Implementing Sim with Minimax computer player

AQA A-level Computer Science (7517)

Component 3: Non-exam assessment

Exam Series: June 2023

Tsz Yeung CHUI

Candidate Number: 4164

Centre Number: 61679

Table of Contents

Table of Contents 2
Analysis 5
The Problem 5
Research 5
Other current solutions 6
End-user 7
Third parties/other prospective users 7
Interviews 7
Hardware/Software requirements 8
Objectives 8
IDE and Programming Language 9
Modelling of the problem 10
Minimax algorithm 10
Representation of playing area 11
Documented design 12
High-level overview 12
Project timeline 12
Decomposing the problem 12
Class diagrams 13
Algorithms 15
Minimax 15
Evaluation function 17
Test for losing condition 19
Merge sort 20
Data structures 21
Database design and queries 22
User interface 22
Technical Solution 24
List of objectives 24
Techniques used 24
Graphical User Interface (gui.py) 25

__init__ 25
start 26
checkifendgame 28
playgame 28
checkmove 28
end 29
displayplayer 30
humanmove 30
draw 30
drawvertex 31
drawedge 31
updatestatus 31
updatewarningstatus 31
disablebuttons 31
enablebuttons 32
mouseclick 32
getnode 33
highlight 33
unhighlight 33
Minimax algorithm (minimax.py) 40
maxdepth 40
evaluate 40
TestForTriangle 40
TestForUnconnected 41
minimax 41
Game flow algorithm (control.py) 47
initial 47
getturn 47
turnduring 47
check 47
computermove 47
turnend 48

checkend 48
Game object (game.py) 50
__init__ 50
Access methods 50
UpdatePossibleMoves 50
Graph object (graph.py) 52
__init__ 52
TestForTriangle 52
UpdateEdge 52
GetEdgeList 52
Merge sort (sort.py) 54
Testing 56
List of tests 56
Evidence of testing 58
Evaluation 69
Evaluation against each objective 69
Feedback by end-user 73
Improvements 73
References 74
Appendix A: cli.py 75

Analysis

The Problem

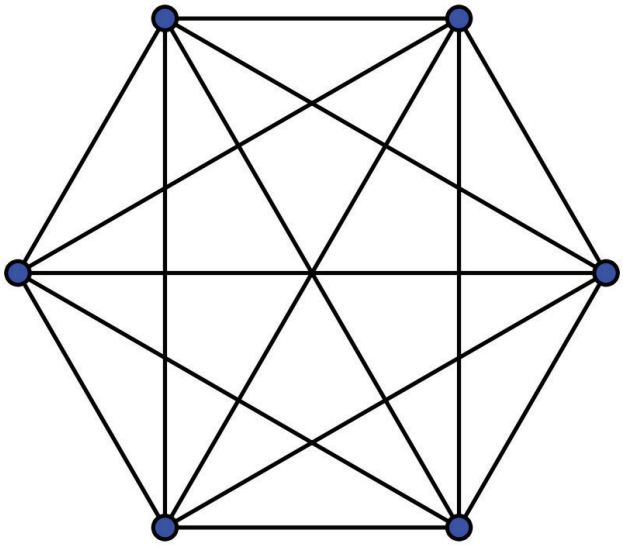
I am implementing a piece of software that enables users to play Sim, where they could play against humans or a computer player which plays according to a minimax algorithm, and provide support for variations of the game.

Research

The Sim pencil game is a two-player game. Traditionally, it has been played on pencil and paper. The rules are as follows:

1. Six dots (vertices) are drawn with no lines (edges) connecting any pair of them initially.
2. Each player, taking turns, would connect a pair of unconnected dots by drawing a line of the player's colour to connect them.
3. The player loses if they draw a triangle of their own colour which contains three vertices.

Here is a simple illustration of the playing area:



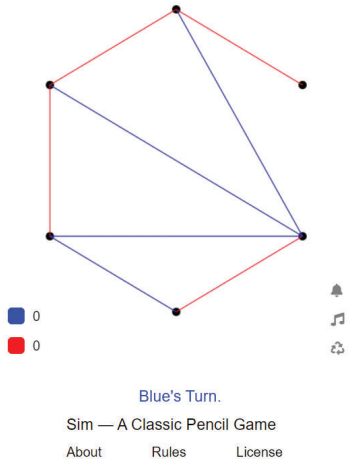
It is proven by Ramsey theory that there always exists a winner in a game of Sim.[1] Any complete graph of 6 vertices must contain a monochromatic (1 colour) triangle when the edges are coloured by 2 colours, and hence the game will never end in a tie. It is also shown by computers that the second player is guaranteed to win if they play perfectly, but such a winning strategy for humans to easily memorise is yet to be found.[2] It is near impossible to find a winning strategy for a multiplayer (3 or more players) variant of the game.[1]

A winning strategy in the standard 2-player variant is found for the second player[3], which is explained in Vol 47 No 5 of the Mathematics Magazine. However, it is out of the scope of

the A-level Mathematics/Further Mathematics/Computer Science to explore further on this winning strategy exactly. A complete search by computers on the game graph will guarantee victory, but due to computational constraints, it is not advisable to implement such algorithm as the time complexity is O(n!) where n is the number of edges in the graph. A more common approach to a computer player is by a minimax algorithm, which is also presented by other implementations in the next section.

Other current solutions

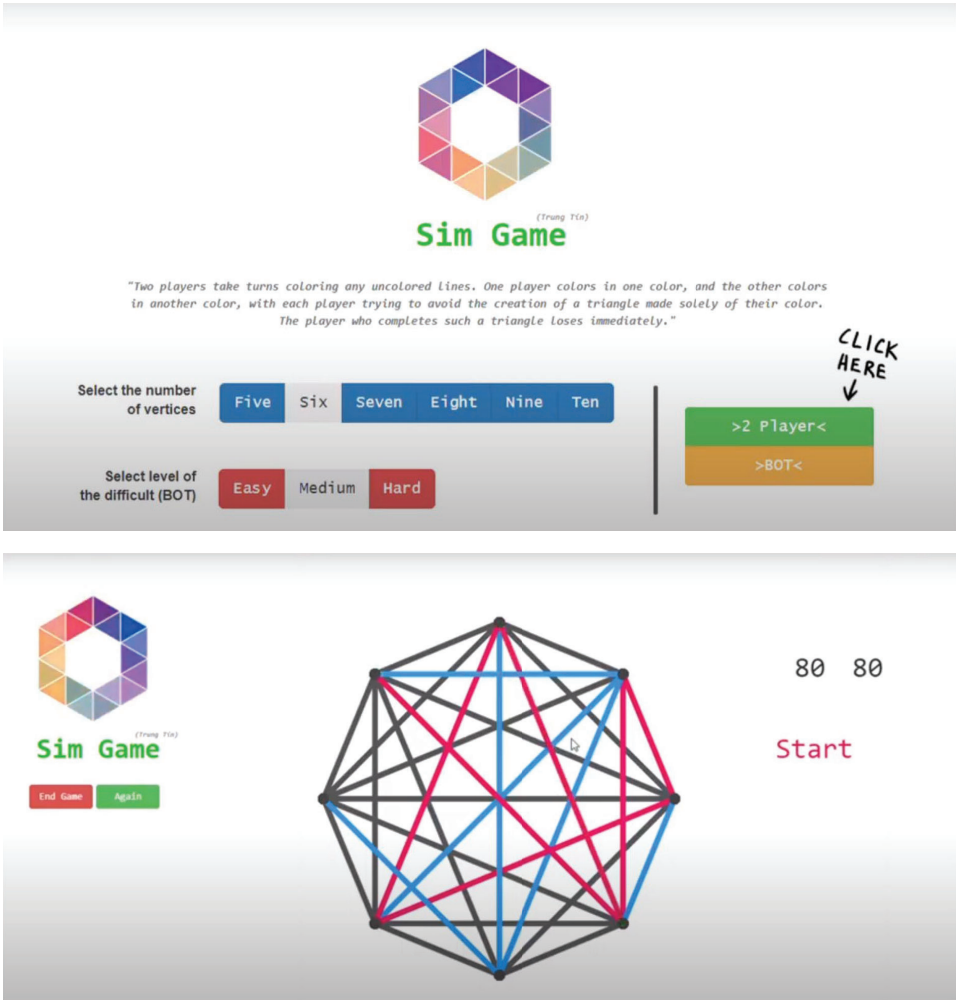
A version of the game is currently hosted on https://joshbraun.umasscreate.net/sim/. A screenshot of the interface is as below:



On this particular website, the difficulty of the computer player could be varied. The user could choose how well the computer plays, and it is a good site to test against my own minimax computer player which is implemented later. The obvious downside of the site is that if a computer player is introduced, the human player always goes first and the computer player always goes second. However, the hardest computer player is not unbeatable despite it technically should have a winning strategy (as discussed previously).

On GitHub, implementations of the game are found. https://github.com/CalvinYL/Sim-Pencil-Game-Project includes an implementation of the game in Java, including a minimax computer player. https://github.com/NicoHinderling/Sim-Pencil-AI also included a computer player, and a graphical user interface as well. According to the person who coded Sim, the evaluation function of that program 'traces down the tree based off of whether it is less likely to encounter cycles and whether the human is more likely to encounter cycles'.

https://github.com/TinDang97/Sim-Pencil-Game is implemented in JavaScript and HTML. It includes a computer player and provides support for multiple numbers of vertices (and not limited to 6). Screenshots of the GUI from his YouTube are found below:



End-user

Mr Chiu, Teacher of Digital Creativity at Tonbridge School

Third parties/other prospective users

Other prospective users include students at the school and any member of the general public who is interested in playing the game. As the game is played offline on the client's own device, as long as a person has an electronic device, the game could be played.

Interviews

Here is a dialogue with Mr Chiu in the initial meeting to facilitate setting objectives:

Q: What functionality would you expect this implementation to have?

A: I would want to have a computer player that I can play against. It would also be nice if it allows human players to play against each other on the same device.

Q: Is there any variations/extensions of the game would you want to see?

A: More human players could be introduced instead of just 2. Alternatively, more vertices could be used instead of just 6.

Q: Would you prefer having multiple levels of the computer player to choose from?

A: I personally do not mind having either – as long the computer sometimes loses then it is fine.

Hardware/Software requirements

No specific hardware is required. A computer with a reasonable CPU will be able to run the minimax algorithm in a short time.

The computer requires Python (and Tkinter, which should be installed alongside Python by default) to be installed in order to run the file.

Objectives

1. The user should be able to choose the number of players in the game.
    a. The number of players is from 2 to 5.
2. The user should be able to choose if the user wishes to play against the computer in the case of a 2-player game.
    a. The user should also be able to indicate if the user would like themselves or the computer to play first.
3. The user should be able to choose the number of vertices for the playing board.
4. The software should assign different colours to represent edges created by different players.
    a. Black and red should be reserved as black means general game announcements and red means warnings.
5. The software should display whose turn it is at the current state of the game.
6. The software should indicate if a player has chosen one vertex and is waiting for a second input.
7. The software should check if a player has attempted to connect vertices that have already been connected, and in such case, disregard the move and ask them to make a correct move.
8. The software should check if a monochromatic triangle has been connected at any point of the game, and in such case, end the game and declare the loser.
    a. The graph should be checked for the losing condition every time a move has been made.
9. The computer player should be implemented using a minimax algorithm with appropriate depth (4 for 7 or fewer vertices, 3 for 8 or more vertices).
    a. The evaluation function should be based on the number of triangles different players can connect, and the number of unconnected triangles with two edges belonging to different players.
    b. Calls in the minimax algorithm should be made recursively.

IDE and Programming Language

Visual Studio Code and Thonny are used as the Integrated Development Environments (IDE) in this NEA. Python is chosen as the programming language to be used. A table comparing the advantages and disadvantages of the different IDEs are found below:

IDE	Visual Studio Code	Thonny	Repl.it
Advantages	- Can install newer versions of Python - Can install various extensions to help with coding - Provides explanations to in-built variable types/functions in Python during mouse hover - Files can be accessed offline	- Debug mode: enables code to be run line-by-line which facilitates debugging	- Code automatically saved online
Disadvantages	(None)	- The Python version is fixed to an older one	- Only accessed online and cannot work offline

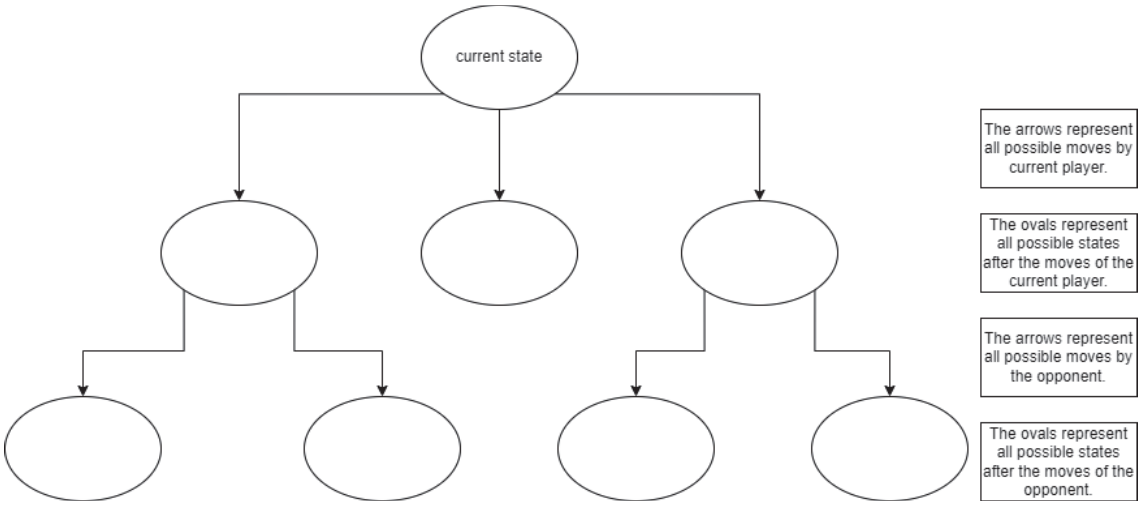
Comparison of the programming languages are as follows:

Programming language	Python (interpreted)	C++ (compiled)
Advantages	- Syntax is easier to understand and debug - Modules are easier to be installed and imported	- Uses less memory and runs faster - Does not need source code when run as it produces an executable file
Disadvantages	- Uses more memory and runs slower - Requires source code to run (as it runs line-by-line at runtime)	- Syntax is more confusing and potentially harder to debug - Only has inbuilt library functions

Modelling of the problem

Minimax algorithm

As the minimax algorithm is a key part of the software, here is a diagram describing how a minimax algorithm works:

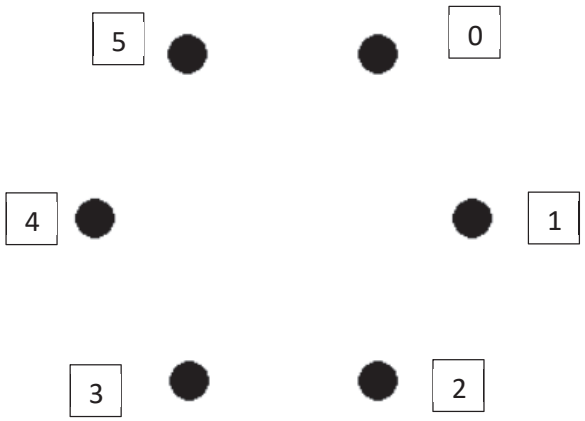


After it has reached the leaf nodes in the search tree, the state is assigned a score by an evaluation function. The score is then returned to the ancestor node, when then that state's score is assigned as either the maximum (when it is the original player's turn) score of its descendant node states, or the minimum (when it is the opponent's turn) score of its descendant node states. After the result has been returned to the top layer, the computer chooses a move based on the highest score in the next layer of states.

Representation of playing area

Below describes how the playing area of the game is interpreted by the computer:

Each vertex is assigned an ID as follows (example).



It then creates a graph containing all these vertices. An adjacency matrix is used over an adjacency list due to the following reasons:

1. The graph is (will be) highly connected. Each player has to create a new edge between two unconnected vertices, and unless they intentionally want to lose, lots of edges are created.
2. Edges are often changed and checked. The game ends when a monochromatic triangle is created, and checks have to be made between triplets of edges.
3. Vertices are not added/removed frequently. The game board is fixed and pre-defined, so once the graph is initialised, there is no need to add new vertices in. Hence an adjacency list is not suitable.

Documented design

High-level overview

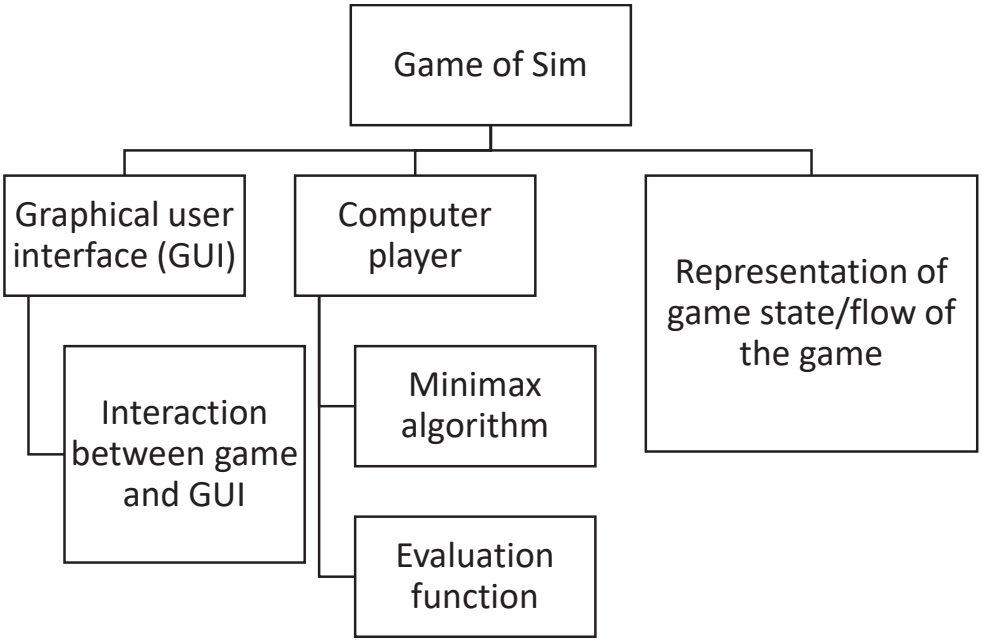
As a proof of concept, a command-line interface version of the game is first coded to make sure the game works, and to improve the evaluation function in the minimax algorithm. After the graphical user interface is developed, the user should be able to interact with the game and the computer player without the need of leaving the application window.

The application should be able to initialise the game and any associated properties from user inputs in the application window, and then display and run the game. It is necessary for it to be able to take interactive input from the user(s) when playing the game as they make moves.

Project timeline

Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Task	Research of the game and viewing current solutions. Initial meeting with client to set out expectations and objectives.				Start designing the software, including classes and algorithms and graphical user interface. Initial development and testing to improve the algorithm.				Developing the software, writing all the code. Documenting the technical solution and adding comments.				Testing the software to make sure it meets the objectives. Speaking to the client for feedback when the software is completed. Finish up the report.			

Decomposing the problem



Class diagrams

Below is a list of class diagrams to illustrate the classes created in the software:

Graph
+ playerno: integer + order: integer + representation: list + edgelist: list
+ TestForTriangle: integer + UpdateEdge(player: integer, nodeone: integer, nodetwo: integer) + GetEdgeList: list

Game
- __currentturn: integer - __playerno: integer - __vertex: integer - __possiblemoves: list - __noofedges: list
+ GetCurrentTurn: integer + GetPlayerNo: integer + GetVertexNo: integer + GetPossibleMoves: list + GetNoOfEdges: list + UpdateCurrentTurn(currentturn: integer) + UpdatePossibleMoves(nodeone: integer, nodetwo: integer)

GUI
+ geometry + resizeable + canvas: Canvas + CANVASCENTER: integer + RADIUS: integer + start_btn: ttk.Button + end_btn: ttk.Button + vertexlabel: ttk.Label + vertex: IntVar + vertexoption: ttk.OptionMenu + mode: integer + modestr: StringVar + modelabel: ttk.Label + modeoption: ttk.OptionMenu + playerlabel: ttk.Label + playerno: IntVar + playernooption: ttk.OptionMenu

+ inputs: list + status: string + warningstatus: string + statuslabel: Label + warningstatuslabel: Label + colour: list + vertices: list + choose: list + clicked: Boolean + move: integer/tuple + computermove: Boolean + currentplayer: integer + gameended: Boolean
+ mainloop + start + checkifendgame + playgame + checkmove + end(playerlost: integer, draw: Boolean) + displayplayer(playerno: integer) + humanmove: integer/tuple of integers + draw(edgelist: list) + drawvertex + drawedge(edgelist: list) + updatestatus (text: string, colour: string) + updatewarningstatus (text: string, colour: string) + disablebuttons + enablebuttons + mouseclick + getnode(x: integer, y: integer): integer + highlight(node: integer) + unhighlight(node: integer)

The minimax algorithm does not use a class, but instead it is achieved purely through recursive function calls.

Algorithms

Minimax

A minimax algorithm is implemented using depth-first search and recursive function calls. To limit the computer run time (and prevent it from becoming too powerful with a complete search), a few base cases are identified to terminate the recursive calls, including when a monochromatic triangle is formed at the current state of the game, or no more possible moves could be made.

If we encounter a state in which a monochromatic triangle has already been formed, the game ends. Hence if that happens during the minimax search, we stop the search immediately (so hence a base case). This needs to be further divided into two cases – where the computer player forms a triangle (undesirable), or the human player forms a triangle (good for the computer player). At this point, we do not send the representation to the evaluation function, but instead we directly assign scores of -99999 and 99999 if the computer and the human form a monochromatic triangle, respectively.

The other base case is when the search depth has been reduced to 0 or there are no more possible moves, indicating the end of the search. Then we return the result as calculated by the evaluation function.

We call the minimax function for every possible move – we update the game board for each such move and call the minimax on this board. In each layer, we alternate the player who is to have their score maximised i.e. change to minimising the computer player's score. In each layer, we return the maximum or minimum score accordingly, depending if which layer is maximising or minimising the computer player's score. We also reset the board after that.

If certain moves give the same score, we just randomly choose a move. If all the moves give a score of -99999 (i.e. losing), we take any move that does not immediately guarantee a loss for the computer.

Pseudocode of the function is found below:

```
MAXDEPTH = 3
Function minimax(computerplayer, representation, possiblemoves, edgespervertex,
depth, maximising)
    listoftriangles = TestForTriangle(representation)
    IF listoftriangles CONTAINS computerplayer THEN
        RETURN -99999
    ELSE
        IF NOT (listoftriangles CONTAINS -1) THEN
            RETURN 99999
        ENDIF
    ENDIF
    IF maximising THEN
        player <- computerplayer
    ELSE
        player <- NOT computerplayer
    ENDIF
    IF depth = 0 OR LEN(possiblemoves) = 1 THEN
```

```
        RETURN evaluate(representation, possiblemoves, computerplayer)
    scorelist <- []
    FOR move IN possiblemoves
        UPDATE representation
        UPDATE possiblemoves
        UPDATE edgespervertex
        scorelist add minimax(NOT computerplayer, representation, possiblemoves,
edgespervertex, depth-1, NOT maximising)
        RESET representation
        RESET possiblemoves
        RESET edgespervertex
    ENDLOOP
    IF depth = MAXDEPTH THEN
        IF NOT (ALL scores IN scorelist = -99999) THEN
            RETURN possiblemoves[scorelist.index(MAX(scorelist))]
        ELSE
            IF move IN possiblemoves NOT immediately lose THEN
                RETURN move
        ENDIF
        RETURN possiblemoves[0]
    ELSE
        IF maximising THEN
            RETURN MAX(scorelist)
        ELSE
            RETURN MIN(scorelist)
        ENDIF
    ENDIF
END Function
```

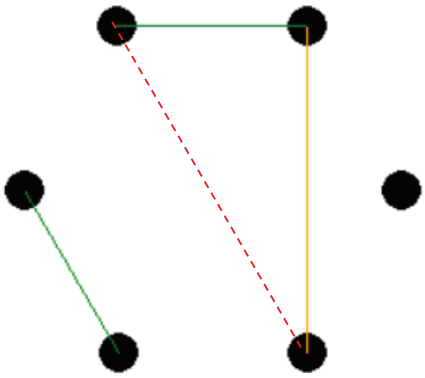
Evaluation function

An evaluation function is designed to assign a score to a state of the board. The evaluation function would only take in the current state of the board as its parameter.

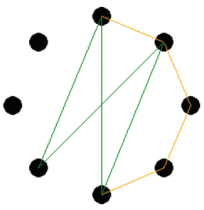
As we have dealt with certain cases in the base cases in the minimax algorithm, there should be no more than one player having formed a monochromatic triangle at the current state. We assign an arbitrary score of -2 for each such triangle connected if it's the player's own colour, and an arbitrary score of +1 for each such triangle connected if it's an opponent's colour[1].

In the case of unconnected triangles (see definition below), we assign an arbitrary score of +10 for each such triangle if one of the existing two edges is the current player's, or -1 for each such triangle if none of the existing two edges is the current player's.

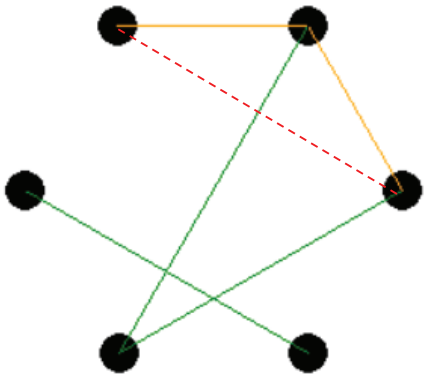
In the case below, there is an **unconnected triangle**. The line indicated in dotted red could be connected by yellow in this move, or if yellow chooses not to, green in future moves.



[1] There could be more than one triangle formed at the same time. See below for example (green can form 2 triangles at the same time by connecting the bottom vertices.



In this case below, no triangles are formed. However, if yellow connects the dotted red line, yellow would form a monochromatic triangle, which is undesirable. Hence green will avoid making that move so as to force yellow to make that move.



Pseudocode of the function is found below:

```
Function evaluate(representation, possiblemoves, playerID)
    localscore <- 0
    For tuple in possiblemoves
        representation[tuple[0]][tuple[1]] <- playerID
        representation[tuple[1]][tuple[0]] <- playerID
        player <- TestForTriangle(representation)
        unconnected <- TestForUnconnected(triangle)
        localscore <- localscore - player.count(playerID)*2
        localscore <- localscore + player.count(NOT playerID)
        localscore <- localscore + unconnected.count(playerID in each tuple)*10
        localscore <- localscore - unconnected.count(NOT playerID in each tuple)
        representation[tuple[0]][tuple[1]] <- -999
        representation[tuple[1]][tuple[0]] <- -999
    End Loop
    Return localscore
End Function
```

Test for losing condition

The game ends when a monochromatic triangle is created between any 3 vertices. Hence we check for the losing condition after every turn in the game. This algorithm is also reused in the minimax algorithm and the evaluation function, but it is slightly modified as we want to keep count of the number of triangles formed, instead of just getting the playerID of the losing player.

We have nested for-loops to check if every triple of vertices has their edges belonging to the same player and return the player number of the triangle created. (note: cannot be -999 or -1 as they indicate either the edges have not been created, or the edges cannot be created). For the algorithm when used in the minimax algorithm and evaluation function, instead of returning the player number, we append the player number and the triple of vertices into a list for each such triangle, and then return the whole list.

Pseudocode:

```
Function TestForTriangle(representation)
    order <- LEN(representation)
    arr <- []
    FOR nodeone IN RANGE (0, order)
        FOR nodetwo IN RANGE (nodeone+1, order)
            FOR nodethree IN RANGE (nodetwo+1, order)
                IF representation[nodeone][nodetwo] =
representation[nodetwo][nodethree] = representation[nodeone][nodethree] THEN
                    player <- representation[nodeone][nodetwo]
                    IF player != -1 AND player != -999 THEN
                        RETURN representation[nodeone][nodetwo] // Only when
actually playing game
                        arr add representation[nodeone][nodetwo] // Only when
using minimax/evaluation Function
                    ENDIF
                ENDIF
            ENDLOOP
        ENDLOOP
    ENDLOOP
    IF arr = [] THEN
        arr add -1
    ENDIF
    RETURN arr
END Function
```

Merge sort

A merge sort algorithm is implemented to deal with all sortings in the programs.

Pseudocode for a merge sort algorithm is found below:

```
Function mergesort(arr)
    IF length(arr) > 1
        left <- arr[:middle]
        right <- arr[middle:]
        mergesort(left)
        mergesort(right)
        arr <- []
        WHILE left NOT EMPTY OR right NOT EMPTY
            IF left[0] > right[0]
                arr add left[0]
                remove left[0]
            ELSE
                arr add right[0]
                remove right[0]
            ENDIF
        ENDWHILE
    ENDIF
    RETURN arr
END Function
```

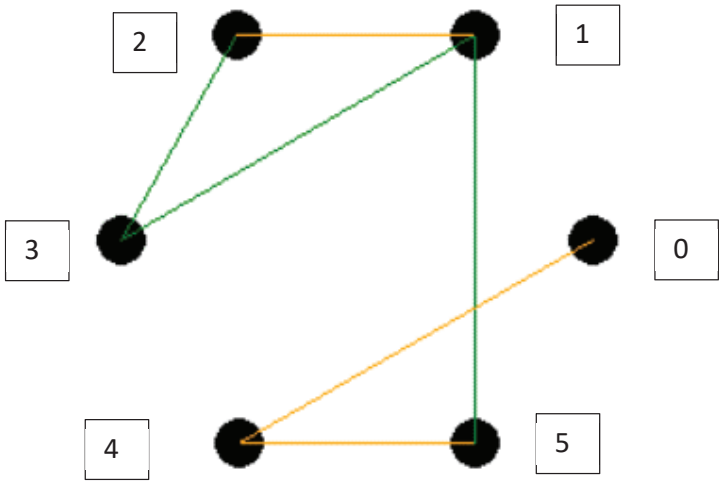
Data structures

The main data structure used in this game is the graph that represents the game board itself. The graph as shown on the game board is represented by an adjacency matrix. The advantages of this have been previously discussed in Analysis/Modelling of the problem/Representation of playing area.

Examples are found below:

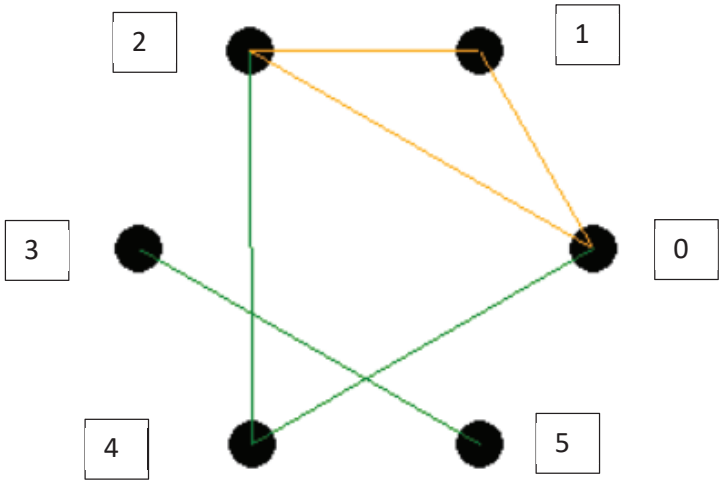
Example 1 (green is player 0, yellow is player 1):

Vertex	0	1	2	3	4	5
0	-1	-999	-999	-999	1	-999
1	-999	-1	1	0	-999	0
2	-999	1	-1	0	-999	-999
3	-999	0	0	-1	-999	-999
4	1	-999	-999	-999	-1	1
5	-999	0	-999	-999	1	-1



The adjacency matrix represents the graph above as in the diagram. In the adjacency matrix, -1 indicates that the vertices cannot be connected (as they are the same vertex), 0 and 1 represent edges created by player 0 and player 1 respectively, and -999 indicates that the vertices are not yet connected.

Example 2 (game ends)



Vertex	0	1	2	3	4	5
0	-1	1	1	-999	0	-999
1	1	-1	1	-999	-999	-999
2	1	1	-1	-999	0	-999
3	-999	-999	-999	-1	-999	0
4	0	-999	0	-999	-1	-999
5	-999	-999	-999	0	-999	-1

As we can see, player 1 has connected a monochromatic triangle (yellow) with vertices 0, 1 and 2. The adjacency matrix has those edges assigned value 1, while edges 0-4, 2-4 and 3-5 are assigned value 0 (player 0). In this case, player 1 has lost.

Database design and queries

No databases are used in this NEA and hence this part is not applicable.

User interface

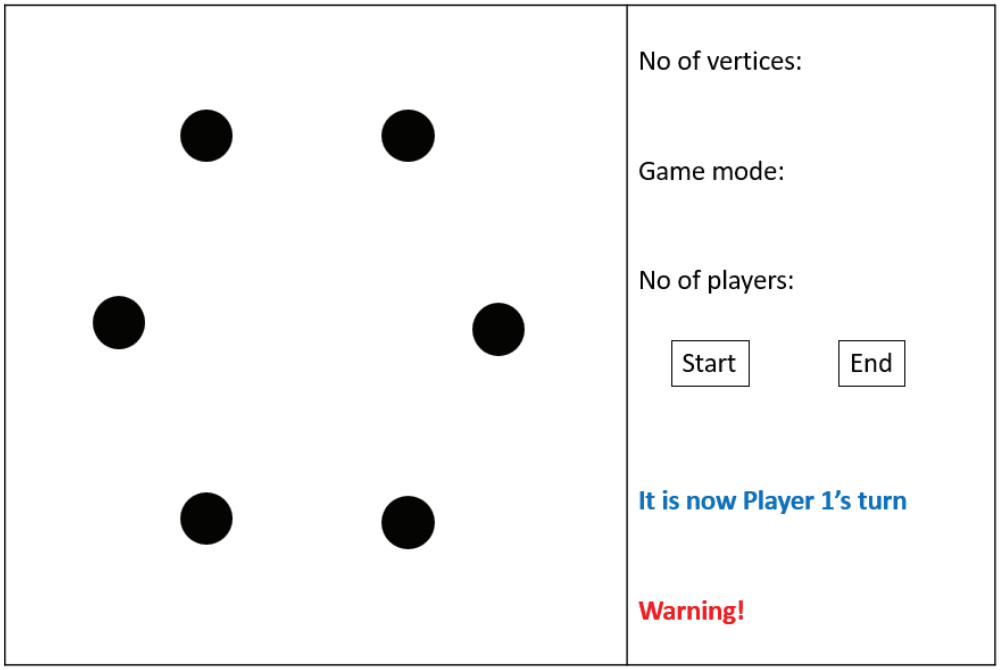
The game uses a graphical user interface (GUI) which is created using Tkinter.

There are two main components of the GUI, including the game board (6-10 vertex graph) and the settings menu. The settings menu includes dropdown menus to get the number of vertices of the graph, game mode and the number of players, and also start/end buttons alongside status messages.

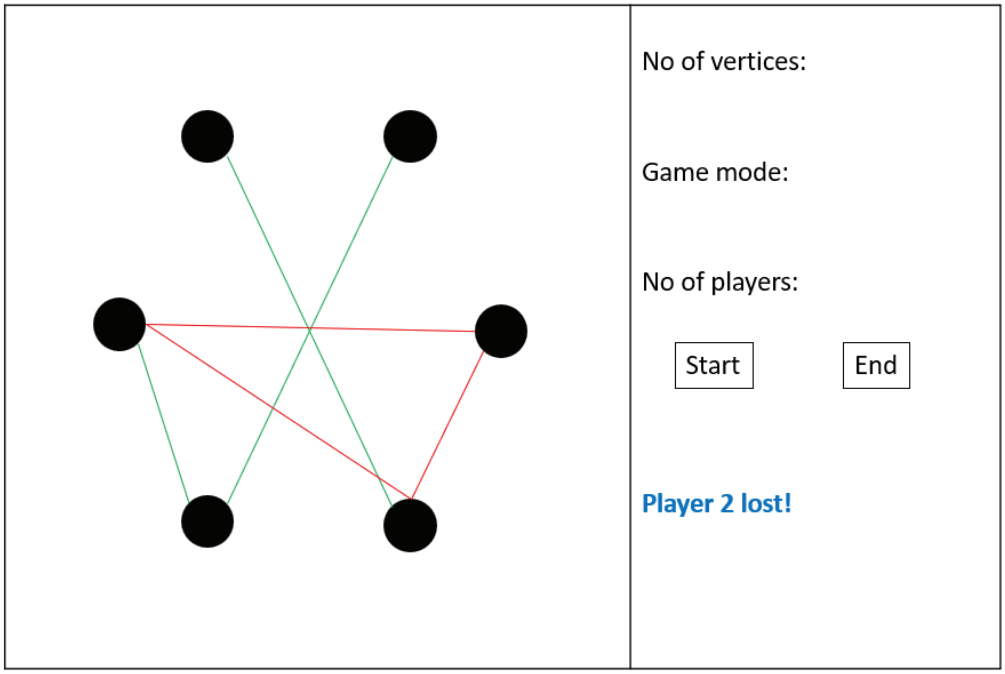
A diagram describing the arrangement of the components is as follows:

<table>
<tr><td rowspan="6">Game board</td><td colspan="2">No of vertices</td></tr>
<tr><td colspan="2">Game mode</td></tr>
<tr><td colspan="2">No of players</td></tr>
<tr><td>Start</td><td>End</td></tr>
<tr><td colspan="2">Status message</td></tr>
<tr><td colspan="2">Warning status message</td></tr>
</table>

Initial designs of the user interface could be found below:



The design above shows the initial state as the game has just been initialised and started, pending the first player to make their move.



This design shows the display when the game ends. As a red triangle has been created as shown, player 2 has lost and it is displayed on the right as well.

Technical Solution

In this section, no code snippets are provided. Instead, parts of code are referenced by the line numbers, and the full code for a particular file could be found immediately after the commentary.

List of objectives

The objectives as listed in Analysis are fulfilled directly by the following parts:

Objective number	Item that fulfils requirement
1	Implementing the dropdown menus *self.playernooption* alongside *self.playerlabel* and *self.playerno*
1a	Implementing *self.playernooption* which limits the number of players to be between 2 and 5
2	Implementing the dropdown menus *self.modeoption* alongside *self.modelabel* and *self.mode*
2a	Implementing *self.modeoption* which allows the player to choose between 'CPU vs Human' and 'Human vs CPU' – different order of playing
3	Implementing the dropdown menus *self.vertexoption* alongside *self.vertexlabel* and *self.vertex*
4	Implementing *self.colour*
4a	Excluding red and black from *self.colour*
5	Implementing the method *self.displayplayer*
6	*self.clicked*, *self.mouseclick* and *self.highlight*
7	Implementing such checks and warnings within *self.checkmove*
8	Implementing *self.checkifendgame* and *self.end*
8a	Every time that *self.checkmove* is called, *self.checkifendgame* is called as well
9	*minimax.maxdepth* limits the depth of the search
9a	*minimax.evaluate* is implemented according to the objective
9b	*minimax.minimax* includes recursively calling itself and having multiple base cases

Techniques used

Below is a table of techniques used in the code:

Technique	Description	Technique group
Graph/Tree Traversal	The *minimax* function (page 41) in *minimax.py* uses depth-first search (implemented recursively).	A
Recursive algorithms	The *minimax* function (page 41) in *minimax.py* recursively calls itself and has base cases to terminate the recursive calls.	A

Mergesort	The *mergesort* function (page 54) in *sort.py* is an implementation of the merge sort algorithm.	A
Graphs	The graph of the game board is implemented in *graph.py* and is stored in an adjacency matrix. (page 52)	A
Complex OOP model (classes and interfaces)	OOP objects in *gui.py*, *game.py* and *graph.py*. Interface (including encapsulation and access methods) in *game.py* and *graph.py*.	A

Graphical User Interface (gui.py)

A GUI object called *startapp* is created in line 320 to run and play the game.

__init__

All methods in *Tk* are inherited as a result of calling *Tk.__init__(self)*. (Line 8)

A window is created with a size of 1000x700, and it is not resizable (hence size is fixed) (Lines 9-10).

A *Canvas* object is created, which is stored in *self.canvas* with size 700x700 and a white background.

2 constants are defined, namely *self.CANVASCENTER* and *self.RADIUS*, which represents where the centre coordinates of *self.canvas* is and the 'radius' of the graph (distance from centre coordinates to any vertex on the graph) respectively.

Different buttons and labels are initialised, in which each of their purposes is explained in the comments in the code. Refer to 'Grid layout' in this section on the next page to see the corresponding line numbers.

To obtain the human input from the dropdown menus, a few objects are created as subclasses of *Variable* class in Tkinter. Those include *self.vertex* (IntVar, integer from 6 to 10), *self.mode* (StringVar, string) and *self.playerno* (IntVar, integer from 2 to 5). These variables are self-explanatory (as in the variable name) and contents stored in these could be obtained by the get() method. **Objectives 1, 1a, 2, 2a, 3**

self.inputs is then initialised as a list to store references to all the dropdown menus. This is useful as the objects stored in *self.inputs* are disabled during gameplay, and by storing all of them in a list, the dropdown menus do not have to be disabled one by one manually.

self.status and *self.warningstatus* is also declared to initialise the messages in *self.statuslabel* and *self.warningstatuslabel*. (Lines 54-56)

self.colour is initialised to assign colours to different players. Each player has its own colour, which is used when displaying the game status and drawing the edges in the graph. (Lines 66-67) **Objective 4, 4a**

self.vertices is initialised as an empty list, which will be used later to store the coordinates of the vertices of the graph. (Lines 69-70)

self.choose is initialised as an empty list, where it is used later to store the vertices that a human player has clicked. It only stores the most recent 2 vertices clicked, and if any one of them is invalid, the list is cleared. (Lines 72-73)

self.clicked and *self.computermove* are both Booleans which are self-explanatory; *self.move* is initialised which is used to store the current player's move. (Lines 75-82) **Objective 6**

We then bind a single click on the canvas with *self.mouseclick* (so when a single click occurs, *self.mouseclick* is called). (Line 87-88)

self.currentplayer stores the player ID of the current player. (Lines 90-91)

self.gameended is a Boolean that indicates if the game has ended. (Lines 93-94)

We then call *self.mainloop()* to initialise the window and keep it running. (Lines 96-97)

Grid layout

Corresponding lines of code where the objects are placed into the grid could be found in Lines 14, 22, 26, 30, 33, 39, 41, 45, 48, 59 and 63.

<table>
<tr><td rowspan="6">*self.canvas* (13-15)
Height = 700
Width = 700
Background = 'white'</td><td>*self.vertexlabel* (29-30)</td><td>*self.vertexoption* (32-33)</td></tr>
<tr><td>*self.modelabel* (38-39)</td><td>*self.modeoption* (40-41)</td></tr>
<tr><td>*self.playerlabel* (44-45)</td><td>*self.playernooption* (47-48)</td></tr>
<tr><td>*self.start_btn* (21-23)</td><td>*self.end_btn* (25-27)</td></tr>
<tr><td colspan="2">*self.statuslabel* (58-60)
Max line length = 240, font: Ariel, 25</td></tr>
<tr><td colspan="2">*self.warningstatuslabel* (62-64)
Max line length = 240, font: Ariel, 25</td></tr>
</table>

start

We start by determining what game mode did the player select. The three options are 'CPU vs Human', 'Human vs CPU' and 'Multiple Humans', which correspond to modes 0, 1 and 2 respectively. (Lines 100-108)

We set *self.gameended* to False to indicate the game has started and has not ended, and we override the number of players if a CPU is involved in the game as CPU only supports 2-player games. (Lines 109-113)

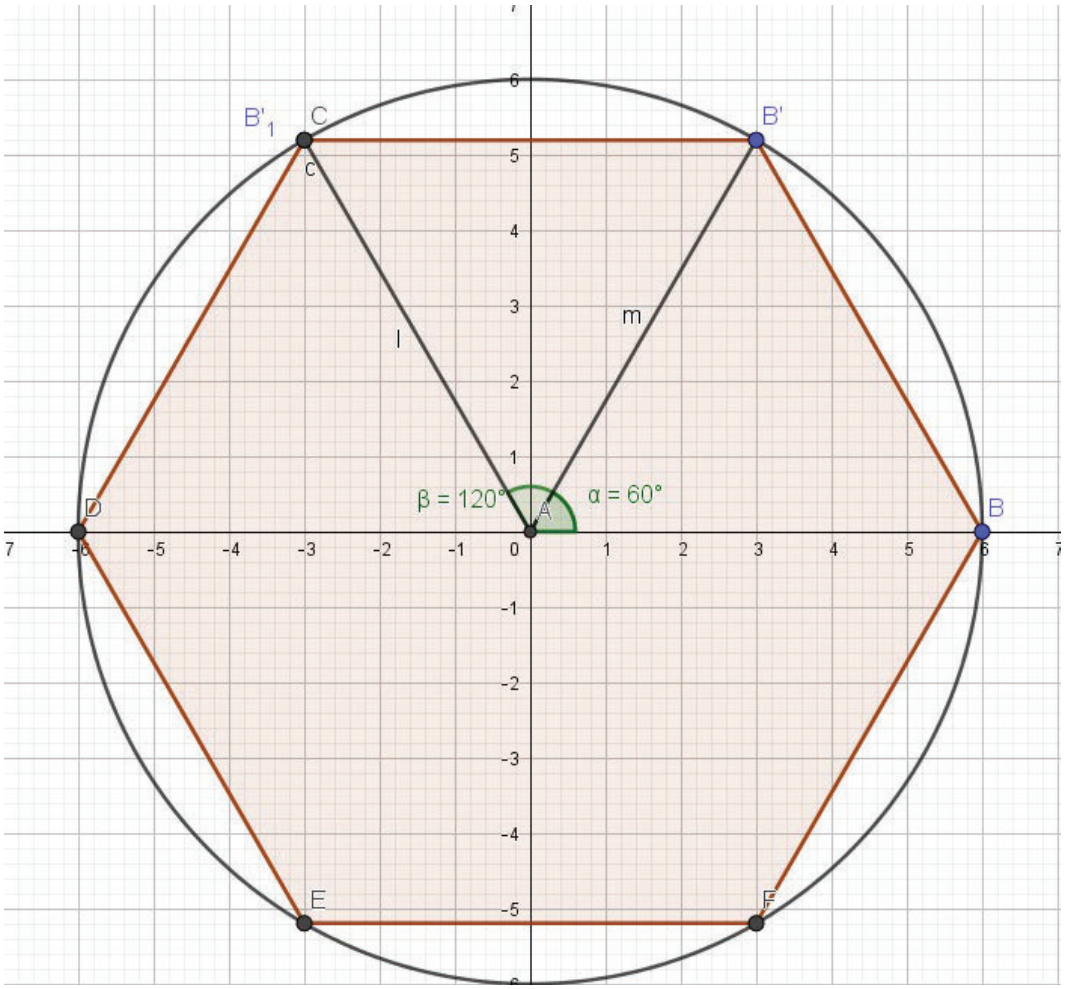
self.disablebuttons is called to disable all the dropdown menus and the start button, and enable the end button. We then call *control.initial* to initialise the game flow in a separate file to the GUI. (Lines 114-117)

Lines 118-120 describe how the coordinates of the vertices are found. This is discussed more in detail in 'Finding the coordinates of the vertices' in this section on the next page.

We draw the initial graph with no edges created by calling *self.draw*, and we start playing the game by calling *self.playgame*. (Lines 121-124)

Finding the coordinates of the vertices

To calculate the coordinates of the centre of each vertex, we first find the centre of the entire graph, which is (350, 350) as the canvas size is 700x700. We have pre-defined the radius of the graph as 100, so each vertex is at a distance 100 away from (350, 350). A diagram to illustrate the coordinates is found below:



So for example, if we are to construct a graph with 6 vertices, we can calculate the angle that OB makes with each of the vertices. This is found by $\frac{360}{6} = 60$ degrees with 6 vertices, or in general, $\frac{360}{n}$ degrees. As Python can only work with radians in sin and cos functions, we convert degrees into radians by dividing it by 180 and multiplying by pi. This gives the angle that OB makes with the first vertex $\frac{2\pi}{n}$, and the kth vertex $\frac{2k\pi}{n}$ (vertex count starts at 0).

Then each coordinate can be found by $(350 + 100\cos\theta\,, 350 - 100\sin\theta)$ where θ is the angle between OB and the vertex.

checkifendgame

This method simply returns *control.checkend()*, which is a Boolean that contains True if the game should end, or False if the game should not end (and hence should continue). (Lines 126-128) **Objective 8, 8a**

playgame

First, *self.currentplayer* is updated by updating its value to the return result by *control.getturn()*. Then we call *self.displayplayer* to update the status message and display the fact that it is the current player's turn. (Lines 131-134)

We then proceed to check if the current player is a computer player; this is achieved by comparing the game mode against the current player ID[2], and *self.computermove* is set to True or False depending on the result of the comparison. (Lines 135-142)

If it is the computer player's turn, we obtain its move by calling *control.turnduring*. *self.move* is updated with the return result, and we can proceed to check if the move is valid by calling *self.checkmove*. (Lines 143-148)

checkmove

We proceed by calling *control.check* on the current move (*self.move*), and we check the return result (*status*). (Lines 151-153)

If *status* is 0, this indicates it is a valid move and all corresponding parts of the *Game* object in *control.py* are updated accordingly. We then clear all existing warnings and obtain a list of edges created (with the playerID that connected the vertices) by calling *control.turnend*. We also reset *self.move* to -1, and we proceed by drawing the new edges by calling *self.draw* on the list of edges that has just been returned by *control.turnend*. (Lines 154-163)

If *status* is not 0, this indicates that the move is not valid. The warning status is updated to warn the current player to choose a valid pair of vertices. (Lines 164-167) **Objective 7**

After checking if the move is valid or not, we check if the game should end by calling *self.checkifendgame*. **Objective 8a** If it returns anything but -1, it means a player has lost, and we call *self.end* to end the game. (Lines 168-171)

If it returns -1, there are two possible outcomes. Either the game ended in a draw, or the game has not finished. To see which case it is, we check if there are no more possible moves. If no more possible moves are found, that means the game ends in a draw and that is indicated by passing True as one of the parameters when calling *self.end*. If there are possible moves, that means the game has not ended, and we call *self.playgame* again to keep playing the game. (Lines 172-178)

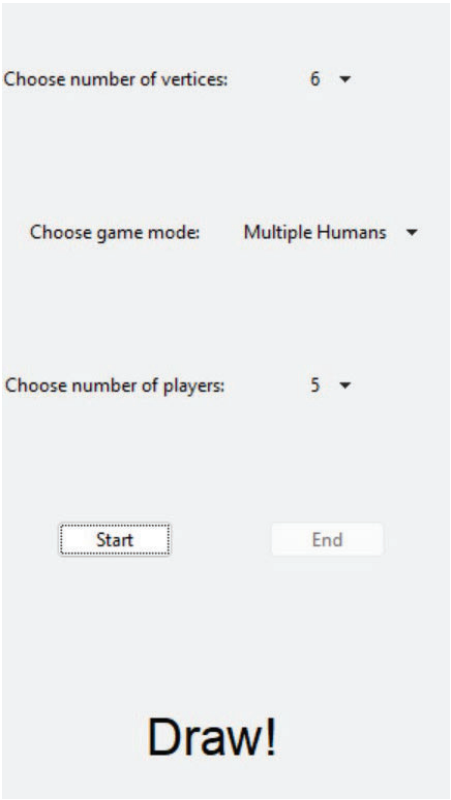
[2] This only works when the game mode is 0 or 1, in which the game involves a computer player. Game mode 2 (Multiple Humans) does not involve a computer player and hence this is irrelevant.

end

The *self.end* procedure takes in two parameters, *playerlost* (integer; player ID of the player who lost; the default value is -1) and *draw* (Boolean; indicates if the game ended in a draw; the default value is False). (Line 180)

We set *self.gameended* to True to indicate that the game has ended, and we clear all warning messages that are displayed by calling *self.updatewarningstatus*. (Lines 181-184)

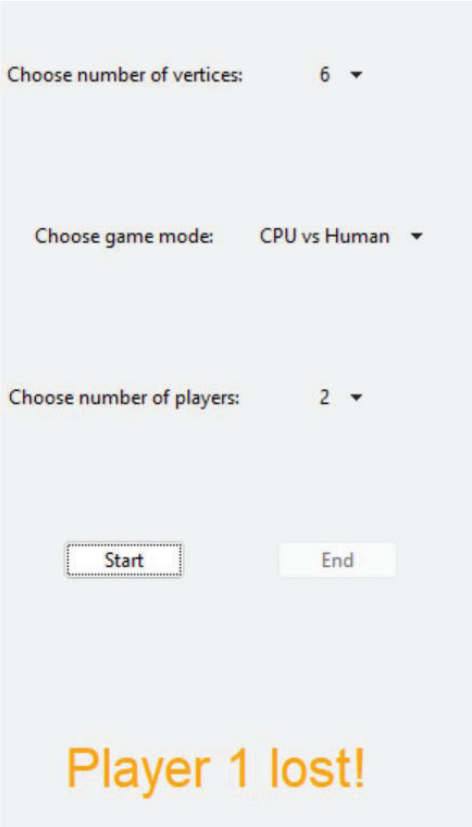
We first deal with the case where there is a draw (indicated as True in *draw*). In this case, we simply update the message displayed on the screen to say that there is a draw. (Lines 185-188)

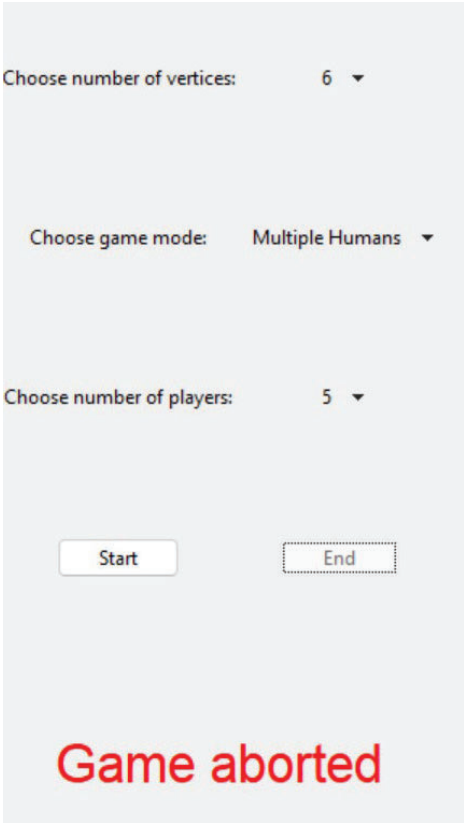


If there is not a draw, it could either be the case that a player has lost, or a player pressed the end button. If a player has lost (so *playerlost* in that case is the playerID and not -1), the message on the screen is updated to indicate who lost; if someone pressed the end button, the message is updated to reflect the fact that the game has been aborted. (Lines 189-197)

Objective 8

Screenshots demonstrating the above are found below:





All the dropdown menus and the start button are enabled, and the end button is disabled. *self.vertices* is cleared so it could be used again next time. (Lines 198-201)

displayplayer

This function acts as a tool to easily update the status and display whose turn is it. It takes in one parameter *playerno* (self-explanatory) and its only purpose is to call *self.updatestatus* with the message and the associated player colour. (Lines 203-206) **Objective 5**

humanmove

The function takes in no parameters and it returns the sorted elements in *self.choose*. If it is called when a computer player is making a move, it does not return a move but instead returns -1 to indicate that a human move should not be made. Otherwise, it calls the *mergesort* function to sort *self.choose* and then return a tuple of the two elements in ascending order. (Lines 208-216)

draw

This function takes in one parameter *edgelist*, which should be a list containing triples of information of edges (player number, vertex 1, vertex 2). It clears all existing canvas by calling *self.canvas.delete('all')* and redraws the vertices and edges by calling *self.drawvertex* and *self.drawedge*. (Lines 218-224)

drawvertex
The method draws circles of radius 10, centring at the coordinates specified in each element of *self.vertices*. (Lines 226-229)

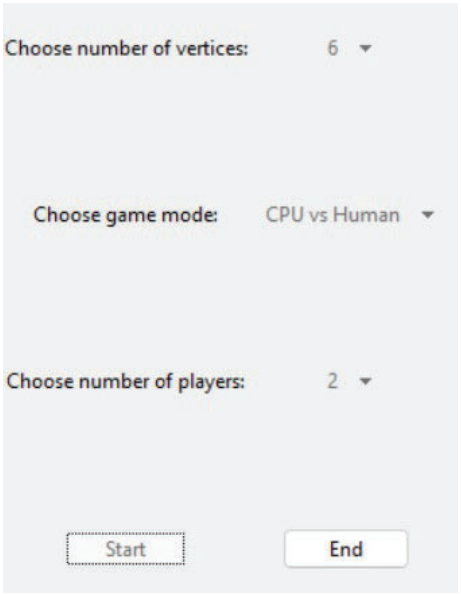
drawedge
The method draws lines between vertices as defined in the triples in *edgelist*. For each triple *edge* in *edgelist*, *edge[0]* is the player number, *edge[1]* is the vertex number of the first vertex and *edge[2]* is the vertex number of the second vertex. Thus the coordinates could be obtained by looking that up in *self.vertices*, and creating a line between them. The colour of the line is the same as the assigned colour of the player, which could be found in *self.colour* (as the colours were pre-defined). (Lines 231-234)

updatestatus
This method takes in two parameters, *text* and *colour* (self-explanatory). It configures *self.statuslabel* to have its message updated to *text* and the text colour updated to *colour*. (Lines 236-241)

updatewarningstatus
This method takes in two parameters, *text* and *colour* (self-explanatory). It configures *self.warningstatuslabel* to have its message updated to *text* and the text colour updated to *colour*. (Lines 243-248)

disablebuttons
Takes in no parameters. It disables all the dropdown menus in *self.inputs*, disables the start button (*self.start_btn*) and enables the end button (*self.end_btn*). (Lines 250-257)

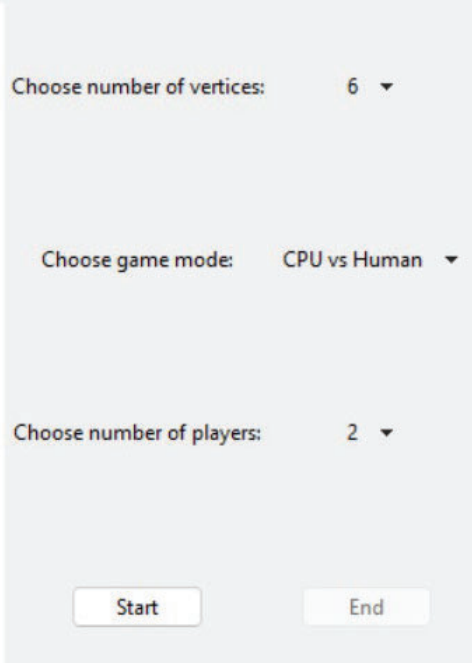
Below is a picture of what it looks like when the method has finished running.



enablebuttons

Takes in no parameters. It enables all the dropdown menus in *self.inputs*, enables the start button (*self.start_btn*) and disables the end button (*self.end_btn*). (Lines 259-266)

Below is a picture of what it looks like when the method has finished running.



mouseclick

This method is called only when there is a click on the canvas as bound in line 87. It takes that click event as a parameter (*event*).

If the computer player is making a move (and *self.computermove* is True), we ignore all clicks on the canvas by the human player, and we update the warning status to ask the player to wait for the computer to finish its move before proceeding. (Lines 299-301)

If it is a human player's turn, we will attempt to get the x-coordinate and y-coordinate of the click, in which we check for the corresponding vertex with *self.getnode*, and we append that result to *self.choose*. (Lines 269-276)

As mentioned, *self.clicked* is a Boolean which indicates if the human player has clicked before or not. If previously a vertex has been clicked, we reset *self.clicked* to False, unhighlight the highlighted vertex, and set *self.move* to the result as returned by *self.humanmove*. We also reset *self.choose* to clear existing clicks by the current player. Finally, we check if the move is valid by calling *self.checkmove*. (Lines 277-290)

If the current player has not previously made a click, we simply set *self.clicked* to True and highlight the vertex that was just clicked. (Lines 291-298) **Objective 6**

getnode

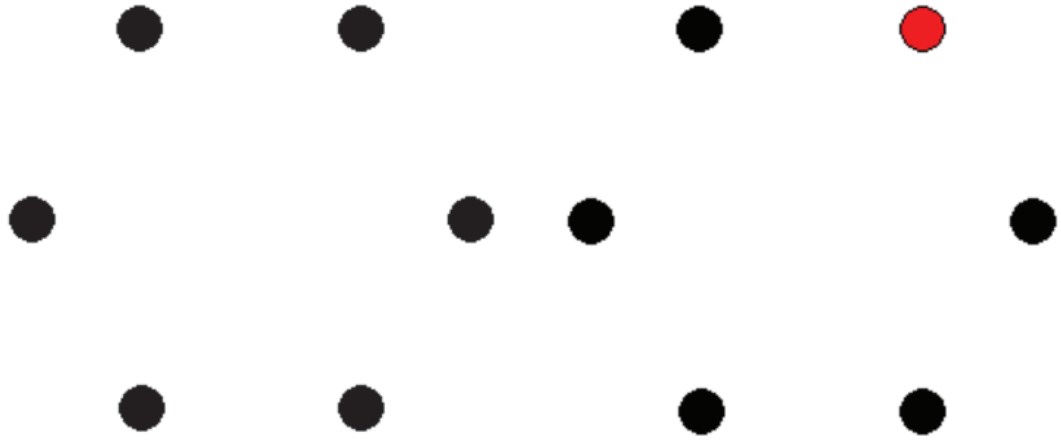
This method takes in two parameters, *x* and *y*, which are the coordinates that are to be checked against. Every vertex is checked to see if *x* and *y* are in the circle (i.e. x- and y-coordinates are within 10 of the centre of the vertex as the circle of the vertex has a radius of 10). If it falls in any vertex, the number of that vertex is returned. Else we just return -1 to indicate that it does not fall in any vertex. (Lines 303-310)

highlight

This method takes in one parameter *node* (which is the vertex number of the vertex to be highlighted). The way to highlight the vertex is to create a red circle which covers the existing black circle of the vertex, which indicates that it has been chosen. (Lines 312-314)

Objective 6

Before highlight and after highlight:



unhighlight

This method takes in one parameter *node* (which is the vertex number of the vertex to be unhighlighted). The way to unhighlight the vertex is to create a black circle which covers the existing red circle of the highlighted vertex, which would reset the highlight effect of the red circle. (Lines 312-314)

```
from math import pi, cos, sin
from tkinter import Tk, Canvas, Label, ttk, StringVar, IntVar
import control
from sort import mergesort

class GUI(Tk):
    def __init__(self):
        Tk.__init__(self)
        self.geometry("1000x700")
        self.resizable(False, False)
        # initialise window with size 1000x700 and it cannot be resized

        self.canvas = Canvas(height=700, width=700, background='white')
        self.canvas.grid(column=0, row=0, rowspan=8)
        # initialise canvas for game board

        self.CANVASCENTER = 350
        self.RADIUS = 100
        # define constants to facilitate drawing of game board

        self.start_btn = ttk.Button(self, style='W.TButton', text='Start',
command=self.start)
        self.start_btn.grid(column=1, row=3)
        # create start button

        self.end_btn = ttk.Button(self, style='W.TButton', text='End',
command=self.end)
        self.end_btn.grid(column=2, row=3)
        # create end button

        self.vertexlabel = ttk.Label(text='Choose number of vertices:')
        self.vertexlabel.grid(column=1, row=0)
        self.vertex = IntVar(self)
        self.vertexoption = ttk.OptionMenu(self, self.vertex, 6, 6, 7, 8, 9, 10)
        self.vertexoption.grid(column=2, row=0)
        # create the dropdown menu for choosing number of vertices and the
associated label

        self.mode = 0
        self.modestr = StringVar()
        self.modelabel = ttk.Label(text='Choose game mode:')
        self.modelabel.grid(column=1, row=1)
        self.modeoption = ttk.OptionMenu(self, self.modestr, 'CPU vs Human', 'CPU vs
Human', 'Human vs CPU', 'Multiple Humans')
        self.modeoption.grid(column=2, row=1)
        # create the dropdown menu for choosing game mode and the associated label

        self.playerlabel = ttk.Label(text='Choose number of players:')
        self.playerlabel.grid(column=1, row=2)
        self.playerno = IntVar()
        self.playernooption = ttk.OptionMenu(self, self.playerno, 2, 2, 3, 4, 5)
        self.playernooption.grid(column=2, row=2)
        # create the dropdown menu for choosing number of players and the associated
label

        self.inputs = [self.vertexoption, self.modeoption, self.playernooption]
        # create a list that contains the dropdown menus so they could be
enabled/disabled

```

```
54          self.status = 'Welcome to Sim!'
55          self.warningstatus = ''
56          # initialise default status/message
57
58          self.statuslabel = Label(self, text=self.status, wraplength=240, font=
    ("Arial", 25))
59          self.statuslabel.grid(column=1, row=4, rowspan=2, columnspan=2)
60          # intialise label to display current turn/status of game
61
62          self.warningstatuslabel = Label(self, text=self.warningstatus,
    wraplength=240, font=("Ariel", 25))
63          self.warningstatuslabel.grid(column=1, row=6, rowspan=2, columnspan=2)
64          # initialise label to display warning messages
65
66          self.colour = ['aqua', 'purple', 'pink', 'green', 'orange']
67          # initialise colours of players
68
69          self.vertices = []
70          # initialise empty list that will contain coordinates of each vertex in the
    graph
71
72          self.choose = []
73          # initialise empty list that will contain what vertices has a player clicked
74
75          self.clicked = False
76          # initialise boolean that indicates if one vertex is clicked already or not
77
78          self.move = -1
79          # initialise variable to store player's move
80
81          self.computermove = False
82          # initialise boolean that indicates if it is the computer player's turn or
    not
83
84          self.enablebuttons()
85          # enable all the dropdown menus
86
87          self.canvas.bind('<Button-1>', self.mouseclick)
88          # if there is a mouse click in the canvas, it calls self.mouseclick
89
90          self.currentplayer = 0
91          # initialise variable to store player ID of current player
92
93          self.gameended = True
94          # initialise boolean that indicates if the game has ended
95
96          self.mainloop()
97          # calls mainloop
98
99      def start(self):
100         if self.modestr.get() == 'CPU vs Human':
101             self.mode = 0
102             # indicates computer plays first, human plays second
103         elif self.modestr.get() == 'Human vs CPU':
104             self.mode = 1
105             # indicates human plays first, computer plays second
106         elif self.modestr.get() == 'Multiple Humans':
107             self.mode = 2
108             # indicates there are multiple human players
109         self.gameended = False
```

```
        # sets self.gameended to False as game has not ended
        if self.mode != 2 and self.playerno.get() != 2:
            self.playerno.set(2)
            # override any number of players set if a computer player is playing
        self.disablebuttons()
        # disable all the dropdown menus and start button, enable the end button
        control.initial(self.vertex.get(), self.mode, self.playerno.get(), self)
        # initialise other aspects of the game from the initial procedure in control
        for i in range(self.vertex.get()):

 self.vertices.append([int(self.CANVASCENTER+self.RADIUS*cos(i*2*pi/self.vertex.get(
))), int(self.CANVASCENTER+self.RADIUS*sin(i*2*pi/self.vertex.get()))])
            # calculate the coordinates of the vertices
        self.draw(edgelist=[])
        # draw the graph with no edges connected
        self.playgame()
        # calls playgame which plays the game

    def checkifendgame(self):
        return control.checkend()
        # calls the checkend function in control to see if the game has ended

    def playgame(self):
        self.currentplayer = control.getturn()
        # gets player ID of current player's turn and stores in self.currentplayer
        self.displayplayer(self.currentplayer)
        # updates status and displays current player ID on the screen
        if self.currentplayer == self.mode and self.mode != 2:
        # self.mode also indicates the computer player ID if self.mode is 0 or 1
        # hence in this case if self.currentplayer is same as self.mode, then it is
the computer player's turn
            self.computermove = True
            # indicate that it is the computer player's turn
        else:
            self.computermove = False
            # indicate that it is not the computer player's turn
        if self.computermove:
            self.move = control.turnduring(self.currentplayer)
            self.checkmove()
            # if it is computer player's turn, we get the computer's move by calling
the turnduring function
            # it returns a move after calling minimax
            # then self.checkmove is called to check the validity of the move

    def checkmove(self):
        status = control.check(self.move, self.currentplayer)
        # the check function in control is called to check if the move is valid or
not
        # if the move is valid, it should return 0
        if status == 0:
        # move is valid
            self.updatewarningstatus('', 'black')
            # clear all warnings
            edgelist = control.turnend()
            # call the turnend function in control for a list of edges with player
numbers and vertices connected
            self.move = -1
            # reset self.move
            self.draw(edgelist)
            # calls the draw function to redraw the whole canvas
```

```
164         else:
165         # move is not valid
166             self.updatewarningstatus(f'Player {self.currentplayer}, please choose a
valid pair of vertices!', self.colour[self.currentplayer])
167             # warn the player to make a valid move by updating warning status
168         if self.checkifendgame() != -1:
169         # indicates game has ended and not in a draw
170             self.end(self.checkifendgame(), False)
171             # calls the self.end function to indicate which player has lost and it
is not a draw, or if the game is aborted
172         else:
173             if len(control.currentgame.GetPossibleMoves) == 0:
174             # if there are no more possible moves any player could make, the game
ends in a draw
175                 self.end(-1, True)
176             else:
177             # the game has not ended so we continue
178                 self.playgame()
179 
180     def end(self, playerlost=-1, draw=False):
181         self.gameended = True
182         # indicates that the game has ended
183         self.updatewarningstatus('', 'black')
184         # clear all warnings
185         if draw:
186         # game ended in a draw
187             self.updatestatus(f'Draw!', 'black')
188             # updates message to show that the game did end in a draw
189         else:
190             if playerlost != -1:
191             # there is a loser in the game
192                 self.updatestatus(f'Player {playerlost} lost!',
self.colour[playerlost])
193                 # updates message to indicate who lost
194             else:
195             # game aborted as the end button is clicked
196                 self.updatestatus(f'Game aborted', 'red')
197                 # updates message to reflect game aborted
198         self.enablebuttons()
199         # start button and drop down menus are enabled again
200         self.vertices = []
201         # clear all coordinates of the vertices
202 
203     def displayplayer(self, playerno):
204         message = f'It is now Player {playerno}\'s turn'
205         self.updatestatus(message, self.colour[playerno])
206         # displays the message 'It is now Player X's turn'
207 
208     def humanmove(self):
209         if self.computermove:
210         # so it is the computer player's turn
211             return -1
212             # the human player should not be making a move
213         self.choose = mergesort(self.choose)
214         # sort self.choose using merge sort
215         return (self.choose[0], self.choose[1])
216         # returns the move which is a tuple of two vertices
217 
218     def draw(self, edgelist):
219         self.canvas.delete('all')
```

```
220         # delete all existing canvas
221         self.drawvertex()
222         # draw all vertices
223         self.drawedge(edgelist)
224         # draw all edges created
225 
226     def drawvertex(self):
227         for i in range(len(self.vertices)):
228             self.canvas.create_oval(self.vertices[i][0]-10, self.vertices[i][1]-10,
    self.vertices[i][0]+10, self.vertices[i][1]+10, fill='black')
229             # draw each vertex as a circle with radius 10, centred at the
    coordinates as specified in self.vertices[i]
230 
231     def drawedge(self, edgelist):
232         for edge in edgelist:
233             self.canvas.create_line(self.vertices[edge[1]][0],
    self.vertices[edge[1]][1], self.vertices[edge[2]][0], self.vertices[edge[2]][1],
    fill=self.colour[edge[0]])
234             # draw each edge connecting the centre of the circles
235 
236     def updatestatus(self, text, colour):
237         self.status = text
238         # update status message to text
239         self.statuslabel.config(text=self.status)
240         self.statuslabel.config(fg=colour)
241         # update the status message and its colour
242 
243     def updatewarningstatus(self, text, colour):
244         self.warningstatus = text
245         # update warning status message to text
246         self.warningstatuslabel.config(text=self.warningstatus)
247         self.warningstatuslabel.config(fg=colour)
248         # update the warning status message and its colour
249 
250     def disablebuttons(self):
251         for object in self.inputs:
252             object.config(state = 'disabled')
253         # this disables all the dropdown menus
254         self.end_btn.config(state = 'normal')
255         # this enables the end button
256         self.start_btn.config(state = 'disabled')
257         # this disables the start button
258 
259     def enablebuttons(self):
260         for object in self.inputs:
261             object.config(state = 'normal')
262         # this enables all the dropdown menus
263         self.end_btn.config(state = 'disabled')
264         # this disables the end button
265         self.start_btn.config(state = 'normal')
266         # this enables the start button
267 
268     def mouseclick(self, event):
269         if not self.computermove:
270         # it is human's move
271             xcoor, ycoor = event.x, event.y
272             # get the coordinates of where the click occured
273             vertex = self.getnode(xcoor, ycoor)
274             # check which vertex do the coordinates correspond to
275             self.choose.append(vertex)
```

```
276             # adds that to the list that contains vertices that the human player's
    has clicked
277             if self.clicked:
278             # one vertex has already been clicked
279                 self.clicked = False
280                 # reset self.clicked
281                 self.unhighlight(self.choose[0])
282                 # unhightlight the first vertex
283                 self.move = self.humanmove()
284                 # call and get the move
285                 self.choose = []
286                 # reset self.choose
287                 if self.move != -1:
288                 # if the move is not attempted during the computer player's turn
289                     self.checkmove()
290                     # check if move is valid
291             else:
292             # no vertex has been clicked
293                 self.clicked = True
294                 # indicate a vertex is clicked
295                 if vertex != -1:
296                 # so it is a valid vertex
297                     self.highlight(vertex)
298                     # highlight the vertex that has just been selected
299         else:
300             self.updatewarningstatus('Please wait for the computer to finish its
    move!', 'red')
301             # warn the user to wait for the computer player to finish its move
302 
303     def getnode(self, x, y):
304         for i in range(len(self.vertices)):
305             if x >= self.vertices[i][0]-10 and x <= self.vertices[i][0]+10 and y >=
    self.vertices[i][1]-10 and y <= self.vertices[i][1]+10:
306             # checks each vertex to see if the clicked coordinates are within 10 of
    the centre of the vertex (in both x and y)
307                 return i
308                 # returns the vertex number
309         return -1
310         # indicates checked all vertices and none match
311 
312     def highlight(self, node):
313         self.canvas.create_oval(self.vertices[node][0]-10, self.vertices[node]
    [1]-10, self.vertices[node][0]+10, self.vertices[node][1]+10, fill='red')
314         # draws a red circle covering the existing vertex to indicate it has been
    chosen
315 
316     def unhighlight(self, node):
317         self.canvas.create_oval(self.vertices[node][0]-10, self.vertices[node]
    [1]-10, self.vertices[node][0]+10, self.vertices[node][1]+10, fill='black')
318         # draws a black circle covering the existing vertex to cover the red circle
    (which meant highlighted)
319 
320 startapp = GUI()
321 # creates a GUI object and starts the application
```

Minimax algorithm (minimax.py)

maxdepth

The maxdepth function (*maxdepth*) is found in lines 3-4 of the code. It creates different maximum search depths depending on the order of the graph.

If the graph has order 7 or below (less than or equal to 7 vertices), the maximum depth is 4. If the graph has order 8 or above (more than or equal to 8 vertices), the maximum depth is 3. This is to reduce the time that the computer player needs to return a move (too large depth will cause the algorithm to run slowly), while managing to run a reasonable depth of minimax (too small depth will cause inaccurate moves).

This function returns a boolean to see if the current search depth is the maximum search depth permitted i.e. it only returns true if the order is 7 or below and the current search depth is 4, or if the order is 8 or above and the current search depth is 3. **Objective 9**

evaluate

The evaluation function (*evaluate*) is found in lines 6-36 of the code in this document above, where each state of the board is evaluated according to the rules below:

For each possible move that could be made, the score is computed as follows:

1. The score is initialised as 0.
2. For each triangle that has been connected which is of the current player's colour, deduct the score by 2.
3. For each triangle that has been connected but not of the current player's colour, add the score by 1. It is implemented in the code by first counting the number of total triangles, and then deducting the number of triangles with the current player's colour.[3]
4. If there are unconnected triangles (2 edges exist but not the third), with one of the edges being the current player's colour, add 10 to the score for each such triangle.
5. If there are unconnected triangles (2 edges exist but not the third), with neither of the two edges being the current player's colour, deduct 1 from the score for each such triangle.

The total score for each state is determined by the sum of the scores of each possible move. **Objective 9a**

TestForTriangle

Lines 38-58 are the *TestForTriangle* function. The function checks if any triangle of the same colour appears on the board using a nested for-loop. The triangles that have been connected are stored using a list. It goes through every triple of vertices of the board and checks if all 3 of the edges connecting the vertices are of the same colour. If there are, then

[3] In the case where there are no triangles connected, the *TestForTriangle* function would return -1. Hence the code also takes that into consideration and takes away the number of '-1' to calculate. In such case, the score for that possible move would be 0.

the player's playerID is appended to the list. In the end, if no triangles are found, the function would return -1 indicating such case. Else it will return the list containing the playerID of the players which has triangles connected.

TestForUnconnected

Lines 60-83 are the *TestForUnconnected* function. The function checks if any unconnected triangle appears on the board using a nested for-loop. An unconnected triangle is defined as follows:

Suppose there are 3 vertices, A, B and C. If AB and BC are connected and of different colours, and AC is not connected, then we call triangle ABC an unconnected triangle.

The unconnected triangles are stored using a list. It goes through every triple of vertices of the board, and checks if all 3 of the edges connecting the vertices satisfy the condition as stated in the definition. If they satisfy the definition, then the player's playerID and the two unconnected vertices (A and C) are appended to the list. In the end, if no unconnected triangles are found, the function would return -1 indicating such case. Else it will return the list containing the playerID of the players who can form an unconnected triangle, along with the two vertices that they could connect.

minimax

Lines 85-177 are the minimax function (*minimax*) which uses depth-first search. It takes in 6 parameters, namely *computerplayer* (integer; playerID of the computer player), *representation* (2D array; adjacency matrix of the graph in the game), *ppmoves* (list; contains tuples containing each pair of unconnected vertices and the player could make those moves), *noofedges* (list; contains the number of edges connected to each vertex), *depth* (integer; current minimax search depth), *maxi* (boolean; indicating if the score is to be maximised (True) or minimised (False) at this node of the search tree).

To prevent any potential overwrite to the initial objects, a copy of *ppmoves* and *noofedges* is made. They are referred to as *possiblemoves* and *edgelst* in the remainder of the function. (Lines 92-93)

Base case 1: We start by checking if any triangles of the same colour have already been formed at the current state. The *TestForTriangle* function is called, with *representation* (the adjacency matrix of the graph) being passed on as a parameter. The return result is stored in *triangleplayer*. If *triangleplayer* contains *computerplayer* i.e. the player ID of the computer player, then the computer player has lost by this point. The evaluated score for this is -99999. If *triangleplayer* contains any other player that is not the player ID of the computer, then the other player has lost and the computer has won. The evaluated score for this is 99999. In these both cases, we do not proceed further as the game has already ended. Hence we simply return the evaluated scores at this layer to the last minimax call. (Lines 94-100)

In *player*, the playerID of the player whose score is being maximised at this layer is stored. If *maxi* is True, which indicates the score on this player should be maximised, then *player* is

being set to the computer playerID (*computerplayer*). As the computer player only plays in a 2-player game, if the computer's score is not being maximised, then the human's player score is being maximised i.e. the computer's score is being minimised. Then the human's playerID is stored in *player*. (Lines 101-104)

Base case 2[4]: We then proceed to a special case of the minimax function, which is the opening moves. Initially, some vertices are not connected to any other vertex, and we would attempt to connect them (connect two vertices that each have 0 edges). We add all indices of *edgelst* where the element is equal to 0 (indicating that no edges are connected to them) to a list called *notconnected*. If there are 2 or more indices in *notconnected*, then the computer randomly chooses 2 of the indices and returns that as their move. (Lines 105-119)

Base case 3[5]: If *possiblemoves* only includes 1 element, and we are in the root of the search tree (so the minimax search has just started), then the computer player has to make the only possible move which is specified in *possiblemoves*. The tuple containing the only possible move is hence returned. (Lines 120-122)

Base case 4: If we have reached the bottom layer of the minimax search, or if *possiblemoves* only includes 1 element and we are not currently in the root, then we call *evaluate* (the evaluation function) to assign scores to each move. (Lines 123-125)

It can be shown that in any state of the game, it falls into one of the 4 base cases above.

Objective 9b

score is a list that stores the scores assigned to each possible move (in order of the moves in *possiblemoves*). A for-loop is implemented to update the adjacency matrix accordingly to each possible move, and call the minimax algorithm recursively (**Objective 9b**) with that updated representation of the graph. *edgelst* is also updated as it stores how many edges are connected to each vertex. After the minimax call, the assigned score is appended to *score*, and the graph (*representation*) and *edgelst* is reset to its original state, preparing for checking the next possible move. (Lines 126-146)

We call *maxdepth* to check the current search depth. (Line 147-148)

If *maxdepth* returns True, that means we are in the root of the search tree, and a tuple should be returned as the computer player's move. We first check if all moves guarantee loss, which is indicated by *Flag*. If *Flag* is True, that means there is at least a move that does not guarantee a loss. Hence the computer can play the move with the highest score assigned to it, and that value is returned. Otherwise, if *Flag* is False, then every possible move is checked in a for-loop. If there is a possible move that does not make the computer player lose immediately, then that move is made by the computer. Else if every single move

[4] This is added during implementation as when testing the minimax, the computer player always starts with the same moves which makes the game boring. This base case adds a bit of randomness.
[5] This is added during implementation as during testing, it is released that the minimax call cannot proceed when the number of possible moves is 0. Hence this case is added to fix that problem.

makes the computer player lose immediately, the computer just plays the first one of them. (Line 149-169)

If *maxdepth* returns False, that means we are not at the root of the search tree. In this case, we should return the maximum or minimum score, depending on if we are in a maximising/minimising layer of the search. The maximum/minimum value in *score* is then returned. (Lines 170-177)

```
from random import randint

def maxdepth(depth, representation):
    return (depth == 4 and len(representation) <= 7) or (depth == 3 and
len(representation) >= 8)

def evaluate(representation, ppmoves, playerID):
# evaluation function
    possiblemoves = ppmoves.copy()
    localscore = 0
    # initialise evaluation score variable
    for (node1, node2) in possiblemoves:
        # try every possible move and update board accordingly
        representation[node1][node2] = playerID
        representation[node2][node1] = playerID
        # above 2 lines update board
        player = TestForTriangle(representation)
        # test if someone wins/loses
        specialmove = TestForUnconnected(representation)
        # test if there is an unconnected triangle with two edges belonging to
different players
        localscore -= player.count(playerID)*2
        # 2 is an arbitrary constant
        localscore += (len(player)+(-1)*player.count(playerID)+
(-1)*player.count(-1))
        # -1 is an arbitrary constant
        if specialmove != [-1]:
            for tup in specialmove:
                if tup[0] == playerID:
                    localscore += 10
                    # 10 is an arbitrary constant
                else:
                    localscore -= 1
                    # -1 is an arbitrary constant
        representation[node1][node2] = -999
        representation[node2][node1] = -999
        # above 2 lines reset board
    return localscore
    # return evaluation score

def TestForTriangle(representation):
    order = len(representation)
    # number of vertices in board
    lst = []
    # used to store which player(s) has connected triangles
    for nodeone in range(order):
        for nodetwo in range(nodeone+1, order):
            for nodethree in range(nodetwo+1, order):
                if representation[nodeone][nodetwo] == representation[nodeone]
[nodethree] and representation[nodeone][nodethree] == representation[nodetwo]
[nodethree] and representation[nodeone][nodetwo] != -999 and representation[nodeone]
[nodetwo] != -1:
                    lst.append(representation[nodeone][nodetwo])
    # checking each triple of vertices using a nested for-loop.
    # If the edges connecting are of the same player, it is added to lst.
    # Note that if the player is -999, it means it is unconnected.
    # Note that if the player is -1, it means the pair cannot be connected (two
vertices are the same).
    if len(lst) == 0:
    # indicating no triangles are connected
        lst.append(-1)
        # no player has connected triangles
    return lst
    # returns the list of players that has triangles connected.
    # It also indicates how many triangles are connected for each player.

def TestForUnconnected(representation):
    order = len(representation)
    # number of vertices in board
    lst = []
    # used to store which player(s) and which pair(s) of vertices can do an
unconnected triangle
    for nodeone in range(order):
        for nodetwo in range(order):
            for nodethree in range(order):
                if representation[nodeone][nodetwo] != representation[nodeone]
[nodethree] or representation[nodeone][nodethree] == representation[nodetwo]
```

```
[nodethree] and representation[nodeone][nodetwo] == 0 and representation[nodeone]
[nodethree] != -999 and representation[nodeone][nodethree] != -1:
                    if (representation[nodeone][nodetwo], nodeone, nodetwo) or
(representation[nodeone][nodetwo], nodetwo, nodeone) in lst:
                        continue
                    else:
                        lst.append((representation[nodeone][nodetwo], nodeone,
nodetwo))
    # checking each triple of vertices using a nested for-loop.
    # If among the pairs of vertices of the 3, 2 edges belong to different players
and 1 edge is unconnected, it is recorded in lst.
    # Note that if the player is -999, it means it is unconnected.
    # Note that if the player is -1, it means the pair cannot be connected (two
vertices are the same).
    # This loop goes through every triple 3 times. Duplicate records are checked and
hence not stored.
    if len(lst) == 0:
    # indicating there are no such unconnected triangles
        lst.append(-1)
        # no player has unconnected triangles
    return lst
    # returns the list of unconnected triangles and the corresponding player that
can make a move.

def minimax(computerplayer, representation, ppmoves, noofedges, depth, maxi):
    # computerplayer stores the playerID of the computer player
    # representation is adjacency matrix of graph
    # ppmoves is list of tuples containing (node1, node2) which are possible moves
    # noofedges is list of number of edges connected to each vertex
    # depth is minimax search depth
    # maxi is True if we are maximising the score, else False if we are minimising
the score
    possiblemoves = ppmoves.copy()
    edgelst = noofedges.copy()
    triangleplayer = TestForTriangle(representation)
    if computerplayer in triangleplayer:
    # the computer player has connected a triangle (and hence lost); base case
        return -99999
    elif -1 not in triangleplayer:
    # the human player has connected a triangle (and hence the computer did not
lose); base case
        return 99999
    if maxi: player = computerplayer
    # we are maximising the score of the computer player (as it is the computer
player's turn)
    else: player = int(not computerplayer)
    # we are maximising the score of the human player (as it is the human player's
turn) i.e. minimising the score of the computer player
    notconnected = []
    # list of vertices that are not connected to any other vertex
    for i in range(len(edgelst)):
        if edgelst[i] == 0:
            notconnected.append(i)
    # this loop stores the vertices that has 0 edges connected to it
    if len(notconnected) >= 2:
    # this is only used in when at least 2 vertices are not connected to anything
else; base case
        a = 0
        b = 0
        while a == b:
            a = randint(0, len(notconnected)-1)
            b = randint(0, len(notconnected)-1)
        return (notconnected[min(a, b)], notconnected[max(a, b)])
        # return a random unconnected pair of vertices as computer's move
    if maxdepth(depth, representation) and len(possiblemoves) == 1:
    # if there is only one possible move, that move must be made; base case
        return possiblemoves[0]
    if depth == 0 or len(possiblemoves) == 1:
        return evaluate(representation, possiblemoves, computerplayer)
        # evaluates the current state of the game; base case
    score = []
    # stores the score for each move
    for (node1, node2) in possiblemoves:
        # loop to minimax the board with every single possible move
        lst = possiblemoves.copy()
        lst.remove((node1, node2))
        # removes the current pair of vertices from the temporary list of possible
moves
        representation[node1][node2] = player
```

```
        representation[node2][node1] = player
        # above 2 lines update the board
        edgelst[node1] += 1
        edgelst[node2] += 1
        # above 2 lines updates the edge count for each vertex
        score.append(minimax(computerplayer, representation, lst, edgelst, depth-1,
not maxi))
        # calls the minimax function recursively
        representation[node1][node2] = -999
        representation[node2][node1] = -999
        # above 2 lines reset the board
        edgelst[node1] -= 1
        edgelst[node2] -= 1
        # above 2 lines reset the edge count
    if maxdepth(depth, representation):
    # At maxdepth, a move has to be returned for the computer player to play
        flag = False
        # to check if all scores evaluated are -99999 (all losing for computer)
        for scr in score:
        # score is the list of scores for every pair of possible moves
            if scr != -99999:
                flag = True
        if flag:
        # so not all scores are -99999
            return possiblemoves[score.index(max(score))]
            # maximising the score for the computer
        # if all -99999, do any move that does not immediately lose
        for (node1, node2) in possiblemoves:
            representation[node1][node2] = player
            representation[node2][node1] = player
            if computerplayer not in TestForTriangle(representation):
                # computerplayer does not immediately lose
                return (node1, node2)
            representation[node1][node2] = -999
            representation[node2][node1] = -999
        return possiblemoves[0]
        # if every move is immediately lose, just do the first move
    else:
    # not at maxdepth; so only max/min score has to be returned depending on layer
of minimax
        if maxi:
            return max(score)
            # max(score) when in maximising layer
        else:
            return min(score)
            # min(score) when in minimising layer
```

Game flow algorithm (control.py)

This algorithm controls the flow of the Sim game and interacts with the GUI.

initial

The *initial* procedure is called by the GUI object when a new game is being initialised. 4 parameters are taken in, including *vertex* (integer; order of the graph, number of vertices in the game), *mode* (integer; game mode; 0 is the computer playing first in a 2-player game, 1 is the computer playing second in a 2-player game, 2 is a multiplayer human game), *playerno* (integer; the number of players in the game), *gui* (*GUI* object; the *GUI* object that initialised the procedure call). (Lines 5-12)

The *currentgame* then creates and stores a *Game* object for the game, and *gamegraph* creates and stores a *Graph* object for the game. *playermode* stores the playermode as indicated in *mode*, and *interface* stores the *GUI* object as specified in *gui*. (Lines 13-20)

getturn

The *getturn* function returns the playerID of the player whose current move is now. It does that by getting the *GetCurrentTurn* property from *currentgame* (which is a *Game* object). (Lines 22-26)

turnduring

The *turnduring* function controls the interaction between the GUI and the game when it is the computer player's turn. It takes in 1 parameter, *currentplayer* (integer; the playerID of the player whose current move is now). If it matches the computer player ID, then *computermove* is called and its result (*move*; tuple that contains the computer player's move) is returned to the GUI. (Lines 28-34)

check

The *check* function checks if a move is valid or not. It takes in 2 parameters, *move* (tuple; contains the move to be made) and *currentplayer* (integer; the playerID of the player whose current move is now). It first checks if *move* is in the list of possible moves (*currentgame.GetPossibleMoves*). If it is not, this means the attempted move is invalid, and a return value of -1 alerts that to the GUI. If it is, then the attempted move is valid. The move is accepted, with the move being removed from the list of possible moves and the game graph representation is updated. This is achieved by calling *currentgame.UpdatePossibleMoves* which removes that tuple, and calls *gamegraph.UpdateEdge* to update the adjacency matrix of the graph. It returns 0 to tell the GUI that it has accepted the move and the properties of the game have been updated. (Lines 36-46)

computermove

The *computermove* function controls the initial *minimax* call. It checks the order of the graph (*currentgame.GetVertexNo*) and calls the *minimax* function with different search

depths. If the order of the graph is less than or equal to 7, the search depth is 4. Else the search depth is 3. (Lines 48-53)

turnend

The *turnend* function calls the *currentgame.UpdateCurrentTurn* function to reflect that it is the next player's turn. It then returns the edge list (*gamegraph.GetEdgeList*) which contains tuples of vertices alongside the player that connected the two vertices. The GUI receives the return value and it could update the canvas. (Lines 55-62)

checkend

The *checkend* function returns if the game has ended or not i.e. if a triangle of the same colour has been drawn. That is achieved with the *gamegraph.TestForTriangle* function. (Lines 64-66)

```
from game import Game
from graph import Graph
from minimax import minimax

def initial(vertex, mode, playerno, gui):
    # this procedure is called when a new game is being initialised
    # vertex is the number of vertices in the game
    # mode is the game mode; 0 indicates computer plays first, 1 indicates computer plays second, 2 indicates no computer player is involved i.e. multiplayer
    # playerno is the number of players in the game
    # gui is the GUI object that calls this function
    global currentgame, playermode, gamegraph, interface
    # the contents of these variables can be reused in other function/procedure calls in this file
    currentgame = Game(playerno, vertex)
    # creates a Game object for the game
    gamegraph = Graph(currentgame)
    # creates a Graph object for the game
    playermode = mode
    # indicates the playermode as discussed above
    interface = gui
    # stores the GUI object that has initialised the game

def getturn():
    global currentgame
    currentturn = currentgame.GetCurrentTurn
    # gets the playerID of the current player from the method in the Game object
    return currentturn

def turnduring(currentplayer):
    if playermode != 2 and currentplayer == playermode:
        # indicating it is the CPU's turn
        move = computermove(currentplayer)
        # calls the computermove function
    return move
    # returns the computer's move back to the GUI

def check(move, currentplayer):
    # checks if a move is valid or not
    if move not in currentgame.GetPossibleMoves:
        return -1
        # indicating that the move is invalid and nothing has been updated
    currentgame.UpdatePossibleMoves(move[0], move[1])
    # removes the current move from the list of tuples of possible moves
    gamegraph.UpdateEdge(currentplayer, move[0], move[1])
    # adds the edge of the current move to the graph
    return 0
    # indicates that the move is valid and has been accepted; it is the next player's turn now

def computermove(currentplayer):
    # returns a move for the computer player based on the result of the minimax algorithm
    if currentgame.GetVertexNo <= 7: # graph order less than or equal to 7 has minimax max depth 4
        return minimax(currentplayer, gamegraph.representation, currentgame.GetPossibleMoves, currentgame.GetNoOfEdges, 4, True)
    else: # graph order greater than or equal to 8 has minimax max depth 3
        return minimax(currentplayer, gamegraph.representation, currentgame.GetPossibleMoves, currentgame.GetNoOfEdges, 3, True)

def turnend():
    global currentgame
    currentplayer = currentgame.GetCurrentTurn
    # indicates the playerID of the current player
    currentgame.UpdateCurrentTurn(currentplayer)
    # updates the current turn to reflect that it is the next player's turn
    return gamegraph.GetEdgeList
    # returns the edge list to the GUI for it to update the canvas

def checkend():
    return gamegraph.TestForTriangle()
    # returns what triangles have been formed in the graph
```

Game object (game.py)

__init__

The class *Game* takes in two parameters when it is created by the constructer, *playerno* (integer; the number of players in the game) and *vertex* (integer; the number of vertices in the graph). It has 5 private properties, including *__currentturn* (integer; holds the player ID of player of the current turn), *__playerno* (integer; the number of players in the game), *__vertex* (integer; the number of vertices in the graph), *__possiblemoves* (list of tuples; holds tuples for every pair of unconnected vertices) and *__noofedges* (list of integers; stores the number of edges connected to each vertex). (Lines 1-7)

__possiblemoves is initialised by using a for-loop to add every single pair of vertices into the list. The tuples will be removed when the corresponding move has been played. *__noofedges* is initialised by using a for-loop to set 0 for all the vertices (as each vertex has 0 edges connected to them initially). (Lines 8-11)

Access methods

As *Game* is encapsulated and has private properties, access methods are created to return what's stored in the properties. These include *GetCurrentTurn*, *GetPlayerNo*, *GetVertexNo*, *GetPossibleMoves* and *GetNoOfEdges*, in which the method names are self-explanatory. (Lines 13-34)

UpdatePossibleMoves

After a move has been made, it has to be removed from *__possiblemoves*. Hence a method *UpdatePossibleMoves* is created, taking in 2 parameters *nodeone* and *nodetwo* (which specify which vertices have just been connected). It removes the tuple from *__possiblemoves* and updates *__noofedges* accordingly as each of the two vertices has 1 more edge connected to each of them. (Lines 36-39)

```
class Game:
    def __init__(self, playerno, vertex):
        self.__currentturn = 0 # indicate player number of current turn
        self.__playerno = playerno # indicate number of players in the game
        self.__vertex = vertex # indicate number of vertices in graph
        self.__possiblemoves = [] # initialize empty list to indicate possible moves
        self.__noofedges = [] # initialize empty list to indicate number of edges connected to each vertex
        for node in range(vertex):
            for node2 in range(node+1, vertex):
                self.__possiblemoves.append((node, node2)) # append each tuple of possible moves into list
            self.__noofedges.append(0) # initialize number of edges connected to each vertex to 0

    @property
    def GetCurrentTurn(self):
        return self.__currentturn # return who's turn it is now

    @property
    def GetPlayerNo(self):
        return self.__playerno # return how many players are there

    @property
    def GetVertexNo(self):
        return self.__vertex # return number of vertices in game

    @property
    def GetPossibleMoves(self):
        return self.__possiblemoves.copy() # return a copy of possible moves that can be made by all players (list)

    @property
    def GetNoOfEdges(self):
        return self.__noofedges.copy() # return a copy of the number of edges connected to each vertex (list)

    def UpdateCurrentTurn(self, currentturn):
        self.__currentturn = (currentturn+1)%(self.__playerno) # indicates that it is next player's move. player numbers are from 0 to self.__playerno-1 so % is used to deal with the loop

    def UpdatePossibleMoves(self, nodeone, nodetwo):
        self.__possiblemoves.remove((nodeone, nodetwo)) # remove tuple from possible moves as that move has been made
        self.__noofedges[nodeone] += 1 # indicate the first node has one new edge connected to it
        self.__noofedges[nodetwo] += 1 # indicate the second node has one new edge connected to it
```

Graph object (graph.py)

__init__

The class *Graph* takes in one parameter when it is created by the constructer, *Game* (*Game* object; provides *Game* object of the current game). It has 4 public properties, *playerno* (integer; the number of players in the game), *order* (integer; the number of vertices in the graph), *representation* (2-dimensional list; adjacency matrix of the graph) and *edgelist* (list of triplets; contains information about each joined edge, including the player that created the edge and the vertex number of the two ends of the edge). *playerno* is initialised by accessing *__playerno* in the *Game* object (*Game.GetPlayerNo*), and *order* is initialised by accessing *__vertex* in the *Game* object (*Game.GetVertexNo*). (Lines 1-6)

representation is then initialised by assigning -999 (indicating that the pair of vertices have not been connected) for all elements in the adjacency matrix. However, as one obviously cannot join a vertex to itself (vertex 1 to vertex 1, vertex 2 to vertex 2, etc), those values are assigned as -1 instead, to indicate an edge cannot be created between them. These are implemented using 2 for-loops. (Lines 7-13).

TestForTriangle

The *TestForTriangle* method is the exact same as the one in the minimax algorithm. The function checks if any triangle of the same colour appears on the board using a nested for-loop. The triangles that have been connected are stored using a list. It goes through every triple of vertices of the board, and checks if all 3 of the edges connecting the vertices are of the same colour. If there are, then the player's playerID is appended to the list. In the end, if no triangles are found, the function would return -1 indicating such case. Else it will return the list containing the playerID of the players which has triangles connected. (Lines 15-21)

UpdateEdge

The *UpdateEdge* method updates *representation* and *edgelist* when a move is made. It takes in 3 parameters, *player* (integer; player ID of the player who made the move), *nodeone* and *nodetwo* (integer; vertex ID of vertices to be connected). It updates the adjacency matrix (*representation*) so that the edge connecting the two vertices is now assigned value *player* instead of -999. It then updates *edgelist* to include the triplet (*player, nodeone, nodetwo*) of the edge just connected. (Lines 23-26)

GetEdgeList

The *GetEdgeList* method returns *edgelist* when it is called. (Lines 28-30)

```
class Graph:
    def __init__(self, Game):
        self.playerno = Game.GetPlayerNo # number of players in game
        self.order = Game.GetVertexNo # number of vertices (order of a graph)
        self.representation = [] # adjacency matrix to represent the graph
        self.edgelist = [] # list to store triplets of playerno, vertex1, vertex2 of joined edges
        for nodeone in range(self.order):
            lst = [] # each row in the adjacency matrix
            for nodetwo in range(self.order):
                lst.append(-999) # sets it to -999 to indicate the edge is not occupied
            self.representation.append(lst) # initialize the adjacency matrix
        for node in range(self.order):
            self.representation[node][node] = -1 # indicates it cannot be changed

    def TestForTriangle(self):
        for nodeone in range(self.order):
            for nodetwo in range(nodeone+1, self.order):
                for nodethree in range(nodetwo+1, self.order):
                    if self.representation[nodeone][nodetwo] == self.representation[nodeone][nodethree] and self.representation[nodeone][nodethree] == self.representation[nodetwo][nodethree] and self.representation[nodeone][nodetwo] != -999: # check if it forms a triangle; if same player forms triangle and it's not 3 empty edges
                        return self.representation[nodeone][nodetwo] # only 1 triangle at most at the first instance when a triangle is formed
        return -1 # indicating no triangle is formed

    def UpdateEdge(self, player, nodeone, nodetwo):
        self.representation[nodeone][nodetwo] = player # update edge to indicate occupied by player
        self.representation[nodetwo][nodeone] = player # update edge to indicate occupied by player
        self.edgelist.append((player, nodeone, nodetwo))

    @property
    def GetEdgeList(self):
        return self.edgelist
```

Merge sort (sort.py)

A merge sort algorithm is implemented. It splits up the whole list into halves every time (lines 3-5) and recursively calls itself (lines 6-7) until there is only 1 element left, where it fails the condition in line 2 and skips to line 36 as a result (returns the list with 1 element). After the left half of the list (*left*) and the right half of the list (*right*) are sorted, they are then combined again as a sorted list.

i is used to keep track of the current index of *left*, and *j* is used to keep track of the current index of *right*. The current index of *lst* (which is the original list) is kept track of by *k*. We compare *left[i]* and *right[j]*. If *left[i]* is greater, then we assign its value to *lst[k]* (lines 13-16). If *right[j]* is greater, then we assign its value to *lst[k]* (lines 17-20). *i* or *j* are incremented accordingly as we move on to the next element in *left* or *right*. If *left[i]* has the same value as *right[j]*, then we assign their values to two consecutive elements in *lst* (lines 21-27). *k* is incremented accordingly depending on which case we fall into (line 25, line 28). This process is repeated until *i* is equal to the number of elements in *left*, or *j* is equal to the number of elements in *right* (line 12).

If all the elements in *right* are inserted into *lst* but not all the elements in *left* are inserted after leaving the loop of lines 12-28, all the elements of *left* are inserted into *lst* in order (lines 30-35). Similarly, if all the elements in *left* are inserted but not all of *right* is inserted, all the elements in *right* are inserted into *lst* in order (lines 37-42).

Finally, the sorted list *lst* is returned (line 43).

```
def mergesort(lst):
    if len(lst) > 1:
        mid = int(len(lst)/2) # finding the middle index of the list
        left = lst[:mid] # this gives the first half of the list
        right = lst[mid:] # this gives the second half of the list
        mergesort(left) # recursively calls the mergesort function on the first half of the list
        mergesort(right) # recursively calls the mergesort function on the second half of the list
        i = 0
        j = 0
        k = 0
        # initialising indicies for lists; i for left list, j for right list, k for combined list
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
            # first element in left list is smaller than that of right list
                lst[k] = left[i] # we put left element into combined list
                i += 1 # increment index of left list
            elif right[j] < left[i]:
             # first element in right list is smaller than that of left list
                lst[k] = right[j] # we put right element into combined list
                j += 1 # increment index of right list
            else:
            # first element of left list is same as right list so we put both of them into the combined list
                lst[k] = left[i]
                i += 1 # increment index of left list
                k += 1 # increment index of combined list so right list element can be inserted
                lst[k] = right[j]
                j += 1 # increment index of right list
            k += 1 # increment index of combined list for next element

        while i < len(left):
        # this happens when the entire right list has been inserted into the combined list
        # we need to insert every element of left list into combined list now
            lst[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
        # this happens when the entire left list has been inserted into the combined list
        # we need to insert every element of right list into combined list now
            lst[k] = right[j]
            j += 1
            k += 1
    return lst
    # returns the combined list (which is sorted)
```

Testing

In this section, we aim to list the multiple tests that were conducted as part of the testing video. The full testing video could be found at **https://youtu.be/bMLJoBCPlek**.

When testing and improving the minimax algorithm, a command line user interface has been written for testing purposes. The full code for that could be found in **Appendix A**.

List of tests

Below is a table of tests conducted and the associated test number with test results:

Test number	Test description	Related objective	Expected result	Test passed?
1	Start application.	N/A	Message 'Welcome to Sim' displayed and game option menus are enabled.	**Yes**
2	Open dropdown menu to choose number of players and change to 3/4/5.	1, 1a	Number of players should be changed.	**Yes**
3	Open dropdown menu to choose and change game mode to Multiple Humans.	1, 1a	Game mode should be changed.	**Yes**
4	Open dropdown menu to choose and change game mode to Human vs CPU.	2, 2a	Game mode should be changed.	**Yes**
5	Open dropdown menu to choose and change game mode to CPU vs Human.	2, 2a	Game mode should be changed.	**Yes**
6	Open dropdown menu to choose number of vertices and change to 7/8/9/10.	3	Game mode should be changed.	**Yes**
7	Start game with following configuration: Number of players: 4 Game mode: CPU vs Human Number of vertices: 6	1, 1a, 2, 2a, 3, 9, 9a, 9b	Game should start accordingly with dropdown menus all disabled. Number of players should change to 2.	**Yes**
8	Press the end button to abort the game during the game.	4a	Game should be aborted with warning message shown.	**Yes**

9	Play a full game with the computer with following configurations: Number of players: 2 Game mode: Human vs CPU Number of vertices: 6	4, 5, 8, 8a, 9, 9a, 9b	Game would display loser of the game. Expect CPU win as second player has winning strategy. During the game, the game options are disabled.	**Yes**
10	Attempt to connect 2 connected vertices in the graph.	7	Warning message shown in player's colour; move disregarded.	**Yes**
11	As a human player, choose 1 vertex when it's the human player's turn.	6	The vertex should be highlighted with a red circle.	**Yes**
12	As a human player, draw a monochromatic triangle intentionally to lose the game.	8, 8a	Game would display loser of the game.	**Yes**
13	Play a game with the following configurations: Number of players: 5 Game mode: Multiple Humans Number of vertices: 6 Make the game end in a draw. Note: test changed as accidentally failed to end the initial game in a draw.	N/A	When the game ends, a message will display to indicate that it is a draw. During the game, the game options are disabled.	**Yes**
14	Play a game with the following configurations: Number of players: 3 Game mode: Multiple Humans Number of vertices: 10 Deliberately draw a monochromatic triangle to end the game after at most 20 turns.	8, 8a	Game would display loser of the game. During the game, the game options are disabled.	**Yes**
15	Start any game and attempt to press the start button during the game.	N/A	Game options are disabled so the software should not react to such click.	**Yes**

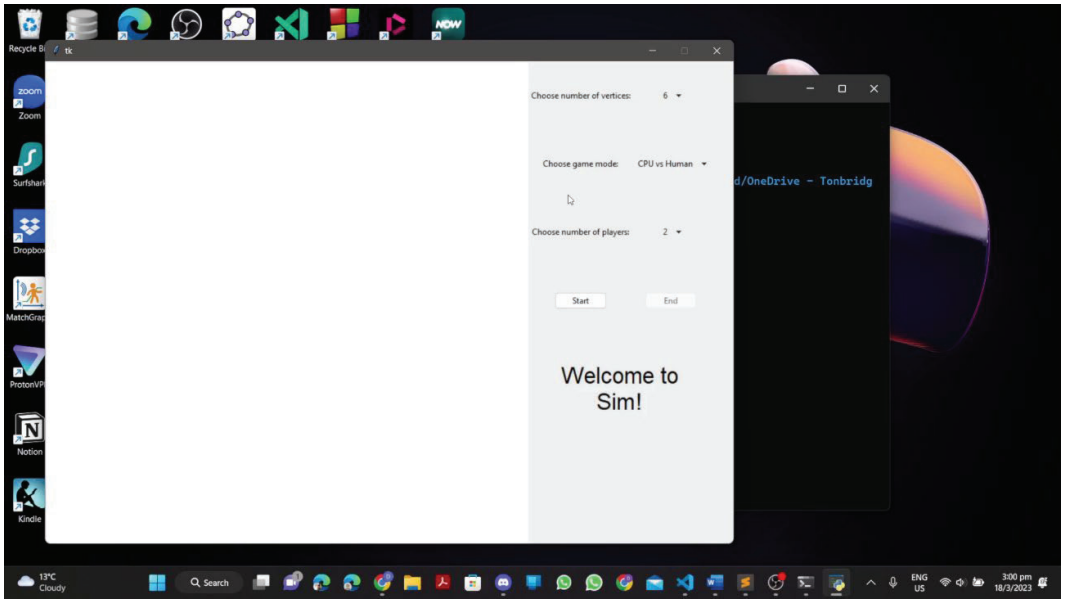
16	Start any game and attempt to change the dropdown menus options during the game.	N/A	Game options are disabled so the software should not react to such click.	**Yes**
17	Press the end button before starting a game.	N/A	End button is disabled so the software should not react to such click.	**Yes**

Evidence of testing

The full testing process is recorded in the testing video (**https://youtu.be/bMLJoBCPlek**).

Test 1

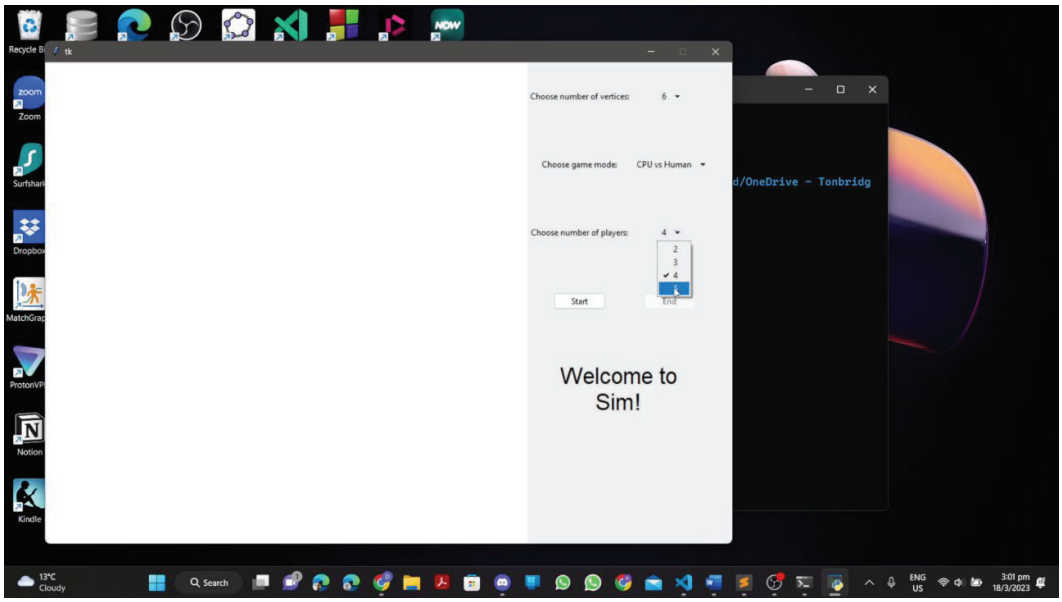
https://youtu.be/bMLJoBCPlek?t=30



Application started from command prompt and correctly displays status message ‘Welcome to Sim!’.

Test 2

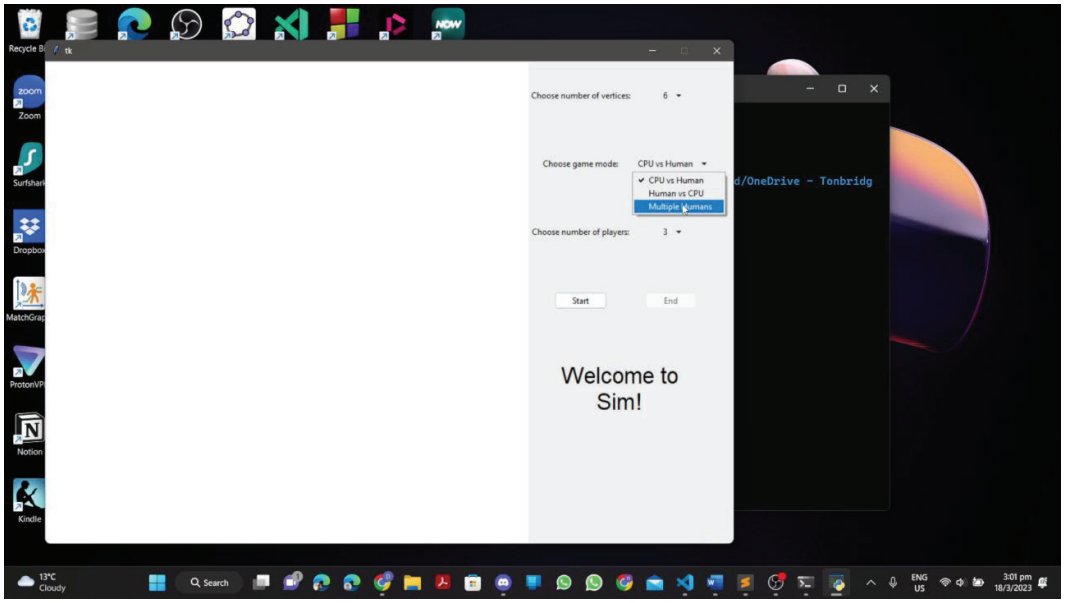
https://youtu.be/bMLJoBCPlek?t=62



Choosing 5 players. Dropdown menu is shown.

Test 3

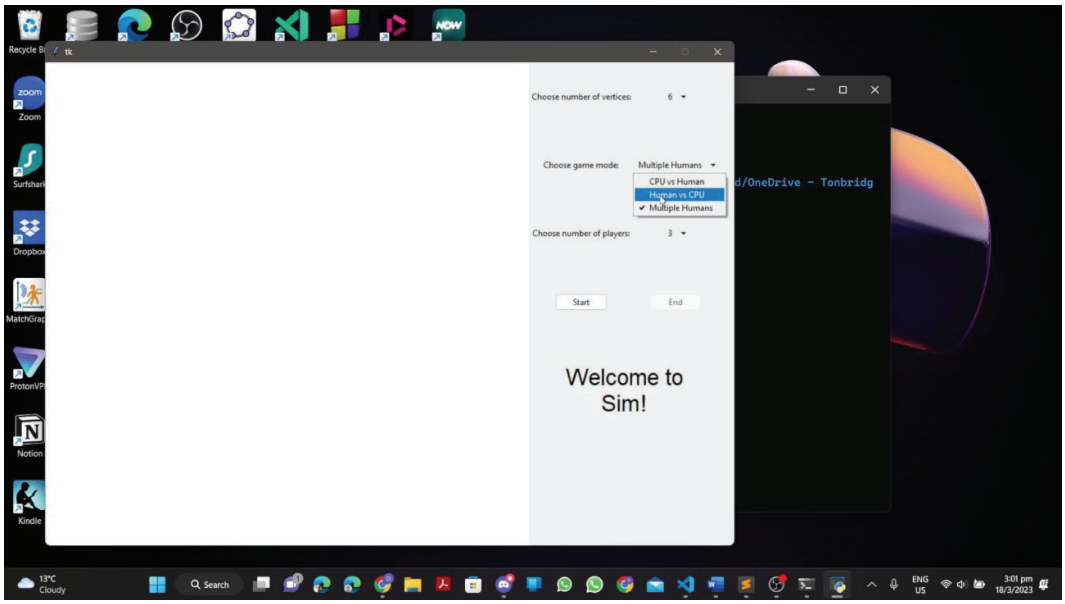
https://youtu.be/bMLJoBCPlek?t=93



Choosing Multiple Humans. Dropdown menu is shown.

Test 4

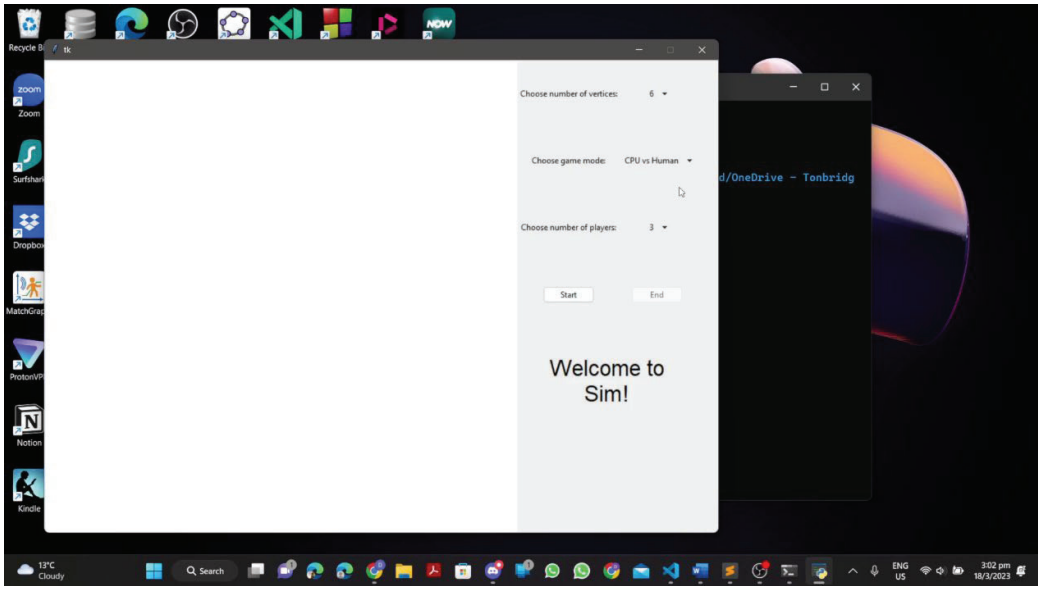
https://youtu.be/bMLJoBCPlek?t=111



Choose Human vs CPU. Dropdown menus as shown.

Test 5

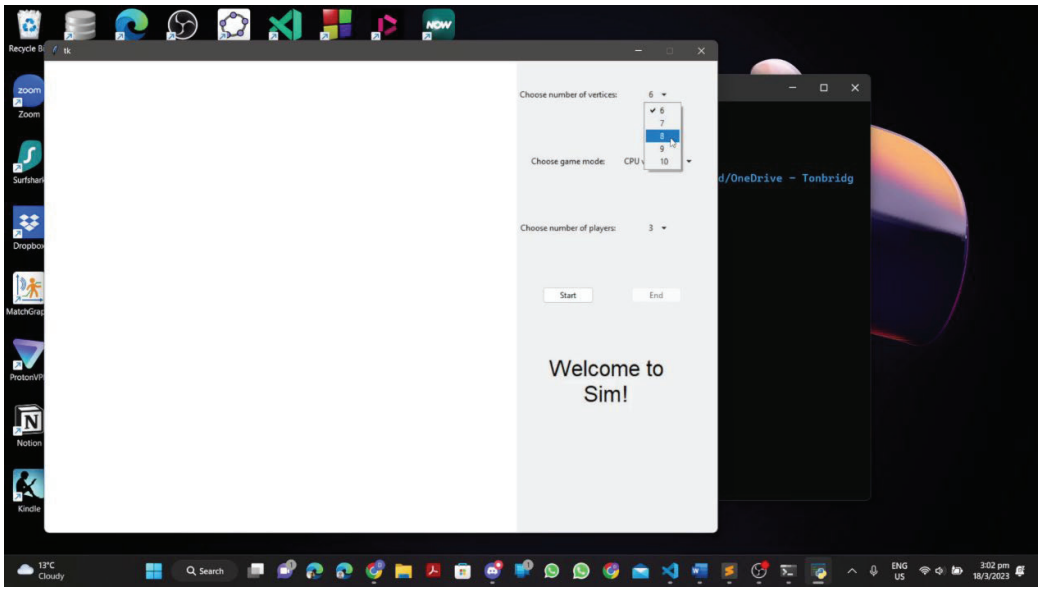
https://youtu.be/bMLJoBCPlek?t=142



Choose CPU vs Human as shown.

Test 6

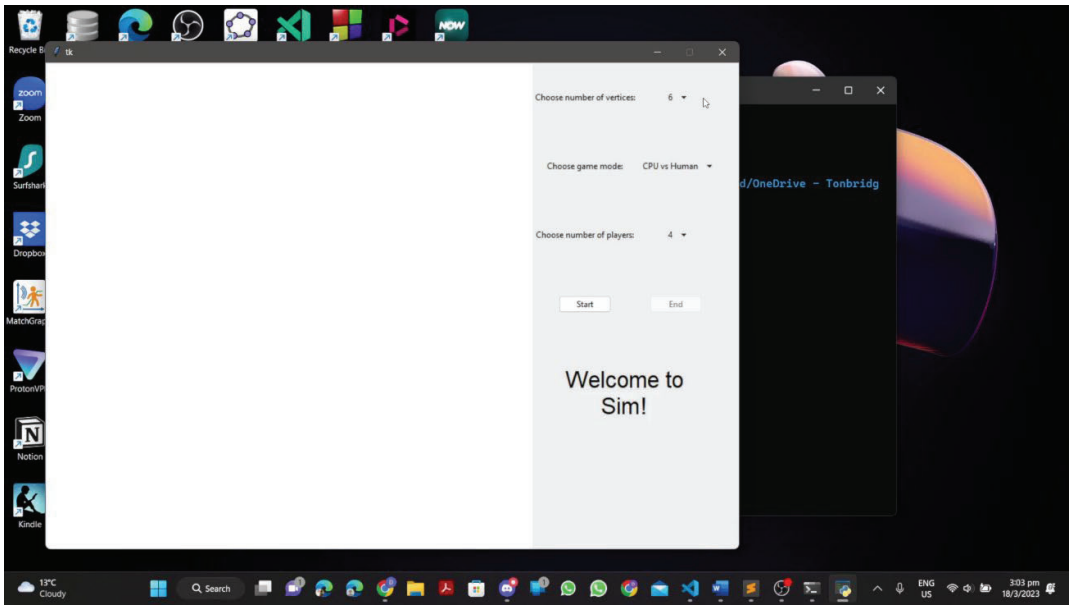
https://youtu.be/bMLJoBCPlek?t=160



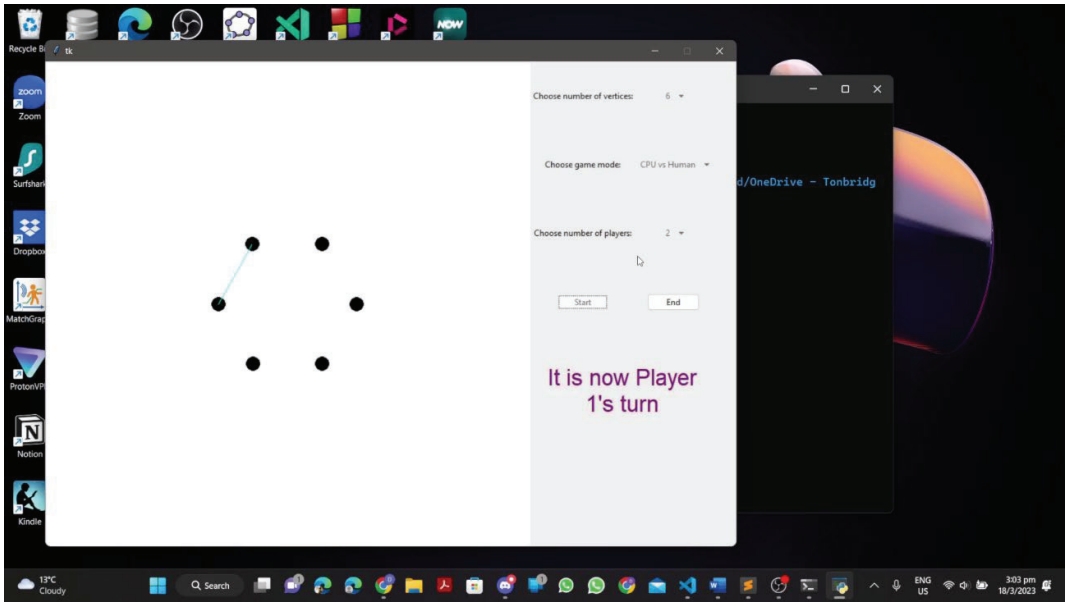
Choose 8 vertices. Dropdown menus as shown.

Test 7

https://youtu.be/bMLJoBCPlek?t=182



Game options initialised as shown.

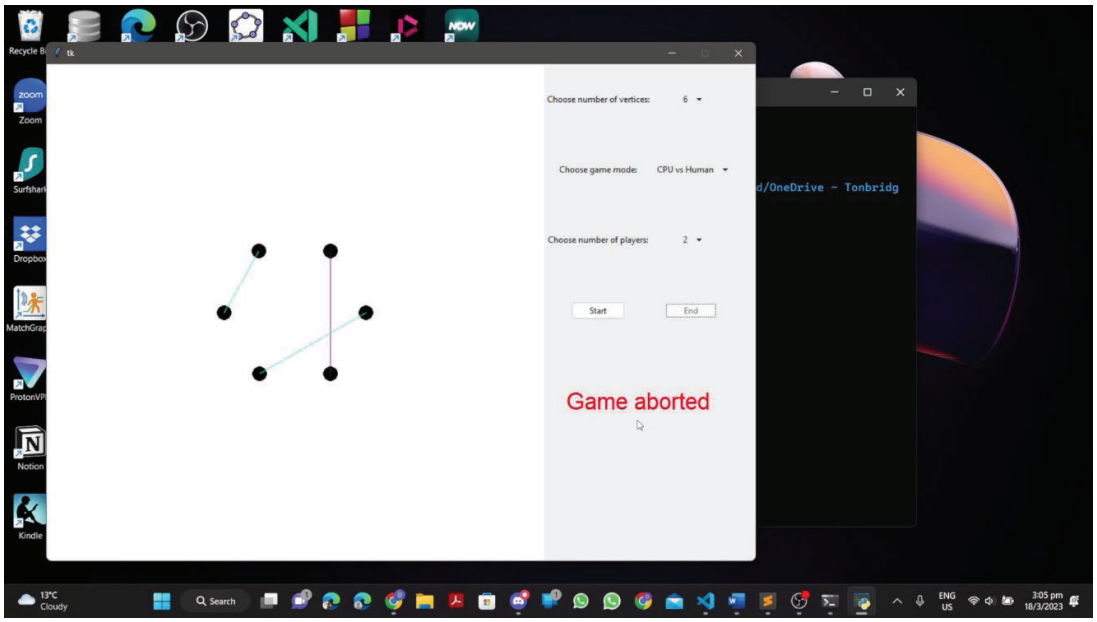


Game option menus is disabled, and number of players is changed to 2.

The game board shows up with the computer player (player 0; first player) made its move already. It is now the human player's turn to make a move.

Test 8

https://youtu.be/bMLJoBCPlek?t=293

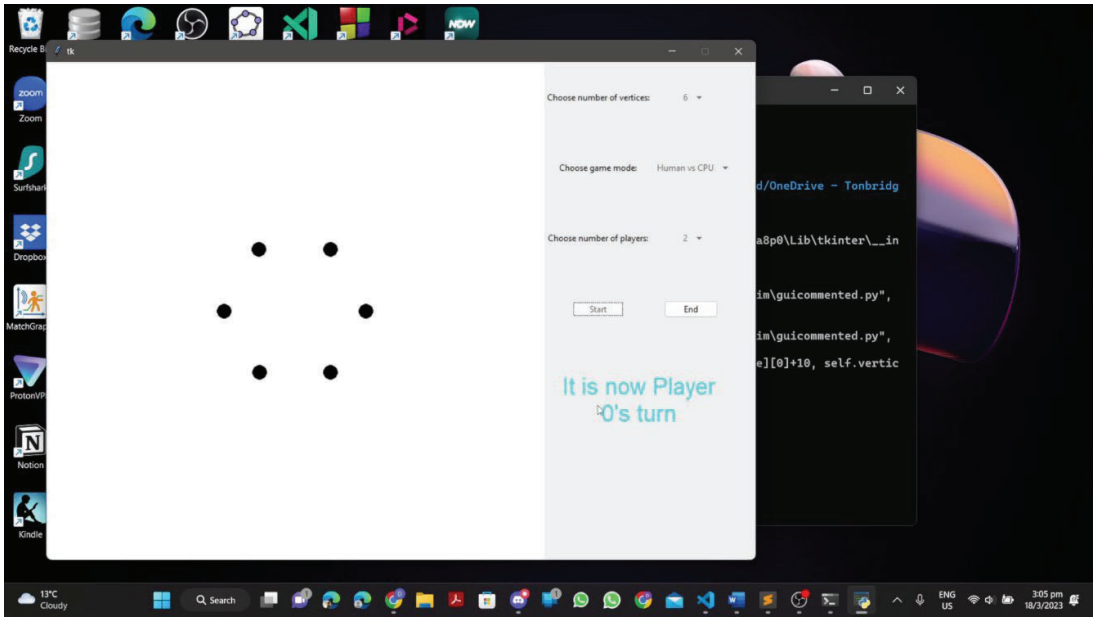


End button pressed and status message ‘Game aborted’ displayed correctly.

Test 9

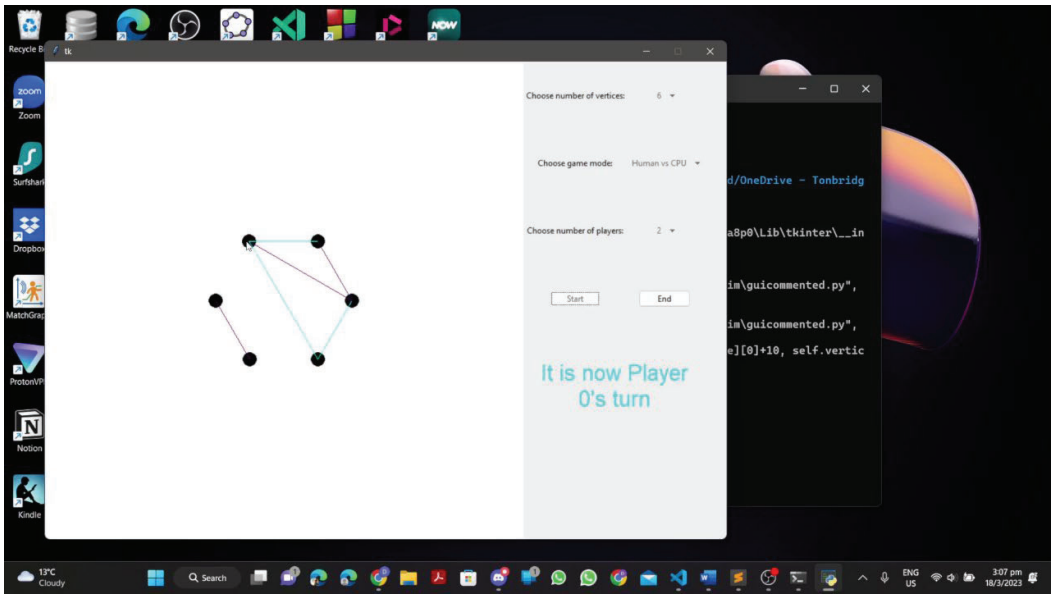
https://youtu.be/bMLJoBCPlek?t=336

Note that Test 10 and Test 11 are both conducted as part of Test 9.

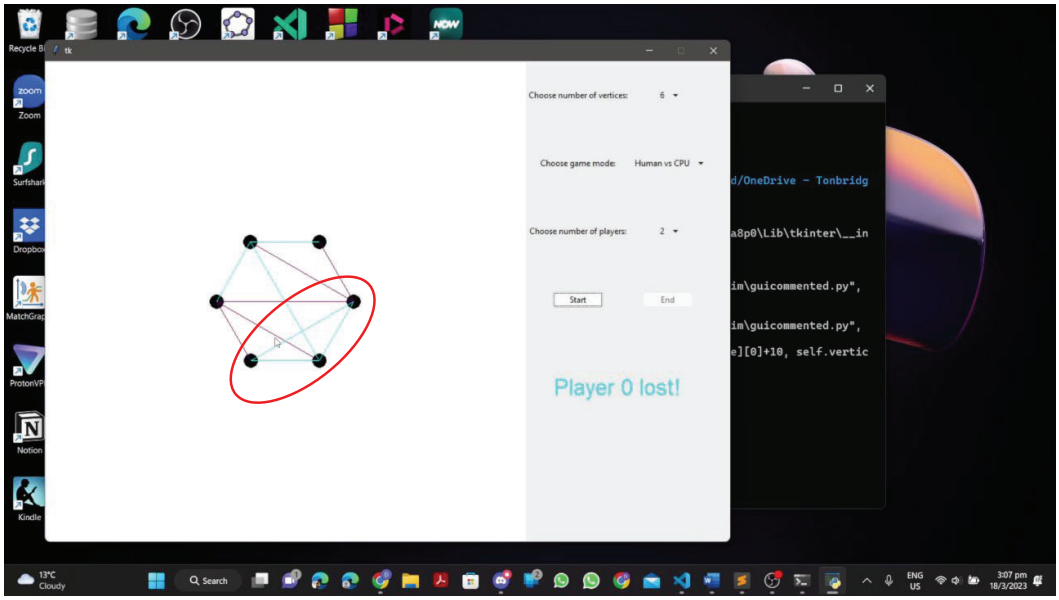


Game settings initialised and it is human player’s turn. Game options are disabled and end button is enabled as expected.

Test 9 (continued)



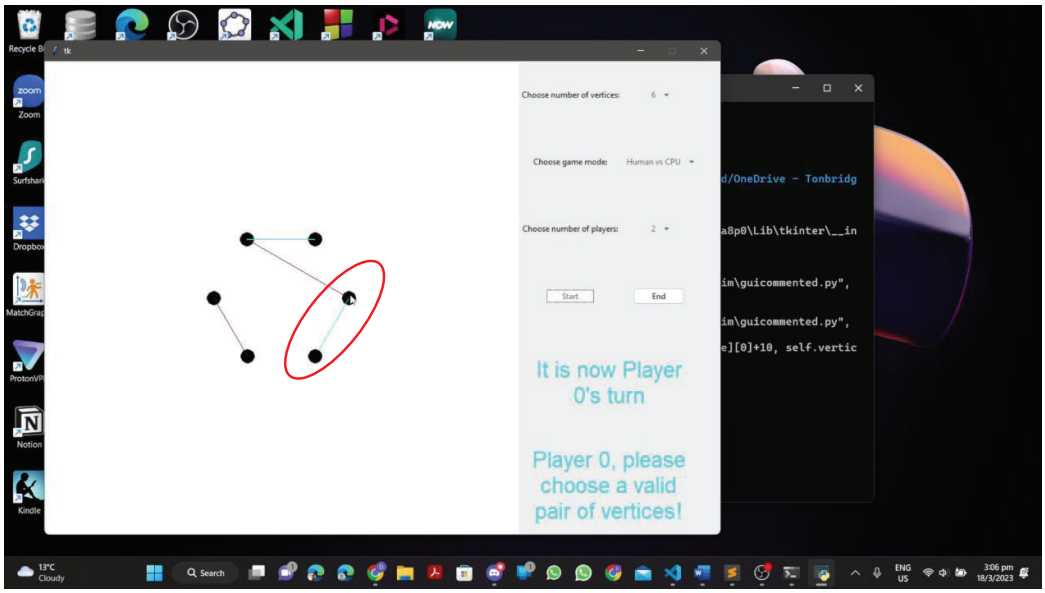
Mid-way during the game; the purple edges belong to the computer player (player 1) and the blue edges belong to the human player (player 0).



Game ends as there is a blue monochromatic triangle formed as circled. The human player has lost as expected as the first player (second player has winning strategy).

Test 10

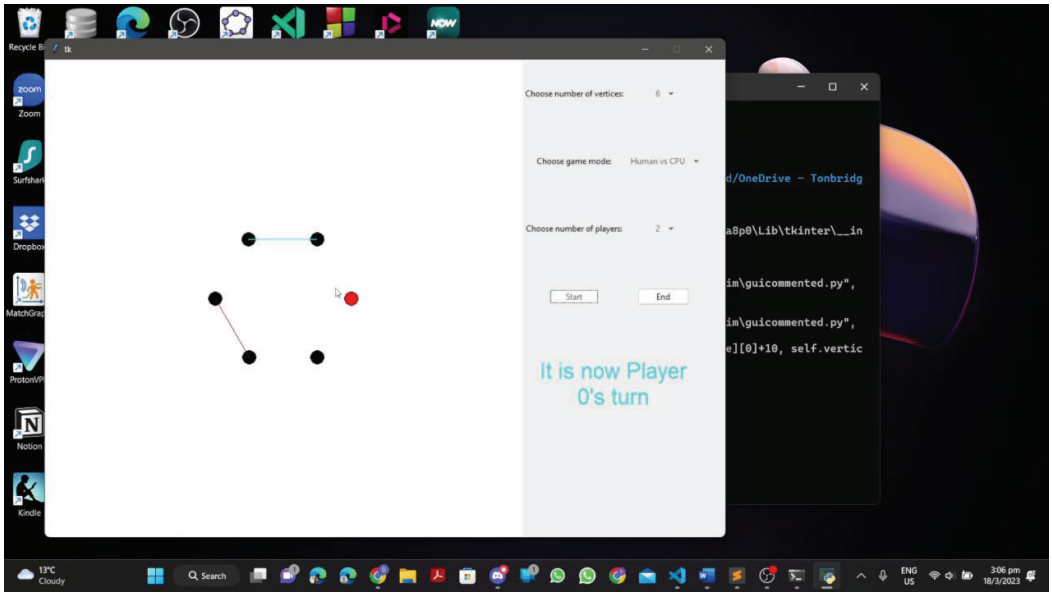
https://youtu.be/bMLJoBCPlek?t=410, https://youtu.be/bMLJoBCPlek?t=514



Player chose vertices that are circled in red. As they have already been connected, the player cannot create an edge between the two vertices and hence a status message is shown, telling the player to choose a valid pair of vertices. The previously chosen vertex is also no longer highlighted, indicating that the previous inputs have been disregarded.

Test 11

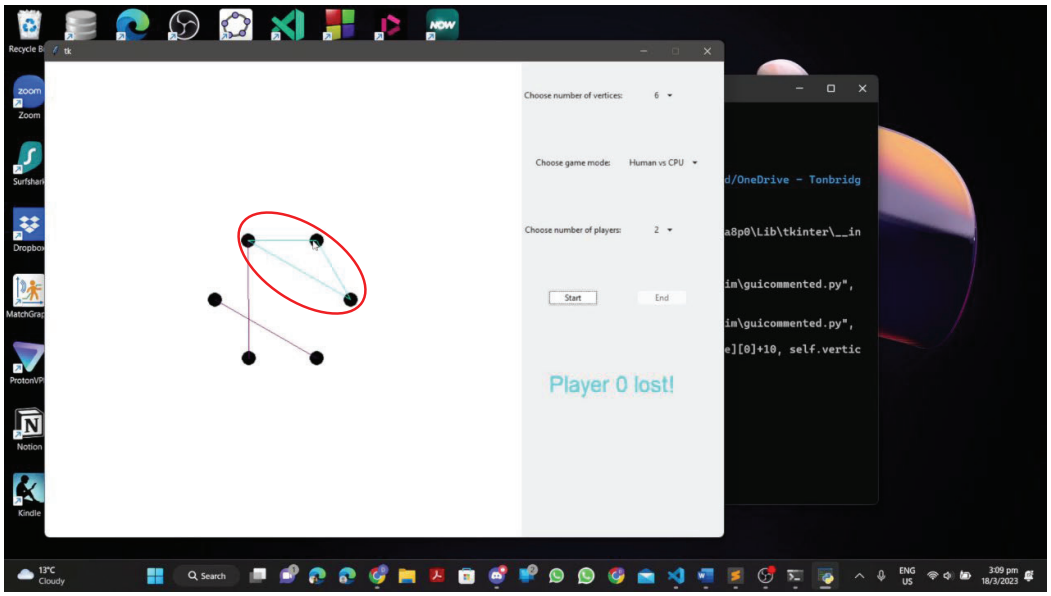
https://youtu.be/bMLJoBCPlek?t=402, https://youtu.be/bMLJoBCPlek?t=545



Player chose vertex which is now being highlighted in red.

Test 12

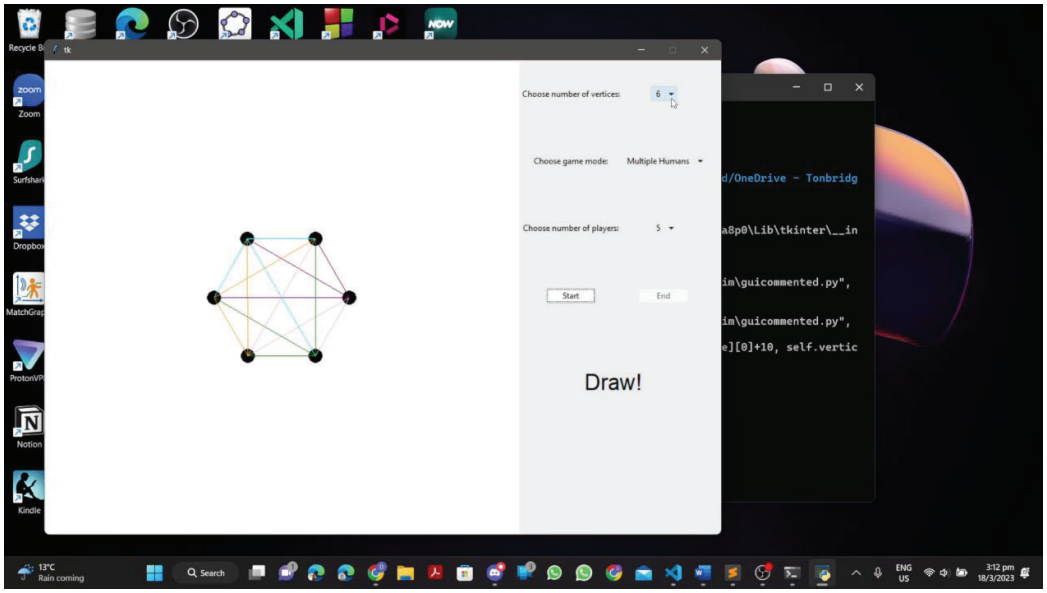
https://youtu.be/bMLJoBCPlek?t=575



Monochromatic triangle formed as shown in circle. Status message displays correctly in the losing player's colour.

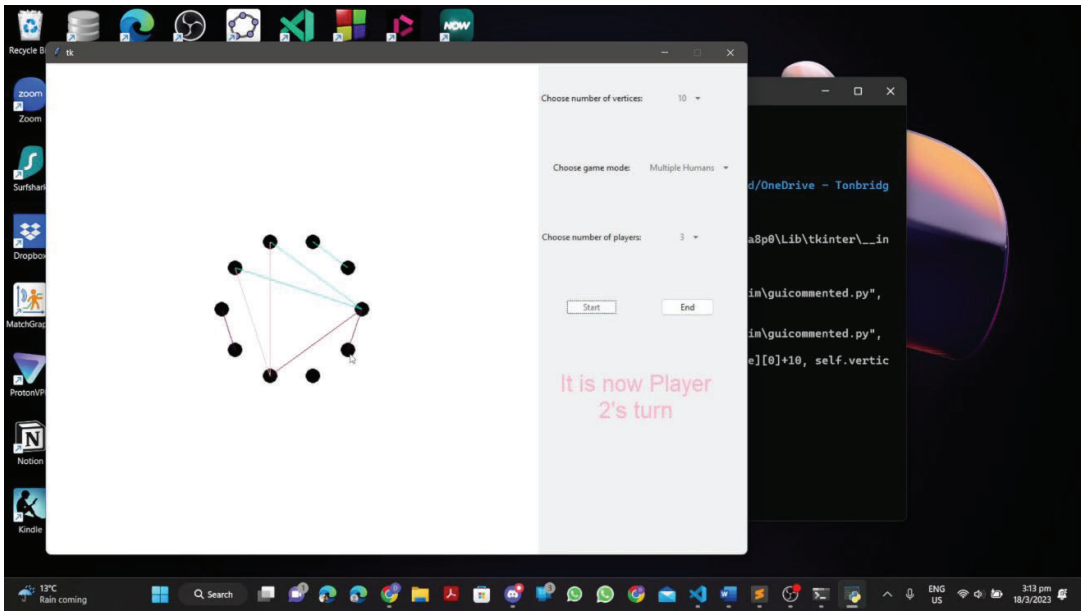
Test 13

https://youtu.be/bMLJoBCPlek?t=736

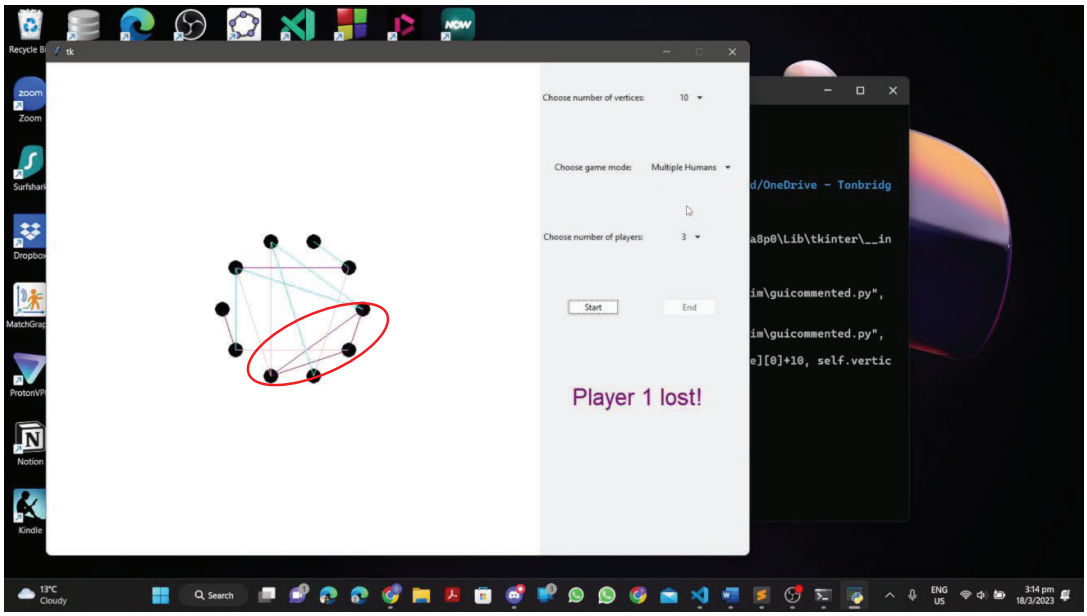


All edges in the graph are connected; however no monochromatic triangles are formed. Hence no player has lost in the game and the game ends in a draw.

Test 14
https://youtu.be/bMLJoBCPlek?t=791

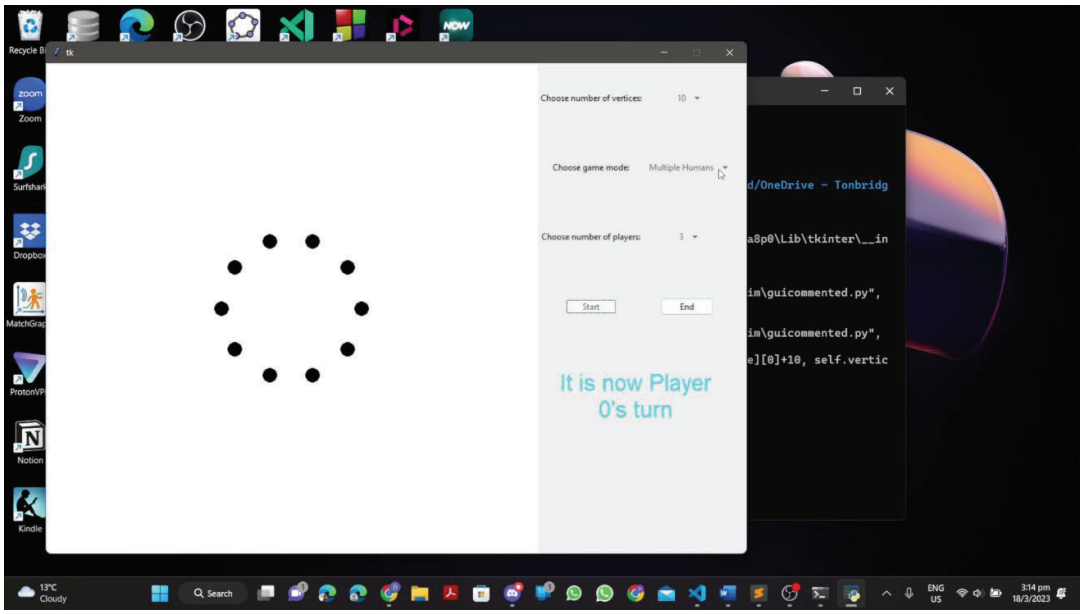


Mid-way during the game; game options disabled as expected.



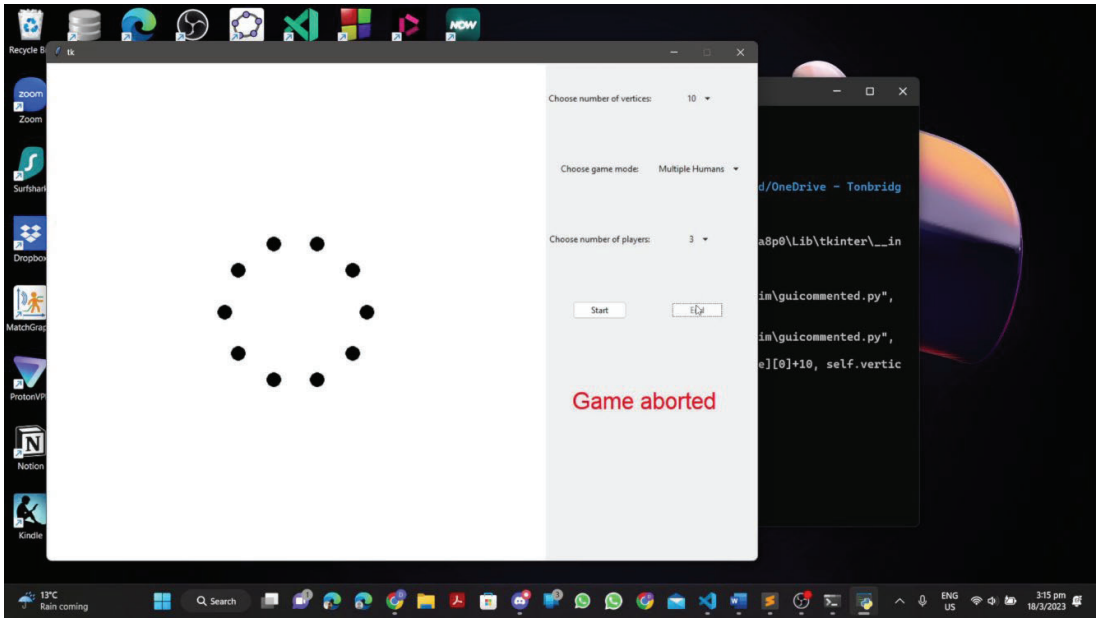
A purple monochromatic triangle is formed as circled. Hence Player 1 lost and the losing message is displayed correctly.

Test 15/16
https://youtu.be/bMLJoBCPlek?t=870



Start button and game options are all disabled (as seen, it is dimmer than the end button). When clicked, the software prompts no response as expected.

Test 17
https://youtu.be/bMLJoBCPlek?t=919



End button is disabled (as seen, it is dimmer than the start button and game options). When clicked, the software prompts no response as expected.

Evaluation

Overall, the proposed problem of implementing Sim with minimax and different variations has been successfully solved. The minimax computer player plays well with the evaluation function, and although it is not implemented as a complete search, it nearly plays perfectly all the time.

Evaluation against each objective

As previously discussed in Technical Solution/List of objectives, all the objectives have been fulfilled through the implementation of different procedures and functions, and the Testing section verifies the effectiveness of the implementation. An evaluation against each of the objectives are found below:

Objective number	Objective	How was it met?	How successfully?
1	The user should be able to choose the number of players in the game.	A dropdown menu for the number of players is created using Tkinter in the GUI. The dropdown menu is enabled when there is no game in progress, hence allowing the user to choose the number of players before starting a game.	Fully complete.
1a	Number of players is from 2 to 5.	In the dropdown menu as described above, there are 4 options for number of players, namely 2, 3, 4 and 5 so the user can choose the number of players as required.	Fully complete.
2	The user should be able to choose if the user wishes to play against the computer in the case of a 2-player game.	A dropdown menu for the game mode is created using Tkinter in the GUI. The dropdown menu is enabled when there is no game in progress, hence allowing the user to choose if they wish to play against a computer player before starting a game. Moreover, if a player chooses to play against a computer but chooses more than 2 players for the game, the software automatically sets the number of players to 2 instead.	Fully complete.

2a	The user should also be able to indicate if the user would like themselves or the computer to play first.	In the dropdown menu described above, there are options for indicating if the computer plays first, namely 'Human vs CPU' and 'CPU vs Human' which indicates if human or computer player plays first respectively.	Fully complete.
3	The user should be able to choose the number of vertices for the playing board.	A dropdown menu for the number of vertices in the game graph is created using Tkinter in the GUI. The dropdown menu is enabled when there is no game in progress, hence allowing the user to choose the number of vertices before starting a game.	Fully complete.
4	The software should assign different colours to represent edges created by different players.	*self.colour* assigns the colours aqua, purple, pink, green and orange to the players 0-4 by default and it can be changed in the code. The edges for the corresponding players are coloured in the assigned colour, and the status messages are in that colour as well.	Fully complete. N.B. improvements listed in later section.
4a	Black and red should be reserved as black means general game announcements and red means warnings.	*self.colour* does not include black and red. By default, the status messages are black (general game announcements) but it changes to red if there are warnings made.	Fully complete.
5	The software should display whose turn it is at the current state of the game.	During each turn, the message 'It's now Player X's turn' is displayed in the status message box in the player's assigned colour.	Fully complete. N.B. improvements listed in later section.

6	The software should indicate if a player has chosen one vertex and is waiting for a second input.	When a vertex is chosen by the player, it is highlighted with the vertex's colour changing from black to red. When the second vertex is chosen, the first vertex is unhighlighted and turns back into black from red.	Fully complete.
7	The software should check if a player has attempted to connect vertices that have already been connected, and in such case, disregard the move and ask them to make a correct move.	The message 'Player X, please choose a valid pair of vertices!' appears when a player attempts to connect two vertices which are already connected. The player's previous clicks on the vertices are disregarded (with no vertices being highlighted at this moment), and the player has to make a legal move before the game could continue.	Fully complete. N.B. improvements listed in later section.
8	The software should check if a monochromatic triangle has been connected at any point of the game, and in such case, end the game and declare the loser.	At the end of a turn, the check is made and when a monochromatic triangle is formed, the software identifies the player who connected the triangle, and displays 'Player X lost!'.	Fully complete.
8a	The graph should be checked for the losing condition every time a move has been made.	Same as 8.	Fully complete.
9	The computer player should be implemented using a minimax algorithm with appropriate depth (4 for 7 or fewer vertices, 3 for 8 or more vertices).	The *maxdepth* function in the minimax algorithm is implemented to return if the maximum search depth has been reached or not, depending on the number of vertices in the graph.	Fully complete. N.B. improvements listed in later section.

9a	The evaluation function should be based on the number of triangles different players can connect, and the number of unconnected triangles with two edges belonging to different players.	The *evaluate* function in the minimax algorithm is implemented as according to the description. The number of triangles different players can connect is checked using *TestForTriangle*, while the number of unconnected triangles with two edges belonging to different players is checked using *TestForUnconnected*.	Fully complete.
9b	Calls in the minimax algorithm should be made recursively.	The minimax algorithm calls itself as part of the search, which is recursive as required.	Fully complete.

Feedback by end-user

Independent feedback has been obtained from the client (Mr Chiu) as below:

Q: Do you think the piece of software has fulfilled all the requirements that you have laid out? Why or why not?

A: Yes, it's completely functionable! I have tried the game and the user interface is clear and easy to understand; and the computer player plays very well! I have never managed to beat the computer when I play first, and the computer sometimes beat me when I play second, which is expected by the game theory as the second player has a winning strategy.

Q: Is there any further improvements you want to see?

A: The minimax runs pretty slowly when the graph has a lot of vertices, particularly at 8, 9 and 10 vertices. It would be better if you can get the computer to make a move quicker. Also there is no indication of which move that the computer has made, so that might be something that could be added.

Q: Would you want me to introduce new functionality if I had the time to code the software again?

A: Multiplayer networking would be something that I want to see so I can play other players over the internet. Maybe changing player colours as well as currently the colours assigned to each player is fixed. It is also desirable for me to choose the difficulty of the computer player as it is really unbeatable! While I don't really mind that, it would be good to see the computer sometimes lose the game.

Improvements

Further improvement, especially on the minimax algorithm, could be done by alpha-beta pruning. With alpha-beta pruning, there could be a smaller number of nodes being searched in the minimax algorithm, and the computer player could make a move faster without the need of searching through the entire minimax graph. Alternatively, the minimax algorithm could be run in another thread, which allows interaction between the human player and the software when the computer is searching and making a move.

Another improvement regarding the computer player would be allowing the user to choose the minimax depth so as to adjust the difficulty of the computer player.

Another area of improvement is the design of the Graphical User Interface. Currently, the software is functionable, but its GUI design is simplistic and minimal. It could be improved with the layout being changed, and possibly introducing more functionality, including customisation of colours or board settings/layout.

References

[1] “Sim (Pencil Game).” Wikipedia, Wikimedia Foundation, 27 Nov. 2022, https://en.wikipedia.org/wiki/Sim_(pencil_game). Accessed 28 Jan. 2023.

[2] Slany, Wolfgang. “Graph Ramsey Games”, 10 Nov. 1999, https://doi.org/10.48550/arxiv.cs/9911004. Accessed 28 Jan. 2023.

[3] Mead, Ernest, et al. “The Game of Sim: A Winning Strategy for the Second Player.” Mathematics Magazine, vol. 47, no. 5, 1974, pp. 243–47. JSTOR, https://doi.org/10.2307/2688046. Accessed 28 Jan. 2023.

Appendix A: cli.py

The command line interface used for testing is as follows:

```
1 from graph import Graph
2 from game import Game
3 import minimax
4
5 computerplayer = 0
6 # 0 for computer playing first, 1 for computer playing second
7
8 def main():
9     print('How many players and how many vertices?')
10    playerno, vertex = map(int, input().split())
11    g = Game(playerno, vertex)
12    game = Graph(g)
13    while game.TestForTriangle() == -1:
14          state = True
15          possiblemoves = g.GetPossibleMoves
16          noofedges = g.GetNoOfEdges
17          print(possiblemoves)
18          if len(possiblemoves) == 0:
19                print('Draw')
20                state = False
21                break
22          currentturn = g.GetCurrentTurn
23          print(game.representation)
24          print(f"It is now player {currentturn}'s turn.")
25          if currentturn != computerplayer:
26                print(f"Which two vertices would you like to connect?")
27                nodeone, nodetwo = map(int, input().split())
28                while (min(nodeone, nodetwo), max(nodeone, nodetwo)) not in
possiblemoves:
29                      print(f"Error: vertices already connected")
30                      print(f"Which two vertices would you like to connect?")
31                      nodeone, nodetwo = map(int, input().split())
32          else:
33                nodes = minimax.minimax(currentturn, game.representation,
possiblemoves, noofedges, 4, True)
34                print(nodes)
35                nodeone = nodes[0]
36                nodetwo = nodes[1]
37          if nodeone > nodetwo:
38                temp = nodetwo
39                nodetwo = nodeone
40                nodeone = temp
41          game.UpdateEdge(currentturn, nodeone, nodetwo)
42          g.UpdatePossibleMoves(nodeone, nodetwo)
43          g.UpdateCurrentTurn(currentturn)
44    if state:
45          print(f"Player {game.TestForTriangle()} lost!")
46
47 if __name__ == '__main__':
48    main()
```