CONTENTS

Cmput 496 Assignment 3

Due Mar 19, **11:55pm**. Submit via eClass submission link on the main eClass course page.
Late submission (with 20% deduction) Mar 21, 11:55pm. Submit via separate eClass link for late submissions.

In this assignment, you develop and test an improved simulation-based player for the game of Go. You add your improvements to our sample player called `Go3`.

Setup

1. First, as in previous assignments, make sure you have your Python 3 and GoGui set up.
2. Now download assignment3.tgz. This contains directories `Go3` and a new `util`, which together implement a set of simulation-based Go players, as discussed in class. It also contains a file with public tests.
3. Modify/add code in subdirectory Go3 only to implement your solution.
4. Study the Go3 documentation.
5. For more information about the game of Go and the Go concepts related to this assignment, see the following resources:
   - Lectures 1 and 2: Game of Go, Basic data structures and algorithms for Go
   - Lecture 12 - random simulations, basics of `Go3`
   - Lecture 13 - Improving simulations, filtering rules and pattern-based playouts

Goals of the Assignment

The main goal of this assignment is to develop a stronger simulation-based Go player. The assignment has two parts: you will implement two different improvements to the *rule-based* `Go3` simulation policy.

SIMULATION POLICY IMPLEMENTATION IN `GO3`

The simulation policy to generate one move in a simulation is implemented in file `util/board_util.py`, method `generate_move_with_filter`. Your work will be to improve this method. The current policy implements two rules: `generate_pattern_moves` and `generate_random_move`. It also implements two filters, which are used in different circumstances: `filleye_filter` (always used) and `selfatari_filter` (only used if switched on with --movefilter, see `Go3` documentation).

LAST MOVE

Both of your improvements will implement local responses to the opponent's last move. The Go board in `util/simple_board.py` stores the last move in field `self.last_move`. If `self.last_move == None`, it means that there is no last move, which happens at the very beginning of a game. In this special case, your new code for parts 1 and 2 should not generate any moves and "fall through" to the other rules.

PART 1: ATARI CAPTURE

Implement the following rule: if you can capture the last move, then generate only this move. This should be the highest priority rule, and always tested first. You can capture the last move if two conditions are true: 1. the block containing the last move must be in atari (have only one liberty), and 2. the move on that liberty must be legal (for example, not forbidden by the ko rule).

Move filter: apply the `selfatari_filter` (which also calls the `filleye_filter`) for potential atari capture moves.

The `policy_moves` command should return the capture move. See GTP commands below.

See the examples.

PART 2: ATARI DEFENSE

Implement the following rule: If the opponent's last move put one of the player's own blocks in atari, then try to save that block.

Details: In general, there are two ways to save a block that is in atari: run away, or capture the opponent's adjacent block.

1. Run away means to play on the last liberty. However, run away only makes sense if the block has more than one liberty after that move.
2. Capture means gaining liberties by capturing opponent adjacent stones.

In any case, only legal moves should be generated. You should generate all moves that defend atari according to the rules above. This should be the second-highest priority rule, applied if Atari Capture does not find a move. In that case, the `policy_moves` command should return the list of all Atari Defense moves, while during simulation, if this rule applies, your program will choose uniformly at random among these moves.

Both in `policy_moves` and during simulation, the `selfatari_filter` move filter (which also calls the `filleye_filter`) needs to be used on potential atari defense moves.

REST OF POLICY

If neither part 1 nor part 2 create a move, then "fall back" to the existing policy rules in `Go3`: as third priority, try pattern moves. As last priority, try random moves. Do not change anything about the implementation of these parts of the policy, just call the existing code.

GTP COMMANDS

Modify the following existing GTP command:

1. `policy_moves`

   This command prints the set of moves considered by the simulation policy for the current position, in the format

   ```
   = MoveType movelist
   ```

   Where `MoveType` is one of: `{Random, Pattern, AtariCapture, AtariDefense}`. Currently, only `Random` and `Pattern` are implemented. `Movelist` is an alphabetically sorted list of moves (this is already implemented).

   Example:

   ```
   = Pattern b3 b4 c2 d3 d4
   ```

   See more examples.

2. `genmove color`

   Your `genmove` command should generate a move using the rule-based simulation player with your new improved simulation policy.

Testing Procedure

Test your program with the options
`python3 Go3.py --movefilter --simulations=rulebased`

Public test cases are in assignment3-public-tests.gtp. This file is also in your assignment3.tgz starter code. Our public tests use the `policy_moves` command only.

All the tests of `policy_moves` which expect an answer of type AtariCapture or AtariDefense currently fail, since the `Go3` starter program does not implement these. Note that our given examples cover only a small subset of possibilities. For evaluation, we will use a more comprehensive set of tests.

Tests of modified player behaviour in `genmove` will be done in our private tests. The generated move will be checked in positions where the improved policy makes a big difference to which move has the best winrate.

Other GTP Commands

Many other GTP commands are already implemented in the `Go3` sample code. Do not break their functionality, since we will test your program using those commands, such as:

```
boardsize, clear_board, name, version, play,...
```

These commands are used by the test script. In this assignment, for convenience we will use multiple calls to `boardsize` and `clear_board` within the same gtp file, so make sure this use case still works in your solution. See the examples.

Examples

Study examples carefully to understand details.

- Some examples were given in the assignment preview at the end of Lecture 13. See the course slides.
- Start testing with the public test cases in assignment3-public-tests.gtp .
- Also study files a3-sample.sgf and a3-ko-example.sgf. These are sgf format game records with similar examples. You can load these files in GoGui with File - Open, navigate to test positions that you are interested in, and (for example) attach your program and run `policy_moves`. These sgf files are added in the hope that they will be useful for your testing, but you do not have to use them.

Do Your Own Pre-submission Test

Follow the steps below on a standard undergraduate machine, and submit a text file `presubmission.log` that shows a copy of your command line presubmission testing.

1. Create `assignment3.tgz` with your solution as described under Submission Checklist below.
2. Use the Linux script command to log your testing session.
3. Copy `assignment3.tgz` into a new directory
4. unpack it
5. run gogui-regress with your program on file `assignment3-public-tests.gtp` as input
6. Stop `script` logging here with the command `exit`.
7. Add file `presubmission.log` to your submission, and compress it again
8. Use `tar -tf` for a final check of your `assignment3.tgz` contents. Use the checklist below.
9. Fix any problems, then submit on eClass when you're ready

Submission Checklist

Submit a single tgz file called `assignment3.tgz` which contains exactly the following (and nothing else):

A single directory `Go3` which contains all the files in your solution, namely:

1. All the files from the original Go3 directory, but with the python code modified to solve the assignment. You may add more .py files but do not remove or rename any existing ones.
2. The file `presubmission.log` from your presubmission testing.
3. A file `readme.txt` which lists the names and student IDs of your team.

Marking

There will be a total of 5 marks for this assignment.

- 2 marks for passing our public test cases in `assignment3-public-tests.gtp`
- 1 mark for doing your own presubmission check and supplying the evidence in `presubmission.log`.
- 2 marks for passing our private test cases

Code that does not run will receive a mark of 0.

Useful Tools

These will be useful for developing this assignment.

- GoGui visualisation of policy moves.
- Watching games in GoGui with gogui-twogtp.

- Policy-only player. Go3 implements a second player, which randomly chooses among all policy moves, and does not do any simulation. You can use this player by running `Go3/PolicyPlayer.py` instead of `Go3/Go3.py`. See the Go3 PolicyPlayer documentation.

EXTRA COMMENTS

- No special error handling code is required in this assignment. We will only test with valid moves and valid GTP commands which have valid arguments.
- We will only test with alternating play, so any `play` commands will be for the color toPlay. Clearing the board resets toPlay to Black.
- By default, we assume that teams will stay the same as in assignment 2. If you change your team, inform the TA before the assignment 3 deadline.
- The Go board in Go3 also implements `self.last2_move`. This is the second-last move, and is used for the pattern moves. It is not needed for this assignment.

Created: Feb 28, 2018 Last modified: Mar 3, 2018

Martin Müller
