Towards Generating Text-Based Adventure Games Using A Reverse Planning Problem Approach

Final Report

Author: Thomas Tallis
Supervisor: Dr. Richard Booth
Moderator: Professor David Walker

Module Code: CM3203
Module Title: One Semester Individual Project

Credits Due: 40

Table of Contents

Table of Figures

1 Introduction

1.1 Project Aim

The aim of this project is to build the structure of a text-based adventure world, based on a basic game idea inputted by the user, and then develop the world into something more difficult to solve, than the original idea. This aim will be achieved by implementing software that should take, as input, a file detailing the goal state of the game – e.g. the player should end at this location with this item – and any number of sequences of actions that take the player from an unspecified initial state of the game, to the described goal. The program will then generate the initial state of the game, according to the inputted action sequences, and add to the initial state in an attempt to obscure the path to the goal. The initial state, along with the allowable actions and goal state, will then be saved to another file. The structure chosen to record a generated environment is that of a planning problem. This format will allow easy confirmation that the generated world can be solved.

1.2 Text-Based Adventure Games

A text-based adventure game in the context of this report is a game where the player must navigate a world, often interacting with items and non-player characters (NPCs), to reach a predefined goal. This goal may be to reach a particular room in the world, or to obtain a particular item, etc.. The player interacts with the game, via a console, by inputting commands that tell the program what the player wants the controlled person to do. The list of commands depends on the game being played, but every game will have a command to move to another room, usually defined by moving in a compass direction, e.g. “go south” or “move south”. Other commands may involve picking up items or talking to NPCs and some games will include non-functional commands like checking the player’s inventory or looking around the current location.

Examples of existing text-based adventure games include: “Colossal Cave Adventure” [3], the first ever text adventure game, where the aim is to gain points by exploring a cave to find treasures and escaping the cave afterwards; “Adventureland” [4], where the aim is to find thirteen items hidden in the world by exploring rooms and solving puzzles; and “Zork” [5], where the player controls an adventurer exploring the world with no explicit goal. All three games are controlled using the standard commands of “go” followed by a compass direction and have a narrator telling the player how their actions have affected the game through text printed to the console.

A generated text-based adventure world from this project will have the basic move commands and take item commands that are often seen in other games. Other commands will include unlocking doors with a key and lighting dark rooms with a torch. Non-functional commands will not appear in this project as they do not change the state of the game, instead they only provide information to the player. A full list of commands, from here on referred to as actions, can be found in Appendix A at the end of this report.

1.3 Planning Problems

A planning problem is a method of formatting information about a problem in a way that makes it possible for a computer to solve the problem. This formatting is done by defining three distinct pieces of information, described below. The basic data type required for all three pieces of information is a propositional variable, i.e. a variable whose value is either true or false. This data type allows a set of what is true at a particular point in the plan, known as a state, to be created.

The first piece of information defined, using propositional variables, describes the initial state of the plan. This state contains information as to what is true at the start of the problem, and what is false. The second piece of information defined describes the goal state of the plan, again in terms of what must be true, and what must be false, by the end of the plan. The third piece of information contains any actions which allow the planner to move from one state to another. For this transition, two sets are defined for each action. These are the preconditions and the postconditions. The preconditions define the values of any variables that must be fulfilled, in the current state, for the action to take place. The postconditions define what value will be assigned to any variables that change in the next state.

As an example, an action allowing a player to move from one room to an adjacent room must have the preconditions that the two rooms are adjacent, and that the player is in the room from which they will move. The postconditions of such an action will include the player being in the adjacent room, and the player no longer being in the initial room.

A plan, therefore, is a sequence of actions that, when followed starting with the initial state, arrives at the goal state.

What this project aims to do is reverse the process of finding the action sequence that travels from a known initial state to a known goal state. Instead, the software will take a known goal state and any number of action sequences, and return the initial state from which those sequences arrive at the goal.

1.4 Running the Code

To run the produced code, the user must have Python 3 [9] installed on their machine as well as Pygame [10] in their Python installation to be able to create the visualisation of a generated world. The program the user must run is named “generate_world.py” and, to run, it must be passed two arguments on the command line of the file names referring to the input file the user has created, and the name of a file that the generated world will be saved to. An example command is “`python generate_world.py input.txt output.txt`”.

In this example, the code will get the goal and action sequence(s) from a file named “input.txt” and save the full plan that it generates to a file named “output.txt”. During runtime the program will print out a list describing the initial state of the world it generated, display a window showing a visual representation of the generate world, and save a full plan to the file named in the second argument of the command.

2 Background

2.1 Describing Adventure Games with Description Logic

In the paper “Building a Text Adventure on Description Logic” [2], the authors use description logic to improve natural language parsing in text-based adventure games. The basic concepts of such a model are defined using propositional variables and actions are defined as STRIPS actions, which are described in more detail in the next section.

This attempt to resolve natural language commands by modelling actions as STRIPS actions is close to the attempt in this project to represent a game world using a planning problem definitions. Furthermore, an attempt to extend this project by producing a playable game from the generated world could utilise such a model because the data structures are implemented in the same way.

2.2 STRIPS

The Stanford Research Institute Problem Solver (STRIPS) can be used to solve planning problems and also provides a syntax to describe a specific planning problem. More information about the original definitions of STRIPS can be found in [8], from here on this report will discuss STRIPS with regard to an implementation in Python, that can be found on github [1], which forms the basis for the syntax that will be used to create the software. A basic instance of this syntax can be seen in figure 2.1.

```
Initial state: At(Kitchen), NextTo(Kitchen,Office), NextTo(Office,Kitchen)
Goal state: At(Office)

Actions:

Move(From,To)
Preconditions:At(From),!At(To),NextTo(From,To)
Postconditions:At(To),!At(From)
```

Figure 2.1 - An example plan in STRIPS format

As can be seen in the figure, a STRIPS instance in this implementation has three distinct sections. The first section describes all the statements that are true at the start of the plan. The second section describes which statements must be true in order to reach the goal state, but does not have to contain all statements that will be true once the goal state is reached. The final state of this plan, for example, will also contain NextTo(Kitchen,Office) and NextTo(Office,Kitchen), however these statements are not included in the second section, as they are not part of the goal. The third section contains all the actions that may be taken by the planner. In this example the planner can only take one action, and that is to move between rooms. As can be seen in the figure above, an action is defined in such a way that the action can be generalised to more than one set of

propositional variables. In the figure, the move action is specified with two arguments – from and to – that allow the action to take in any pair of rooms rather than just providing the move action between two specific rooms.

A state in STRIPS planning differs slightly from a state in other planning methods as it only contains those variables that are currently true. Any variables that do not appear in the state are considered false. This is not the case in an action however because it is often necessary to say that a variable does not exist in the state – i.e. the variable is false - , or that a variable will become false, and therefore be deleted from the state, once the action has taken place. Therefore, a way to represent that a variable must be false or will become false is required. This representation is usually achieved by preceding the variable with either the word ‘not’ or the negation sign used in propositional logic (¬). In the example above any variable preceded by an exclamation mark is considered false.

2.3 Maze Complexity and Difficulty

The aim of the project inherently requires some measure of how difficult it is to solve a particular game. Research on the topic of how difficult mazes are to solve seemed an appropriate place to start as the set of rooms in the game can be considered a maze.

The research revealed a paper on this topic [6] and a simplified version of the same calculations applied to rectangular mazes [7]. Of the two options, the simplified version seemed more relevant initially as it discussed only rectangular mazes, which is what the typical adventure game world looks like in practice. Further inspection of the paper, however, revealed that the particular measure it discussed related to the complexity of a maze, defined in terms of the lengths of hallways in a maze. This definition of complexity, however, is not what is required of the measure in this project, because it does not reflect the difficulty to solve the maze, only the difficulty of describing the maze. An example given in the paper in figure 6 shows three mazes that have increasingly higher complexity than the previous one, due to an increase in the length of the hallway, but each only has one path through the maze, and is therefore just as easy to solve. The original paper, on the topic of difficulty and complexity of mazes, appears more useful as it discusses a measure of difficulty that is more applicable to the requirements of the project. However, an implementation of the required formula to calculate the difficulty rating would be time-consuming to achieve, and a simpler method would allow for a reasonable estimate of what would be calculated while allowing more time to be spent elsewhere on the code.

3 Design and Implementation

Designing the software required some way of representing the basic data types that would be used throughout the runtime of the program. The representation that has been chosen is the same as the representation used in the Python implementation of a STRIPS planner [1]. This compatibility allows the use of the STRIPS planner to verify the generated plan is solvable. Within the program, a propositional variable is stored as a tuple of information. The first item in the tuple is the identifier of the variable, e.g. “At”. This is followed by the items that come inside of the brackets in the standard representation that can be seen in figure 2.1. For example, the propositional variable At(Kitchen) would be stored in the program as (“At”, “Kitchen”). To represent a variable as false, that variable’s identifier is preceded by an exclamation mark, e.g. (“!At”, “Kitchen”).

From this definition of a propositional variable, it is possible to implement a state as a set of tuples representing all the variables that are true at that point in time. An action in the code is represented as three tuples. The first defines the name of the action and any arguments that are passed to the action to generalise the input. The second contains all the tuples of propositional variables that make up the preconditions of the action, with reference to the arguments of the action where necessary. The third defines the postconditions of the action in the same way as the preconditions are defined. An example of theses tuples can be seen in figure 3.1, below.

```
(("MoveEast", "From", "To"),
 (("At","From"),("!At","To"),("NextTo","From","To"),("NextTo","To","From")),
 (("At","To"),("!At","From"))
)
```

Figure 3.1 - An example action as stored in the code

3.1 Receiving and Interpreting the Inputted Action Sequence

The first step in creating the software was to build a function that could interpret the input from the user. To make it possible for the program to interpret the input from the user, a set of rules was created to define how the text file would be formatted. The rules are as follows:

- The first line in the text file is the goal state of the game in the following format:
  - It is represented as a list of variables separated by a comma and a space
  - each variable is represented as an identifier then, without spaces, a comma separated list of inputs to the variable enclosed by brackets touching the identifier.
  - Example:
    - At(Library), Has(Key)

- Each following line in the text file is an action sequence that ends at the goal state and must be of the same format as the goal state, except each item is an action and not a variable.
    - Example:
        - Take(Key), Unlock(Kitchen,Library), Move(Kitchen,Library)

3.2 Generating the Initial State

To generate the initial state, from the inputted goal and action sequence(s), a reverse planning problem approach is used. Specifically, the algorithm takes in the sequence(s) of actions and the goal state, and reverses the process to find the initial state that arrives at the goal state when the sequence of actions is taken. This reversal is achieved by performing the sequence of actions in reverse order, with the preconditions and postconditions swapped, i.e. for the last action in the original sequence (which will be applied first) to be taken, all the postconditions must exist in the goal state and, after the action is taken, all the preconditions will be added to make the next state. This process is repeated on the current state until all the actions have been applied and the initial state is produced.

To find the associated preconditions and postconditions of an action in the sequence, the program performs a dictionary lookup using the action name as the key. Any variable propositions that exist in the action are then substituted for the actual value that has been passed with that variable name in the action sequence. For example, the action “Take()” requires the name of the item being taken and the room from which the item is being taken. The preconditions of Take in the program are: “(At,Room), (In,Item,Room),(!Has,Item)” and, in generating the initial state, “Item” and “Room” are replaced with whatever item is being picked up and from where, e.g. a key from the kitchen.

The post conditions of the next applicable action can then be used to check that the action took place by comparing the postconditions of an action to the current state. If any postconditions in the action conflict with what is in the current state, the action sequence is deemed invalid and a SequenceError is thrown. These conflicts are: the postconditions state that the player is currently in a different location to where the player is in the state; and the postconditions contain a false propositional variable that is currently true in the state.

Once the program confirms that the action took place, it adds the preconditions of the action to the current state. This is done by adding any true propositional variables in the preconditions to the state, ignoring any duplicates, and removing any true propositional variables in the state that are false in the preconditions.

In some cases, the user will want to include multiple paths that lead to the same goal. The process that takes place when the user inputs multiple sequences is to apply each action sequence individually to a copy of the original goal state, using the same method as for one action sequence. The resulting initial states are then combined together, ignoring any duplicate propositional

variables, to create the initial state from which all sequences lead to the goal state. The states can be combined in this way without any conflicting variables because, for all sequences to lead to the goal state, there must be no conflicts in the way they describe the world. For example, one sequence may say that the library is connected to the kitchen, and another may say that the library is connected to the office, but they will not say that they are connected in the same direction as, if they did, this would become a conflict and both sequences would not be able to reach the goal state. If any conflicts do arise because of an error in one of the action sequences, the program will throw a SequenceMergeError with some text describing why the error was thrown. If the example of two rooms connected in the same direction, or a related conflict of the same room connected in the same direction to two different rooms does occur, the program will throw an error stating “Invalid grid structure of the rooms”. This error statement will also be printed if two sequences follow a path from the same room in different directions and reach the same endpoint. As an example of when this final case occurs, if one action sequence says that by going north twice, from the start location, the player arrives at the goal, and another says that by east twice, from the start location, the player arrives at the goal, this error will be accompanied by the text: “Invalid grid structure of the rooms.”. The final time an error will be thrown is if at least one of the action sequences has a starting location that differs from a starting location in another action sequence. In this case, the accompanying text will say “at least two action sequences start in different places”.

3.3 Formatting the Output

After the initial state is generated, it needs to be saved to file in a format that could be used, in future development, to build the game. The format chosen for saving the information was the same as what is required by the Python STRIPS program [1]. This format was chosen because it could then be immediately inputted into the STRIPS program to test the generated initial state had a valid path to the goal state.

The initial state, goal state and set of used actions need to be outputted to the file in a particular format. For compatibility with the STRIPS program, the initial and goal states must be outputted on one line each, preceded by the words “initial state:” and “goal state:” respectively. Each variable is comma separated with no whitespace and the items inside the variables are also comma separated with no whitespace.

The actions are outputted on 3 lines each, separated by a blank line, with each action having a line for the identifier and any variable item references; a line for all the preconditions of the action, preceded by the word “preconditions:”; and a line for all the postconditions of the action, preceded by the word “postconditions:”.

An example of a simple world represented as the outputted planning problem can be seen found above in section 2.2, figure 2.1.

3.4 Difficulty Measurements

The difficulty measurement of the generated world can be split into two categories: a measurement of the difficulty based on the arrangement of the rooms in the world; and a

measurement of the difficulty based on the number of items in the world. This distinction was made because the program will increase the difficulty associated with the rooms and the difficulty associated with the items separately, so a separate measure of difficulty is required for the program to know when to stop in each case. To get an overall difficulty measure, the two measurements are added together.

3.4.1 Room Difficulty Measurement

To confirm that any measurement devised actually represented the difficulty of a particular set of rooms, three basic room maps were created that could be ranked, without using the difficulty measure, as to which would be more difficult to solve. This ranking could then be compared to the scores that the measurement assigns to each map to decide if the measurement works in practice. The below figure shows the three examples with difficulty increasing from left to right.

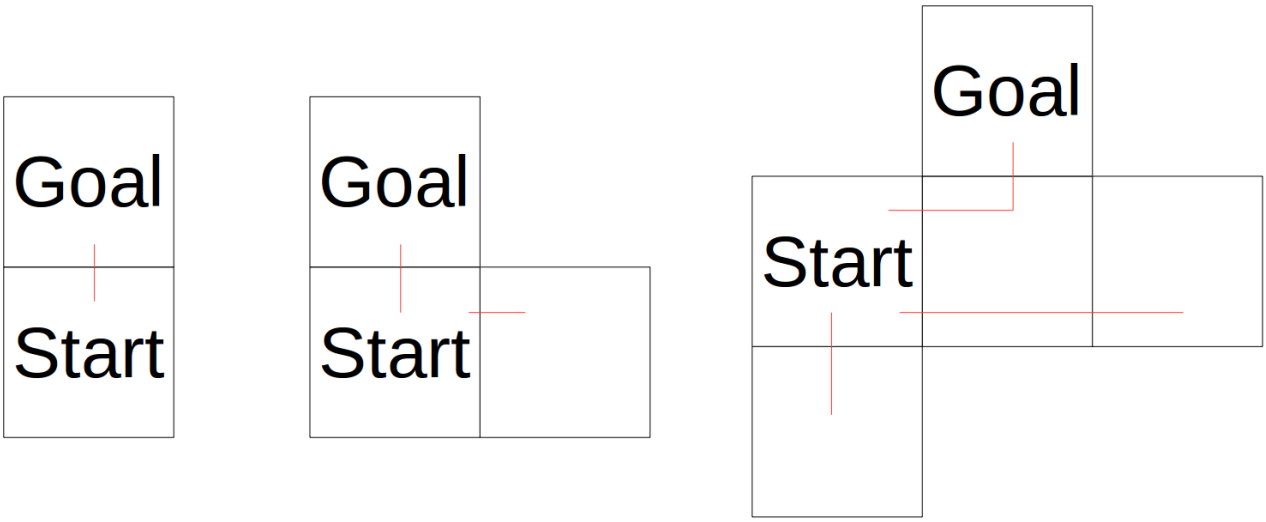

Figure 3.2 - Three simple examples of room maps

The red lines on the maps follow unique paths from the start room to either dead ends or the goal room. With these examples ranked, it should become clear that an increase in the number of paths, leads to an increase in the difficulty measure of a map. This relationship occurs because a greater number of paths in the map means that there are more decisions that must be made to find a path that leads to the solution. An increase in the length of a path, without increasing the number of paths, should not, therefore, increase the difficulty measure. As an example, adding a room between the start and the goal (with no additional exits) in the first example in figure 3.2 would not increase the number of paths, and thus does not increase the difficulty associated with reaching the goal.

While counting the paths of the example maps is trivial enough, when a map becomes sufficiently complex, an algorithm for counting the number of paths becomes infeasible to implement within a reasonable time limit. Therefore, a simpler measure must be found that

emulates a count of the number of paths. An approach to this method could be based on the number of neighbours of each room in the world. This approach should approximate the number of paths because each neighbour of a room should increase the number of paths.

A formula can now be devised for the difficulty measure, based on the number of neighbours of each room in the world. An initial idea proposed that it should simply be the sum of the number of neighbours of each room. This can be seen in the below formula where D_r represents the difficulty of the room-based component, and n_i represents the number of neighbours at a given room i.

$$D_r = \sum(n_i)$$

This idea, however, would result in a greater value when a path is lengthened, by adding a room to either the middle or the end of an existing path, without creating a new path. With this in mind a new formula was created that ignores any room with two or fewer neighbours.

$$D_r = n_1 + \sum(n_i - 2)$$

In this approach, the first term (n_1) is the number of neighbours of the start room. The second term is a sum of all the remaining room's number of neighbours with a reduction of two for each count, with the caveat that any negative terms in the formula become 0 to avoid reducing the difficulty measure when a dead-end is added. This formula more accurately describes the difficulty of solving a particular arrangement of rooms as it will count the, and gives the desired ranking for the examples in figure 3.3. The actual values for the difficulty measure of the examples are, from left to right, 1, 2 and 3. These values align with the number of paths for each of the examples, and the formula should predict the number of paths from the start room to a dead end or goal room for any map that has a single, unique path to each dead end or the goal room. However, a map that has more than one unique path leading to a dead end will have a higher path count than the measured difficulty value. This is illustrated in the figure below where the map on the left, that has a single unique path to each dead end, has 4 unique paths and a measured difficulty value of 4. Where as, the map on the right, that has multiple paths leading to the goal room , still has a difficulty measure of 4 but a path count of 5.

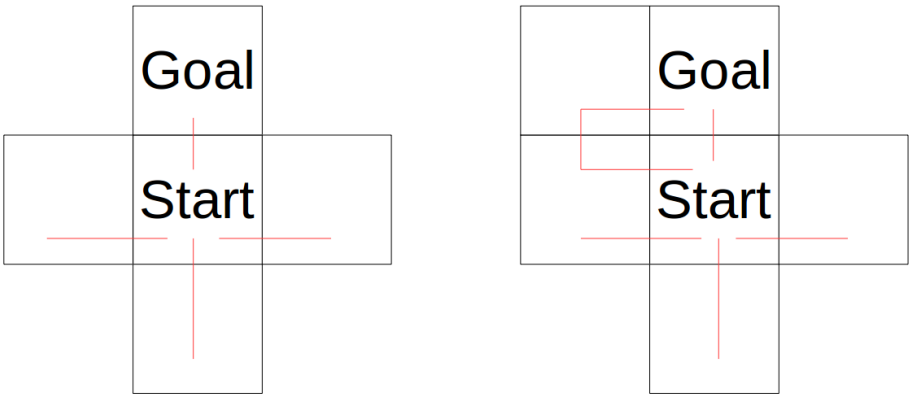

Figure 3.3 – A comparison of two maps with the same measured difficulty

3.4.2 Item Difficulty Measurement

Quantifying the difficulty associated with the items in the world was simpler than for the rooms. It was based on the number of items in the world. Another addition to this measure was a count of anything that the items would allow the player to pass, from here on referred to as a blockage. The current blockages in the world are: locked doors that can be unlocked with a key; hidden passages that can be revealed by talking to an NPC; and dark rooms that can be lit by a torch.

An initial approach to this measurement was to total the number of items and their associated blockages, as shown by the formula below where D_i represents the item component of the difficulty measure, n_i represents the number of items in the world, and n_b represents the number of blockages in the world.

$$D_i = n_i + n_b$$

This total provides a rough estimate of how hard solving a world would be as it should be based on the number of puzzles that need to be solved. However, on further consideration, the measure was changed to be based on the number of blockages only, given that the actual item should not be considered part of the difficulty because any item found by the player while travelling around the rooms would be immediately picked up. Therefore, finding the item is only difficult because of the blockage it unlocks not because of the item itself.

Therefore, the final measurement used was just to total the number of blockages in the world, shown below.

$$D_i = n_b$$

This works as the greater the number of blockages in the world, the higher the difficulty associated with solving the world, as the solution is obscured by the need to find the items for the blockages.

3.5 Increasing the Room-Based Difficulty Measure

The next algorithm to be implemented was one that would increase the difficulty of the original initial state by adding rooms that would neighbour the existing rooms. The first version of this algorithm was implemented in such a way that any room could become connected to any other room, provided that neither of the two rooms that were to become connected did not already have four neighbours. This method was initially acceptable because it was quick to implement and served the purpose of increasing the difficulty. The main disadvantage of this implementation, however, was a difficulty in accurately producing a visual representation of the game’s map. Such a view was implemented later on to provide a clearer overview of how the rooms connected to each other. This implementation also meant that moving in a compass direction, as most previous text-based adventure games implement, becomes meaningless as even though a player moves east from a room, they may end up west of the room.

To take into account these considerations, a redesign of the method for connecting rooms together was performed. This new method maps out the rooms in a grid structure, maintaining that each room had at most four neighbours and that a new room would not join two rooms that were separated by more than one block in the grid.

Adding extra rooms required a way of generating a new room name that was guaranteed to be unique. This was achieved by calling the new room “Room” followed by a number that was higher than any room added previously. As an example, the first room to be added would most likely be “Room1”. Ensuring that the room number was higher required looking through all the room names for any that were formatted the same way – i.e. “Room” followed by a number – and work out which was the highest number, then increment it. This search was achieved with a regular expression - `^Room[1-9]{1}[0-9]*$` - that would separate all the rooms of the required format from the rooms with a different name format. Once the next available number is found, it is stored in a variable and incremented every time a new room is added.

Once generating a new room name was achieved, the problem of selecting a room’s neighbours could be addressed. In the first implementation this selection only required randomly selecting one to four rooms from a list of rooms that did not currently have four neighbours. In the later edition of the algorithm, a new room would be added to a position in the grid, such that it was next to at least one existing room, and then connected to neighbours by which it was surrounded. This design required some way of keeping track of where a room was in the grid, which could be achieved with a coordinate system. This system was implemented by assigning the room in which the player would start the coordinate (0,0) and then all rooms after would be assigned a coordinate based on the direction of their attachment to a room that had already been given a coordinate. This step also required redefining a pair of neighbouring rooms to include the direction the player would have to take to move from the first room to the second. For example, the original implementation might define a pair of neighbouring rooms as NextTo(“Office”,“Library”), while in the later edition of the algorithm this definition would be NextTo(“Office”, “Library”, “east”), specifying that to go from the office to the library, the player must go east.

3.6 Increasing the Item-Based Difficulty Measure

The difficulty of solving the game could also be increased by adding more blockages into the game for the player to get around by finding the required item. Adding blockages has been achieved by implementing a function that locks a certain number of doors depending on a variable passed to the function. There is also a similar function that makes a certain number of rooms dark based on a variable. However, these two functions only work if the player already has the associated item to unblock the blockage. This limitation is because there is no implementation to ensure that, if a new item were to be added alongside an associated blockage, the blockage does not block the player from reaching the item. The current implementation means that, even though the measured difficulty of the world in terms of items increases, the actual difficulty to solve the game does not increase.

The locking doors function is implemented to take in a variable percentage that it uses to determine how many of the unlocked doors will become locked. This number of rooms is calculated based on the “NextTo()” propositional variables in the initial state. The function then randomly selects that number of unlocked doors from the list of “NextTo()” variables and then adds to the state a “Lock()” proposition for each chosen door with the rooms from the “NextTo()” as arguments.

3.7 Displaying the Rooms of the World

Adding a grid structure to the world to allow for better placement of new rooms also opened the possibility of easily creating a display of what the world looked like, which would make it simpler for a user of the program to visualise where the rooms were in the generated world.

A visualisation was achieved by using the coordinate system which was introduced in section 3.5. From there, a grid could be generated by finding the furthest x and y values from the origin. Then, a room could be mapped to its section in the grid by translating the coordinate to a coordinate of the screen. To indicate which rooms were connected to each other, a line was drawn through the grid-lines between the neighbours. The final step was to black out any section in the grid in which there was no room. The below figure shows an example of a generated world represented in this way.

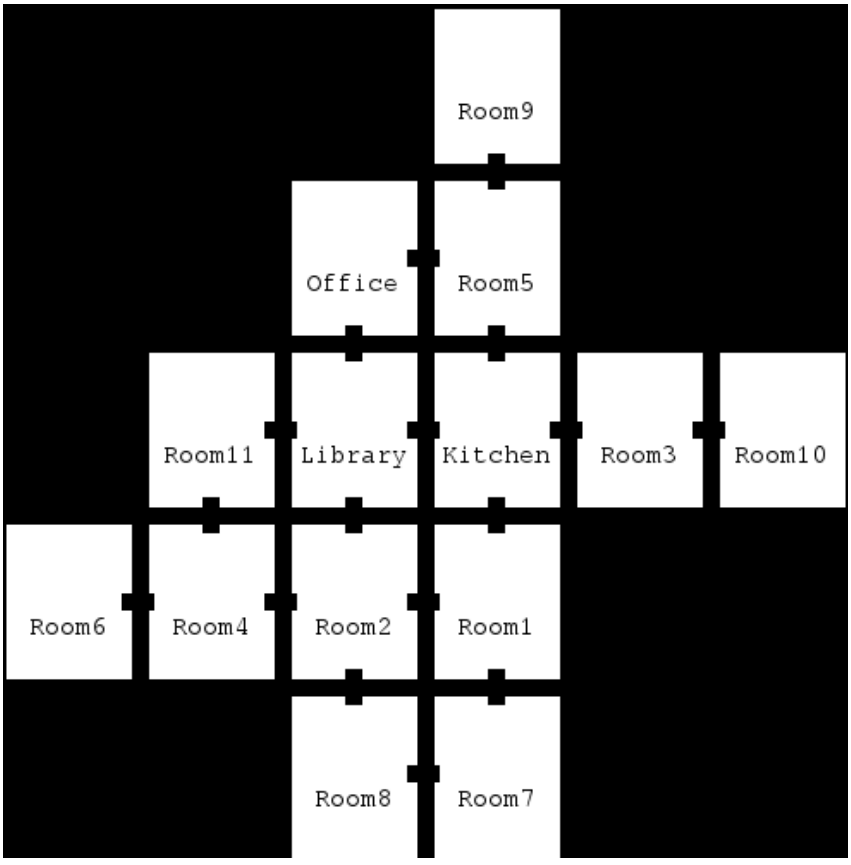

Figure 3.4 - A visualisation of a generated map

4 Results and Evaluation

4.1 Testing the Software

To test that the program passes several requirements, inputs were created that would test all cases in each requirement.

Case 1: Basic input

This case is the most basic, in which the input is a simple goal state, which requires the player to be at a location, and a single action sequence moving from room to room with no item requirements. This case will be used in each test to ensure that the program is able to deal with the most basic requirement of the project.

```
At(Kitchen)
MoveNorth(Office,Library), MoveEast(Library,Kitchen)
```

Figure 4.1 - Test case 1

Case 2: Input contains multiple action sequences

In this case the input contains two action sequences that lead to the same goal state as in the first case. Neither action sequence requires an item. This case will test the programs ability to combine two initial states into one, and should pass without any errors due to conflicts in the sequences being thrown.

```
At(Kitchen)
MoveNorth(Office,Library), MoveEast(Library,Kitchen)
MoveEast(Office,Bedroom), MoveNorth(Bedroom,Kitchen)
```

Figure 4.2 - Test case 2

Case 3: The action sequence requires an item the player does not hold

In this case the input contains a single action sequence that leads to the same basic goal state. It differs from the previous cases by requiring a door to be unlocked to reach the goal state. The key for the door is not in the player's inventory. This case will test the actions that allow the player to pick an item up and unlock a door.

```
At(Kitchen)
MoveNorth(Office,Library), Take(Key,Library), Unlock(Library,Kitchen,east), MoveEast(Library,Kitchen)
```

Figure 4.3 - Test case 3

Case 4: The action sequence requires an item that the player already holds

In this case the input is identical to case 3 except the key is now already in the player's inventory. This case will test the program's ability to be able to deal with the player not needing to pick up an item. It will also be used to test the item-based component of the difficulty measure.

```
At(Kitchen)
MoveNorth(Office,Library), Unlock(Library,Kitchen,east), MoveEast(Library,Kitchen)
```

Figure 4.4 - Test case 4

Case 5: The goal state of the plan is to have an item

In this case the goal state is modified such that the player is no longer required to be in a certain location, but to have a specific item. This will test whether the program is capable of backtracking through the action sequence, without initially knowing where the player is located.

```
Has(Key)
MoveNorth(Office,Library), MoveEast(Library,Kitchen), Take(Key,Kitchen)
```

Figure 4.5 - Test case 5

Case 6: Single invalid action sequence

The aim of this case is to test that the program can recognise when the input is invalid. In this case, the goal will be the same as case 1, but the action sequence will not lead to the goal. When run this input should throw an "invalid action sequence." error.

```
At(Kitchen)
MoveNorth(Office,Library), MoveEast(Library,Bedroom)
```

Figure 4.6 - Test case 6

Case 7: Input contains multiple action sequences that are invalid

The aim of this case is to test that the program can recognise when the combination of multiple action sequences is invalid. It is split into two parts to test the two situations under which errors are thrown. Both situations contain two action sequences that, when on their own, would be valid sequences, but when they are together are invalid.

In case 7a, the input is invalid because each sequence describes a path from the start location that leads in a different direction to the other, but both paths reach the same location. When run, this input should throw an "Invalid grid structure." error.

```
At(Kitchen)
MoveNorth(Office,Library), MoveNorth(Library,Kitchen)
MoveEast(Office,Bedroom), MoveEast(Bedroom,Kitchen)
```

Figure 4.7 - Test case 7a

In case 7b, the input is invalid because the two action sequences start in a different location.When run, this input should throw a SequenceMergeError which states “At least two sequences start in different locations."

```
At(Kitchen)
MoveNorth(Office,Kitchen)
MoveEast(Bedroom,Kitchen)
```

Figure 4.8 - Test case 7b

4.1.1 The Program Can Generate the Initial State Based on User Input

This test will be performed on all the test cases and will ensure that the program can take in a variety of inputs and still produce the initial state that leads to the goal state defined by the user. This test will also confirm, where possible, that the plan outputted by the program can be solved using the Python implementation of STRIPS [1]. This should prove that the planning problem that is generated by the program is valid and that the sequence required to reach the goal state is the same as the sequence given by the user.

As can be seen in the below screenshots, all cases performed as expected. Cases 1 through 5 all produced an initial state that matches the information provided in it’s input. They also all produce valid plans that are solvable using at least one of the given action sequences. Cases 6, 7a and 7b all failed to produce a plan and threw the expected errors. Therefore, all tests were passed, shoing tha the program can generate the initial state based on user input, or print a relevant message if there is an error.

Case 1

```
Initial State:
[('NextTo', 'Kitchen', 'Library', 'west'), ('NextTo', 'Library', 'Office', 'south')
, ('NextTo', 'Office', 'Library', 'north'), ('At', 'Office'), ('NextTo', 'Library',
 'Kitchen', 'east')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('MoveEast', 'Library', 'Kitchen')]

Testing file "testcases/output.txt" using strips:

Goal already solved? False
Solving...
Solved!
Plan: MoveNorth(Office, Library) -> MoveEast(Library, Kitchen)
```

Figure 4.9 - Test 1, case 1

Case 2

```
Initial State:
[('NextTo', 'Kitchen', 'Library', 'west'), ('NextTo', 'Office', 'Bedroom', 'east'),
 ('NextTo', 'Office', 'Library', 'north'), ('At', 'Office'), ('NextTo', 'Bedroom',
'Office', 'west'), ('NextTo', 'Library', 'Office', 'south'), ('NextTo', 'Bedroom',
'Kitchen', 'north'), ('NextTo', 'Library', 'Kitchen', 'east'), ('NextTo', 'Kitchen'
, 'Bedroom', 'south')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('MoveEast', 'Library', 'Kitchen')]
[('MoveEast', 'Office', 'Bedroom'), ('MoveNorth', 'Bedroom', 'Kitchen')]

Testing file "testcases/output.txt" using strips:

Goal already solved? False
Solving...
Solved!
Plan: MoveNorth(Office, Library) -> MoveEast(Library, Kitchen)
```

Figure 4.10 - Test 1, case 2

Case 3

```
Initial State:
[('At', 'Office'), ('NextTo', 'Library', 'Kitchen', 'east'), ('NextTo', 'Library', 'Office',
 'south'), ('NextTo', 'Office', 'Library', 'north'), ('Lock', 'Library', 'Kitchen'), ('In',
'Key', 'Library'), ('NextTo', 'Kitchen', 'Library', 'west'), ('Lock', 'Kitchen', 'Library')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('Take', 'Key', 'Library'), ('Unlock', 'Library', 'Kitc
hen', 'east'), ('MoveEast', 'Library', 'Kitchen')]

Testing file "testcases/output.txt" using strips:

Goal already solved? False
Solving...
Solved!
Plan: MoveNorth(Office, Library) -> Take(Key, Library) -> Unlock(Library, Kitchen, east) ->
MoveEast(Library, Kitchen)
```

Figure 4.11 - Test 1, case 3

Case 4

```
Initial State:
[('Lock', 'Library', 'Bedroom'), ('NextTo', 'Study', 'Bedroom', 'south'), ('Lock', 'Bedroom',
 'Library'), ('At', 'Office'), ('NextTo', 'Study', 'Kitchen', 'north'), ('NextTo', 'Kitchen',
 'Study', 'south'), ('NextTo', 'Library', 'Bedroom', 'east'), ('NextTo', 'Library', 'Office',
 'south'), ('NextTo', 'Bedroom', 'Library', 'west'), ('NextTo', 'Office', 'Library', 'north')
, ('NextTo', 'Bedroom', 'Study', 'north'), ('Has', 'Key')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('Unlock', 'Library', 'Bedroom', 'east'), ('MoveEast', '
Library', 'Bedroom'), ('MoveNorth', 'Bedroom', 'Study'), ('MoveNorth', 'Study', 'Kitchen')]

Testing file "testcases/output.txt" using strips:

Goal already solved? False
Solving...
Solved!
Plan: MoveNorth(Office, Library) -> Unlock(Library, Bedroom, east) -> MoveEast(Library, Bedro
om) -> MoveNorth(Bedroom, Study) -> MoveNorth(Study, Kitchen)
```

Figure 4.12 - Test 1, case 4

Case 5

```
Initial state of plan:

Initial State:
[('NextTo', 'Library', 'Office', 'south'), ('At', 'Office'), ('NextTo', 'Library', '
Kitchen', 'east'), ('NextTo', 'Office', 'Library', 'north'), ('In', 'Key', 'Kitchen'
), ('NextTo', 'Kitchen', 'Library', 'west')]
Goal State:
{('Has', 'Key')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('MoveEast', 'Library', 'Kitchen'), ('Take', 'K
ey', 'Kitchen')]

Testing file "testcases/output.txt" using strips:

Goal already solved? False
Solving...
Solved!
Plan: MoveNorth(Office, Library) -> MoveEast(Library, Kitchen) -> Take(Key, Kitchen)
```

Figure 4.13 - Test 1, case 5

Case 6

```
Traceback (most recent call last):
  File "input_plan.py", line 123, in <module>
    initial_states.append(backtrack(goal_state,sequence))
  File "input_plan.py", line 75, in backtrack
    raise SequenceError("action sequence is not valid.")
exceptions.SequenceError: 'action sequence is not valid.'
```

Figure 4.14 - Test 1, case 6

Case 7a

```
Traceback (most recent call last):
  File "input_plan.py", line 137, in <module>
    initial_state = combine_states(initial_states)
  File "input_plan.py", line 110, in combine_states
    raise SequenceMergeError("Invalid grid structure of rooms.")
exceptions.SequenceMergeError: 'Invalid grid structure of rooms.'
```

Figure 4.15 - Test 1, case 7a

Case 7b

```
Traceback (most recent call last):
  File "input_plan.py", line 137, in <module>
    initial_state = combine_states(initial_states)
  File "input_plan.py", line 101, in combine_states
    raise SequenceMergeError("At least two sequences start in different locations.")
exceptions.SequenceMergeError: 'At least two sequences start in different locations.'
```

Figure 4.16 - Test 1, case 7b

4.1.2 The Program Can Increase the Room-Based Difficulty

In this test, the only case that will be used is case 1 because, past the point of generating the initial state of the input, the program does not take into account any differences in the goal state or sequences and as such, proving this algorithm works for one case is sufficient to show it works for all cases.

The figures below show a contrast between what the map looks like without adding rooms, and what happens when rooms are added. As can be seen in the figures, when the algorithm is applied, it does add more rooms to the map and connects the added rooms to adjacent rooms. This in turn raises the rooms component of the difficulty measure from 1, in the first figure, to 10 in the second figure.

Case 1

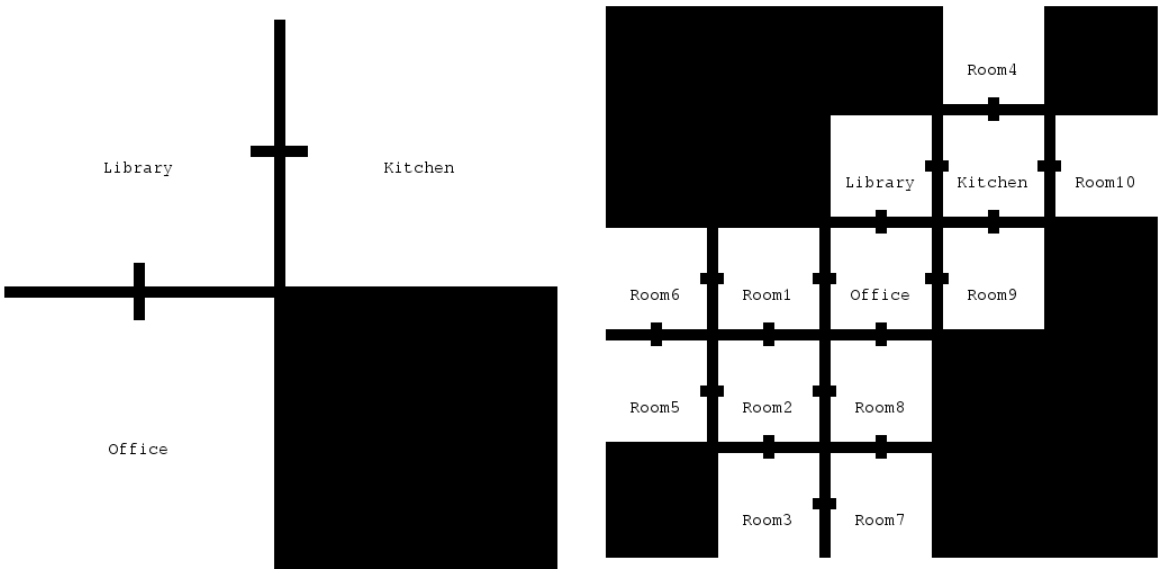

Figure 4.17 – Test 2, case 1, before and after adding rooms respectively

4.1.3 The Program Can Increase the Item-Based Difficulty

This final test will use cases 1 and 4 to show two possibilities of the algorithm that increases the item component of the difficulty measure. The first possibility is that no change occurs to a case where no items are held by the player at the start. The second possibility is that, in a case where the player holds a key, a function will be used that will lock a percentage of the not locked doors.

In both cases, the world will have rooms added first and then the program will attempt to increase the item-based difficulty. As can be seen in the images below for case 1, apart from a slight re-ordering of the initial generated propositional variables between the two figures as a result of running the program twice, the only change in the initial state output is the increase in the number of “NextTo” propositions that means the room-adding algorithm has worked. This means that, as expected, the algorithm has failed to add any blockages to the world. This failure is because the player does not hold the item required to pass the blockage. In the set of figures relating to case 4, it

can be seen that the algorithm has succeeded in adding locks to the doors. This addition is shown by the propositional variables found at the end of the state that define there are locks between some of the doors, and this is down to the fact that the player already holds the key.

Case 1

```
Initial State:
[('NextTo', 'Library', 'Office', 'south'), ('NextTo', 'Library', 'Kitchen', 'east'),
 ('NextTo', 'Office', 'Library', 'north'), ('NextTo', 'Kitchen', 'Library', 'west'),
 ('At', 'Office')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('MoveEast', 'Library', 'Kitchen')]
```

Figure 4.18 - Test 3, case 1, before adding items

```
Initial State:
[('At', 'Office'), ('NextTo', 'Library', 'Office', 'south'), ('NextTo', 'Kitchen', 'Library', 'west'),
('NextTo', 'Library', 'Kitchen', 'east'), ('NextTo', 'Office', 'Library', 'north'), ('NextTo', 'Room1',
 'Library', 'east'), ('NextTo', 'Library', 'Room1', 'west'), ('NextTo', 'Room2', 'Room1', 'east'), ('Ne
xtTo', 'Room1', 'Room2', 'west'), ('NextTo', 'Room3', 'Room2', 'east'), ('NextTo', 'Room2', 'Room3', 'w
est'), ('NextTo', 'Room4', 'Room2', 'south'), ('NextTo', 'Room2', 'Room4', 'north'), ('NextTo', 'Room5'
, 'Room2', 'north'), ('NextTo', 'Room2', 'Room5', 'south'), ('NextTo', 'Room6', 'Room1', 'north'), ('Ne
xtTo', 'Room1', 'Room6', 'south'), ('NextTo', 'Room6', 'Office', 'east'), ('NextTo', 'Office', 'Room6',
 'west'), ('NextTo', 'Room6', 'Room5', 'west'), ('NextTo', 'Room5', 'Room6', 'east'), ('NextTo', 'Room7
', 'Room6', 'north'), ('NextTo', 'Room6', 'Room7', 'south'), ('NextTo', 'Room8', 'Room5', 'east'), ('Ne
xtTo', 'Room5', 'Room8', 'west'), ('NextTo', 'Room8', 'Room3', 'north'), ('NextTo', 'Room3', 'Room8', '
south'), ('NextTo', 'Room9', 'Room4', 'east'), ('NextTo', 'Room4', 'Room9', 'west'), ('NextTo', 'Room9'
, 'Room3', 'south'), ('NextTo', 'Room3', 'Room9', 'north')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('MoveEast', 'Library', 'Kitchen')]
```

Figure 4.19 - Test 3, case 1, after adding items

Case 4

```
Initial State:
[('Lock', 'Bedroom', 'Library'), ('NextTo', 'Kitchen', 'Study', 'south'), ('NextTo', 'Bedroom
', 'Study', 'north'), ('At', 'Office'), ('Lock', 'Library', 'Bedroom'), ('NextTo', 'Library',
 'Bedroom', 'east'), ('NextTo', 'Office', 'Library', 'north'), ('NextTo', 'Bedroom', 'Library
', 'west'), ('NextTo', 'Study', 'Kitchen', 'north'), ('NextTo', 'Study', 'Bedroom', 'south'),
 ('Has', 'Key'), ('NextTo', 'Library', 'Office', 'south')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('Unlock', 'Library', 'Bedroom', 'east'), ('MoveEast', '
Library', 'Bedroom'), ('MoveNorth', 'Bedroom', 'Study'), ('MoveNorth', 'Study', 'Kitchen')]
```

Figure 4.20 - Test 3, case 4, before adding items

```
Initial State:
[('NextTo', 'Study', 'Kitchen', 'north'), ('NextTo', 'Kitchen', 'Study', 'south'), ('NextTo', 'Study',
 'Bedroom', 'south'), ('NextTo', 'Bedroom', 'Study', 'north'), ('NextTo', 'Office', 'Library', 'north'
), ('Lock', 'Bedroom', 'Library'), ('NextTo', 'Library', 'Office', 'south'), ('NextTo', 'Library', 'Be
droom', 'east'), ('At', 'Office'), ('NextTo', 'Bedroom', 'Library', 'west'), ('Has', 'Key'), ('Lock',
'Library', 'Bedroom'), ('NextTo', 'Room1', 'Office', 'north'), ('NextTo', 'Office', 'Room1', 'south'),
 ('NextTo', 'Room2', 'Room1', 'east'), ('NextTo', 'Room1', 'Room2', 'west'), ('NextTo', 'Room3', 'Kitc
hen', 'south'), ('NextTo', 'Kitchen', 'Room3', 'north'), ('NextTo', 'Room4', 'Room3', 'west'), ('NextT
o', 'Room3', 'Room4', 'east'), ('NextTo', 'Room5', 'Room4', 'south'), ('NextTo', 'Room4', 'Room5', 'no
rth'), ('NextTo', 'Room6', 'Room2', 'north'), ('NextTo', 'Room2', 'Room6', 'south'), ('NextTo', 'Room7
', 'Room2', 'south'), ('NextTo', 'Room2', 'Room7', 'north'), ('NextTo', 'Room7', 'Office', 'east'), ('
NextTo', 'Office', 'Room7', 'west'), ('NextTo', 'Room8', 'Library', 'east'), ('NextTo', 'Library', 'Ro
om8', 'west'), ('NextTo', 'Room8', 'Room7', 'south'), ('NextTo', 'Room7', 'Room8', 'north'), ('NextTo'
, 'Room9', 'Bedroom', 'north'), ('NextTo', 'Bedroom', 'Room9', 'south'), ('NextTo', 'Room9', 'Office',
 'west'), ('NextTo', 'Office', 'Room9', 'east'), ('NextTo', 'Room10', 'Library', 'south'), ('NextTo',
'Library', 'Room10', 'north'), ('NextTo', 'Room10', 'Study', 'east'), ('NextTo', 'Study', 'Room10', 'w
est'), ('Lock', 'Room10', 'Study'), ('Lock', 'Study', 'Room10'), ('Lock', 'Room8', 'Library'), ('Lock'
, 'Library', 'Room8'), ('Lock', 'Room6', 'Room2'), ('Lock', 'Room2', 'Room6')]
Goal State:
{('At', 'Kitchen')}
Sequence(s):
[('MoveNorth', 'Office', 'Library'), ('Unlock', 'Library', 'Bedroom', 'east'), ('MoveEast', 'Library',
 'Bedroom'), ('MoveNorth', 'Bedroom', 'Study'), ('MoveNorth', 'Study', 'Kitchen')]
```

Figure 4.21 - Test 3, case 4, after adding items

4.2 Evaluation of Overall Program

As an overview of the total program, most of the requirements that the project aimed to achieve have been implemented and, as shown by the tests, perform as expected over a range of possible inputs. These successes include the program being able to generate the correct initial state based on a sequence of action and a goal state given by the input, as well as being able to save this in a valid format that can be tested by the STRIPS program [1] and also used in future work to create a playable version of the game.

A requirement that has not been fully met is the one to increase the difficulty to solve the game based on items and blockages in the world. While there are functions that have been implemented that successfully add locked doors and dark rooms to the map, these functions only get called if the player already has the required item in their hand. This means that the difficulty does not increase for cases where the player is not holding anything. As well as this, while the theoretical difficult measure does increase for cases where the requirement is met, the actual difficulty for the player to reach the goal state does not increase as they already have the item required to get past the blockage.

An issue also arises with the limitation that the inputted action sequence should remain unchanged with respect to adding rooms in the middle of the path. This issue is that the actual difficulty for the player to reach the goal state will depend heavily on the initial inputted action sequence. For example, a sequence that requires two moves to reach the goal from the start will be easy to solve regardless of the number of rooms that are added to the world, because there will always be the two move route to the goal. In contrast, if the input already has multiple locked doors and dark rooms already in the world as well as a long chain of rooms that separate the start location from the goal location, the final output of the program will be significantly harder to solve, even if the final measured difficulties are equal to the first example.

5 Future Work

The nature of this project allows for further development of the software that goes beyond the scope of the project. In this section, additions to the existing code are discussed, along with any issues that may arise in the implementation of such an addition.

5.1 Create the Game

An additional piece of software could be developed that would take an adventure game, saved in the form of a planning problem, and allow a user to explore the world and play the game. The planning problem syntax would be a good start for creating the playable game, as the state already contains the knowledge the game needs about the world. The interaction with the user, however, would require a parser that would allow the user to enter a command that the program would be able to interpret and then use to update the state accordingly.

5.2 Further Development of Item Placement in the Game

Current implementation of increasing the item-based difficulty measure only adds blockages to the world, and blockages are only added if the item that is required to pass the blockage is already in the player's inventory or in the starting room of the world. Algorithms could be developed that would allow new items to be placed into the world. For example, a world with no locked doors and no key could have a key placed into the world when the locked doors are added. However, as discussed in section 3.6, this new implementation could result in adding items that cannot be reached by the player because their associated blockage blocks them. Therefore, this feature would require a way of checking to make sure an item is not placed behind the barrier that requires the item in order to pass through, for example, a key must be accessible before the door that it unlocks.

In addition, any locked doors in the world are currently opened by the same key, which, while easier to implement, means that adding locked doors behind existing locked doors will not increase the difficulty of reaching the goal as the player must already have the key, to unlock the first door, in order to reach the second door. This issue could be rectified by introducing a multiple key system in which each locked door requires a specific key. This system would be implemented by passing a new argument to the "Unlock()" action that specifies which key is required to open the lock. Because of the current implementation of adding locked doors to the game to increase the item-based difficulty measure, each locked door added in this process would still only be unlocked by a key that the player is holding, but this could be extended such that the player is holding multiple keys to allow for each lock to require a different key.

5.3 More Interactions with NPCs

Currently, the only interaction that can take place with an NPC is to reveal a hidden path between two rooms. An addition to the existing program could be to implement more interactions with an NPC.

Interactions could include a way of acquiring an item from an NPC by buying it or trading the item for something the player finds in the world. This would be fairly simple to implement as it would just require an action to buy items from an NPC or give items to an NPC that would mostly utilise the already defined propositional variable. The only new variable that would be introduced is one to describe that an NPC has an item. This could be done by modifying the current “has(item)” variable to include an argument to say who has the item – i.e. “has(player,item)” or “has(npc1,item)”.

Another interaction could allow an NPC to reveal hints or information about the world that would tell the player what their goal is or how to get past a particular blockage, for example the goal might be to give a particular item to the NPC and by talking to them the player finds out what item they must retrieve. This would be difficult to implement however as the hint will often be a text-based clue that has no effect on the world state and there is no way to describe such information in terms of an action of propositional variables. If the program were able to create the game though, a hint action could be implemented that gives the name under which a hint is stored in a separate data structure, allowing the program to do a lookup and display the hint on the screen.

5.4 Show Items in Visual Representation of the World

A further improvement to the visualisation of the world could be to include the locations of items and NPCs on the map. More details could also be added to represent the blockages, such as locked doors and hidden passages. These additions would allow the user of the program to get a clearer idea of what the world looks like as it would give a complete representation of everything in the world, not just the connections between rooms.

6 Reflection on Learning

6.1 Difficulties Overcome in Development

Throughout the development process certain parts of the software were more difficult to implement that I had expected when planning how my time would be spent. These issues often came from a bug in the code that I hadn't anticipated, but which I found when testing the code.

For example conflicts arising when multiple initial states were being combined were not caught until I created some test cases and they outputted results different to what I expected. When initially designing the code, I made an assumption that no conflicts could arise when merging the action sequences because the input would not contain conflicts. However, I soon realised that this would not be the case most of the time unless the creator of the input gave specific thought when making their action sequences. This first arose when I created a test case in which two action sequences tried to reach the goal by going in different directions. I noticed that this was a conflict because the visualisation of such a world was wrong and therefore wrote some code that would throw an exception if any conflict in the action sequences would cause an error in the grid of the world.

In addition this was also the first time I had defined my own exception in Python that would allow me to better describe the error that had been caught. Learning how to do this cost time that could have been spent catching other bugs or implementing some of the future work, but it was necessary to learn and will help me in future projects in a similar situation.

Another difficulty I overcame was that of trying to change the move action to moving in a compass direction as opposed to just moving between two rooms. This change required some way of making the layout of rooms into a grid-like structure that would result in correct placement of rooms and not just allowing any room to connect to any other room regardless of location. It also meant that my previous implementation of adding rooms to the world to increase the difficulty had to be changed as up to that point, a room could be added that connect any two rooms in the world. To change this code I had to create a way of mapping the set of rooms in a world to a coordinate system that would allow me to add a room to a specific coordinate and instantly find any adjacent rooms that could become neighbours. This extra implementation cost time that I had allocated to increasing the item-based component of the difficulty measure, which would have benefitted otherwise. It did however mean that the worlds the program generated would make more sense to traverse and also allowed for the conventional compass-point movement that is implemented in most games of the same genre.

6.2 Professional Skills

Before this project, I had not done such a large-scale individual software development. I, therefore, came to the realisation that there was a lot more work that I personally needed to do compared with group projects I have previously been a part of that were of a similar time-frame. This realisation gave me an understanding of how important creating a work plan at the start of

development is in order to make sure that the project is completed in time and that your time is spread evenly throughout the process. I believe that this idea of setting out a work plan will help to achieve all the aims of future individual projects of the same scale.

I have also never produced as much code as I did in this project, and this has allowed me to see the importance of organising the code in a way that makes it easy to find specific functions or objects that need to be updated or debugged. Furthermore, organised code is imperative in being able to communicate effectively to another person how the code works and which functions perform what jobs. Both of these issues meant that time I could have spent developing my code was actually spent re-organising functions to make finding a particular function easier.

Also attributable to the amount of code being written, I increased my appreciation of the usefulness of generalised functions that can be used at multiple points in the program. They save time in creating the process, rather than repeating similar lines of code, and are more efficient when changes need to be made. They also allow the code to be understood and debugged more easily.

Appendix A

A full list of actions and their associated preconditions and postconditions:

MoveNorth(From,To)

Preconditions: At(From), !At(To), NextTo(From,To,north), NextTo(To,From,south),
!Lock(From,To), !Dark(To), !HiddenPath(To,From)

Postconditions: At(To), !At(From)

MoveEast(From,To)

Preconditions: At(From), !At(To), NextTo(From,To,east), NextTo(To,From,west), !Lock(From,To),
!Dark(To),!HiddenPath(To,From)

Postconditions: At(To), !At(From)

MoveSouth(From,To)

Preconditions: At(From), !At(To), NextTo(From,To,south), NextTo(To,From,north),
!Lock(From,To),!Dark(To),!HiddenPath(To,From)

Postconditions: At(To), !At(From)

MoveWest(From,To)

Preconditions: At(From), !At(To), NextTo(From,To,west), NextTo(To,From,east), !Lock(From,To),
!Dark(To), !HiddenPath(To,From)

Postconditions: At(To), !At(From)

Open(Container,Item)

Preconditions: Has(Container), Openable(Container), Contains(Container,Item), !Has(Item)

Postconditions: !Has(Container), Has(Item)

Talk(Person,PersonLocation,Room1,Room2,Direction12,Direction21)

Preconditions: Knows(Person,Room1), Knows(Person,Room2), HiddenPath(Room1,Room2),
HiddenPath(Room2,Room1), In(Person,PersonLocation), At(PersonLocation),
NextTo(Room1,Room2,Direction12), NextTo(Room2,Room1,Direction21)

Postconditions: !HiddenPath(Room1,Room2), !HiddenPath(Room2,Room1)

Take(Item,Room)

Preconditions: At(Room), In(Item,Room), !Has(Item)

Postconditions: Has(Item), !In(Item,Room)

Unlock(Current,Neighbour,Direction)

Preconditions: At(Current), !At(Neighbour), NextTo(Current,Neighbour,Direction),
Has(Key), Lock(Current,Neighbour), Lock(Neighbour,Current)

Postconditions: !Lock(Current,Neighbour), !Lock(Neighbour,Current)

ClearDarkness(Room,Item)

Preconditions: Has(Item), Purpose(Item,Light), Dark(Room)

Postconditions: !Dark(Room)

References

[1] Wesley Tansey, Python implementation of STRIPS. https://github.com/tansey/strips

[2] Malte Gabsdil, Alexander Koller, Kristina Striegnitz. Building a Text Adventure on Description Logic. 2001. http://cs.union.edu/~striegnk/papers/dlws.pdf

[3] Will Crowther, Colossal Cave Adventure, 1976. https://en.wikipedia.org/wiki/Colossal_Cave_Adventure

[4] Scott Adams. Adventureland. 1978. https://en.wikipedia.org/wiki/Adventureland_(video_game)

[5] Tim Anderson, Marc Blank, Bruce Daniels, Dave Lebling. Zork. 1977-1979. https://en.wikipedia.org/wiki/Zork

[6] Michael Scott McClendon. The Difficulty and Complexity of a Maze http://t.archive.bridgesmathart.org/2001/bridges2001-213.pdf

[7] McClendon. Complexity of Rectangular Mazes.

http://www.math.uco.edu/mcclendon/complexityrecmazes.pdf

[8] Wikipedia article describing STRIPS https://en.wikipedia.org/wiki/STRIPS

[9] Python 3 documentation. https://docs.python.org/3/

[10] Pygame Wiki. https://www.pygame.org/wiki/GettingStarted