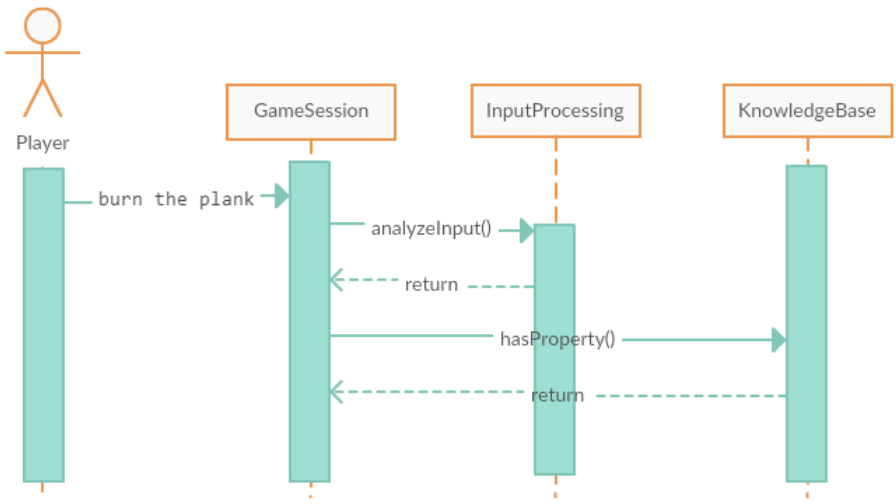
3.3.7 Link

A Link is an object holding two base items and a result item. Its existence signifies that the two base items can be combined – with the action `combine item1 and item2` – to yield the result item. This class exists so that links can easily be edited or removed from the editor and so that the engine knows which item to create. Links will be described further in the Knowledge Representation section.

3.3.8 Action

An action has a name, a base property (of the item undergoing the action), and a set of effects. Either the action can have the single effect of destroying the item, or it can add and remove any number of properties from the item. In addition, it can add any number of items to the inventory.

These classes represent the basics of the game world; some of them also tie in with the knowledge representation and help to provide an interface for the editor to change the knowledge represented.



Collaboration diagram showing the system's behaviour when the user takes an action on an item

3.4 Knowledge Representation

The engine's knowledge about the game world is implemented in Prolog. A set of relations encode this knowledge; an expert system allows the engine to make sense of it; and a Java class provides some code to integrate with the Java-based editor and engine, parse the Prolog return values and generate natural-language feedback.

3.4.1 Prolog Basics

As suggested in the Preparation section, the Prolog code provides the core of the knowledge representation functionality. All of the game's knowledge about the world (other than elementary considerations, such as room-item mappings) is stored as Prolog.

The Prolog is based around relations either between items and other items, or between items and properties, or between properties and properties. Six such relations exist: `opens`, `linkedwith`, `combinable`, `fitsinside`, `inside` and `hasproperty`.

3.4.2 Expert System

Underlying this is an extension of basic Prolog, called the Expert System and modelled on a similar Expert System described by Ivan Bratko [5]. A distinction is drawn between knowledgebase and Prolog statements; the latter are created by labelling the former as either a fact or a rule. For example, consider a plank made of wood. The knowledgebase statement is `plank hasproperty wooden`. The corresponding Prolog statement is `fact : plank hasproperty wooden`, and this statement simply encodes the knowledgebase statement previously given.

As in regular Prolog, facts are atomic and predetermined, and always hold. Rules hold if and only if their premises hold: in this case, the form is `rule : <statement1> if <statement2> and <statement3> …`

Rather than directly querying the Prolog rules laid out in the knowledgebase, querying of the knowledgebase is thus done through a special function, denoted by `answer(Goal, Truth, Reasoning)`, or `answer(G, T, R)` defined with the following cases:

```
answer(Goal, true, specified) :- fact : Goal.
```

This is the simplest case, in which Prolog answers that the goal was specified, when the goal is denoted as a fact in the knowledgebase.

```
answer(Goal, true, specified) :-
      rule: Goal if Premise,
      dealWithConjunction(Premise, true, FPCL).
```

This case identifies a rule with the goal as its conclusion, and considers the premise. It passes the premise to a helper function, `dealWithConjunction`. The variable `FPCL` holds the **failed precondition list**, which is returned to the engine as an explanation for the failure of a query. It consists of those premises within the rule which did not hold, and as a result caused the conclusion of the rule to fail to hold. In this case, however, the rule held and so we are not interested in the precise mechanism by which it came to hold.

```
answer(Goal, false, FPCL) :-
      rule: Goal if Premise,
      dealWithConjunction(Premise, false, FPCL).
```

This is the same case, except that here the rule has failed to hold. Thus the engine *is* requesting the information about which preconditions failed to hold (held in the failed precondition list). Thus the value held in `FPCL` must be returned.

The `dealWithConjunction` function has four cases.

```
dealWithConjunction(A and B, true, []) :-
      dealWithConjunction(A, true, []),
      dealWithConjunction(B, true, []).
```

The conjunction should be denoted as true (and thus have no failed preconditions) if the two terms that make up the conjunction each themselves return true.

```
dealWithConjunction(A and B, false, FPCL) :-
      dealWithConjunction(A, TA, LA),
      dealWithConjunction(B, TB, LB),
      notBothTrue(TA, TB),
      append(LA, LB, FPCL).
```

Conversely, if one of the conjunctions is returning `false` then we must consider where the error arose. This is essentially a depth-first search: keep appending together the failed precondition lists as we recursively explore each conjunction, and return the final result. The condition `notBothTrue(TA, TB)` simply establishes that at least one of them is false (because if it is not, `dealWithConjunction` should not return false at all). Although this case should be captured by the previous rule, it is a good idea to ensure orthogonality between Prolog rules where possible.

```
dealWithConjunction(X, true, []) :-
      answer(X, true, _).
```

If the statement is not in the form of a conjunction, it is simply a single check that can be performed by the answer function: either it is a fact that atomically holds, or there is some sort of rule governing it, in which case we need to continue with the recursive search.

```
dealWithConjunction(X, false, [X]).
```

If no justification can be found to make the statement true, then by the closed-world assumption it is false. It is thus a failed precondition, and the failed precondition list just contains the statement.

Thus carrying out the Prolog query `answer(<knowledgebase query>, T, R)` explores by the above mechanism whether the knowledgebase query holds. When it returns, `T` holds the truth value of the knowledgebase query (either `true` or `false`) and `R` holds either `specified` in the case of a `true` value or the `FPCL` in the case of a `false` value. This is further processed by the Java code in the KnowledgeBase class, described in the Java Interface subsection.

The necessity of this extension becomes clear with the discussion of item properties at the end of the next subsection.

3.4.3 Prolog and Knowledgebase Relations

The `opens` relation takes a key and a locked door. At the time when the designer creates a LockedDoor via the editor menu, the statement `fact : <keyName> opens <doorName>` is added to the Prolog. When a player invokes the open command, the input processing tries to get it into a form whereby one item is being opened by another. If it can be transformed into such a form, a lookup for the above fact is performed. If it is found, the door opens. If not, an error message is returned (the exact error message depends on other characteristics of the two items).

The `linkedwith` relation corresponds to a Link object and connects two items. Creating a link inserts the statement `fact : <item1> linkedwith <item2>` into the Prolog. This relation should be symmetric (if **A** is linked with **B** then **B** should also be linked with **A**), but the naïve method of ensuring symmetry leads to an infinite loop.

Consider the rule `fact : X linkedwith Y :- fact : Y linkedwith X.`

If there is no atomic statement finding that one item is linked with another, Prolog will follow this rule until stack overflow, switching around the arguments and never logically terminating. Thus, it is necessary to define another operator, `combinable`, which does exhibit symmetry, and use this when querying the knowledgebase but `linkedwith` when updating it.

```
fact : X combinable Y :- fact : X linkedwith Y.
fact : X combinable Y :- fact : Y linkedwith X.
```

Clearly, the problem of an infinite loop is avoided. This is encoded as a Prolog rule rather than a knowledgebase rule because the problem it addresses is inherent in Prolog, and not of interest to the game engine.

The `fitsinside` relation also connects two items. The only constraint that needs to hold in order for this relation to hold is that the second item has the property of being a container.

```
rule : I fitsinside C if C hasproperty container.
```

Other methods of determining whether an item could fit inside a container included comparing the volume of the item with the free volume of the container, but this was abandoned due to time constraints and a perceived lack of effect upon the player of the game.

The `inside` relation is set when an item is initialised to being inside a container by the game designer, or when the player puts the item into the container. It is removed when the player removes the item from the container (to prevent them from taking the same item more than once).

The `hasproperty` relation is more involved, and makes use of the expert system developed.

When the game designer sets an item's properties, any `hasproperty` relations concerning that item are cleared, and for each property the statement `fact :<item>`

`hasproperty <property>` is added to the Prolog code. Thus the expert system will clearly agree that the property holds, returning `T = true` and `R = specified` as described in the Expert System subsection above.

The game designer can also set property rules, via option **8** in the game editor. This allows the designer to designate a goal property, and a list of premise properties that together imply the goal property. For example, the designer may wish to have a property **flammable** that applies to an item if the item is **wooden** and **dry.** Thus `flammable` is the goal property, and `[wooden, dry]` is the premise list.

The designer also creates actions, each of which has a base property. When a user attempts to carry out this action on an item (established by the processing of their input, discussed in the next section), the system runs the knowledgebase query `item hasproperty baseProperty`, which is to say the Prolog query `answer(item hasproperty baseProperty, T, R).` An action can itself modify the properties of the object on which it acts, if this is specified when the action is created.

There are three possibilities of concern here.

The first is where the item does indeed have the property (whether this is specified by the designer or inferred from a set of property rules, the response is the same), which returns `T = true` and `R = specified`. In this case the action is successful and its effects are carried out.

The second is where the property does not hold, but some rule could have made it hold. Here `T = false` and `R = FPCL`, which can be used to provide feedback to the player.

The third is where the property does not hold and no rule has been defined on it. In this case `T = false` and `R = []`, and this too can be used to provide feedback.

Through the six relations, knowledge of the game world is encoded in the Prolog code. The expert system allows the analysis of these six relations to allow for conclusions that go beyond reading what has been written to the knowledgebase.

3.4.4 Java Interface

Integrating the Prolog code described in the previous subsection into the main Java editor and game framework requires the use of the TuProlog library with some extensions.

The library works through an object of class `Prolog`. First a theory must be added to this object, and then the object can be queried. My `KnowledgeBase` Java class defines instance properties for `facts` and `rules`, each of which is a `HashSet<String>` in which each string is a single Prolog statement. This modular representation makes it easy to remove existing rules (for example, if an item loses a property as the result of an action

that it undergoes). In addition, it has a field `base` of type `Theory` (a class defined by TuProlog to hold the theory) which is imported directly from the `expertSystem.pl` file.

Each of the knowledgebase relations described in the previous subsection has its own method for adding or removing a relation, or checking the presence of the relation by querying.

When a query is carried out, the instance method `query()` is called. This initialises a new `Prolog` object, adds the contents of `base`, `facts` and `rules` as theories and executes the requested Prolog query.

A different function, `generateFactForQuery()`, wraps a knowledgebase query to convert it into a Prolog query, for example:

`plank hasproperty flammable` → `answer(plank hasproperty flammable, T, R).`

When the query returns, it returns an object of type `SolveInfo`, which contains the variable mappings carried out by the query. Extracting these values is simple, but parsing them can be more difficult, in particular if a `hasproperty` query has failed and it is necessary to parse the failed precondition list.

Although TuProlog accepts infix operators such as `plank hasproperty wooden`, it understands this to mean `hasproperty(plank, wooden)` and uses this format when returning Prolog output, so each entry in the failed precondition list is of this form. This (in particular the presence of the comma) complicates the task of converting the FPCL into a Java list:

```
private static List<String> javaListFromPrologList(String prologList) {
	int bracketDepth = 0;
	List<String> javaList = new LinkedList<String>();
	String currentElement = "";
	for (char c : prologList.toCharArray()) {
		if (c == '[') continue;
		if (c == ']') javaList.add(currentElement);
		if (c == ')') bracketDepth--;
		if (c == '(') bracketDepth++;
		if (bracketDepth == 0 && c == ',') {
			javaList.add(currentElement);
			currentElement = "";
			continue;
		}
		currentElement += c;
	}

	return javaList;
}
```

This handles the beginning and end of the list, and the fact that commas delineate list items only if they are not within a set of parentheses (i.e. within one of the failed premises).

It is then necessary to parse each individual failed premise:

```
private static String parseFailedPremisePrologFunctor(String prologTerm) {
	Pattern p = Pattern.compile("(.*)\\((.*),(.*)\\)");
	Matcher m = p.matcher(prologTerm);
	m.matches();
	String functor = (m.group(1));
	String arg1 = (m.group(2));
	String arg2 = (m.group(3));
	if (functor.equals("hasproperty")) {
		return "the " + desanitiseString(arg1) + " is not " +
desanitiseString(arg2);
	}
	return "it is not the case that " + desanitiseString(arg1) + " " +
functor + " " + desanitiseString(arg2);
}
```

The regular expression matches the previously mentioned format of a Prolog compound term with arity 2, extracting the functor and each argument. Here `arg1` corresponds to the item name and `arg2` corresponds to the property, and so the sentence returned might be *the plank is not fragile* if that were the property queried. In the other case, you might get *it is not the case that blue key opens red door*.

Note that `sanitiseString` is a function that replaces spaces with underscores to make item names Prolog readable; `desanitiseString` merely reverses this.

Finally, the following function puts the above functions together by converting the Prolog list into a Java list and parsing each individual term:

```
public static String fullyParsePrologFailedPremiseList(String prologList) {
	List<String> javaList = javaListFromPrologList(prologList);
	if (javaList.size() == 0) return "This is because that property is
atomic and does not hold.";
	List<String> reasons = new LinkedList<String>();
	String fullReason = "This is because ";
	for (String s : javaList) {
		String reason = parseFailedPremisePrologFunctor(s);
		reasons.add(reason);
	}
	fullReason += String.join(" and ", reasons) + ".";
	return fullReason;
}
```

Here, each individual term in the list is converted into a human-readable reason, and these are joined with the 'and' conjunction in the usual way and terminated with a full stop.

By these methods, the processed version of a command by the user involving the game's knowledgebase can be checked against the knowledgebase, and the success or failure of the query can be converted into a natural-language explanation, which is used to give feedback to the user.

3.5 Input Processing

Input Processing occurs in two stages: the conversion of a user-entered string into an object of type `AnalyzedInput`, and the processing of that `AnalyzedInput` until it is in a form which can be used to query the knowledgebase as described in the previous section.

3.5.1 Building the Analyzed Input

The first stage makes use of Stanford CoreNLP, as well as my custom-defined `AnalyzedInput` class. This class has a single verb, a list of nouns and a list of verb synonyms. The verb corresponds to the action being carried out; the nouns correspond to the items on which that action is carried out. As a simple example, the command `open the red door with the red key` would result in the `AnalyzedInput`:

```
[verb=open, nouns=[red door, red key], verbSynonyms=[…]]
```

Here, the verb synonyms would not be used as the verb **open** corresponds to an existing construct in the game engine.

One unfortunate consequence of using Stanford CoreNLP was that the initial loading of models means that the first analysis carried out is significantly slower than subsequent analyses, leading to an uncomfortably long wait after the player has entered their first command. I got around this by invoking a sample call in the constructor of the `InputProcessing` class (which holds the code that processes user input), parsing the sentence *open red door with red key.* This result is thrown away; the effect of the invocation is that the loading time occurs when a game session is created rather than when the first user command is entered. This is less jarring for the player.

The instantiated `InputProcessing` then calls a method `analyzeInput()`, which at its core carries out `posParse()`:

```
private AnalyzedInput posParse(String input) {
	List<String> nouns = new LinkedList<String>();
	String verb = getFirstWord(input);
	Set<String> verbSynonyms = null;

	Document doc = new Document(input);

	// There should only be one sentence. Ignore others.

	Sentence sentence = doc.sentence(0);
	String currentNoun = "";
	boolean prevWasNoun = false;
	for (int i = 0; i < sentence.length(); i++) {
		String word = sentence.word(i);
		String tag = sentence.posTag(i);
		// Include preceding adjective in the noun.

		if (tag.startsWith("N")) {
			if (prevWasNoun) {
				//handle compound nouns
				currentNoun = nouns.remove(nouns.size() - 1);
				nouns.add(currentNoun + " " + word);
				currentNoun = "";
			} else {
				nouns.add(currentNoun + word);
				currentNoun = "";
			}

			prevWasNoun = true;
		}
		else prevWasNoun = false;
		if (tag.startsWith("J")) currentNoun += word + " ";
		// Only one verb
		if (tag.startsWith("V")) verb = word;
	}
	if (verb != null) verbSynonyms = getVerbSynonymsAndHypernyms(verb);
	return new AnalyzedInput(nouns, verb, verbSynonyms);
}
```

In essence, this carries out a CoreNLP part of speech tagging on the user input, and then examines the input one word at a time. If a verb is seen (indicated by a tag starting with V), it is designated as the verb of the sentence. Each sentence is a single instruction and should only contain one verb. If a noun is seen, it is added to the list of nouns; however, consideration must be made of adjectives describing nouns (red door) and compound nouns (bathroom door): thus both preceding adjectives and preceding nouns are included within the current noun.

The function `getVerbSynonymsAndHypernyms()` calls WordNet, and provides the verb as a search term. WordNet returns a set of synonyms and a set of hypernyms in its own format (with associated information), and just the words themselves are converted into sets and joined together to give a single set of synonyms for the verb in the input. WordNet even allows the calling function to specify that it is a verb being looked up, which reduces the number of extraneous suggestions.

3.5.2 Using the Analyzed Input

Each of the core functionalities of the game session has its own function: taking items, taking items from containers, inspecting items, acting on items, combining items, and opening doors. The action is chosen by performing a `switch` on the first word of the user input; if this is not one of the engine-defined actions, the player is assumed to be taking a game-specific action on an item.

Each of these acts upon the `AnalyzedInput` in a similar but subtly different way. Consider the example of taking an action upon an item, which is fairly representative.

If more than one distinct noun is present, the function returns; in this version of the engine, actions can only act upon one item at a time.

If an item whose name exactly corresponds to the name typed in is present in the inventory, this item is acted upon. If not, a set is constructed from the names of all of the items currently in the inventory and all of those present in the current location. These will be used as a base from which to draw suggestions.

Here, a function `getPossibleItems()` is used to narrow down the field of suggestions. It takes a set of strings representing the universal list of suggestions, and a single string representing the raw item name entered by the user. If the exact name of the item is contained in the set, the singleton set consisting of that item name is returned.

If not, it iterates over the universal set one item at a time. It then considers each of the item's tags in turn (recall from the subsection on Object Classes that an item's tags are simply the distinct single words contained in its name: a **red ball** has the tags **red** and **ball**). It then calculates the Levenshtein edit distance between the raw item name and that tag: the number of insertions, deletions and substitutions required to turn the raw name into the tag.

The maximum Levenshtein distance is the length of the longer word, so the actual distance can be scaled by dividing it by this maximum. Since a low edit distance corresponds to a high similarity, it is then necessary to take one minus this scaled distance.

$$d_{max} = \max(n_1, n_2)$$

$$s = \left(1 - \frac{d}{d_{max}}\right)$$

Here s is the similarity, d is the Levenshtein distance and n_1 and n_2 are the lengths of the words being compared.

If the similarity is greater than 0.5 for one of its tags, the item is added to the set of possible items.

If this set is empty when the function would return, it instead raises an exception and informs the user that it does not recognise the item. Thus it is guaranteed that if the function returns, there is at least one recommendation.

The `getPossibleItems` function is called by a function called `didYouMean`, which returns a string. If the size of the set of recommendations taken in is exactly 1, then the single item is returned. If it is greater than 1, the user is presented with the full list and asked to choose one. If the user does not choose one, the same exception is raised warning the user that the item was not recognised.

In a similar fashion, if the `verb` in the `AnalyzedInput` is recognised as the label of one of the predefined actions, then the action is simply carried out. If not, the function iterates over the `verbSynonyms` field and sees whether any of the synonyms correspond to recognised actions (a constant-time lookup for each synonym, since the actions names are the keys in a `HashMap`). Again, if only one action matches then the action is carried out; if multiple actions match the synonyms of the entered verb, the player is asked to choose one.

Confident that it has a correct action and item name, the engine queries the knowledgebase to see whether the action can be carried out. If it can, then the action takes effect. If not, the failure reason is printed to the terminal so that the player has useful feedback about why the action failed.

There are subtle differences in the other functions: for example, when taking an item from a container the universal set of suggestions is initialised to the container's contents rather than the set of all items present. However, for the most part the above strategy is followed.

By this method, the user's raw natural language input is successively converted into a more structured form corresponding to concepts within the game. This structured form allows the system to query the knowledgebase (as described in the previous section) and return a response to the input.

4 Evaluation

4.1 Sample Run

Automated verification is difficult to perform on a system of this nature. Rather than being dependent upon some input from a possible range, the output generated by the game is dependent upon the human designer's interaction with the game editor, and the human player's interaction with the game session.

However, a few checks can be performed.

Suppose that through the editor, the game designer creates an action **burn**, which can be carried out on **flammable** objects. Suppose an object is flammable if it is wooden and dry. The game designer creates a plank, which is both wooden and dry; a bone, which is dry; and an apple, which is neither wooden nor dry.

Printing the knowledgebase from the editor shows that the properties have been accurately encoded in Prolog:

```
fact : plank hasproperty dry.
fact : bone hasproperty dry.
fact : plank hasproperty wooden.
rule : X hasproperty flammable if X hasproperty wooden and X hasproperty
dry.
```

Next, it is necessary to test the action in-game. This requires the creation of a room to hold all of the items.

The player begins by taking the items, and then types `inventory`. As expected, the game replies

```
[apple, bone, plank]
```

Trying to burn each of these in turn:

```
> burn the apple
burn failed because it is not flammable
This is because the apple is not wooden and the apple is not dry.
> burn the bone
burn failed because it is not flammable
This is because the bone is not wooden.
> burn the plank
The item is burned to a cinder
You lost the plank
```

As expected.

Next, it is useful to test the input processing. The designer creates a bedroom and a kitchen; there is a red door going north from the bedroom to the kitchen; the bedroom contains a red key, a car and a crab.

```
BEDROOM
room with bed

Available Items:
red key
car
crab
red door

Exits:


> take the carb
Did you mean:
[car, crab]
Type an item name or 'no' to return
no
Sorry, that item - carb - was not recognised
> take the red key
Took red key
> open the door with the key
a door opens to the NORTH
> take the cor
Took car
> take the crib
Took crab
```

Where the user names an ambiguous item (carb), they are prompted to correct it. Where the item name can be corrected unambiguously (cor, crib), the correction is automatic. Also note that the red door and red key are referred to simply as `door` and `key`.

The WordNet lookup can be tested in a similar way if the crab is designated as edible, and the `eat` action destroys an edible item.

<Session 1>

```
> eat the crab
The item is devoured.
You lost the crab
```

<Session 2>

```
> consume the crab
The item is devoured.
You lost the crab
```

<Session 3>

```
> devour the crab
The item is devoured.
You lost the crab
```

Although the action is called `eat`, the WordNet lookup allows for the recognition of the word `consume`, as well as the word `devour`.

4.2 User Study

4.2.1 Method

As originally planned, I decided to proceed with a comparison of a game developed using my engine and the classic text adventure title *Colossal Cave Adventure* (CCA), as perceived by a range of users who would then fill in a questionnaire, comparing the range of input, quality of feedback and enjoyability. I also decided to evaluate my own system using the System Usability Scale (SUS) [9] to give a more objective measure of its usability.

The SUS questionnaire involved the following questions, slightly changed from those in the base SUS to better apply to this situation:

1. I think that I would like to play this game frequently.
2. I found the game unnecessarily complex.
3. I thought the game was easy to play.
4. I think that I would need the support of a technical person to be able to play the game
5. I found the various functions in the game were well integrated.
6. I thought there was too much inconsistency in this game.
7. I would imagine that most people would learn to play this game very quickly.
8. I found the game very cumbersome to play.
9. I felt very confident playing the game.
10. I needed to learn a lot of things before I could get going with this game.

Each was scored by each participant from 1 (strongly disagree) to 5 (strongly agree).

To get the SUS score for a single user, sum together $(Answer - 1)$ for each positive (odd-numbered) question and $(5 - (Answer - 1))$ for each negative (even-numbered) question to get a score out of 40, then multiply by 2.5 to get a score out of 100.

$$SUS = 2.5 * \sum_{i=1}^{5} (A(2i-1) - 1) + \big(5 - (A(2i) - 1)\big)$$

Where $A(j)$ denotes the answer to question j. The first term in the summation represents odd answers; the second denotes even answers.

As funding was not available, the incentives I could provide were limited and so I was conscious of the fact that the duration of an individual experiment would have to be short to avoid putting off would-be participants. I judged that the longest possible experimental slot would be 20 minutes. This allowed for 6 minutes on each game (12 in total) and some time at the beginning to sign the consent form and read the instructions, and at the end to fill in the questionnaire and discuss the experiment.

There were 15 participants in total, significantly more than my desired minimum of 10. These were mostly students at my College for reasons of convenience – users are far

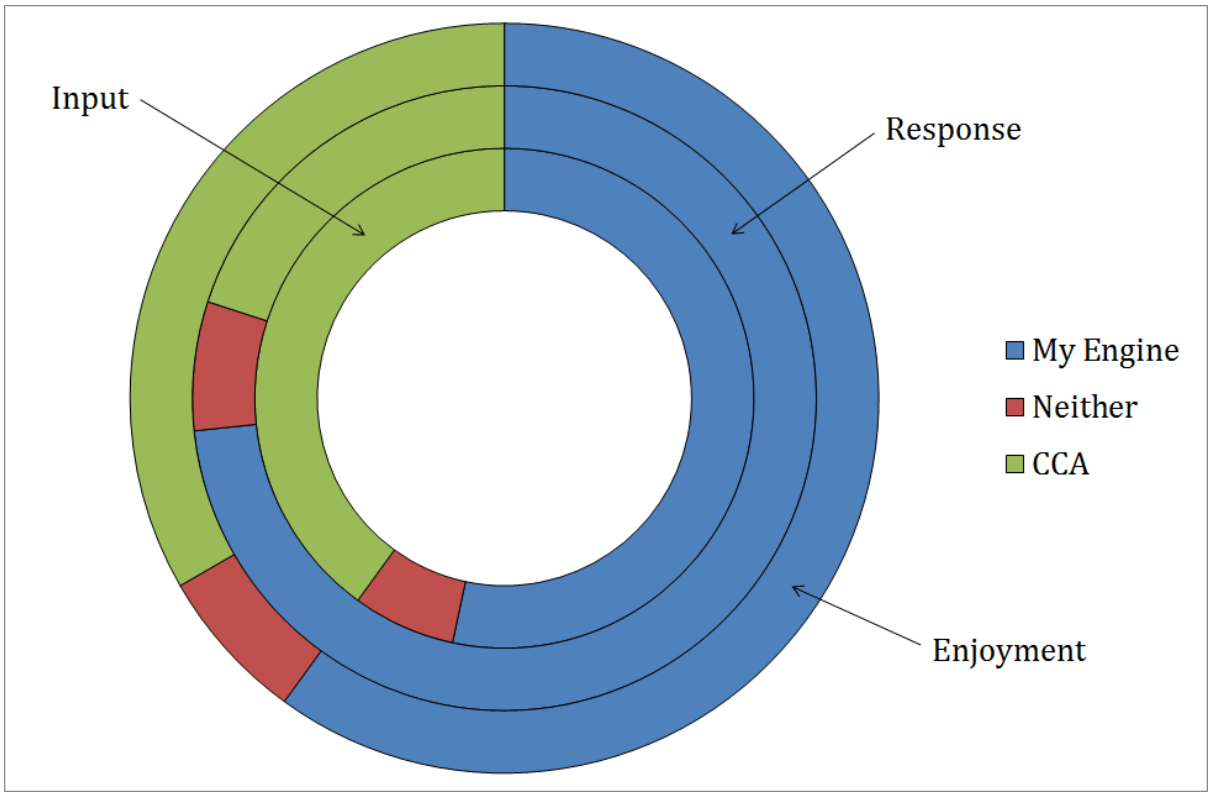
more likely to participate when it requires less effort on their part. Only two of the participants expressed any previous familiarity with interactive fiction, in keeping with its niche and slightly dated nature.

4.2.2 Observations

A few facts came to light based on my observation of the players' interaction with the game, their reactions and their feedback forms:

- Users were visibly frustrated with the inconsistency of CCA's input – to be expected given that it follows a per-state input-output model.
- There was a lot of confusion when the user's first guess at the correct action to proceed did not work – many just repeated the instruction, which naturally gave the same result. This caused visible frustration.
- Users dislike reading instructions, but they prefer help menus that indicate the correct action to not being able to find the correct action at all. Users seem to divide between thinking that only instructions in the help menu can be carried out, and thinking that any arbitrary instruction can be carried out.
- Multiple participants highlighted the lack of time, but it would be very hard to get people to sign up to longer slots given the lack of available funding, and allowing only certain participants to keep playing for longer might bias the data.

4.2.3 Results



Doughnut chart showing the comparative performance of the engine on each measure

Eight participants thought my game supported a wider range of input; **six** thought that CCA did; **one** noticed no discernible difference. Thus my game was supported by **53.3%** of all participants and **57.1%** of those who expressed a preference.

Eleven participants preferred my game's feedback; **three** preferred that of CCA; **one** noticed no difference. Thus **73.3%** of all participants, and **78.6%** of those with a preference, preferred my game.

Nine enjoyed my game more; **five** preferred CCA; **one** enjoyed them both equally. This equates to **60%** of all players and **64.3%** of players with a preference preferring my game.

On the system usability scale (SUS), the data were as follows:

Q\P	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	5	2	4	2	4	2	1	4	1	5	3	4	4	2
2	2	2	4	3	1	1	2	3	2	4	3	3	2	2	4
3	2	2	2	3	4	3	2	1	4	2	3	2	4	4	2
4	2	2	4	2	1	1	4	4	1	3	2	2	1	1	4
5	4	4	3	2	5	5	4	2	5	3	4	3	5	4	2
6	2	3	2	3	2	1	1	3	2	3	1	3	1	2	3
7	4	4	3	4	4	4	3	3	5	2	4	2	5	5	3
8	2	2	4	2	2	1	2	4	1	3	3	4	1	1	4
9	3	1	2	2	4	4	1	2	5	2	2	4	4	4	2
10	2	2	5	2	1	1	4	4	2	4	2	2	2	1	4
Total	25	25	13	23	32	35	19	11	35	13	27	20	35	34	12
Score	**62.5**	**62.5**	**32.5**	**57.5**	**80**	**87.5**	**47.5**	**27.5**	**87.5**	**32.5**	**67.5**	**50**	**87.5**	**85**	**30**

Here, the rows correspond to questions on the questionnaire (as listed above) and the columns correspond to participants in the experiment.

The mean score was $\mathbf{59.8/100}$, with a standard deviation of $\mathbf{22.5}$.

4.2.4 Analysis

My game was preferred on all three measures, though by very different margins. Only a small majority preferred my game's range of input, whereas a very large majority preferred the feedback that it gave.

Considering only those who expressed a preference, it is possible to look at the cumulative binomial distribution to check for statistical significance at the 5% level, with $n = 14, p = 0.5$ in each case.

On the range of input: $P(X \geq 8) = 0.395$ which is not statistically significant.

On the quality of feedback: $P(X \geq 11) = 0.029$ and so this is statistically significant.

On the enjoyability: $P(X \geq 9) = 0.212$ which is not statistically significant (though this measure was always less important than the other two).

Firstly, it should be noted that the original proposal called only for a majority to prefer the intelligent engine; statistical significance would always be difficult to obtain with a relatively small sample size. However, the reasons for the narrowness of the preference for my game's range of input should be explored.

Part of this could be explained by the aforementioned issue of some people not realising that the range of instructions was broader than just what was listed in the help menu; another part could be Stanford CoreNLP's unfortunate struggle with non-grammatical English. Those who entered instructions such as `open ancient door with black key` found that CoreNLP treats **open** as an adjective rather than a verb, anticipating that the imperative would take the form `open *the* ancient door with *the* black key`. Thus the input processing module grows confused and returns an error. I was aware of this issue before the user study and the help menu even instructed players to type in grammatical English, and in particular to avoid omitting articles, but this was not always taken to heart.

It is worth noting that even the perceived range of accepted input depends upon exactly how the game is designed. If the designer specifies more actions and their preconditions and results, more input actually *is* accepted and so naturally, more input will be perceived to be accepted.

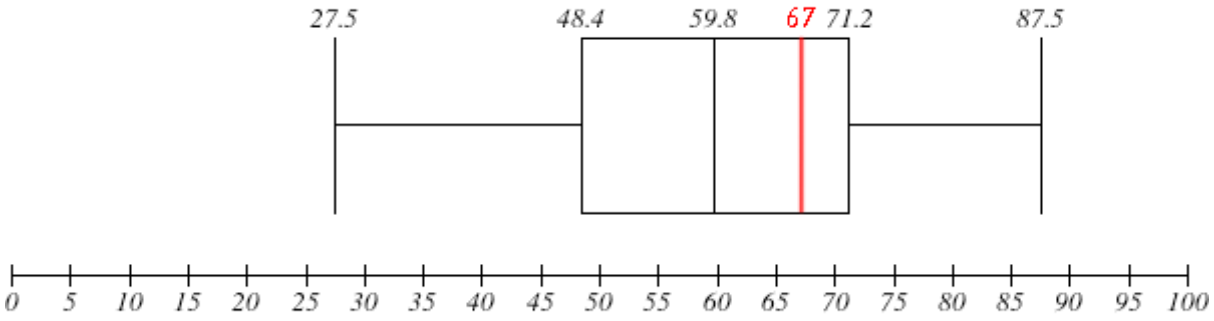
The average SUS score for a software system is **67/100** and so at **59.8/100** my game engine is slightly below average. However, the very high standard deviation of **22.5** (giving a coefficient of variation of 0.376) indicates that this may not be a very reliable estimate. Indeed, the 95% confidence interval for the mean SUS score is

$$59.8 \pm 1.96 * \frac{22.5}{\sqrt{15}} = 59.8 \pm 11.4$$

Giving a range of:

$$[48.4, 71.2]$$

which clearly includes the overall average of 67.



A box plot of the SUS score, with the overall average (67) marked in red. The endpoints denote the extremes of the data set and the box extends for one standard deviation on either side of the mean of 59.8.

Thus, although the project could conceivably have performed better both on the objective System Usability Scale and in comparison with *Colossal Cave Adventure,* it did succeed by the terms set out in the proposal (a majority of participants preferring it).

Given the difficulty of getting to grips with the idea of interactive fiction and most participants' lack of experience with it, the SUS results were somewhat encouraging.

5 Conclusions

This dissertation has laid out the process of designing a more intelligent interactive fiction engine, which tries to move towards the perfect AI (or equivalently, the human game master) by improving upon the efforts of previous interactive fiction engines to recognise and understand input provided by the player, and use this to modify the game world and give feedback to the player.

The project has attempted to accomplish this aim by using state-of-the-art natural language processing in the form of Stanford CoreNLP, and knowledge representation and inference with Prolog.

CoreNLP serves as the basis for an input processing module that converts semi-arbitrary natural language commands into a more uniform format, which is checked against a knowledgebase in Prolog to give an accurate response.

The results of the user study were encouraging: on all three metrics (range of input, quality of response, enjoyability) the game I developed using the new engine is preferred to *Colossal Cave Adventure.* Although the System Usability Scale ranks the interface as slightly below average, there is a large range in the scores; moreover interactive fiction can be daunting, especially when the player only has a few minutes in which to interact with it, and much of the SUS is focused on ease of use. Had funding been available, I would have run a lengthier study, in which participants were invited to play each game for 15 minutes and were properly compensated. This may have given them more time to get used to both systems and thus get a feel for them, and improved their overall impressions.

I feel that I committed to the idea of graph-based knowledge representation too early — even putting it in the project proposal — only to take two weeks to realise that it was not appropriate for the project and that Prolog was a better fit. This was a result of having to make a major decision about implementation in the few days that the project proposal was being written rather than the two weeks allocated to such research in the first phase of the project. I felt pressured by the overseers into including a lot of implementation detail in the proposal, and in retrospect I should have resisted this a little more, saying that such considerations should be left to the research phase.

Later on in the development process, more time should have been spent designing the example game. Reasoning that each player would only have a few minutes to interact with it, I did not want to spend too much effort meticulously planning out scenarios that few users would ever see (only three players managed to reach the final room even in this simple example). However, a traditional interactive fiction game, which is geared towards entertaining a player, will include more side material which does not progress the game but which nevertheless amuses the player. Given that the comparison being made was with such a game, perhaps an increase in the amount of side material would

have improved the players' perception of the game, and would naturally have increased the possible range of input.

Given more time, a range of extensions could have been made. Although part of speech tagging turned out to be an admirably successful basis for parsing user commands, dependency analysis would have been more thorough and rigorous. However, the form in which CoreNLP would tag these dependencies (in a semantic graph) appeared to be too complex to process further.

Some of the functionality of the knowledgebase was held back by limitations on how much of the editor could realistically be developed in the timeframe. For example, the expert system was able to process arbitrary knowledgebase rules to perform inference, but rules only ended up applying to object properties when checking for action success. It is hard to imagine a situation in which a rule determines whether a key can open a door, but it would have been interesting to explore further applications of the expert system.

It would similarly have been interesting to look into improving the experience of the designer by performing input processing on the editor, in the vein of Inform 7 [3]. The issue (other than time) was that this would be very difficult to test except by having users interact with the editor itself, which was always outside the scope of the project. Nonetheless had there been several weeks remaining with the above features all implemented, it would have been informative to attempt this.

Thus although improvements could naturally have been made, and the development process was not quite ideal, I am pleased with the progress that was made. There is a functioning engine with a fairly intuitive editor; significant processing is performed on the input; knowledge is stored in a methodical way and inference is performed upon it; an example game designed using the editor compares favourably to one of the classics of interactive fiction and shows promise for an even more favourable comparison and greater enjoyment by players with a little more work. This is certainly an area that merits more research than an undergraduate dissertation can provide, but I feel that I have provided a proof of concept and a solid starting ground.

Bibliography

1) Montfort, Nick. "Toward a Theory of Interactive Fiction." IF Theory Reader. Boston, MA: Transcript On, 2011. 25-58. Print.
2) "TADS - the Text Adventure Development System, an Interactive Fiction Authoring Tool." TADS - the Text Adventure Development System, an Interactive Fiction Authoring Tool. TADS, 2013. Web. 14 Mar. 2016. <http://www.tads.org/>.
3) Nelson, Graham. "Natural language, semantic analysis, and interactive fiction." IF Theory Reader 141, 2006.
4) Lally, Adam, and Fodor, Paul. "Natural Language Processing With Prolog in the IBM Watson System." Association for Logic Programming. Association for Logic Programming, 31 Mar. 2011. Web. 15 Mar. 2016. <http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/>.
5) Bratko, Ivan. "Expert Systems." Prolog Programming for Artificial Intelligence. Wokingham, England: Addison-Wesley, 1986. 314-58. Print.
6) "JPL A Java Interface to Prolog." JPL A Java Interface to Prolog. n.p., n.d. Web. 15 Mar. 2016. <http://www.swi-prolog.org/packages/jpl/java_api/>.
7) "TuProlog Home." TuProlog @ UniBo. n.p., n.d. Web. 15 Mar. 2016. <http://apice.unibo.it/xwiki/bin/view/Tuprolog/>.
8) Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55-60
9) "System Usability Scale (SUS)." System Usability Scale (SUS). Usability.gov, n.d. Web. 17 Mar. 2016. <http://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>.
10) Jerz, Dennis G. "Somewhere Nearby Is Colossal Cave: Examining Will Crowther's Original "Adventure" in Code and in Kentucky." Digital Humanities Quarterly. Alliance of Digital Humanities Organisations, 2007. Web. 07 May 2016.

Project Proposal

Nakul Khanna, Gonville & Caius College

Directors of Studies: Prof **Peter Robinson,** Dr **Graham Titmus**

Supervised by Dr **Sean Holden**, Computer Laboratory

Description

Interactive fiction was a highly popular genre of computer game through the 1980s, dominated by the now-defunct Infocom, founded at MIT at the beginning of the decade. The player is provided with a text-only interface and must describe their actions through text.

Much of the accepted instruction set had to be manually hard-coded into the game along with the main story and gameplay thread. This would lead to the frequent, immersion-breaking response "sorry, I didn't understand that command" (or words to that effect) whenever the player attempted to do something different.

Significant strides have been made in natural language processing and knowledge representation since the golden era of interactive fiction, and this project will explore the use of these new techniques in allowing a wider set of interactions and a more enjoyable and immersive experience for the player.

The end result of the project will be an engine for the (human) design of interactive fiction games. It will create a game with which a player can interact, and will extend the vocabulary of available actions in this game beyond what is directly specified by the designer.

Thus the two main aspects of the project are the designer interface for creating the game and the experience of the user in playing it. As it is not feasible to have somebody else design a game using my engine in the timeframe, more focus will be put on the player experience than that of the designer.

Starting Point

Many excellent tools and frameworks exist for natural language processing. One such tool is Stanford CoreNLP, a Java framework. Using this makes more sense than reinventing the wheels of part-of-speech tagging and parsing. I intend to code the knowledge representation and inference from scratch, however, as it is more heavily influenced by the setting in which the project takes place – that of an interactive fiction game. In keeping with the language used by CoreNLP, I intend to use Java for the bulk of the project, though some AI processing may be easier in Python; this will be established during the initial research phase.

The CoreNLP API takes in a natural language sentence (or set of sentences) in English and runs a set of analyses: it constructs a parse tree, does part of speech (POS) tagging

and carries out named entity recognition (NER). This data is all returned in JSON format for easy processing.

This analysis allows the nouns and verbs in the natural language input to be identified. The nouns can be looked up in the knowledge representation, and the verbs that the user is attempting to apply to them can be compared with what the game knows about the object in question.

Knowledge representation is frequently done with graph databases. Current knowledge is represented as a set of nodes and edges in the graph and inference is done through graph traversal. However, other techniques for knowledge representation exist and will be explored in the initial research phase.

For example, the instruction "put badger in cage" is parsed as:

```
(ROOT (S (VP (VB Put) (NP (NN badger)) (PP (IN in) (NP (NN cage))))))
```

It can thus easily be determined that "put" is being applied to "badger" and "in cage." The engine can look up what happens when you try to put a badger somewhere, whether something can be put in a cage, and the effects upon a cage and a badger of putting a badger in a cage.

I will use another library, SimpleNLG, for the language generation itself. SimpleNLG allows a grammatical English sentence to be constructed from a provided subject, object and verb with various other customisation possibilities. Inference from the knowledge graph, combined with the sentence structure provided by NLG, should be sufficient to generate a meaningful response.

Evaluation Criteria

Testing will be by means of my designing a game using the engine to emulate the early portions of an open-source (or at least, fully studied and mapped) game from the golden era and have a set of human players play through the two without knowing which is which, and rate both games on how free they felt they were to carry out arbitrary actions, how pleased they were with the responses, and how much they enjoyed the experience.

The project will be considered a success if more than half of players correctly determine that the game designed using my engine allows for more expressiveness, and if more than half view it as more enjoyable than the original and are pleased with the responses they receive.

Alternatively, if a clear majority recognise the greater expressiveness but do not agree that the more expressive game is more enjoyable, we could conclude that the enjoyability of an interactive fiction experience is not strongly linked with the feeling of player freedom.

Structure and Plan of Work

The development will be split into ten packages. Each will consist of roughly 20 hours of work.

12th October 2015 – Initial thoughts and summary

16th October 2015 – Draft proposal submission

23rd October 2015 – Final proposal submission

Package 1 – Initial Research and Reading (6th November 2015)

Familiarise myself with the CoreNLP framework, with looking into and trying out classic interactive fiction and with studying modern techniques for knowledge representation and intelligent inference (focusing on graph-based knowledge representation). This is frequently done with graph databases. Current knowledge is represented as a set of nodes and edges in the graph and inference is done through graph traversal. Other options will also be explored during this phase.

Package 2 – Experiments with NLP and state representation (20th November)

Create a basic program that accepts user input, parses it and prints some conclusions based on what the user has told it, and a small corpus of pre-input data.

Package 3 – The game design interface (4th December)

Create an interface that could be used by a technical but non-expert game designer to create an interactive experience. At this point in the development, the game designer's ability to edit the graph (which specifies a set of entities, their properties and the connections between them) should be in place.

A stretch goal later in the project could involve making this interface more user-friendly, perhaps even carrying out some NLP on it. As specified in my opening, however, the emphasis is on the experience of the player rather than the designer.

Package 4 – Accepting user input (18th December)

The focus here is on getting from the graph created by the designer to a playable experience with which a user can interact.

This is where the NLP comes in: the user's natural language instruction should be interpreted by the engine and related to the system's knowledge. At the basest level, the player should be able to interact with objects in the game in exactly the way specified by the designer.

Additionally, inference should be carried out on what the designer has specified in terms of knowledge, to increase the available range of instructions. This is achieved by traversal of the knowledge graph. This is continued in the next phase.

Package 5 – Returning interesting feedback to the player, and progress report (15th January 2016)

Focus on improving the sense of immersion by increasing the set of understood instructions, and work on how it takes the instructions given to it by the designer, its knowledge of the world, and the arbitrary user input provided, and uses all of these to return informative and interesting feedback to the player.

Write a progress report based on how well I have kept to the schedule outlined in the first five packages of this report.

Package 6 – Designing an example game (29th January)

Identify a suitable early interactive fiction experience, preferably one which is either open source or comprehensively mapped and described, and re-implement the opening portion using my engine.

At this point, the development will be mostly finished. An executable version will be made available over SRCF to be tested by the Overseers and anybody else who may wish to try it.

Package 7 – Human testing and evaluation (12th February)

This will consist of having players try out both the original game and my engine's version of the game (preferably without knowing which is which) and filling in a brief questionnaire:

- Which game allowed you greater freedom of action?
- Which game gave more interesting feedback based on your instructions?
- How would you rate the enjoyability of each game out of ten?

I foresee having ten subjects try out this test. Ideally they will have some interest in interactive fiction but will not be experts in it.

Package 8 – Begin writing up (26th February)

Complete the introduction and implementation sections of the write-up.

Continue carrying out human tests if necessary.

Package 9 – Continue writing up (11th March)

Complete the evaluation section of the write-up based on comments from the subjects.

Package 10 – Finish writing up and submit draft (25th March)

Proofread the dissertation in full and improve grammar, narrative flow and structure. Obtain feedback from my Director of Studies.

Package 11 – Incorporate feedback and submit (22nd April)

Follow the feedback laid out by my Supervisor, have a final read through and submit.