Comprehensive Exercise Report: Connect Four

Team administration of Section ?

Balasuriyage Aritha Dewnith Kumarasinghe 203AEB014
Mio Tabayashi 201ADB057
Kohei Miyamoto 191ADB103

Requirements/Analysis

Week 2

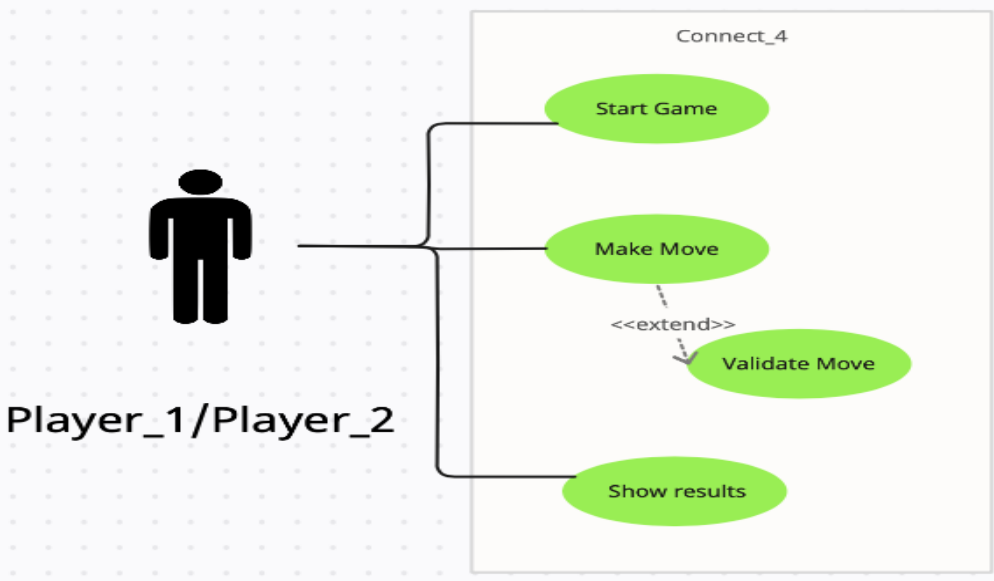
Journal

- After reading the client's brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
    - Connect 4 consists of a grid that is 7 across by 6 down that should be filled by discs of two different colors one for each player until either a straight line of four discs of the same color are created in any orientation or both players run out of the 21 discs that are given to them.
        - This game can only be played by two players.
        - The game uses a grid structure that has 42 empty spaces at the start of the game.
        - These spaces are filled using discs of two different colors (red, yellow) by the players in a sequential manner.
        - Each of the players get 21 one discs of a single color.
        - A player wins when they have used four discs given to them to create a line within the grid in any orientation (connecting 4).
        - The game is considered a draw when neither of the players have accomplished the goal of connecting 4 and all 21 one of the discs have been used to fill the grid.
- After reading the client's brief (possibly incomplete description), what questions do you have for the client? Are there any pieces that are unclear? After you have a list of questions, raise your hand and ask the client (your instructor) the questions; make sure to document his/her answers.
    - On a scale of 1 to 10 with one being unimportant, 10 being very important and 5 being indifferent , rate the feature of a player of the game being able to play against a computer
    - Can we as the developers change the rules of the game as we see fit?
- Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.
    - The development of graphic user interface
    - Minimizing consumption of memory space
- Describe the users of this software (e.g., small child, high school teacher who is taking attendance).
    - The potential users of this software will be small kids, teenagers, grownups and family members. We expect people from a variety of age groups will play in our game.
- Describe how each user would interact with the software
    - While the wide variety of user groups is expected, the interaction with users must be intuitive, language-less and real-time. For example, small kids may not understand long descriptions about the game and difficult vocabulary may disturb understanding of game rules. As well, response time of user action should be kept to a minimum for reducing the struggle of users towards the game.
- What features must the software have? What should the users be able to do?
    - To play connect the game connect four according to the defined rules.

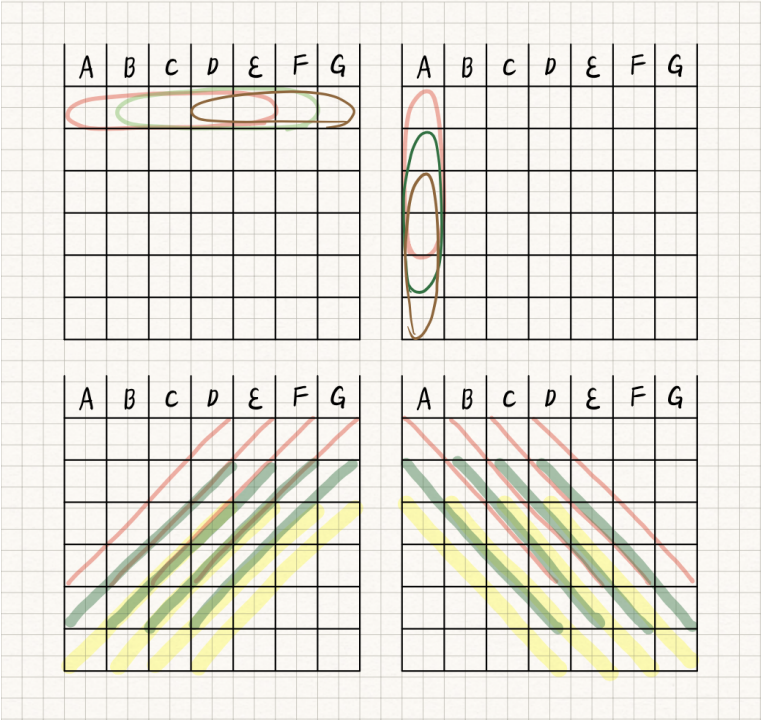
- Other notes:
    - The rules of the game connect four have been defined according to the instructions given on the official hasbro website (link).

Use Case

Two people are in a room together and wish to spend time playing a simple board game. They have the option of playing a computerized version of connect four on their computer. The use case diagram is as follows:



Possible connection of 4:



Prototype:

As of 2023/03/04,
In order to make sure that the created requirements, the prototype without GUI was created.
https://github.com/koheitech/rtu_ass_connect-four/blob/main/connect4_prototype.py

API specification:
Input: Column (either A, B, C, D, E, F, G)
Output: game board matrix in 2D array (so that GUI can display)

SRS: Software requirements

- Functional
    - Receive input
    - Fill in the slot
    - Check the winning condition
    - Display the winner
- Nonfunctional
    - English
    - Can be executed in Windows, MacOS and Linux
    - Can be executed without the Internet connection

URS: User requirements

- User can choose several play modes
    - vs CPU
    - vs AI
    - vs Other user
    - User can play using GUI (button)
    - User can watch the state of the game after each choice

SysRS

- Python 3.10.6
- GUI: PyQt5

Software Requirements

Project Description

Design
Define the user requirements for the computerized game (Connect 4) that will be developed within the project.
Design the system using software modeling language.
Define the technical requirements of the system.
Finalize the software requirements within a SRS.

Build
Define the tasks that are needed to fulfill the requirements of this project..
Select a team workflow management tool.
Assign tasks to the team members.
Complete tasks.

Test
Identify the errors/bugs within the software and fix them.
Verify that all defined requirements have been met by the system.

Project Overview
Develop a computerized version of the board game called Connect 4 according to the requirements that are defined inorder to receive the highest possible grade for the course Applied System Software(DIP392).

Requirements

1.0 The software will be a computerized version of the board game Connect 4.
1.2 The software must be installable.
1.3 The software must have a Graphical User Interface.
1.6.1 The software will have a game rule implementation(GRI) mechanism.
1.6.2 The GRI will use a game state matrix in order to keep track of the moves made.
1.6.3 The GRI will analyze the game state matrix for 4 slots filled in the same color and decide the winner based on the color.
1.4 The game within the software must be able to be played between two users, or between a user and the computer.
1.5.1 The game within the software can only have one player or the computer start by making a move then follow a sequential manner of where one player after the other makes moves.
1.5.2 The game will only allow each of the players to make a maximum of 21 moves.
1.5.3 The moves will consist of the players filling the slots within the grid with 21 tokens of a specific color(either blue or orange)given to them at the start of the game.
1.3.1 The GUI within the software will consist of a grid with 42 slots (7 across by 6 down) with all slots being empty at the start of the game.

1.3.2 The GUI within the software must allow each player to fill one of the slots within the grid for each of their turns.
1.3.3 The GUI within the software must announce the winner of the game based on the results produced by the GRI.

Black-Box Testing

Journal

Remember: Black box tests should only be based on your requirements and should work independent of design.

The following prompts are meant to aid your thought process as you complete the black box testing portion of this exercise. Please review your list of requirements and respond to each of the prompts below. Feel free to add additional notes.

- What does input for the software look like (e.g., what type of data, how many pieces of data)?
    - String (specified coordinate of the game board)
- What does output for the software look like (e.g., what type of data, how many pieces of data)?
    - Visual change of the software (movement of both user and computer)
- What equivalence classes can the input be broken into?
    - 7 classes (variety) of input exist for the possible input that system expects to take
- What boundary values exist for the input?
    - The input value is expected to be upper letter alphabet from A to G
- Are there other cases that must be tested to test all requirements?
    - Appropriate feature of the graphic user interface
    - Does the CPU make the correct movement for the game play?
    - Does the algorithm recognise the finish condition of the game correctly?
- Other notes:
    - Versioning started

Black-box Test Cases

Use your notes from above to complete the black-box test plan section of the formal documentation by writing black box test cases (other than actual results since no program currently exists). Remember to test each equivalence class, boundary value, and requirement.

Test ID	**Description**	**Expected Results**	**Actual Results**
1.1	(vs person mode) If the user inputs the column to put the pack, it should insert to the bottom of the row based on the gravity factor, then the other player adds another pack in the same manner.	The visualization of the game board with 2 newly added packs.	Success The implemented GUI is enforced to insert only at the bottom part of the pack based on the gravity rule.
1.2	(vs AI mode) If the user inputs the column to put the pack, it should insert to the bottom of the row based on the gravity factor, then the AI adds another pack in the same manner.	The visualization of the game board with 2 newly added packs.	Success The implemented GUI is enforced to insert only at the bottom part of the pack based on the gravity rule.
1.3	The player turn should come once in every two turns	User adds the pack onto an odd number of packs or even number of packs depending on the turn.	Success
1.3.1	(vs player mode) The player1 inputs in odd turn	Input happens on the 1st, 3rd, 5th … 2N-1th time	Success
1.3.2	(vs player mode) The player2 inputs in even turn	Input happens on the 2nd, 4th, 6th … 2Nth time	Success
2	If there are 4 sequential packs of the same color in horizontal, vertical or diagonal gamespace, it should stop the game and display who wins the game.	Message saying who is the winner along with the final result of the game board.	Success
3.1	If the user inputs the invalid data, it should	Error message and another input try	- With the implementation

	reject the input and let the user do it again		of GUI, it is not even allowed to input invalid data. GUI is implemented in a way that only valid positions can be clicked.
3.2	If users inputs letters other than column name (A, B, C, D, E, F and G), rejects the input	Error message and another input try	- Likewise, input is only accepted by valid clicks with GUI implementation.
3.3	If the user inputs the column which is already full, rejects the input	Error message and another input try	- Likewise, input is only accepted by valid clicks with GUI implementation.

Design

Instructions: Week 6

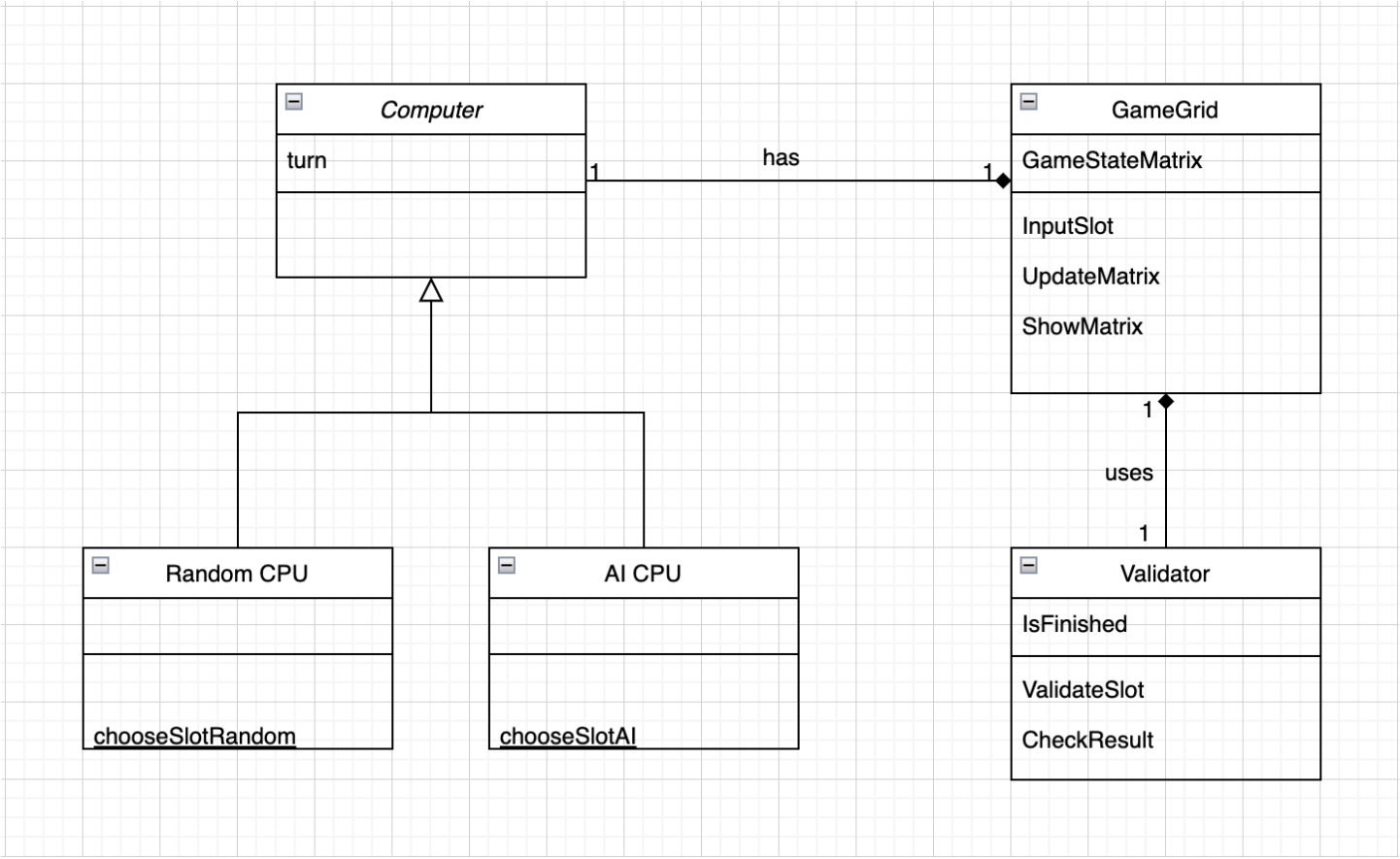
Journal

Remember: You still will not be writing code at this point in the process.

The following prompts are meant to aid your thought process as you complete the design portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- List the nouns from your requirements/analysis documentation.
    - Software
    - Graphical User Interface(GUI)
    - Game Rule Implementation
    - Game state matrix
    - Slots
    - Player
    - Computer
    - Token
    - GameGrid
    - Winner
- Which nouns potentially may represent a class in your design?
    - GameGrid
    - Computer
- Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
    - Game state matrix
    - Winner
- Now that you have a list of possible classes, consider different design options (***lists of classes and attributes***) along with the pros and cons of each. We often do not come up with the best design on our first attempt. Also consider whether any needed classes are missing. These two design options should not be GUI vs. non-GUI; instead you need to include the classes and attributes for each design. Reminder: Each design must include at least two classes that define object types.

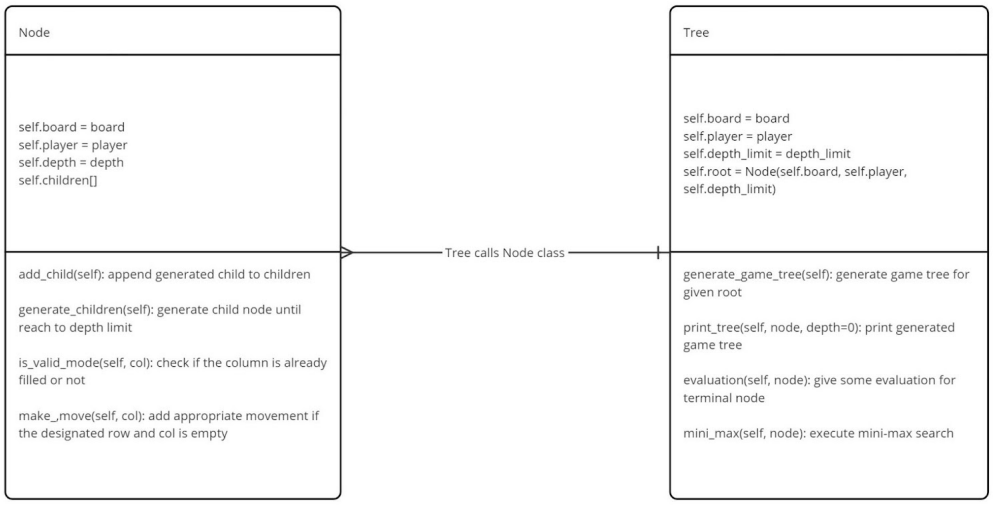
Design1



Pros:
- loosely coupled
- inheritance clearly separates the types of CPU

Cons:
- too many classes
- scattered class requires the methods to be public, which cause the exposure of methods

Design 2: AI part class diagram



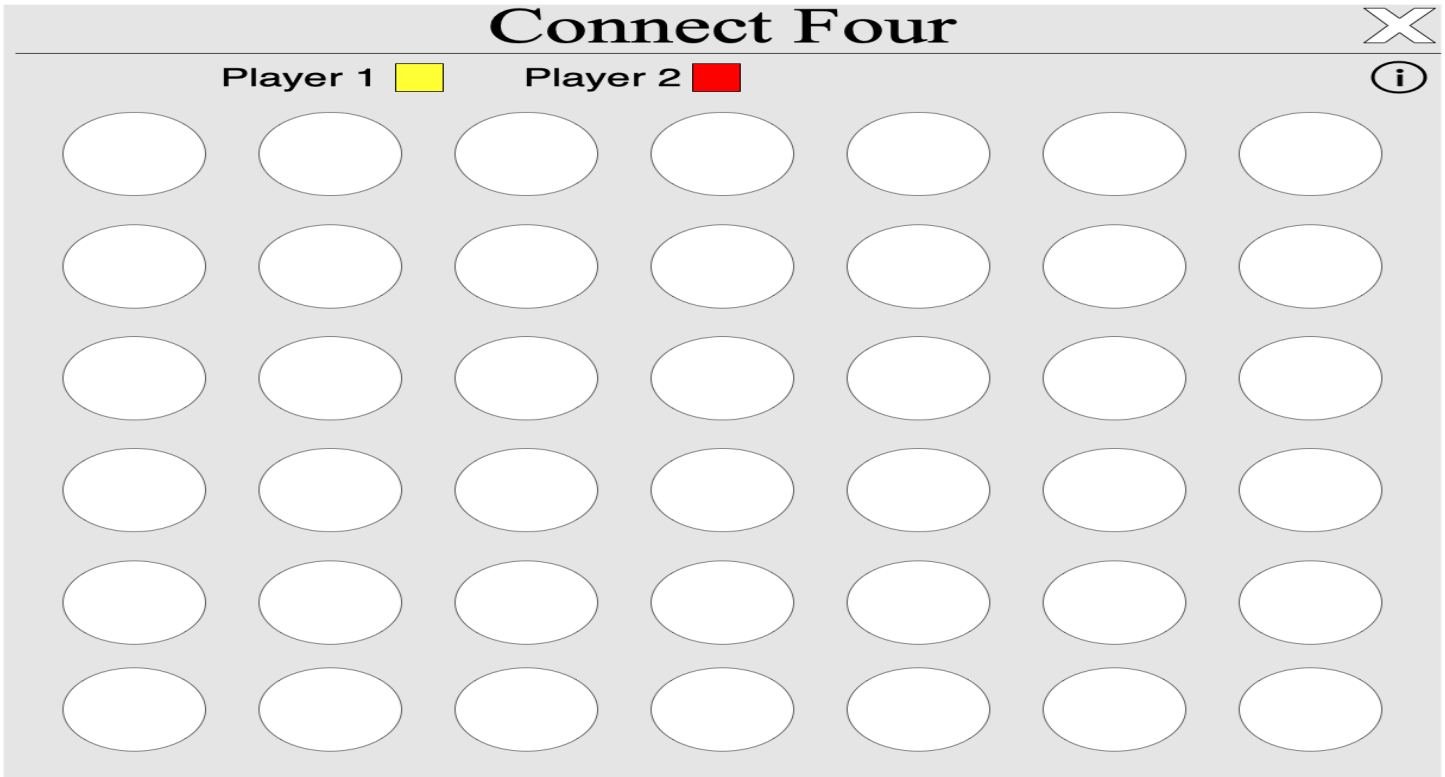
The basic direction is proposed and the basic algorithms are implemented but the integration test with main body has not been done yet

- Which design do you plan to use? Explain why you have chosen this design.
- List the verbs from your requirements/analysis documentation.
    - Play
    - Fill slot
    - Update
    - finish
- Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
    - Choose slots
    - Validate slots
    - Check winner
    - update matrix
- Other notes:
    - Link to AI logic:
      https://colab.research.google.com/drive/1GjIJvOInb3eouo6LAdhV_xsK1Jy_ErgW?usp=sharing

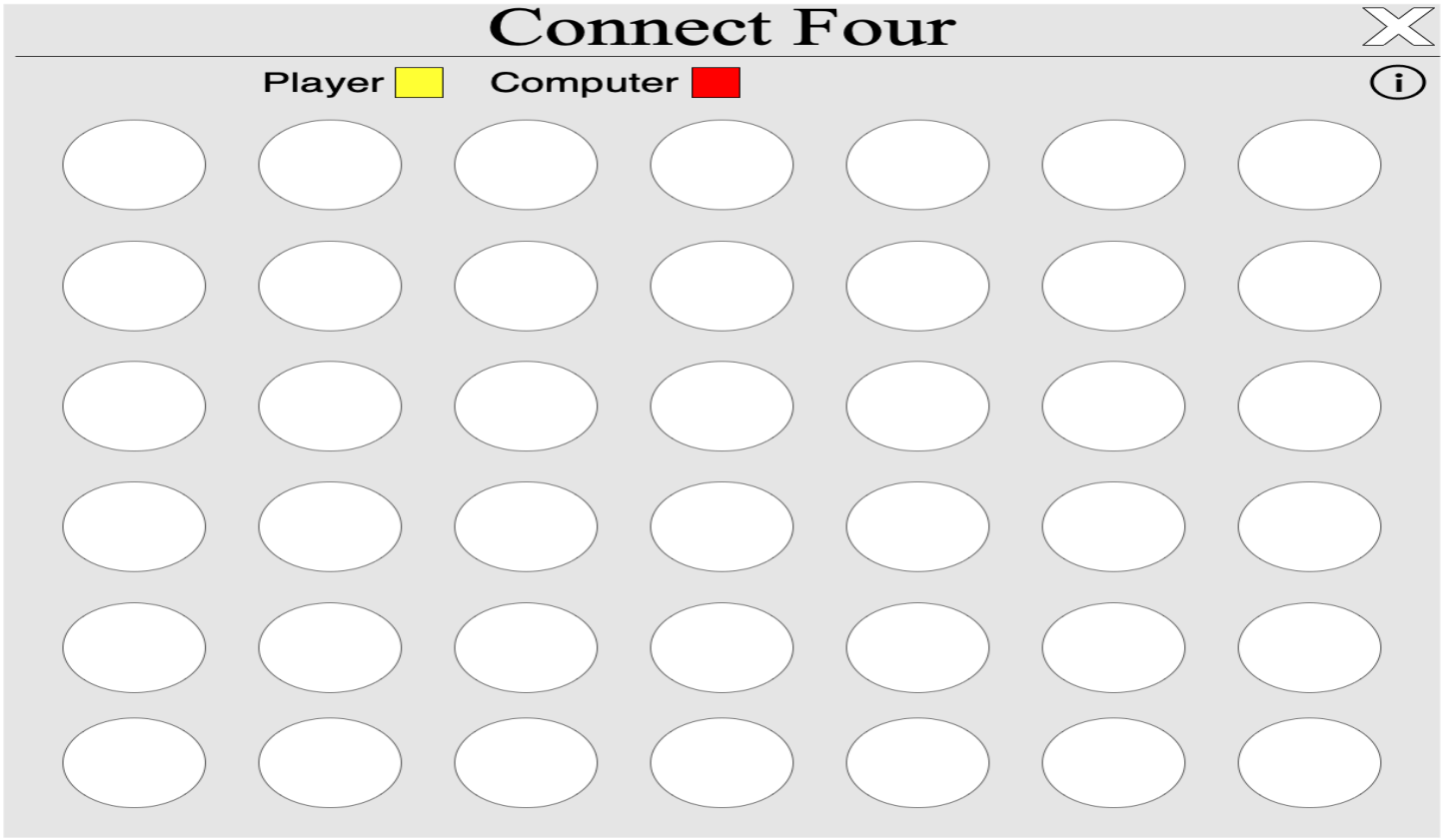
Software Design

WireFrame 1: Start screen of the game:

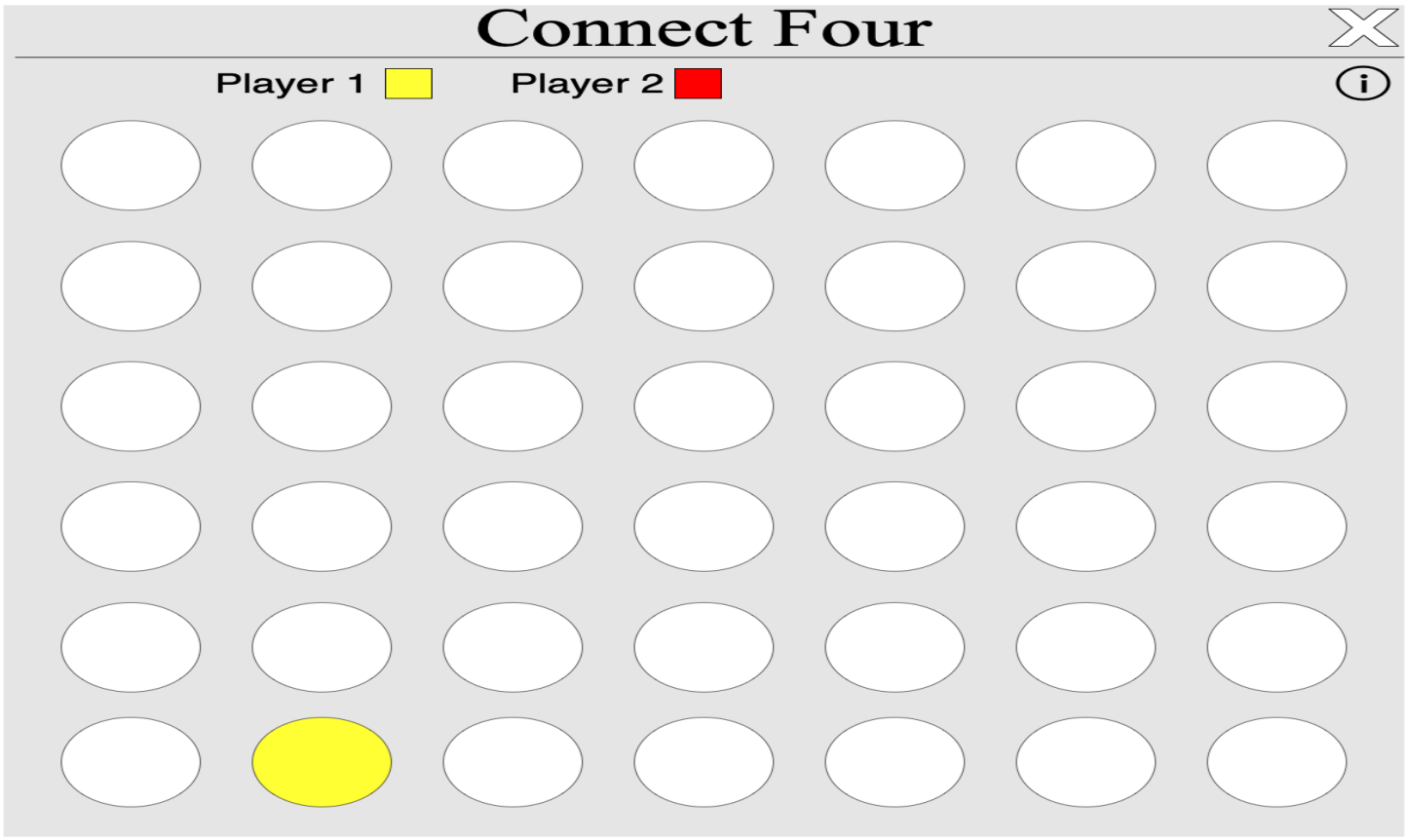
WireFrame 2.1: Player1 vs. Player 2 is selected by user(showing an empty grid with no slots filled):



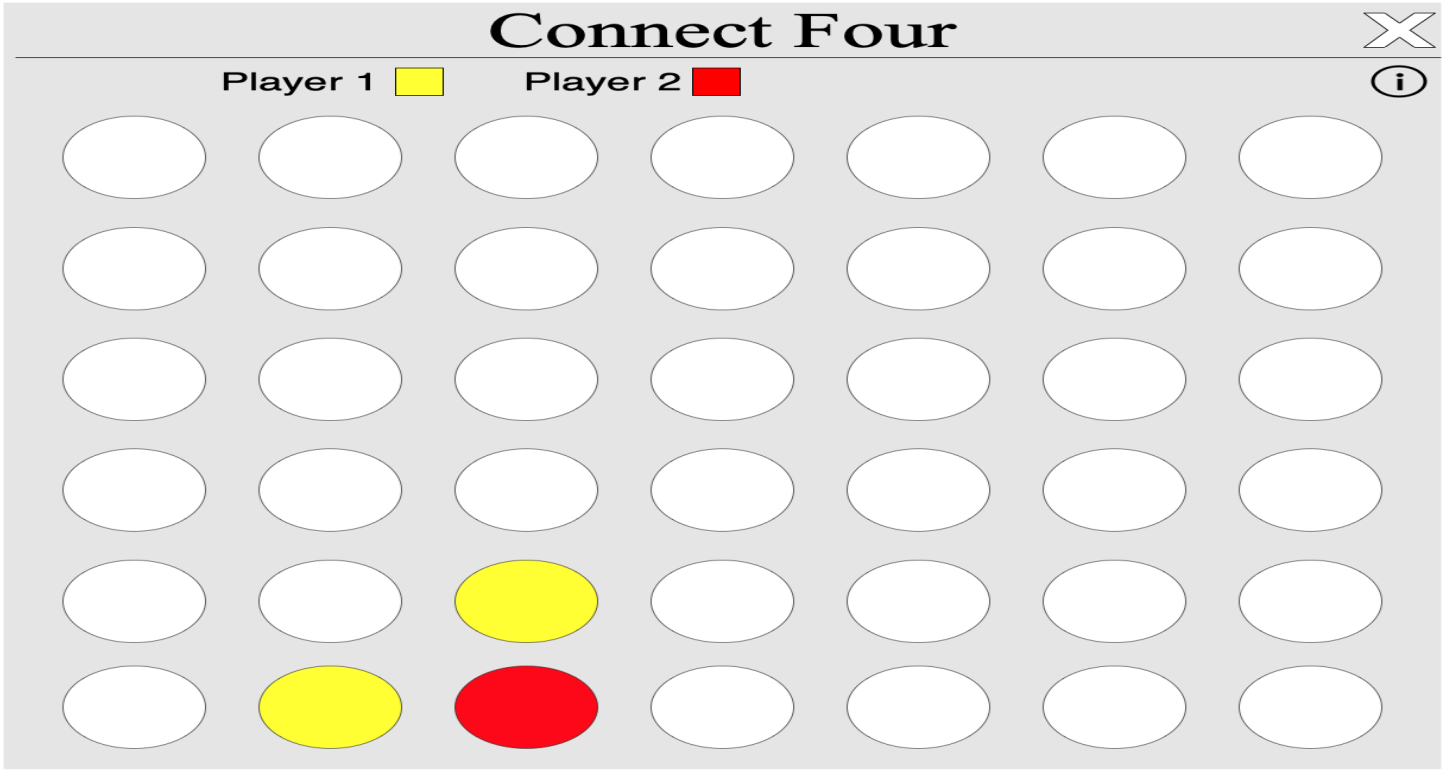
WireFrame 2.2: Player1 vs. Computer is selected by user(showing an empty grid with no slots filled):



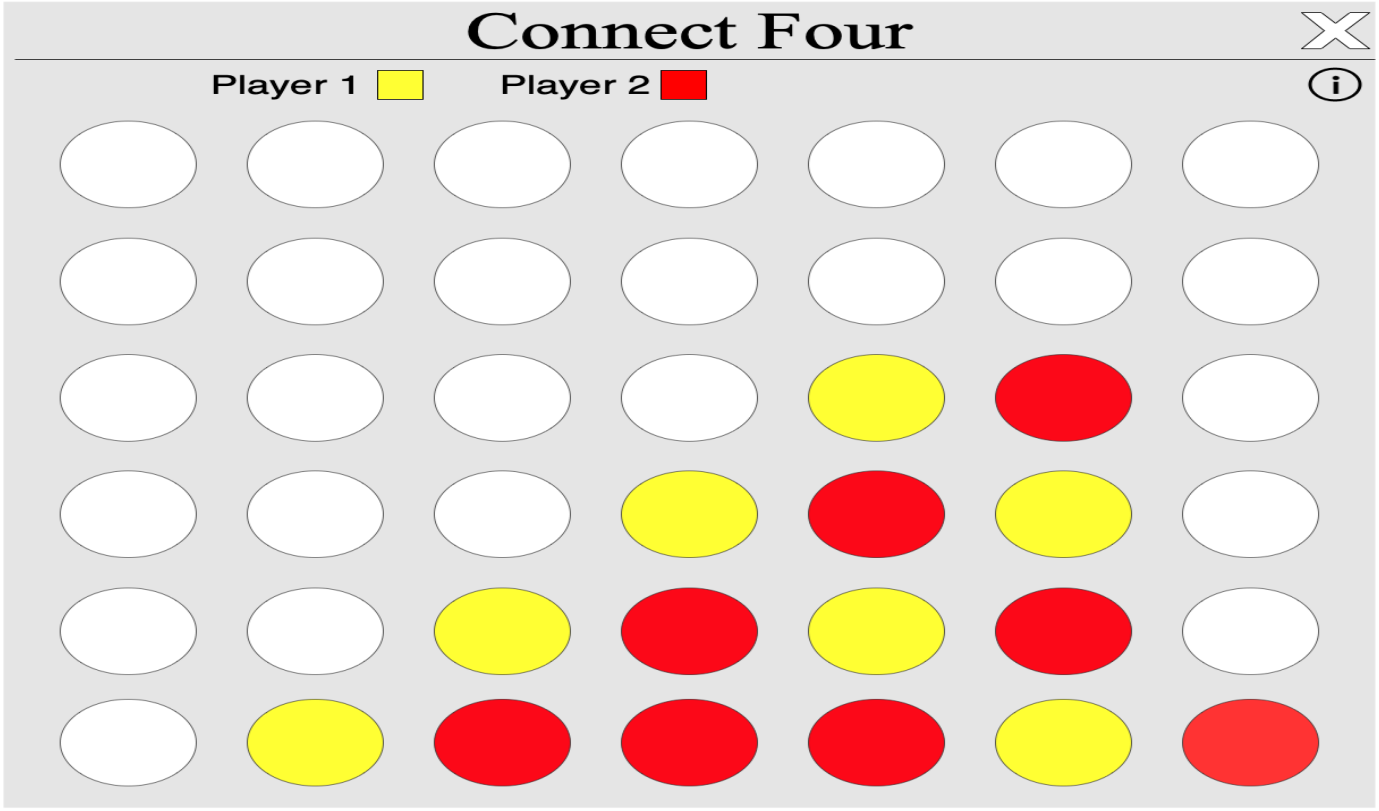
WireFrame 3: Player 1/Player makes a move(clicks on a button on the bottom most row(only)) by clicking one of the slots with their color.



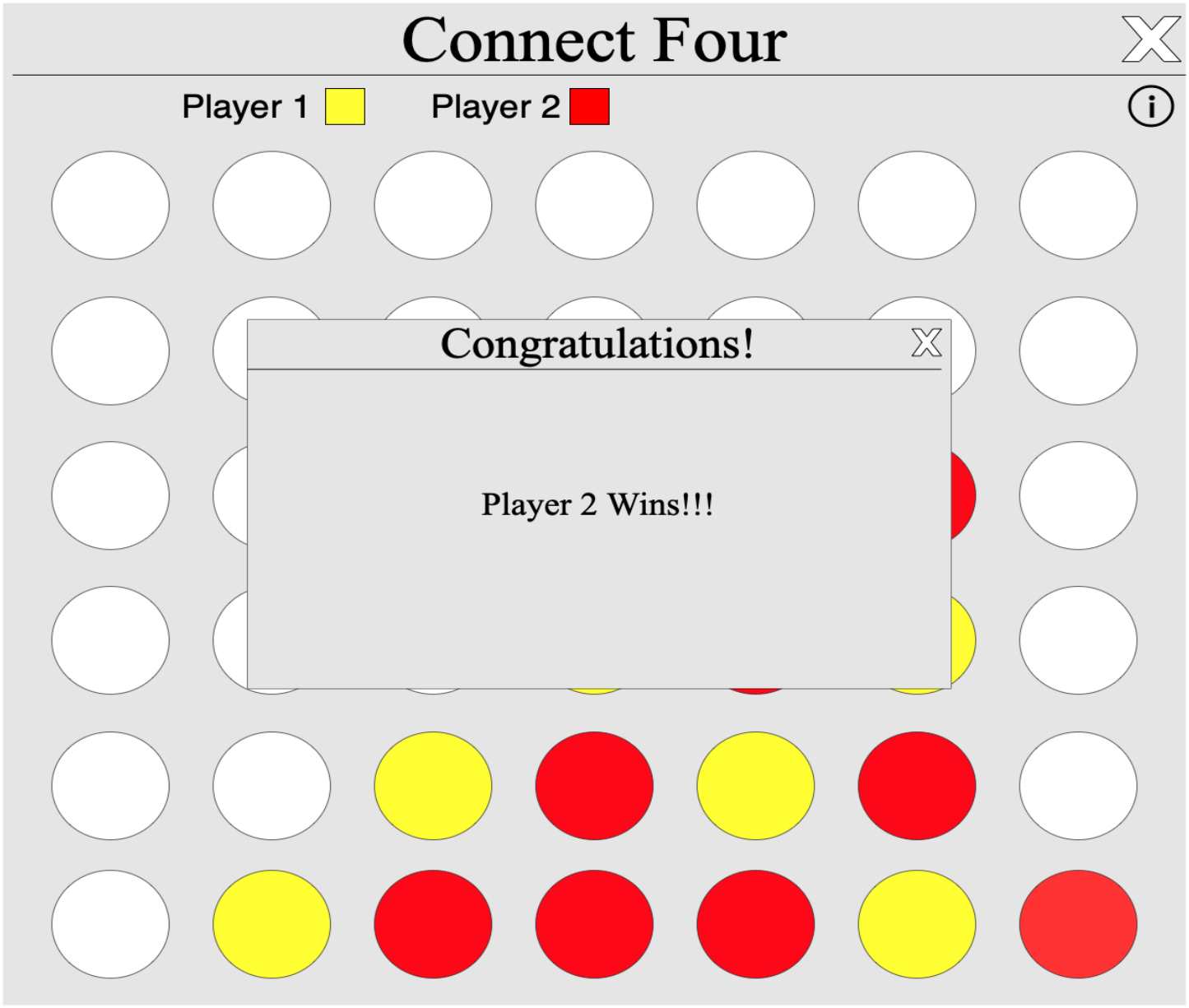
WireFrame 4: Player 1/ Player makes another move on a column that already has some(bottom most row) rows within the same column already filled.



WireFrame 5: Player Two has made the winning move(connect four of the slots using their moves):



WireFrame: Window showing the announcement of the winner(Which in this case is player 2).



Implementation

Instructions: Week 8

Journal

The following prompts are meant to aid your thought process as you complete the implementation portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- What programming concepts from the course will you need to implement your design? Briefly explain how each will be used during implementation.
    - Implementation of AI part

      The AI part of the software aimed to work as a stand alone program file that takes the state of the game and makes predictions from the given game state. In the integration, the AI part is implemented as an object instance that is imported from a separated python file. The game state is given in a 2 dimensional array, the human player's movement is represented by 1, the computer's movement is represented by 2, and the empty state is represented by 0 in the game state array. While the traverse of the full scale game tree is failed, the algorithm repeatedly generates the limited depth of the game tree from the given game state in every computer's turn. It consumes more RAM space than the generation of a single full-scale game tree at the beginning. However, Python has an automatic RAM release functionality and consumption of RAM is not a big problem.

      In GUI part, AI file is called right after the player's movement if the player's movement didn't satisfy the winning condition and makes a move according to the alpha-beta pruning of the generated game tree. The depth of the game tree is changeable from a variable of the object instance and it influences the behavior of AI. In other word, AI makes more rational movement if the depth of game tree gets deeper.

    - Implementation of GUI part
- Other notes:
    - AI code

```
class Node:
    #provides necessary things for generating trees
    def __init__(self, board, player, depth):
        self.board = board
        self.player = player
        self.depth = depth
        self.children = []

    def add_child(self, node):
        self.children.append(node)

    def generate_children(self):
        if self.depth <= 0:
            return

        for col in range(7):
            if self.is_valid_move(col):
```

```
            child = Node(self.make_move(col), not self.player, self.depth - 1)
            self.add_child(child)
            child.generate_children()

    def is_valid_move(self, col):
        return self.board[0][col] == 0

    def make_move(self, col):
        board = [row[:] for row in self.board]
        for row in range(5, -1, -1):
            if board[row][col] == 0:
                board[row][col] = 1 if self.player else 2 #1 for player and 2 for computer
                break
        return board

class Tree:
    #generate tree from Node class
    def __init__(self, board, player, depth_limit):
        self.board = board
        self.player = player
        self.depth_limit = depth_limit
        self.root = Node(self.board, self.player, self.depth_limit)

    def generate_game_tree(self):
        self.root.generate_children()

    def print_tree(self, node, depth=0):
        print(' ' * depth, end='')
        print(f'Player: {1 if node.player else 2}')
        print(' ' * depth, end='')
        print(f'Board: {node.board}')
        for child in node.children:
            self.print_tree(child, depth+1)

    def check_win(self, node):

        for col in range (7): #check vertical
            tmp = [row[col] for row in node.board]
            for i in range(3):
                if tmp[i:i+4].count(2) == 4:
                    return (True, float('inf'))
                elif tmp[i:i+4].count(1) == 4:
                    return (True, -float('inf'))

        for row in node.board: #check horizontal
            for i in range(4):
                if row[i:i+4].count(2) == 4:
                    return (True, float('inf'))
                elif row[i:i+4].count(1) == 4:
                    return (True, -float('inf'))
```

```
    for row in range(5, 2, -1): #check upper right line
        for col in range(4):
            count_P = 0
            count_C = 0

            if node.board[row][col] == 0:
                break

            for i in range(4):
                if node.board[row-i][col+i] == 0:
                    break
                elif node.board[row-i][col+i] == 1:
                    count_P = count_P + 1
                else:
                    count_C = count_C + 1

            if count_C == 4:
                return (True, float('inf'))
            elif count_P == 4:
                return (True, -float('inf'))

    for row in range(5, 2, -1): #check upper left line
        for col in range(3, 7):
            count_P = 0
            count_C = 0

            if node.board[row][col] == 0:
                break

            for i in range(4):
                if node.board[row-i][col-i] == 0:
                    break
                elif node.board[row-i][col-i] == 1:
                    count_P = count_P + 1
                else:
                    count_C = count_C + 1

            if count_C == 4:
                return (True, float('inf'))
            elif count_P == 4:
                return (True, -float('inf'))

    return (False, None)



def evaluation(self, node):
    evaluation_score = 0
    count_P = 0
    count_C = 0

    for col in range (7): #evaluate vertical line
```

```
    tmp = [row[col] for row in node.board]
    for i in range(3):
        zero_count = tmp[i:i+4].count(0)
        two_count = tmp[i:i+4].count(2)
        one_count = tmp[i:i+4].count(1)

        if zero_count + two_count == 4:
            if two_count == 3:
                count_C += 1000
            else:
                count_C += two_count

        elif zero_count + one_count == 4:
            if one_count == 3:
                count_P += 1000
            else:
                count_P += one_count

if count_P < count_C:
    evaluation_score += count_C * 10
elif count_P > count_C:
    evaluation_score += count_P * -10

count_P = 0
count_C = 0

for row in node.board: #evaluate horizontal line
    for i in range(4):
        zero_count = row[i:i+4].count(0)
        two_count = row[i:i+4].count(2)
        one_count = row[i:i+4].count(1)

        if zero_count + two_count == 4:
            if two_count == 3:
                count_C += 1000
            else:
                count_C += two_count

        elif zero_count + one_count == 4:
            if one_count == 3:
                count_P += 1000
            else:
                count_P += one_count

if count_C > count_P:
    evaluation_score += count_C * 10
elif count_C < count_P:
    evaluation_score += count_P * -10

count_P = 0
count_C = 0
```

```
    for row in range(5, 2, -1): #evaluate upper right line
        for col in range(4):
            if node.board[row][col] == 0:
                break

            else:
                for i in range(4):
                    if node.board[row-i][col+i] == 0:
                        break
                    elif node.board[row-i][col+i] == 1:
                        count_P = count_P + 1
                    else:
                        count_C = count_C + 1

    if count_P > count_C:
        evaluation_score += count_P * -2
    elif count_P < count_C:
        evaluation_score += count_C * 2
    else:
        evaluation_score += 0

    count_P = 0
    count_C = 0

    for row in range(5, 2, -1): #evaluate upper left line
        for col in range(3, 7):
            if node.board[row][col] == 0:
                break

            else:
                for i in range(4):
                    if node.board[row-i][col-i] == 0:
                        break
                    elif node.board[row-i][col-i] == 1:
                        count_P = count_P + 1
                    else:
                        count_C = count_C + 1

    if count_P > count_C:
        evaluation_score += count_P * -2
    elif count_P < count_C:
        evaluation_score += count_C * 2
    else:
        evaluation_score += 0

    return evaluation_score

def AlphaBeta_search(self, node, alpha, beta):
    if self.check_win(node)[0]:
        return self.check_win(node)[1]

    if node.depth <= 0:
```

```
        return self.evaluation(node)

    if node.player:
        #minimizing layer
        possible_minimum = float('inf')

        for child in node.children:
            value = self.AlphaBeta_search(child, alpha, beta)
            if value < possible_minimum:
                possible_minimum = value
                if possible_minimum < beta:
                    beta = possible_minimum
            if beta <= alpha:
                break

        return possible_minimum

    else:
        #maximizing layer
        possible_maximum = -float('inf')

        for child in node.children:
            value = self.AlphaBeta_search(child, alpha, beta)
            if value > possible_maximum:
                possible_maximum = value
                if possible_maximum > alpha:
                    alpha = possible_maximum
            if alpha <= beta:
                break

        return possible_maximum

def make_best_move(self, node):
    maximum = -float('inf')
    promise_row = None
    promise_col = None
    alpha = -float('inf')
    beta = float('inf')

    for child in node.children:
        value = self.AlphaBeta_search(child, alpha, beta)
        if maximum <= value:
            maximum = value
            for col in range(7):
                for row in range(5, -1, -1):
                    if node.board[row][col] == child.board[row][col]:
                        continue
                    elif child.board[row][col] == 1:
                        continue
                    else:
                        promise_row = row
                        promise_col = col
```

```
    return (promise_row, promise_col)
```

- GUI code

Implementation Details

Completed README: https://github.com/koheitech/rtu_ass_connect-four/blob/main/README.md

Based on the code above, the implementation is conducted.
First, our implementation is separated into two modules: GUI component and AI logic component.
GUI component is responsible for the overall operation of the connect-4 game, and the AI component is responsible for the algorithm for finding the best path to take for winning the game for the CPU.
Once the two modules are created independently, the integration of two modules are conducted. Since we were agreed upon regarding the requirement and direction of the implementation, two separate modules are implemented in a way that they can be integrated with ease.
However, as always, the integration does not proceed as we expected.
After the integration, the logic of switching turns for each player did not work well. After debugging, it was found out that the action when a button is clicked is assigned multiple times and it makes the function call multiple times. so I assigned the action only once when the button is made.
Throughout the bug fix, the working code is pushed into the github.
https://github.com/koheitech/rtu_ass_connect-four

Testing

Instructions: Week 10

Journal

The following prompts are meant to aid your thought process as you complete the testing portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

- Have you changed any requirements since you completed the black box test plan? If so, list changes below and update your black-box test plan appropriately.
    - Test No. 3.1, 3.2 and 3.3 were designed to catch the invalid input based on the assumption that we would build a CLI-based application. Thus, those test are no longer applied to our final implementation.
- List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.
    - Main class (GUI)
        - Chip cannot be inserted to invalid slots.
        - When the winning condition is met, it displays the winner and result
    - AI class
        - Unit test is conducted.

```
for i, state in enumerate(state_list):
    tree = Tree(state, False, 5)
    tree.generate_game_tree()
    row, col = tree.make_best_move(tree.root)
    print(f"{i}th promising row:{row} and col:{col}")
```

```
0th promising row:2 and col:0
1th promising row:5 and col:3
2th promising row:5 and col:6
3th promising row:2 and col:5
4th promising row:5 and col:2
5th promising row:4 and col:3
```

Testing Details

In summary, the AI module was separately tested by unit testing. Then, the overall application is tested by actually playing the application.
The black-box testing is successfully completed and the table above is filled out.

Presentation

Instructions:Week 12

Preparation

The following prompts are meant to aid your thought process as you complete the presentation portion of this exercise. It is recommended that you examine the previous sections of the journal and your reflections as you work on the presentation as it is likely that you have already answered some of the following prompts elsewhere. Please respond to each of the prompts below and feel free to add additional notes.

- Give a brief description of your final project
    - Connect-4 vs AI
- Describe your requirement assumptions/additions.
  1.0 The software will be a computerized version of the board game Connect 4.
  1.2 The software must be installable.
  1.3 The software must have a Graphical User Interface.
  1.3.1 The GUI within the software will consist of a grid with 42 slots (7 across by 6 down) with all slots being empty at the start of the game.
  1.3.2 The GUI within the software must allow each player to fill one of the slots within the grid for each of their turns.
  1.3.3 The GUI within the software must announce the winner of the game based on the results produced by the GRI.
  1.4 The game within the software must be able to be played between two users, or between a user and the computer.
  1.5.1 The game within the software can only have one player or the computer start by making a move then follow a sequential manner of where one player after the other makes moves.
  1.5.2 The game will only allow each of the players to make a maximum of 21 moves.
  1.5.3 The moves will consist of the players filling the slots within the grid with 21 tokens of a specific color(either blue or orange)given to them at the start of the game.
  1.6.1 The software will have a game rule implementation(GRI) mechanism.
  1.6.2 The GRI will use a game state matrix in order to keep track of the moves made.
  1.6.3 The GRI will analyze the game state matrix for 4 slots filled in the same color and decide the winner based on the color.

- Describe your design options and decisions. How did you weigh the pros and cons of the different designs to make your decision?
    - We decided to go with the GUI-based application over the CLI-based application because GUI offers a more intuitive playing experience of the game, which can contribute to the better user experience.
- Describe your tests (e.g., what you tested, equivalence classes).
    - Blackbox testing for the whole application
    - Unit testing for the AI module
- What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
    - Communication
        - Our daily chats can reduce the miscommunication throughout the project.

    - Architecture for Integration
        - The successful integration depends on the successful architecture of the software.
    - Test it right, sleep well at night
        - The more elaborate test is expected for the product of better quality.
- What functionalities are you going to demo?
    - Play vs AI
    - Play vs Another person
- Who is going to speak about each portion of your presentation? (Recall: Each group will have ten minutes to present their work; minimum length of group presentation is seven minutes. Each student must present for at least two minutes of the presentation.)
    - Kohei Miyamoto