Reflective Essay

- Naomi Christie
- 200724012
- Matthew Huntbach
- Computing & Information Systems

Introduction

The following essay will describe the work on enabling a computer to play go against a human player. It will cover the development process, including steps taken to test the code and to handle errors. Some of the successes from the project will be highlighted, as well as some areas where things didn't work as well as planned. Decisions taken along the way will also be described and explained. Finally future avenues will be outlined.

Code Development

Version control

During development Git was used for version control yielding a number of benefits. The major benefit of using version control is allowing a remote backup of the system to exist, so that were the local development machine to break code could continue to be developed on a separate machine. In my particular case I sometimes worked on the code from my personal computer, and sometimes from my work computer and the ease of being able to pull down the code base whichever machine I was using was helpful, as was the possibility of choosing to run the code from the more powerful of the two machines I had available. Source control also helped me structure my work as I attempted to make each commit tackle one feature. In reality I was less disciplined about this than I would have been had I been working on a team where clarity of purpose of commits is more critical. None the less, I found writing descriptive commit messages helped me understand my progress when coming back to the code a day or two after last working on it.

Example commit message:

```
commit a040b14da0815cc3ee6264621b3cfd7179e4e787
Author: Naomi Christie <naomi.christie@fundingcircle.com>
Date:   Mon Aug 15 11:41:44 2022 +0100

    Time the recursive function

    I've added at timer which logs after the recursive function runs.
    Seeing poor performance all round. Going to try removing the game
    tree building part to see if that gives improvement.
```

Test driven development

Code was written for the most part using test driven development (TDD). I would first write a failing test for a new feature I wanted to develop, then write the code to allow that test to pass. A big advantage to this

style of working is as I developed new features they often broke old features, and I was able to see this straight away by running the test suite. Once I saw things which had broken during development it helped guide me as to what had gone wrong in order to fix it. There were some drawbacks to the TDD approach however. One was that I encountered a lot of test pollution: where fixtures set up for one test ended up transferring over to new tests. This slowed down development quite a lot while I was establishing what had gone wrong, as my first assumption was that the software I was developing had a bug, and it took some time to establish that it was issues in the test environment. Had I more time I would have done greater research into the causes of test pollution and solutions for it, but with limited time I found inelegant work-arounds including creating unique variable names for each test and manually deleting object members in the test code in order to ensure they were clean before the test took place.

Debugging

One incredibly useful took during development was Python's debugger which can be invoked using the following line of code: `import pdb; pdb.set_trace()`. Once the debugging line is in place I was able to run the test suite and enter the code at the point the debugger was placed in order to inspect objects and variables at that point. Using the debugger I was able to establish that test pollution was occurring, for example. I also employed the debugger to initially establish why the algorithm didn't block winning moves from a human player, a problem I've labelled the 'despondent machine' as the computer appears to give up at this stage. The output from the debugger displayed below (reference: https://github.com/nchristie/robogo/commit/4c5731ae4320c0327befac50bae0e5c8a3a5f345) shows the path from a root node onwards in a game on a 5x5 board when the win condition is three stones in a row. You will see that when following the path after the blocking the human's winning move eventually the board gets to a state where there is no single move the computer can make to block the human from winning.

Python debugger output:

```
(Pdb) [print(row) for row in tree_0809_0414.root_node.get_children()
[3].board_state]
['●', '○', '+', '+', '+']
['○', '+', '+', '+', '+']
['●', '+', '+', '+', '+']
['+', '+', '+', '+', '+']
['+', '+', '+', '+', '+']
[None, None, None, None, None]
(Pdb) [print(row) for row in tree_0809_0414.root_node.get_children()
[3].get_children()[7].board_state]
['●', '○', '+', '+', '+']
['○', '+', '+', '+', '+']
['●', '●', '+', '+', '+']
['+', '+', '+', '+', '+']
['+', '+', '+', '+', '+']
[None, None, None, None, None]
(Pdb) [print(row) for row in tree_0809_0414.root_node.get_children()
[3].get_children()[7].get_children()[7].board_state]
['●', '○', '+', '+', '+']
['○', '+', '+', '+', '+']
['●', '●', '○', '+', '+']
['+', '+', '+', '+', '+']
```

```
['+', '+', '+', '+', '+']
[None, None, None, None, None]
(Pdb) [print(row) for row in tree_0809_0414.root_node.get_children()
[3].get_children()[7].get_children()[7].get_children[10].board_state]
*** TypeError: 'method' object is not subscriptable
(Pdb) [print(row) for row in tree_0809_0414.root_node.get_children()
[3].get_children()[7].get_children()[7].get_children()[10].board_state]
['●', '○', '+', '+', '+']
['○', '+', '+', '+', '+']
['●', '●', '○', '+', '+']
['+', '●', '+', '+', '+']
['+', '+', '+', '+', '+']
[None, None, None, None, None]
```

Logging I employed logging using Python's logger library. One area this was useful was was helping to unpick what was happening while the recursive algorithm was running, for example I was able to print when the algorithm reached a return statement and then see what the utility of the node at that level was, and what the node-id was:

```
logger.debug(
    f"Returning at depth of {depth} with score of {parent_utility} at
node: {parent_node_id}"
)
```

I also employed print statements in the test code to show the computer's path down the game tree - formalising what I had learnt from using the Pythong debugger for the despondent machine issue outlined above. I employed the following block of code in order to do this:

```
def print_game_path(depth, print_node):
    for i in range(depth):
        print(
            f"Move {i} score: {print_node.get_score()}, path_depth:
{print_node.get_path_depth()}"
        )
        for row in print_node.board_state:
            print(row)
        if not print_node.is_leaf_node():
            print_node = print_node.get_optimal_move()
        else:
            break
```

(Reference: https://github.com/nchristie/robogo/commit/7e2b8364b959f95ed77c8422f033a5ecf217793c#diff-acc9f5d2892f3a2255a022dc3d77e88d81eb60bb3b1d0e6271a1dd3b0629815eR195-R203)

Manual testing I found manually testing by playing the game itself helped to uncover edge cases which I hadn't thought of during test driven development. I was then able to add those test cases to the

automated test suite. Manual testing was also a way to run through scenarios and see how the code performed time-wise. For example I established that the algorithm takes a lot longer to choose a move at the start of the game when there are many potential alternatives compared to later on in the game where a number of moves have already been made on the board, reducing the number of branches to investigate. This led me to add a time-gain to the code whereby it searches to a shallower depth early on in the game, and searches more deeply as the game progresses and the branching reduces.

Error handling

During the course of coding I found it very beneficial to include error handlers for a range of scenarios. Placing the errors at the lowest possible level and providing meaningful messages was the best strategy for example the MinimaxNode class is the generic class which creates nodes in the game tree. Each node can have a score and there is a setter for that score:

Setter for node score

```
def set_score(self, score):
    logger.debug(f"In set_score for node: {self.get_node_id()}, score:
{score}")
    if type(score) not in [int, float]:
        raise Exception(
            f"set_score error: score must be int or float got
{type(score)} for node: {self.get_node_id()}"
        )
    self.score = score
```

Example of set_score in use

```
best_score = func(
    res["best_score"],
    best_score,
)
parent.set_score(best_score)
```

If there were an error in the function which returns the best score above and it returned the wrong type, then this would be raised by the error handler in the ser_score function.

Metrics

In order to weigh up different options I included some simple metrics in the logging using python's `perf_counter` from the `time` library. Using this I was able to see that using a generator to create node children took less time than creating all the children up front. In an example I ran on a 9x9 board with a win condition of five stones in a row, a tree depth of 3 and while calculating the minimizer's second move in the game it took 60 seconds to run the minimax algorithm if all node children were generated up front, and 55 seconds if a generator was used.

Notes taken during manual testing

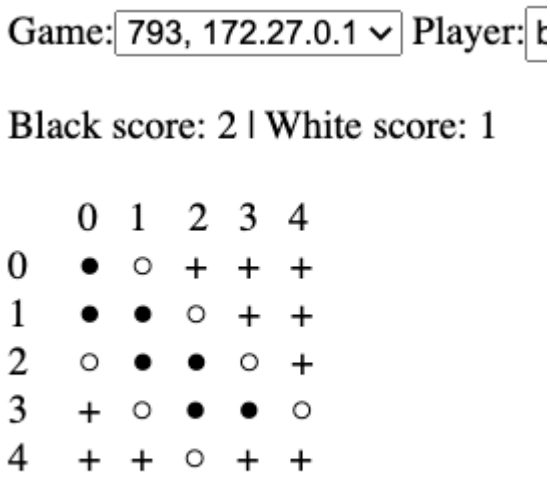
```
BOARD_SIZE = 9
WINNING_SCORE = 5
MAX_TREE_DEPTH = 3
child getter = get_all_children_and_rank_by_proximity
Calculated white move: (1,1)
Minimax seconds to execute: 60.1060

BOARD_SIZE = 9
WINNING_SCORE = 5
MAX_TREE_DEPTH = 3
child getter = generate_next_child_and_rank_by_proximity
Calculated white move: (1,1)
Minimax seconds to execute: 55.3870
```

Successes

The major success in this project is that the result is a computer which plays a game against a human player. In all manual tests the computer blocks the human from making winning moves, and games end in stalemate. The computer also takes moves in a fairly reasonable time frame - normally within a few seconds, but can stretch to over a minute for moves early in the game when the branching factor is higher owing to the greater number of open positions on the board.

A breakthrough moment when writing the code was when I conducted a manual test and saw the computer played in such a way as to create a pattern on the board known in Go as a ladder (reference: https://en.wikipedia.org/wiki/Ladder_(Go)) (see image below).

Robogo playing in ladder formation on 5x5 board with win condition of 3 in a row



The decision to remove as many non-functional requirements from the code for the algorithm as possible increased overall performance, and this process being guided by metrics helped a great deal.

Things which didn't go so well

The biggest compromise I had to make during development was to forego implementing the capture and ko rules of Go in favour of achieving an alpha-beta pruning algorithm which worked, as the time constraints of the project meant I was unable to achieve both. However, the way that I have implemented the minimax algorithm with a base class and then a class which inherits from that base and a separate module for game logic means that implementing capture and ko rules can be done in isolation without breaking the existing code. It was ambitious to attempt to implement the capture and ko rules and alpha beta pruning. In hindsight I think it would have been better to aim to do a simpler game such as tic tac toe or connect four in the first instance and save the idea of implementing five-in-a-row Go as a potential extension to the project. Alternatively I could have done this project with an interface on the command-line rather than as a website and that might have bought back enough time to develop more of the game of go instead as while it was interesting to learn Django, that learning process took up a significant amount of development time. After watching the lecture by XXX I was expecting to be able to achieve a tree depth in the order of 14, however the maximum tree depth even on a 5x5 board is 5. Experimentation with different board sizes showed me that with each increment in the size, the time for the computer to calculate moves went up a large amount. While I understand that the original lecture focussed on chess which has greater constraints therefore fewer potential moves, I wasn't able in the time to research fully what was holding up my algorithm. Had I more time I'd calculate how many possibilities the algorithm has to search for the current board size and how the possibilities expand as the board grows in order to better understand why the algorithm couldn't run to a greater depth. One thing which took up a lot of time during development was that once I'd implemented minimax with alpha beta pruning the computer continued not to make blocking moves in some circumstances, which led me to question whether the algorithm was functioning. I had to eventually log step-by-step what the computer's expectation for the upcoming moves were before I realised what was happening. In the situation where the computer didn't attempt to block it was because if both players played optimally the computer would definitely lose, and therefore all nodes had the same value (equally bad). The computer was therefore choosing the first of many bad nodes rather than the one which would block its opponent, extend the game length and allow the possibility of the opponent making a bad move and losing the game. Once I realised what was happening I tried out two things to mitigate: I tried ranking the nodes with a preference for the longest path before a lose state with varying success. I found through trial and error that when I ranked nodes according to proximity to current stones on the board that this also allowed blocking moves to take priority so chose this in the end. A fairly minor error I made was to presume that IP addresses could be unique machine identifiers. I was hoping to use the IP address in order to find a user's game which they hadn't completed in an earlier session and retrieve the details from the database. I discovered after introducing this feature that the IP address that was received by the system wasn't fixed, and so user games will naturally time out as the IP address rotates. I did some research to try to ascertain if there was an alternative way to uniquely identify a machine without the use of cookies, but didn't find an answer. Were I doing this project again I'd have either learnt how to work with cookies or used Django's username and password features to allow the user to log in and resume an old game.

Decisions made along the way

Once I had the game logic in place and a system for playing whereby the computer could recognise when a set of stones were lined up in a row and inform the user who had won I focused on getting minimax with alpha-beta pruning functioning. As mentioned above this meant that I had to forego implementing the capture and ko rules as intended. The first implementation of alpha-beta pruning I made both found the best move and built the game tree by adding children nodes to the root node and adding children to the children and so on. This was very useful for allowing me to inspect the path the computer expected the game to go down, and debug for example why it was that the computer wasn't blocking winning moves

from the human player, however retaining the game tree wasn't needed in order to execute minimax with alpha beta pruning, so I stripped this element out of the code in the hopes of saving processing effort. As a larger board increases the time it takes the algorithm to perform by a great deal, I compromised on my initial plans by doing a 5x5 board instead of 9x9. I found on a 5x5 board game play became much less interesting if the win condition was five stones in a row, as the blocking move for five stones in a row on a 5x5 board is any position in line with a given stone. I found it was surprisingly easy to adapt the board size in the game as it appears on the web browser with just a couple of code modifications. Having spent some time manually testing on a 9x9 board and therefore dealing with either very long processing time between moves or having to restrict the tree depth to 2, introducing the option of a smaller board for manually testing helped a great deal to debug the code.

Future directions

The main feature I would want to work on next were I to continue this project is to implement the capture and ko rules in order to meet the original project specifications and make a game which can help beginners learn the basic rules of Go. As previously mentioned this should not be a huge amount of work, as the space for implementing these rules is isolated from where the algorithm is implemented. I would then want to get the game to work on a larger board, and introduce options for the human player to choose their board size from the web browser at the start of a new game. The most interesting avenue to explore would be to experiment with other algorithms, such as Monte-Carlo search trees. Another interesting avenue to pursue would be to resolve the 'despondent machine' issue where the computer stops trying to block the human from winning when it's on a losing path.

Conclusion The main successes in this project were implementing alpha-beta pruning and ranking moves. Achieving this was facilitated by using a number of developer techniques including test driven development, error handling, logging and version control. The scope of the project was fairly ambitious, and therefore I had to compromise by not implementing the full game of Go. Aiming at the start of the project to implement a simpler game such as connect four would have been easier to achieve.