BDL – S1960565 – JEET NAVINDGI – CW4

Ruleset from https://www.britgo.org/intro/intro2.html

Basic requirements:

- Two players are able to start a game.
- While the game is active, players must take turns to play.
- Each turn should have a maximum time limit of 30 seconds.
- For each turn, the contract should take player input as a play. If this play is valid, allow the play and update game board accordingly.
- Continue game until two consecutive passes are played.
- Calculate the winner.
- Reset game state.

Advanced Requirements:

- Randomly assign black and white to each player.
- Once a player has taken their turn, notify opposing player that it is their turn.
- Notify players of important developments of the game state.

I will first focus on the basic requirements, and then I will cover the advanced requirements after.

Contracts:
2 Contracts: **GoPlay** and **GoBoard**
GoPlay concerns how the players will interact with the game. This will keep track of the main game state as well as important statistics of the players.
GoBoard concerns the game itself. This class will be able to keep track of the board as well as validate and handle moves made on the board.

Public API of GoBoard: State Variables

- *board* : internal 2D array of type uint8. 2D array size should be 20x20 to encompass the intersections of a 19x19 board. The choice of a 2D array allows us to use (x, y) coordinates to select intersections on the board. Each element of this array (each [x][y] of the board) will have the uint8 value of 0, 1 or 2. Where 0 = no stone, 1 = player 1 stone and 2 = player 2 stone at those coordinates.

Public API of GoBoard: Functions

- *constructor()* : Constructor which creates an empty board, i.e., a 2D array filled with 0's.

- *getStoneAt(uint256 x, uint256 y, uint8 stone)* : internal view function that returns the uint8 value (0, 1 or 2) at coordinates *x, y* of the board.

- *setStoneAt(uint256 x, uint256 y, uint8 stone)* : internal function that changes the value at coordinates *x, y* of the board to the *stone* value. (*stone* should be 0, 1 or 2, but no check is required because this is an internal function, and none of my functions will use this improperly; this in turn saves gas!).

- *checkPlay(uint256 x, uint256 y, uint8 stone)* : private view function (only ever called by *setStoneAt* function). This should check that placing the *stone* at *x, y* is a valid play and complies with the rules of the game. So, it should check that *x, y* is actually on the board, there is no self-capture, and it complies with the ko rule (in which case true should be returned, otherwise false). This function should return a Boolean value, true for valid play and false for invalid play.

- *captureStones()* : an internal function that captures (removes/sets to 0) stones from the board that should be captured. This function will be called after the *setStoneAt* function, when a stone is placed, but nothing else is updated yet. This function should check if there are any such stones yet to be captured and removes them (using *setStoneAt* with *stone* value 0). This function should return the number of stones removed as a uint256. Because of the self-capture rule, we can always assume that captured stones will be of the opposing players stones.

- *gameEndCheck()* : an internal function that checks whether two consecutive passes have been played. If so, then this function should return true, else false.

- *calculateTerritories()* : an internal function that calculates and returns the amount of surrounded (empty) territory for each player (after removal of dead strings). Returns an array of length 2 containing the value for player 1 and player 2 respectively. This is only called when the game is over (after the above function returns true).

- *resetBoard()* : an internal function which resets the board by setting all elements of the *board* to 0, i.e., no stones on the board.

Public API of GoPlay - GoPlay is GoBoard (inheritance): State Variables

- *players* : internal array of addresses. Size should be 2. Holds the two current players' addresses. address(0) should be used when there is no player. From here on I will refer to player 1 as *player[0]* and player 2 as *player[1]*.

- *capturedStones* : internal 2D array of type uint256. Array size should be 2 (for the 2 players).
  capturedStones[0] = number of stones captured by player 1.
  capturedStones[1] = number of stones captured by player 2.

- *withdrawables :* mapping from address to uint256. Stores the withdrawable 'winnings' of players. This allows for a push-pull mechanism.

- *gameState* : internal enum. Values: *gameOver, game Open, gameFull, gameEnding*. Represents state of the contract/game. Where:
  - *gameOver* – default state. When we are ready for a new game.
  - *gameOpen* – after execution of the *openGame* function (explained later) has been successful.
  - *gameFull* – after execution of the *joinGame* function (explained later) has been successful.
  - *gameEnding* – when two consecutive passes have been made, the game is over, but the winner is yet to be determined. No plays can be made.

- *itsPlayer1sTurn* – internal Boolean which is true if it is player 1's turn or false if it is payer 2's turn. Note, default value will be false, so player 2 will start with only basic requirements met.

Public API of GoPlay : Events

- *Winner(address winner, uint8[][] board)* : emitted when the winner has been decided. This allows the players to find out who won as well as how the *board* looked like in the end.

Public API of GoPlay - GoPlay is GoBoard (inheritance): Functions

- *openGame()* : public function that allows a player to join the game as player 1.
  - **Checks** – *gameState* should be *gameOver*.
  - **Checks** – 0.01 ether should be passed in for contract profit and to minimize disruptive players.
  - **Effects** – *players[0]* should be updated to callers address.
  - **Effects** – *gameState* should be updated to *gameOpen.*

- *joinGame()* : public function that allows a player to join the game as player 2.
  - **Checks** – *gameState* should be *gameOpen.*
  - **Note** – no check needed on player being the same player who opened the game as this poses no security implications (and saves gas!).
  - **Checks** - 0.01 ether should be passed in for contract profit and to minimize disruptive players.
  - **Effects** – *players[1]* should be updated to callers address.
  - **Effects** – *gameState* should be updated to *gameFull*.

- *cleanUp()* : private function that resets the game by defaulting all the values. No checks required as it is a private function. This should reset the *players, capturedStones, gameState* and *itsPlayer1sTurn* to default values. This should also call the *resetBoard* function to reset the *board*.

- *play(uint256 x, uint256 y)* : public function that allows a player to play their turn by placing a stone on the board (aka adding a *1* to the *board* array if *itsPlayer1sTurn* else adding 2).
  - **Checks** – *gameState* should be *gameFull*.
  - **Checks** – the address of the caller should be present in the *players* array.
  - **Checks** – if caller is player 1, *itsPlayer1sTurn* should be true, else it should be false.
  - **Checks** – checks if the play complies with the rules of Go using the *checkPlay* function passing in *x, y,* and the stone based on who's turn it is. This returns a boolean value. This boolean value must be true.
  - **Effects** – update the *board* using *setStoneAt* with the coordinates *x*, *y* and the stone based on who's turn it is.
  - **Effects** – capture any stones that should be captures as a result of the play made in the previous step. Do this using the *captureStones* function. This will return an integer value of the number of stones that were captured. This value should be added onto the respective callers count of captured stones in the *capturedStones* array.
  - **Effects** – update the boolean value *itsPlayer1sTurn* to the opposite value.

- *pass()* : a public function that allows a player to play their turn by passing (aka conceding 1 stone without placing anything on the board).
  - **Checks -** *gameState* should be *gameFull*.

    - **Checks** – the address of the caller should be present in the *players* array.
    - **Checks** – if caller is player 1, *itsPlayer1sTurn* should be true, else it should be false.
    - **Effects** – the number of captured stones of the opposing player should be incremented by 1.
    - **Effects** – using the *gameEndCheck* function, check whether 2 consecutive passes have been made. If *gameEndCheck* returns true, update *gameState* to *gameEnding*.
    - **Effects** – if *gameState* is *gameEnding*, use the *calculateTerritories* function as well as the values in *capturedStones* (remember to add komi points) to determine the winner. Return the winners funds into their corresponding entry in the *withdrawables* mapping. Return half the losers funds into their corresponding entry too. Emit the *Winner* event passing in the address of the winner, and the *board*.
    - **Effects** – if *gameState* is *gameEnding*, call the *cleanUp* function.

- *quit()* : a public function that allows player 1 to quit the game in case they've had enough of waiting for a player 2 to join (or some other valid reason).
    - **Checks** – *gameState* should be *gameOpen.*
    - **Checks** – caller should be player 1.
    - **Effects** – return all funds back to player 1 via *withdrawables* mapping.
    - **Effects** – change *players[0]* and *gameState* to default values.

- *Withdraw()* : public function that allows previous players to withdraw funds.
    - ***Checks*** – callers address must not be present in the *players* array at the time of the call.
    - **Effects** – set the callers withdrawable value to 0.
    - **Interactions** – make the transfer.

- *Timeout functions*. I leave this for the coder. Once it is a players turn, they should have 30 seconds to make their play, otherwise, it becomes an automatic pass. If this happens 2 times in a row for the same player, then the game ends with the other player winning. The idle player will lose their eth while the winning player gets their eth back. As before this should emit the *Winner* event passing in the address of the winner. This should also set up for a new game.

The above API's makes a functioning game of Go. However, there are a few security and convenience issues. Firstly, I will go over some of the decisions I have made in the above API's.

Player commitments and rewards:
I have decided to make players pay 0.01 ether to play the game. Doing so means that the contract now has a source of profit. This will also divert malicious users; if the game were free to play, malicious users could come in and play unethically for free, whereas now there is a penalty. In the end, the winner of the game gets their 0.01 ether back. If the loser interacts as expected till the end of the game, they will receive 0.005 eth back. If however, the user idles for too long (avoiding calling anymore functions), they will automatically lose the game and lose their 0.01 eth completely. This way, players in a losing position are encouraged to finish a game to at least get some of their funds back. I use a push pull mechanism for rewards to increase gas fairness; players will withdraw their own rewards using the withdraw function.

Handling idling players:
An unethical player might stop interacting with the game because they feel they have lost, and they don't want to waste eth on gas for the functions. I have explained above how we handle this, but here I will go into more detail. I have enforced a maximum time of 30 seconds per turn. If a user exceeds this time limit for 3 turns in a row, they will be considered idle and will automatically lose the game along with their funds. If they were to continue playing until the end of the game, they would receive half of their funds back.

How players interact with the game's contract to make a move:
The GoPlay contract is used by the players to interact with the game. Functions pass and play are used to make a move.

Here are a few issues with the above:

Who gets to start the game:
For simplicity, the player who opens the game is player 1 and the player who joins the game is player 2. My current specification makes it so that player 2 always starts. This isn't fair because player 2 will always have the advantage. My advanced requirements section will cover this. We can use a commit reveal scheme to randomize the process.

Player notification for game updates:
Because players are on a time limit each turn, it would be nice to know when it actually is their turn. In the above API's, a player will not be able to know when the other player has played, nor will they know what their play was! It would be very hard for a player to know what is happening in the game. More events will fix this. We should emit an event when any of the following happens:

- A player has played
- A player has passed
- A player has quit
- A player has joined
- A player has won

Now, players can check for these events to know what is happening!
Additionally, if it was a play, it would be nice (and essential) for the player to know how the updated board looks like. For this we should add a getter function that shows the 2D array of the board. To save gas, I will decide that we will not do any formatting to make it look like a real board, instead players are encouraged to do this offline (maybe this makes the 30 second time limit too harsh, but the idea of some time limit is still essential).

Two malicious players causing a game running forever:
If two players get in a game and never make two pass moves in a row, the contract would be forever stuck in the *gameFull* game state. To counter this we must set a time limit on the whole game. For this example, we can set the time limit to 10 minutes.

Fixing these issues:
For the notification issue, we just need to add events to the GoPlay API as well as say when they are emitted. Now players can log these event being emitted and know what is happening.

- *Played(address player, uint8[][] board)* : emitted at the end of *play* function.
- *Passed(address player)* : emitted at the end of the *pass* function.
- *Quitted(address player)* : emitted at the end of the *quit* function.

- *Joined(address player)* : emitted at the end of the *joinGame* function.
- *Opened(address player)* : emitted at the end of the *openGame* function.
- *Won(address player, uint8[][] board, uint256 player1points, uint256 player2points)* : emitted at the end of the *pass* function (for case when there is a winner)

I also said I would add a getter for the board. The following should be added to GoPlay API:

- *getBoard()* : public view function that returns the 2D array *board* (no formatting).

For the issue of who gets to start the game, we should implement a commit – reveal scheme. For this we need to make small changes to the functions *openGame* and *joinGame*. Specifically, we should require players to commit a random value (hashed offline) when calling these functions. We would then require a new game state where we are waiting for both players to reveal their random value via a new function called *reveal*. Once both have revealed, we could xor both values, take it to the modulus of 6, and say that if the value is ≤ 3 then player 1 goes first.
To implement this, we need to make the following changes/additions to the GoPlay API:

- State variable *gameState* should have a new possible value called *gameWaiting,* which will be for when both players are committed into the game but haven't revealed yet.
- New state variable *commitments* is needed which will be a mapping from addresses to string.
- Functions *play* and *pass* should take in an additional argument (*string commitment*). These should be stored in the *commitments* mapping.
- New function to set a time limit for revealing stage, to make it so that players can't freeze the state of the contract in this state. Players who do not reveal after a certain amount of time will lose their eth and will lose the game. The Other player will win.
- New function : *reveal(string rand).* Allows players to reveal their commitment. Once both players have revealed, and the starting player has been decided, the game state should update to *gameFull.*

The implementation of the game time limit is tricky because it cannot be done automatically. My way around this would be to create a function *expireGame()* which can only be called by the owner if 10 minutes have passed since the game has entered the *gameFull* state. The *expireGame()* function should terminate the game normally (by calculating the winner), rewarding normally and resetting. The reason I choose not to take the funds away from these players is because it is difficult to tell whether they are malicious or not (just from the time limit running out). They might just have had a long game. We should set the time limit to a suitable amount so that it isn't too much such that users would need to wait so long to get in a game, and that it isn't too less such that players can't finish their games properly, this is a trade-off. Note: All the time limits I have used in this assignment (30 seconds/10 minutes) may not be suitable; I have no experience of playing Go, so these may need to be altered.

Comment on gas fairness vs efficiency:
If a player makes a pass move, and this happened to be the game ending move, that player will require to pay the gas to completely reset the entire game. This isn't very good in terms of gas fairness. A way to improve this is to make functions that each player would need to call separately to reset their portion of the game (i.e., player 1 would reset *players[0], capturedStones[0], etc…*). This is just a trade-off. I have tried to make the rest of the game as fair as possible in terms of gas; each player calls the same functions the same number of times (or roughly).

Coder decisions:

In my API, I have implied using the checks-effects-interaction pattern in the *withdraw* function. If the coder decides to stick to this pattern, a good idea would be to use *call* instead of *transfer* or *send*. This is because *call* has no gas limit, so it would never fail due to gas running out. *Transfer* and *send* do have gas limits (2300), so they are vulnerable to fail, however because of this capped gas limit, re-entrancy attacks are impossible. With *call*, re-entrancy attacks are possible, so it is important to protect against this, and we have, by encouraging a checks-effects-interaction pattern.

DoS : Griefing:

We have used a pull-over-push architecture to pay, so there is no DoS griefing attack possible. This way, a failed transfer (or call in my case) doesn't affect the state of the contract since it is outside and away from any important code regarding the game.

DoS : Unbounded operation

Loops over arrays cost more gas as the array increases. This is because there is more code to execute (the same line gets executed more and more). If the array becomes too large, then it may become impossible (beyond gas limits) to execute the code containing the loop. In my API, there is no obvious use of this, however the coder will need to make sure to avoid such operations if they are needed and the coder will need to use an implementation that doesn't look efficient if such a situation arises. This is a trade-off, security for gas efficiency.

Forcibly Sending Ether to the contract:

Since there are no strict equality checks w.r.t. the contracts balance, anyone can send ether to my contract without any security concerns.

Delegate calls

If the coder decides to use libraries, and they decide to use delegate calls, they must make sure to check the libraries code thoroughly to make sure that there is no code that can overwrite important aspects of the contract. Since delegate calls run the libraries code in our contracts context, this is a possibility the coder must be weary of. Also, if the coder decides to use tx.origin, they must do so in a safe way. Coder should understand how tx.origin and msg.sender changes as it goes through multiple contracts.

Front running

We are safe against any front running attack as we have employed a cryptographic commitment scheme at the start of all games, which helped us randomly assign black and white to the players.