Enabling Computer to Play Go Against a Human Player

Naomi Christie
200724012
Matthew Huntbach
Computing & Information Systems

***Abstract*—This project creates a computer opponent for the game of Go using minimax and alpha-beta pruning and is implemented using the Python programming language, the Django web framework, and a PostgreSQL database. The game itself is a variation of Go aimed at facilitating the learning of basic moves in the game for beginners.**

***Keywords*—minimax, alpha-beta pruning, Go game**

I. Introduction

A. Background

The ancient Chinese two-player board game Go has attracted media attention in recent years following Google's success in writing Artificial Intelligence capable of beating the world's best human Go player (Cook, 2016). While there are many resources to play Go online both against other humans and against computers (Sensei's Library, n.d.), there are fewer resources featuring computer opponents aimed at assisting beginners to understand the basic rules. Gomoku is a game played on a Go board in which the aim is to place five stones in a row, horizontally, vertically, or diagonally before one's opponent manages to get five in a row, but it loses much of the rest of the game logic, including captures (Wikipedia Contributors, 2022). The Robogo project aims to implement a new variation of Gomoku in which all the rules of the game of Go are maintained aside from the scoring method, which is simplified down to: the winner is the first to get five stones in a row vertically or horizontally (not diagonally). The human player will play against a computer allowing them to learn how moves and captures work on a Go board without needing to find a knowledgeable human opponent. While there are resources to play Gomoku against computers online (gomokuonline.com, n.d, gomoku.yjyao.com, n.d.) on investigation I could not find this unusual variation in which capture rules are upheld available and therefore it represents a new offering to the Go playing world.

B. Software summary

At present Robogo is partially implemented: the game is currently played on a five-by-five board, with a win condition of four stones in a row either horizontally or vertically (not diagonally). Capture and ko rules are yet to be implemented in the game. The computer blocks the human from winning in most circumstances, and this has been achieved by using minimax with alpha-beta pruning. In the following report I will explain how I brought Robogo to its current state, and the next steps anticipated in its development.

C. Terminology

This section of the report will cover some terms common in the game of Go.

- Stone: a player's piece, can be black or white

- Group: a collection of stones which are touching on the board

- Capture: when a stone or group of stones is surrounded on all sides they become the prisoners of the opponent

- Ko rule: the board must not return to an identical state during gameplay. This rule prevents the game from stalling

- Intersection: stones are played not within the lines of the grid, but on the cross-shapes which are made by the lines, these are known as intersections

- Liberty: a stone or group of stones which are yet to be captured have free spaces around them horizontally and vertically, these are known as liberties. Once a group has zero liberties it is captured

- Jump: a move in the game which doesn't connect to a stone, but is some one or more places away

- Connecting move: a move in the game which links directly to another stone either horizontally or vertically

II. Literature Review

A. Language and framework

The decision to write Robogo in Python was based on two things - firstly the popularity of the language; 58% of respondents to the StackOverflow 2022 developer survey said they had 'done extensive development work' in Python over the last year or 'want to work in' Python over the next year (Stack Overflow, 2022), and secondly its status as a back-end language, therefore suitable for the most significant feature of this project which was allowing a computer to play the game.

Flask and Django were both considered for the project. Flask is a lightweight web framework best suited for building APIs, whereas Django is geared towards fuller projects featuring some element of user interface alongside the back-end (Campbell, 2022; Robinson, 2017). In addition Django allows for easy integration with databases through its models, providing a 'single, definitive source of information' about the data in the project (djangoproject.com, n.d.). A range of online Django tutorials (Parker, 2017; djangoproject.com, n.d.; Portella, 2021) were used as models for how to build the project, with the greatest reliance on on Django's own official starter project (djangoproject.com, n.d.).

Various front-end point-and-click graphical options were investigated. Initially the pygame library for Python was considered (Clark, 2022), but eventually was ruled out as it would have been complex to integrate this with a Django back-end in the available time. Using css, html and javascript to render the board was also considered (Lung, 2020; viethoang, 2017), but again on further investigation this implementation would have required a significant amount of effort to be devoted to integration into the Django ecosystem, and can easily become unwieldy (saaspegasus.com, 2021). Eventually the decision was made to rely only on Django's in-built templates and views, and to render the Go board using unicode characters. There are various conventions for preparing ascii Go diagrams (Wedd, n.d.), and the eventual design used these as an inspiration, but adapted from these to make the board easier to read as moves were made. Using Django forms (djangoproject.com, n.d.) was chosen over point and click, again in order to reduce the amount of time spent on front-end coding.

B. Minimax and Alpha-Beta Pruning

The main resources for understanding minimax and alpha-beta pruning were Winston's 2010 lecture entitled 'Search: Games, Minimax and Alpha-Beta' (Winston, P. H, 2010), and Jain's Blog post on minimax and alpha-beta Pruning (Jain, 2017). Additionally Lane's blog post on writing binary trees in Python was used as a model for how to build a tree by using a Python class to create the node objects (Lane, 2021), although as the code developed it diverged from that model as will be discussed later.

III. Robogo User Guide

The user should make sure they have Docker (Docker, n.d.) installed on their machine. Then open a terminal at the robogo directory and type in the following command: `./start_game.sh`. Note that if the user gets an error message reading: `permission denied: ./start_game.sh` then the user should run the following command: `chmod u+x start_game.sh` and then try `./start_game.sh` again. The game should now open in a web browser. The user should see a page showing a simple five-by-five Go board where the intersections are labelled as plus signs, and the coordinates, 0-4, are labelled along the top and the right hand side.

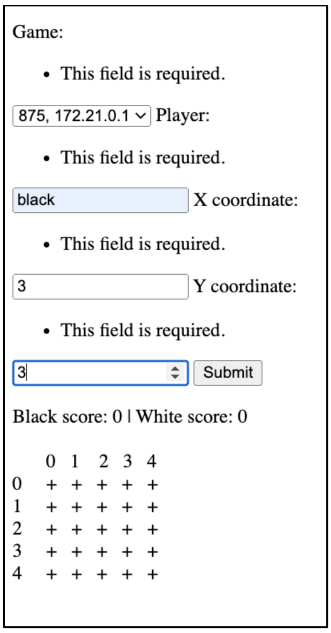

Figure 1. (*Opening state of website and user inputs*)

A form at the top of the screen will allow the user to state which game they are playing (based on their IP address), which colour player they're playing (human user should always play the black player), and which X and Y coordinates they want to play at (see figure 1). In this iteration the form on the first page is a bit confusing for the player as the label for the subsequent input fields appears to the right of the previous one. In future iterations this bug will be fixed. When the user gives coordinates for their move the white response will be calculated and both moves will appear on the board simultaneously. The formatting for the form inputs now becomes easier to interpret (see figure 2).

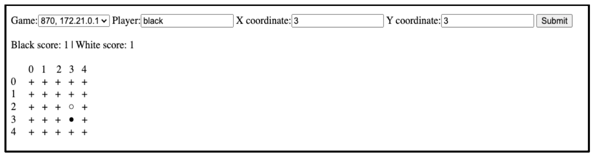

Figure 2. (*Appearance of website following first move on board*)

It can take up to two minutes for the moves to display on the board as the algorithm calculates white's best move. In future iterations of the game there will be a visual indication that the computer is calculating the next move. Once the moves show on the board the human user can then input the coordinates of their next move, and so on. The game detects when the human or the computer has four stones in a row and then displays a game over message including the outcome of the game from the human's perspective. If all positions on the board are occupied without a win state having been achieved, then the system simply displays 'Game Over'. A running score for the two players is displayed at the top of the board throughout (see figure 3).

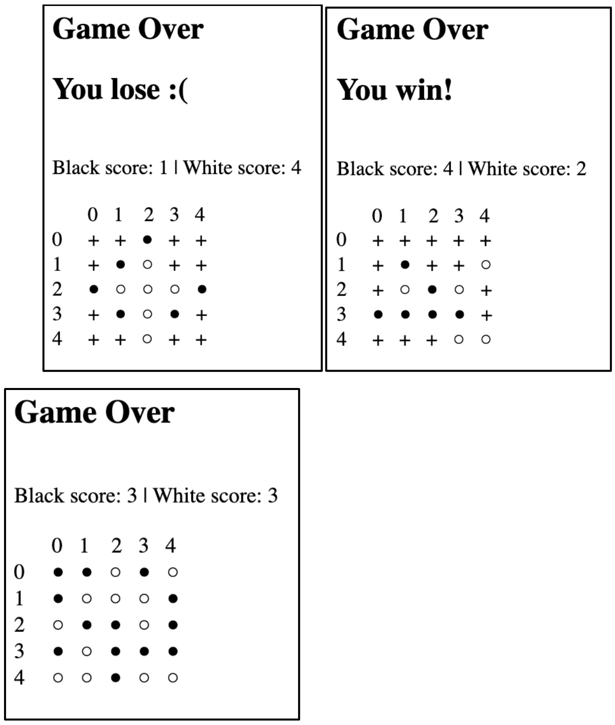

Figure 3. (*Lose, win and stalemate*)

If the user inputs coordinates which are out of range, they will forfeit that move and the computer will play instead. In later iterations this will be replaced with warnings to the user and the opportunity to input valid coordinates.

Capture logic and the ko rule are yet to be implemented in this iteration of the software but will be introduced at a later stage.

IV. ROBOGO PROGRAM

A. Overview

Robogo is implemented using the Django web framework for the Python language and a PostgreSQL database. Docker containerisation has been employed to ease the running of the code from different computers or servers by automating the process of setting up dependencies. Online resources (Thagana, 2022; Docker, n.d.) were referred to in order to set up Docker for this project. The Django framework uses Model, View, Template structure. The Model represents both the classes in the program and the items in the database. The View handles the logic to prepare information for the Template, which is the means of viewing the relevant information from a web browser. Django was chosen to simplify the coupling of concepts in the code to the database, create some ready-made structure to the code and to ease the rendering of information as a web page.

B. Minimax with Alpha-Beta Pruning

The code to allow the computer to select the best move is stored in `games/minimax.py` and `games/go_minimax_joiner.py` and implements the minimax algorithm.

The minimax algorithm allows the computer to optimise its choice of next move by looking ahead several steps in the game by creating a game tree made up of potential moves for each point in the game. The algorithm then assesses the utility of the moves at the given future point in the game (Winston, 2010). For ease this future point in the game will be referred to as the terminal state, although it will normally be a few moves on in the game and not the end state of the game. From this terminal state, the algorithm works back up the tree to establish the optimal move for each player at each point by inheriting the utility from the terminal state. For a two-player game such as Go the computer will always choose the move with the minimum utility to its opponent and presume that its opponent will select their move for maximum utility, and this alternating minimising and maximising of utility is where the term minimax derives. Alpha-beta pruning is layered over minimax to reduce the amount of computing time required to calculate the optimal move. It works by navigating the game tree to the terminal state of one branch and establishing the utility of the optimal move on that branch, now when the next branch is assessed, as soon as the utility for the opposing player breaches the threshold value set by the first branch, we can ignore all paths below that level and avoid these calculations (Crack Concepts, 2019).

In this project minimax is implemented in the file `games/minimax.py`. The core elements in this file are a base class called `MinimaxNode` and the algorithm itself, which is a free function called `minimax_with_alpha_beta_pruning_algorithm`. The code in this file is agnostic to which game is being played, and although it is used in Robogo to play this variation of Go, it could equally be used for any adversarial two-player pure logic game, such as Chess.

C. MinimaxNode

In earlier iterations of the code, `MinimaxNode` featured member variables to allow a tree to be built for future inspection such as score for that node and an array of its children, which in turn were also MinimaxNodes, see commit with hash beginning ef2090 for details (Christie, 2022). It also had method called `get_optimal_move` which was used on the penultimate nodes to determine which terminal node would be selected. The method looped over the leaves of the penultimate node and selects the one with the best score for the player, so if it's the minimiser this is the score which is worst for the opponent, and if it's the maximiser it's the score which is best for itself.

During the course of developing the code and reading online resources (Jain, 2017; Serrão, 2021; GeeksforGeeks, 2021) it was realised that it was possible to do all the relevant calculations during the algorithm without storing the outcomes in the node objects, so the code in its current state does this to reduce the number of actions the algorithm has to perform in an attempt to speed up processing time. Almost all the functions and members which used to reside in the MinimaxNode class are now gone.

D. minimax_with_alpha_beta_pruning_algorithm

The `minimax_with_alpha_beta_pruning_algorithm` function was developed using several online resources such as Jain's blog post (Jain, 2017). The function takes a `MinimaxNode` as its only required argument, and then has optional arguments for `depth`, `winning_score` and `start_time`. Depth and winning score both have default values set elsewhere in the code, while `start_time` is used if we want to apply a timeout to the function but is not required in all cases (for example it isn't used in the tests). The function then checks if its base case has been met: if we've reached full depth or if the node has a winning utility. If either of these conditions are met, then it returns a dictionary with the best score and the move node which will take the computer down the path with that best score. If the base case hasn't been met, then the code sets up some variables for use in the recurse case. `alpha` is the variable representing the best score for the maximizer and is initialised at -infinity, which is the worst possible score for the maximizer. Then `beta` is the variable representing the best score for the minimizer, and is initialised at the worst possible score for the minimizer; infinity. The variable for storing the best utility seen so far while navigating the game tree is entitled `best_score` and is set to -100 if the player to move is the maximizer or 100 if it's the minimizer - different values from the alpha and beta values in order to make it easier to examine what was happening in the code from the logs and the debugger, and still initialised at bad values from the perspective of the player to move. Finally, `func` is a variable which is set to the Python inbuilt function max if the player to move is the maximizer, or min if the player to move is the minimizer - `func` will be used during the recursion to select the best move and assign alpha and beta values.

We now begin examining the children of the parent node. This is done by looping over the output of the generator: `generate_next_child_and_rank_by_proximity`, which will be discussed in further detail later in this report. At this stage the algorithm recurses, i.e. calls itself. The effect of this is that the first child of the root node will find its first child and so on until the base case is met, at which point the utility of the node is evaluated and returned alongside the node itself. Now the best score is found by taking the `func` of the best score from the result and the existing best score - i.e. the minimum of the two if it's the minimizer's turn to play or maximum if it's the maximizer. If our best score at this stage matches the score of this node, then the current child node is assigned the status of best node. At this stage if it's the maximizer's turn to move we set `alpha`, or we set `beta` if it's the minimizer's turn. We use `func` again to get the correct value comparing `best_score` to `alpha` or `beta`.

At this stage we check if break conditions have been met. The following cases are valid break conditions: `alpha` is greater or equal to `beta` - in this case we know that any remaining nodes down this branch of the tree cannot be an improvement on what we have available now, or put another way, continuing down this path would inevitably give the human opponent the advantage against the computer. The other break conditions are if we're at a winning node as we don't need to examine the tree beyond a winning state, or if the process has timed out. The time out was set to 120 seconds, and this is to prevent highly prolonged lagging but comes with the downside that not all options will have been evaluated and the move suggested after timeout may not be the best move in the game, simply the best of all the moves evaluated over the course of two minutes. The function returns the best score and the move node from which the score came either once all children nodes on that level have been evaluated or when a break condition is met. The result is returned to the next level up in the loop, i.e. the parent node until it reaches the root node at which point we have the best move available for the root node which can then be presented in the UI.

For Minimax to work it is necessary to be able to determine the utility of the terminal nodes, which is where the logic of the given game becomes relevant. To separate out the Go logic from the pure Minimax code a new class called `GoNode` was created in `games/go_minimax_joiner.py`. `GoNode` inherits all the variables and methods from `MinimaxNode` and in addition can examine any given board state. To determine the utility of the board state for a terminal node `GoNode` looks for all stones grouped in horizontal or vertical lines and keeps a running score of the highest number of stones per player and then returns the highest group length as that player's score.

E. Getting node children; various approaches

There are several approaches available for generating the children of a given node. Two options are to generate all the children up front, or to use a generator to yield the children one at a time. Both approaches were experimented with, and it was found that the generator improved the algorithm speed.

The other set of options when generating node children is whether to generate the next move sequentially on the board, or to rank moves according to some known strategy. The main advantage to ranking moves is it can speed up the algorithm as it evaluates moves with a higher likelihood to have a good outcome first. Initially a generator which simply iterated over all board positions yielding the next valid move was used. One drawback of this approach is that the gameplay doesn't appear very natural, as when all moves were equally bad the algorithm would simply make the first available move and it becomes clear very quickly that it's moving through the array one cell at a time.

The code for generating ranked node children looks at each place on the board where a stone has already been placed and generates node children connecting to those stones. Once all connecting positions have been generated it then generates moves with a jump of 1 away from each stone, and then a jump of 2 and so forth. A side benefit found of this node ranking strategy is it also led to more natural moves and increased the likelihood of blocking moves when the computer was on a losing path.

The list of potential moves was kept artificially short by limiting the jump size to a maximum of a third of the board's width to preserve computing resource. Another benefit to restricting move options is creating a beginner-friendly computer player who takes a simplistic strategy like that which human beginners tend to adopt.

F. Database

The database software used for this project is PostgreSQL. The database consists of two tables: Games and Moves. Each user can play one game per device as their game is tied to their IP address. Each game can have zero to many moves, and if the game is deleted, all its moves also get deleted from the database.

G. Front end

The front-end is implemented using Django's templating functionality which in turn uses which in turn uses Jinja scripting. The board is rendered using "+" to represent empty intersections, "●" to represent black stones and "○" to represent white stones.

V. Code Complexity

A. Findings

Through trial and error, it was found that the best conditions to allow interesting gameplay in which the human user didn't need to wait unreasonably long for the computer to play and the computer would play robustly in most scenarios were a five-by-five board with a win condition of four stones in a row. This was discovered by adding a timer around the minimax with alpha-beta pruning algorithm and then trying out the timing on the second move in the game for a variety of board sizes and win conditions (see figure 4). The second move was chosen as it was found that the first move was usually quick to calculate, but the second move would take longer. For consistency the first move was always with coordinates x=3, y=3, and the second move with coordinates x=1, y=1. Following is a brief discussion on why the board size needed to be restricted, and why four-in-a-row was conducive to better gameplay.

Time To Process On Varying Boards

Variables (<u>underlined</u> variable changed in each iteration)			**Result**
Board size	***Win condition (stones in a row)***	***Search depth***	***Processing time (seconds)***
4x4	3	8	20
4x4	3	<u>7</u>	8
4x4	3	<u>6</u>	2
4x4	<u>4</u>	6	38
4x4	4	<u>5</u>	5
<u>5x5</u>	4	5	61
<u>6x6</u>	4	5	128
<u>7x7</u>	4	5	>600
7x7	4	<u>4</u>	150
<u>8x8</u>	4	4	262

Variables (<u>underlined</u> variable changed in each iteration)			Result
Board size	***Win condition (stones in a row)***	***Search depth***	***Processing time (seconds)***
<u>9x9</u>	4	4	322
9x9	<u>5</u>	4	331
9x9	5	<u>5</u>	>600

Figure 4. (*Table with processing times*)

B. Board Size and Big O Notation

Big O notation is a way of describing how the run time of an algorithm grows as its input size grows (Wikipedia Contributors, 2022), for example O(n) would mean that the run-time of the algorithm would be proportionate to the number of inputs where n is the number of inputs, and O(n^2) would mean it's order of n-squared etc.

In his 2003 lecture notes on artificial intelligence, Megalooikonomou writes that the "time complexity of minimax is O(b^m) […] where b is the number of legal moves at each point and m is the maximum depth of the tree." When alpha-beta pruning is added to the algorithm the complexity becomes O(b^(d/2)) where d is the cutoff depth (Megalooikonomou, 2003).

This equation explains why processing time was seen to be longer at the beginning of games, when there were more open legal moves than at the end and explains why processing time increases with board size as there are more open moves on larger boards. Finally, it explains why processing time increases with greater tree depth.

In the case of the game of go, to examine the board from early in the game to full depth without pruning the tree would be O(b!), as the legal moves on the board are similar in number to the size of the board size itself so each board position has something in the order of the board size of potential moves to examine, and each of those again has similar. The complexity goes down quite fast on a board with a smaller size, particularly as in the current state of the game where captures haven't been implemented as the proportion of open positions declines.

VI. Testing and Error Handling

A. Test driven development

This section of the report will discuss the concept of test-driven development (TDD) and how it was used during the writing of the code. Please see `games/tests` for the tests written on the code in this project.

Code was written for the most part using TDD. First a failing test would be written for a new feature, then the code was written to allow that test to pass. A big advantage to this style of working is as new features were developed, they could break old features. It was possible to see when features had been broken straight away by running the test suite, and the tests also guided as to what had gone wrong.

There were some drawbacks to the TDD approach. One was that a lot of test pollution was encountered: where fixtures set up for one test ended up transferring over to new tests. This slowed down development quite a lot while investigating the source of the errors. When tests failed the initial assumption was that the software had a bug, and it took some time to establish that it was issues in the test environment. Greater research into the causes of test pollution and solutions for it would have been researched given more time on the project, but with limited time inelegant workarounds were used including creating unique variable names within each test and manually deleting object members in the test code to ensure they were clean before the test took place.

One incredibly useful took during the TDD cycle was Python's debugger which can be invoked using the following line of code: `import pdb; pdb.set_trace()`. Once the debugging line was in place, it was possible to run the test suite and enter the code at the point the debugger was placed to inspect objects and variables. Using the debugger it was possible to establish that test pollution was occurring, for example.

The debugger was also used to establish why the algorithm didn't block winning moves from a human player, a problem casually entitled the 'despondent machine', as the computer appears to give up at this stage. To summarise the issue, the computer was presented the possibility to block when the human was about to place three stones in a row on a game where the win condition was to attain three stones in a row, however it chose not to block. The output from the debugger showing the path the computer *didn't* take, proving that it was an inevitable lose is displayed in Appendix A of this report.

B. Manual Testing

Testing by playing the game itself helped to uncover edge cases which hadn't been envisaged during TDD. Once uncovered those test cases were added to the automated test suite. Manual testing was also a way to run through scenarios and see how the code performed timewise. For example, it was established that the algorithm takes a lot longer to choose a move at the start of the game when there are many potential moves compared to later in the game where several moves have already been made on the board, reducing the number of branches to investigate. This led to a code adaptation whereby the algorithm is instructed to search to a shallower depth early in the game, and searches more deeply as the game progresses and the branching reduces.

C. Error Handling

While coding, it was 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, in earlier iterations of the code the `MinimaxNode` class had a score and there was a setter for that score (see figure 5).

```
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
```

Figure 5. (*Example of error handling*)

Originally `set_score` was called from within the algorithm which executed minimax with alpha-beta pruning (see figure 6). If there were a type error in the code which returns the best score and it returned neither an integer nor a float, then this would be raised by the error handler in the `set_score` function. This code was deprecated eventually as discussed elsewhere in the report but was highly useful during development to observe what the algorithm was doing and flag when there had been an error and why.

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

Figure 6. (*Example of `set_score` called elsewhere in the code*)

VII. Conclusion and Future Directions

This project began with the aim of creating a program which would allow a computer to play a simplified variation of the game of go against a human player. This goal was met by using a minimax algorithm with alpha-beta pruning.

Some initial aims of the project were not met in the timeframe, namely allowing the game to be played on a nine-by-nine board (it uses five-by-five instead), allowing a win-condition of five stones in a row (it uses four instead), and implementing all usual rules of Go aside from scoring (i.e. capture and ko rules). Through investigation it was found that the code lacked the efficiency to permit play on a a nine-by-nine board with a five-stone win condition as the time it took to perform moves was too high.

Some future directions worth investigating are as follows:

- Assess which part of the code is causing the most lagging when calculating game moves. It is suspected that this will be in the code for generating child nodes, but each part of the code used in the algorithm would benefit from metrics to establish where efficiencies could be made. One option to reduce lagging would be to pre-calculate the game tree for early moves in the game and implement a database of initial board states and responses which could be drawn on at the stage in the game where complexity is highest owing to higher branching. Another option would be to implement in a lower-level language such as C in order to reduce latency.

- Investigate other algorithms and compare performance, for example Monte Carlo tree search

- Implement the capture and ko rules to complete the project to the initial specifications. Further investigation into the complexity of the game should be conducted at this stage.

- Improve the user experience: it was not a priority to have a good user interface for this initial implementation of the code as the goal was to have a computer which could play against a human rather than to have an elegant interface, however, the interface could be much improved. As the code is separated into separate areas for back end and front-end logic, introducing a fully functional point-and-click front end such as the ones researched during early development should be possible.

References

Cook, M (2016) 'AlphaGo: beating humans is one thing but to really succeed AI must work with them' *The Guardian, 15 March 2016*. [online] Available at: https://www.theguardian.com/technology/2016/mar/15/alphago-ai-artificial-intelligence-beating-humansgoogles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol [Accessed 23 August 2022]

Sensei's Library (n.d.), 'Go servers.' [online] Available at: https://senseis.xmp.net/?GoServers [Accessed 23 August 2022]

Wikipedia Contributors (2022). 'Gomoku'. [online] Available at: https://en.wikipedia.org/wiki/Gomoku. [Accessed 23 August 2022]

gomokuonline.com. (n.d.). 'Play Gomoku online'. [online] Available at: https://gomokuonline.com/ [Accessed 23 August 2022]

gomoku.yjyao.com. (n.d.). 'Gomoku'. [online] Available at: https://gomoku.yjyao.com/ [Accessed 23 August 2022]

Stack Overflow (2022). 'Developer Survey'. [online] Available at: https://survey.stackoverflow.co/2022/#most-popular-technologies-language-learn [Accessed 23 August 2022]

Campbell, S (2022) 'Flask vs Django: What's the Difference Between Flask & Django?' [online] Available at: https://www.guru99.com/flask-vs-django.html [Accessed 23 August 2022]

Robinson, S (2017). 'Flask vs Django', *Stack Abuse*, 27 September 2017. [online] Available at: https://stackabuse.com/flask-vs-django/ [Accessed 23 August 2022]

djangoproject.com (n.d.). 'Models'. [online] Available at: https://docs.djangoproject.com/en/4.0/topics/db/models/ [Accessed 23 August 2022]

Parker, C (2017). 'Tutorial: Create a real-time web game with Django Channels and React'. [online] Available at: https://codyparker.com/django-channels-with-react/8/ [Accessed 23 August 2022]

djangoproject.com (n.d.). 'Writing your first Django app'. [online] Available at: https://docs.djangoproject.com/en/4.0/intro/tutorial01/ [Accessed 23 August 2022]

Portella, L (2021). 'Django Essential Training', *Linkedin Learning* 23 September 2021. [online] Available at: https://www.linkedin.com/learning/django-essential-training/ [Accessed 23 August 2022]

Clark, T. H (2022), 'Python Miniproject: Making the Game of Go from Scratch in PyGame', *Medium* [online] Available at: https://towardsdatascience.com/python-miniproject-making-the-game-of-go-from-scratch-in-pygame-d94f406d4944 [Accessed 23 August 2022]

Clark, T. H (2022), 'gogame', GitHub. [online] Available at: https://github.com/thomashikaru/gogame [Accessed 23 August 2022]

Lung, L-H (2020), 'How To Make A Go Board With CSS', *Medium* [online] Available at: https://levelup.gitconnected.com/how-to-make-a-go-board-with-css-ac4cba7d0b72 [Accessed 23 August 2022]

viethoang (2017), 'Go (Game) 9x9', *Codepen* [online] Available at: https://codepen.io/viethoang012/pen/ygzZaB [Accessed 23 August 2022]

saaspegasus.com (2021) 'Organizing your Front-End Codebase in a Django Project' *SaaS Pegasus*, June 2021. [online] Available at:

https://www.saaspegasus.com/guides/modern-javascript-for-django-developers/client-server-architectures/ [Accessed 23 August 2022]

Wedd, N (n.d.) 'Presentation of ascii Go diagrams' [online] Available at: http://www.weddslist.com/goban/ [Accessed 23 August 2022]

djangoproject.com (n.d.). 'Working with forms'. [online] Available at: https://docs.djangoproject.com/en/4.0/topics/forms/ [Accessed 23 August 2022]

Winston, P. H (2010) 'Search: Games, Minimax, and Alpha-Beta', MIT Open Courseware, [online] Available at: https://www.youtube.com/watch?v=STjW3eH0Cik [Accessed 23 August 2022]

Jain, R (2017) 'Minimax Algorithm with Alpha-beta pruning', *hackerearth*, March 31 2017. [online] Available at: https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/ [Accessed 23 August 2022]

Lane, W (2021) 'Writing a Binary Search Tree in Python with Examples', *boot.dev*, 20 July 2021. [online] Available at: https://blog.boot.dev/computer-science/binary-search-tree-in-python/ [Accessed 23 August 2022]

Thagana, K (2022). 'Dockerizing a Django app', *LogRocket*, April 8 2022. [online] Available at: https://blog.logrocket.com/dockerizing-django-app/ [Accessed 23 August 2022]

Docker (n.d.). 'Quickstart: Compose and Django' [online] Available at: https://docs.docker.com/samples/django/ [Accessed 23 August 2022]

Docker (n.d.). 'Get Docker', [online] Available at: https://docs.docker.com/get-docker/ [Accessed 23 August 2022]

Crack Concepts (2019). ‘Alpha beta pruning in artificial intelligence with example.’ Available at: https://www.youtube.com/watch?v=_i-lZcbWkps . [Accessed 26 August 2022]

Wikipedia Contributors (2022). Big O Notation. [online] Available at: https://en.wikipedia.org/wiki/ Big_O_notation. [Accessed 25 August 2022]

Megalooikonomou, V (2003). Artificial Intelligence course notes, Lecture 8. [online] Available at: https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html. [Accessed 25 August 2022]

Serrão, R. G (2021) 'Minimax algorithm and alpha-beta pruning', *Mathspp* [online] Available at: https://mathspp.com/blog/minimax-algorithm-and-alpha-beta-pruning [Accessed 23 August 2022]

GeeksforGeeks (2021). 'Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)' *GeeksforGeeks*, 18 August 2021, [online] Available at: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/ [Accessed 23 August 2022]

Christie (2022). 'robogo, commit:054e4e3a1540a5f08fb07f3ebb9f9a2d7c01d7fd'. Github [online] Available at: https://github.com/nchristie/robogo/commit/054e4e3a1540a5f08fb07f3ebb9f9a2d7c01d7fd#diff-525cccac87c344225da6bc8ac769aa34cadac221d1e66ea7b872ad7d6abe85f5L80-L109 [Accessed 23 August 2022]

Christie (2022). 'robogo, commit:ef2090c6e2ef01e543117b80b43c82d6a8fc53eb. Github [online] Available at: https://github.com/nchristie/robogo/blob/ef2090c6e2ef01e543117b80b43c82d6a8fc53eb/games/minimax.py [Accessed 23 August 2022]