Chapter 1

Maintenance Manual

1.1 Pre-requisites

This program has been ran and tested on Mac OSX 10.9.5 and Debian 7 Wheezy 64-bit systems. Python 3.4.2 is the preferred runtime[1]. This software is incompatible with the Python 2.x branch.

Ruby[2] is needed for the digraph_generator.

Both systems are designed to have as few dependencies as possible so *no external package installations are needed* outside of Ruby and Python3 languages.

Both languages chosen are interpreted languages so no installation, building or compilation is needed.

1.2 Bug log

Though the system makes great effort to accommodate all possible pathological cases for graphs and move combinations, the system makes no attempt to defend against improper use, and so many things such as incorrect file formats or using the API incorrectly will cause it to crash in a variety of colourful ways.

Only one instance of the Game class and the program should be run due to the use of SQLite persisted to disk.

1.3 File Listing

Path Description

main.py This file contains GameShell class which inherits CMD and provides the interactive command line interface. The purpose of this class is for user interface only and avoids containing game logic. Within the postcmd method, there is logic to manage the automated playing behaviour. Most behaviour for the other methods is delegated to the Game class. The main method in this file also can handle command line arguments provided.

game.py This file provides the Game class and behaves as the adjudicator for the rules. This class should be a singleton due to the use of SQLite. This class provides current_player to give the correct proponent or opponent according to the rules, and will throw InvalidMove error if either tries to make an invalid move. The game state can also be queried through

[1] https://www.python.org/

[2] https://www.ruby-lang.org/en/

is_game_over and game_over_reason. Game provides the from_file and from_af methods which instantiate the game with a knowledge base.

player.py This file provides two classes, Proponent and Opponent which are both initialized with an instance of Game and delegate the moves to it. Proponent implements has_to_be and next_move, Opponent implements could_be, concede, retract, and next_move. The next_move method on both classes returns the string representation of the desired move and the instance of the argument it suggests.

argument.py This file contains the entire SQLite section. It implements the Argument class which has class methods for managing the SQLite database, loading graphs and files, and searching for arguments and relations. Instance methods on Argument allow for accessing the name, label and step (minmax numbering)

labelling.py This labeller is based on SASsY ASPIC- implementation with modifications to make it appropriate for the use of this system. It has one class method, grounded, which accepts an Argument Framework (In this case the Argument class) and calls get_arguments on it and updates the knowledge base with correct labelling and minmax numbers.

generator/digraph_gen.rb This file contains several graph generators and saves to the format seen in Appendix **??**. The DiGraph class is instantiated with nodes and contains the attribute nodes and the method add_attack. The graph generators are fully_connected_builder, random_graph_builder, balanced_tree_builder, unbalanced_tree_builder, worst_case_tree_builder, looping_graph_builder. All these methods take an array of names as one parameter, and balanced_tree_builder, unbalanced_tree_builder, worst_case_tree_builder take the number of child nodes each node should have. generate_node_names provides arrays of node names when provided the number of names you desire.

runner.rb This file is a naive script that will attempt to run every file in the graphs directory through main.py and output to the results folder.

1.4 Future Usage

This software is highly modular in the design, meaning that future usage will be straight forward. To interface programmatically in Python, you will need the files *game, player, argument, labelling* and in your new file run `from game import Game`. This will grant you access to both proponent and opponent through the current_user variable. For instance if you were to make a new web interface, your controller file would include Game, send the user a prompt for input dependant on the current_user and retractable_arguments variables, then with the return value for a move just delegate directly to the available player.

If you need to make changes to how the SQLite Database is used, please inspect the Argument.py file, in which you can find database creation SQL in the _create_db private class method. There are also _reset_db and _delete_db methods for database control. All SQL used within should be database agnostic with the exception of data types in the _create_db method as SQLite[3],

[3] https://www.sqlite.org/datatype3.html

PostgreSQL[4] and MySQL[5] data types are all different.

You can also use the Argument.py file in isolation to load a previously ran discussion with the from_database method. Once loaded you will be able to instantiate arguments and use the plus and minus methods to explore attacks without using SQL directly.

[4] http://www.postgresql.org/docs/9.3/static/datatype.html
[5] http://dev.mysql.com/doc/refman/5.6/en/data-types.html