Connect the dots

Analysis

Contents

Problem Definition

I will be creating a solvable puzzle game where the player has to connect all corresponding colour dots while filling out the entire given grid without crossing opposing lines or leaving any squares empty.

This can be achieved by creating a Python program that uses Pygame to create a display so objects can be drawn onto it allowing the user to solve the game. I will use objects and classes to create the dots, the lines connecting the dot and the grid.

My stakeholders for this project will be my Year 13 classmates from all classes- Stakeholder 1- Finley Whetstone, Stakeholder 2- Ethan Venton, Stakeholder 3- James Hatswell and Stakeholder 4- Evie-May Apted. They are experienced with puzzle games and understand the concept of the game so they can offer feedback on features of the game and any potential additional features that can be added in the future. They will also be able to test the game to see if the puzzle is solvable and can give me an opportunity to increase the difficulty if proven to be simple to solve.

Justification of Computational Approach

Clear initial situation

The puzzle will start with an evenly sized grid split by grid lines to create numerous mini squares inside of the grid. The game will then draw on a pre-defined set number of dots, 2 per colour, to a randomly assigned square on the grid.

Clear goal

I have a thought-out plan to create the game with some suggestions from the stakeholders for what they would like to see as well, and more specific/ additional features will be discussed with stakeholders as the project progresses. Some of the features for the game are:

· A 7 block by 7 block grid with several pairs of dots on it is generated

· The user can click on a dot and drag it to generate that same colour line as the dot clicked,

· The user can then connect the line to the same colour dot,

· If lines cross one another, they break causing them to disappear

· The program randomly generates the dots in solvable positions on the grid

Clear inputs and outputs

Inputs for the game would be the user using a mouse to click and hold dots to navigate them to the other dot. There will be a menu where the user will be able to press a button to start the game and with some additional feature that could be added as the project progresses. Outputs will include the program generating dots, the user moving lines and connecting to other dots, and once completed a win screen will be shown, and a new level will be generated.

Clearly defined logic

The program will generate a solved puzzle, with the lines already being connected to the dots, and then the algorithm will remove the connected lines so that just the dots remain in a determined solvable position all before the puzzle is shown to the user. The user will then need to be able to interact with the grid/ dots and navigate them to solve the puzzle, this can be done by using a mouse tracking algorithm and when the user presses down on the mouse button the program will recognise this and show the appropriate output.

Initial Research

Researching potential algorithms

I began by researching possible ways to generate dots onto a grid. One way I found was that you could use grid coordinates to place the dots on whatever square you choose or have it randomly assigned to a square. Another way you could do this is by creating an array and creating a set grid size using zeros to represent any empty cells and using constants to determine where the dots will be. During my research I found a potential worded solution on stack overflow:

Consider solving your problem with a pair of simpler, more manageable algorithms: one algorithm that reliably creates simple, pre-solved boards and another that rearranges flows to make simple boards more complex.

The first part, building a simple pre-solved board, is trivial (if you want it to be) if you're using `n` flows on an `n` x `n` grid:

- For each flow...
  - Place the head dot at the top of the first open column.
  - Place the tail dot at the bottom of that column.

Alternatively, you could provide your own hand-made starter boards to pass to the second part. The only goal of this stage is to get a valid board built, even if it's just trivial or predetermined, so it's worth keeping it simple.

The second part, rearranging the flows, involves looping over each flow, seeing which one can work with its neighboring flow to grow and shrink:

- For some number of iterations...
  - Choose a random flow `f`.
  - If `f` is at the minimum length (say 3 squares long), skip to the next iteration because we can't shrink `f` right now.
  - If the head dot of `f` is next to a dot from another flow `g` (if more than one `g` to choose from, pick one at random)...
    - Move `f`'s head dot one square along its flow (i.e., walk it one square towards the tail). `f` is now one square shorter and there's an empty square. (The puzzle is now unsolved.)
    - Move the neighboring dot from `g` into the empty square vacated by `f`. Now there's an empty square where `g`'s dot moved from.
    - Fill in that empty spot with flow from `g`. Now `g` is one square longer than it was at the beginning of this iteration. (The puzzle is back to being solved as well.)
  - Repeat the previous step for `f`'s tail dot.

The approach as it stands is limited (dots will always be neighbors) but it's easy to expand upon:

- Add a step to loop through the body of flow `f`, looking for trickier ways to swap space with other flows...
- Add a step that prevents a dot from moving to an old location...
- Add any other ideas that you come up with.

The overall solution here is probably less than the ideal one that you're aiming for, but now you have two simple algorithms that you can flesh out further to serve the role of one large, all-encompassing algorithm. In the end, I think this approach is manageable, not cryptic, and easy to tweek, and, if nothing else, a good place to start.

Link to page here.

```
for row in range (1,ROWS-1):
  for column in range (1,COLS-1):
    color = grid[row][column]
    if color>0:
      if color in startNodes:
        endNodes[color] = [row,column]
        distances[color] = abs(row-startNodes[color][0]) + abs(column-startNodes[color][1])
      else:
        startNodes[color] = [row,column]
      allNodes.append([row,column,color])
```

This function found on 101 computing.net checks the pre-defined/ already set grid to see where all the start and end nodes are so it can identify where each flow needs to connect to.

```
def solvePuzzle(grid):

    drawGrid(30) #30 is the width of each square on the grid
    myPen.getscreen().update()
    time.sleep(1-SPEED)
    #Applying heuristics to see ff the grid can be discarded at this stage, this will save a lot of steps
    if checkGrid(grid)==False:
      return False

    #Is the grid fully solved?
    if solved(grid):
      return True

    #Let's investigate the next move!
    for color in startNodes:
      startNode = startNodes[color]
      endNode = endNodes[color]

      #Check if the dots from this color are already connected
      if (abs(endNode[0],startNode[0]) + abs(endNode[1],startNode[1])>1):
        #If not let's find out in which direction to progress...
        directions = []
        if grid[startNode[0]][startNode[1]+1]==0:
          directions.append("right") #Lower Priority!
        if grid[startNode[0]][startNode[1]-1]==0:
          directions.append("left") #Lower Priority!
        if grid[startNode[0]+1][startNode[1]]==0:
          directions.append("down") #Lower Priority!
        if grid[startNode[0]-1][startNode[1]]==0:
          directions.append("up") #Lower Priority!

        if len(directions)==0:
          return False

        #Let's investigate possible moves in different directions...
        for direction in directions:
          if direction=="right":
             startNode[1] += 1
             grid[startNode[0]][startNode[1]]=color
             if solvePuzzle(grid)==True:
               return True
             else:
               #backtrack...
               grid[startNode[0]][startNode[1]]=0
               startNode[1] -= 1

          elif direction=="left":
             startNode[1] -= 1
             grid[startNode[0]][startNode[1]]=color
             if solvePuzzle(grid)==True:
               return True
             else:
```

```
        grid[startNode[0]][startNode[1]]=0
        startNode[1] +=1
    elif direction=="up":
      startNode[0] -= 1
      grid[startNode[0]][startNode[1]]=color
      if solvePuzzle(grid)==True:
        return True
      else:
        grid[startNode[0]][startNode[1]]=0
        startNode[0] +=1
    elif direction=="down":
      startNode[0] +=1
      grid[startNode[0]][startNode[1]]=color
      if solvePuzzle(grid)==True:
        return True
      else:
        grid[startNode[0]][startNode[1]]=0
        startNode[0] -=1
return False
```

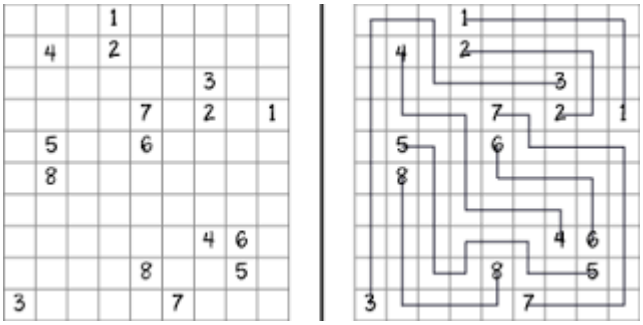
This function checks all possible moves for every node on the grid and states whether it is solvable or not. It does this recursively/ backtracks as it progresses, it starts off by taking the first node and checking if they are already connected, if it is connected, then it moves onto the next node, if not, the algorithm will first try path find to the other node of that colour in every direction, using the shortest path possible, the algorithm first checks right, and if a solution isn't found it then checks left and if no solution is found then etc. And when the algorithm connects the 2 nodes, it moves onto another node, it will do the same thing for this node, however if a solution cannot be found for that node, the algorithm will backtrack and find

a different solution for the first node. It will repeat this process until either the grid is solvable, or there are no other possible moves to make, and will say that the puzzle is not solvable.

Investigating existing systems

Numberlink

The concept of numberlink is similar to the game I plan to make, you have to connect each number to its match via a path that doesn't, cross any other path or double back on itself and you must also use every empty square on the grid.



Feature I like:

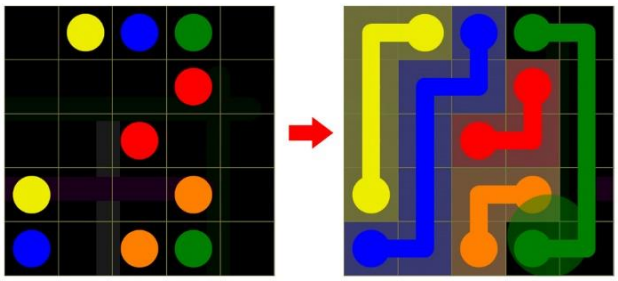
- Simple design
- Can choose different difficulty/ grid size
- Paths are easy to draw

Features I dislike

- Visually it looks boring
- For some puzzles you can connect all the dots without filling every grid square
- Hints provided only tell you a single square that a path will cross instead of giving you the entire path

Flow free

Each puzzle has a grid of squares with pairs of coloured dots occupying some of the squares. The objective is to connect dots of the same colour by drawing paths between them so that the entire grid is occupied by paths. However, paths may not intersect and will be broken if they do.



Features I like:

- Clear colour scheme
- All puzzles have only 1 solution
- Hints give you 1 solved path
- Can choose difficulty/ grid size

Features I dislike

- You can get unlimited hints so the puzzle can be solved for you
- Puzzles aren't generated randomly; they are pre-set by a person beforehand

Stakeholder interview

I will be interviewing stakeholders so that I can identify what features stakeholders would like to see implemented, how the game will look and what needs to be done.

Question	Reason
Have you played a game similar to this type of game before?	To determine whether the stakeholders have some knowledge of the game.
Would you say how the interface looks would be a concern?	To see if I should how much interface should be focused.
Is there a colour scheme or theme you would be interested in?	To get information on what they would like the interface to look like.
Would you like there to be different difficulties?	So more experienced players can have a challenge.
Would you like the game to keep track of how many moves you have made?	To see whether the user would find it useful.
Would you like an option to reset the level?	To see if they find it to be a useful feature.
When completing a level would you rather the game automatically go to the next level or give you the option to exit/ move to the next level?	To see what would be preferred when completing a level.

Do you have any extra ideas you would like to add to the game?	So that they can include any missed feature.

Interview Transcript

Have you played a game similar to this type of game before?

Stakeholder 1: *Yes, I have played connect the dots before as a mobile app*

Stakeholder 2: *Yeah*

Stakeholder 3: *Yes*

Stakeholder 4: *Yes*

Would you say how the interface looks would be a concern?

Stakeholder 1: *Yes, should be clean and simple looking, while effective*

Stakeholder 2: *To a point yes. It doesn't matter too much but if it looks terrible then people won't want to play*

Stakeholder 3: *Not really, as long as it's easy to navigate*

Stakeholder 4: *Yes, it's important it looks appealing*

Is there a colour scheme or theme you would be interested in?

Stakeholder 1: *No, any simple colour scheme should work as long as colours don't clash*

Stakeholder 2: *Not really, I don't think the colour scheme would be very impactful however it may be beneficial to add a colour-blind mode for extra points*

Stakeholder 3: *A Retro Style or modern feel would be best*

Stakeholder 4: *Neon*

Would you like there to be different difficulties?

Stakeholder 1: *Yes, you could increase grid size to increase difficulty*

Stakeholder 2: *Yes maybe 3 levels*

Stakeholder 3: *Yes*

Stakeholder 4: *More squares of higher difficulty like 14X14 instead of funny shapes*

Would you like the game to keep track of how many moves you have made?

Stakeholder 1: *Yes*

Stakeholder 2: *Could be used for a more difficult mode or for a point system to be competitive with friends?*

Stakeholder 3: *Yes, it will help the player improve*

Stakeholder 4: *Yes*

Would you like an option to reset the level?

Stakeholder 1: *Yes, depending on difficulty setting*

Stakeholder 2: *Yes*

Stakeholder 3: *Yes*

Stakeholder 4: *Yes*

When completing a level would you rather the game automatically go to the next level or give you the option to exit/ move to the next level?

Stakeholder 1: *Give options between levels*

Stakeholder 2: *Being given the option to leave to main menu or continue would make it appear more professional*

Stakeholder 3: *Give the option to exit or move onto the next level as the player might not want to carry further on*

Stakeholder 4: *Automatically go to next level*

Do you have any extra ideas you would like to add to the game?

Stakeholder 1: *Make a leader board of the highest scoring individuals*

Stakeholder 2: *Nope*

Stakeholder 3: *Maybe variable levels, so joining up two certain colours will unlock a pathway for another colour to go through*

Stakeholder 4: *No that's all*

Summary of Findings

The game I will be making will be a solvable puzzle type game. The player will be able to control the mouse to draw lines from dots to solve the puzzle. There will be multiple levels which will have dark, neon colour palette as requested by stakeholder 4 as I believe that will make the game look more aesthetically pleasing. The player should aim to complete each level in as little moves as possible as there will be a leader board the player can access in the menu displaying these statistics.

I aim to allow the player to have a choice of what level difficulty they wish to attempt before starting the game, requested by the stakeholders, as it allows for more advanced players to challenge themselves, while beginners can get a feel for how to "tackle" the game. This will be done by increasing the grid size for the levels, for example, "easy" difficulty would be a 5x5 square grid, "medium" difficulty would be a 7x7 square grid, and a "hard" difficulty would be a 10x10 square grid, and then potentially an "impossible" difficulty with 14x14 square grid once I have completed everything else I aim to achieve.

In order to keep the player interested in the game, they will have the option to use a "hint" feature which will give them 1 solved line to assist them if they get stuck, to prevent the player from overusing this feature, the "hint" feature will be limited to 1 use per level and then for the "impossible" and "hard" difficulty, this limit may be increased due to the amount of lines there are to solve. There will also be an option to reset all moves made if the player is stuck and wishes to start over, this will also reset the move tracker, which will be a visible game mechanic which allows the player to see how many moves they have made, requested by the stakeholders.

Project Limitations

Limited input/ output methods

Pygame is very efficient from creating images and shapes to create a successful interface like the one I will create for the game. However, it is more difficult implement other input methods than keyboard and mouse, such as buttons, text boxes etc, and when I want to display a leader board neatly, it will be difficult to implement in pygame, therefore creating a separate window using tkinter allowing users to input any details so they can recognise themselves in the table may be more appropriate.

Using random generation

The program will have to randomly generate a solvable solution that can be solved with a minimum amount of moves as how many dots there are. So, I decide to estimate the total number of combinations of dots for a 5x5 square grid for complete random generation with no bias:

- We have a 5x5 grid, which has 25 cells.
- We need to place 5 pairs of dots, meaning 10 cells will be occupied by dots (each pair consists of 2 dots of the same colour), and the remaining 15 cells will be empty.
- Each pair is distinct (e.g., red pair, blue pair, etc.), so the order in which we assign colours matters.
- The positions of the two dots in each pair are distinguishable only by their location on the grid, not by any inherent order (i.e., the red dot at (1,1) and the red dot at (1,2) is the same as the red dot at (1,2) and the red dot at (1,1)).

To find the total number of possible combinations, we can approach it as follows:

1. Choose Positions for the First Pair:
   - Select 2 distinct cells out of 25 for the first colour
   - The number of ways to do this is $C(25, 2)$, where $C(n, k)$ is the combination formula "n choose k".

2. Choose Positions for the Second Pair:
   - After placing the first pair, we have 23 cells left.
   - Choose 2 distinct cells out of the remaining 23 for the second colour
   - Number of ways: $C(23, 2)$.

3. Continue for All Five Pairs:
   - Similarly, for the third pair: $C(21, 2)$.
   - Fourth pair: $C(19, 2)$.
   - Fifth pair: $C(17, 2)$.

4. Multiply the Choices:
   - The total number of ways to place all five pairs is the product of the combinations for each step:

     Total = $C(25, 2) \times C(23, 2) \times C(21, 2) \times C(19, 2) \times C(17, 2)$.

5. Calculating Each Combination:
   - $C(n, 2) = n! / (2! \times (n-2)!) = n \times (n - 1) / 2$.
   - So:
     - $C(25, 2) = 25 \times 24 / 2 = 300$.
     - $C(23, 2) = 23 \times 22 / 2 = 253$.
     - $C(21, 2) = 21 \times 20 / 2 = 210$.
     - $C(19, 2) = 19 \times 18 / 2 = 171$.
     - $C(17, 2) = 17 \times 16 / 2 = 136$.

6. Multiplying Them Together:
   - Total = $300 \times 253 \times 210 \times 171 \times 136$

- So, the total number of ways is 370,677,384,000.

However, some of these combinations may be equivalent due to rotation or reflection (i.e., symmetric boards might be considered the same) of the puzzle. But the problem asks for "complete random generation" without considering such symmetries, so the value won't need to be adjusted to account for that. Due to this being an approximation, we can't say that this value is the exact number of possible combinations there are for a 5x5 square grid, but we can assume that the actual number of combinations will be around this value.

After calculating, the number of possible combinations for placing 5 pairs of dots on a 5x5 grid is approximately:
370,677,384,000 (370 billion, 677 million, 384 thousand).

Reasons a 5×5 Flow Free Puzzle Is Unsolvable
These are the most common "failure modes" that make a random dot placement unsolvable:

1. Isolated Regions
- If dots or walls (other paths) split the grid into disconnected sections where a pair is trapped, it's unsolvable.
- Example:
  - Red dots at (1,1) and (5,5), but a cluster of other dots blocks all possible paths between them.

2. Forced Crossings or Overlaps
- Even if paths exist, they might require crossing another path or overlapping, which isn't allowed.
- Example:
  - Two pairs (red and blue) are placed such that their only possible paths intersect diagonally.

3. Odd-Parity Problems
- If a pair's dots are separated by an odd number of cells in both rows and columns (e.g., one dot at (1,1) and the other at (2,2)), and the surrounding paths block "checkerboard" movement, it may be unsolvable.

4. Colour Trapping
- A dot is surrounded by other colours with no room to "escape" and form a path.
- Example:
  - A blue dot at (3,3) is bordered by red/green/yellow dots at (2,3), (3,2), (3,4), (4,3), leaving no space to start a path.

5. Unpaired Dots
- Rare in random generation (since we're explicitly placing pairs), but possible if generation allows single dots.

Quantifying Unsolvable Configurations
To estimate how many of the 370 billion configurations fail these rules:

A. Isolated Regions (Partitioning)
- Likelihood: ~20–30% of random configurations.
- Why: As the grid fills with pairs, the chance of accidental partitioning increases.
- Example: If two pairs form a "wall" splitting the grid, a third pair on opposite sides is doomed.

B. Forced Crossings
- Likelihood: ~10–20% of configurations.
- Why: With 5 pairs, path interference grows quadratically.

C. Odd-Parity and Trapping
- Likelihood: ~5–10% of configurations.
- Why: Less common than partitioning but still significant.

D. Combined Effect
- These failures often overlap (e.g., a partition also forces crossings).
- Estimated unsolvable: ~50–70% of configurations.
- Thus, solvable: ~30–50% (aligning with empirical observations).

Refined Estimate
If we assume:
- Lower bound: 30% solvable → 111 billion solvable.
- Upper bound: 50% solvable → 185 billion solvable.
- Realistic midpoint: ~40% → 148 billion solvable.

Computational Verification
For exact numbers, we'd need to:
1. Brute-force test all 370 billion configurations (impossible without large computer processing).
2. Use Monte Carlo sampling:
- Randomly generate 1 million configurations.
- Test solvability using a Flow Free solver algorithm (e.g., backtracking).

- Extrapolate the ratio.
- Example: If 400,000/1,000,000 are solvable, assume 40% overall.

Existing Data
Smaller grids have been studied:
- 3×3 grid: ~60% solvable.
- 4×4 grid: ~45% solvable.
- 5×5 grid: Likely 30–50% (as above).

Final Answer
For a 5×5 grid with 5 pairs:
- Total configurations: 370,677,384,000.
- Estimated solvable: 100–200 billion, so let's take the median which would be 150 billion.

Therefore, using these values we can assume that the chances of a solvable grid be generated would be approximately 40.5%, and while the chance of the grid being solvable is high, however just because the probability of an event is high, doesn't mean that the event is likely to occur, you can also say that the increase in possible outcomes lowers specific event likelihood as when there are many possible outcomes, the probability of one particular event happening becomes smaller, even if the probability for a single attempt is relatively high. This would pose a problem as the program would have to identify whether there is a solvable solution before showing it to the user. As well as the fact that it is unlikely that the puzzle would be solvable assuming complete random generation therefore this way may be inefficient.

Another way to randomly generate levels would be to randomly assign a single dot to a square on the grid, then draw a line in any direction that can also change direction, for a set number of grid squares e.g. between 3-7 squares per coloured dot and line, and then once the algorithm finished drawing the second dot for that colour is placed at the end. This way is good as it guarantees that the grid is solvable as the program will draw in the dots 1 by 1. but this may prove to be difficult as the program would have to draw the paths and dots, check if it is solvable to make sure no unexpected errors occur, remove the connected paths from the dots, and then draw the dots, while making sure that the user doesn't see the solved version of the puzzle beforehand.

Additional features

When I interviewed my stakeholders, I asked whether they would like some additional features to be added to the game, these include: a button to reset any moves made and start over, an option for different level difficulties, and a feature to track how many moves have been made. I think these ideas would be a good addition to the game adding a sense of professionalism therefore making the game more engaging but with time limitations only 1 or 2 of these may be

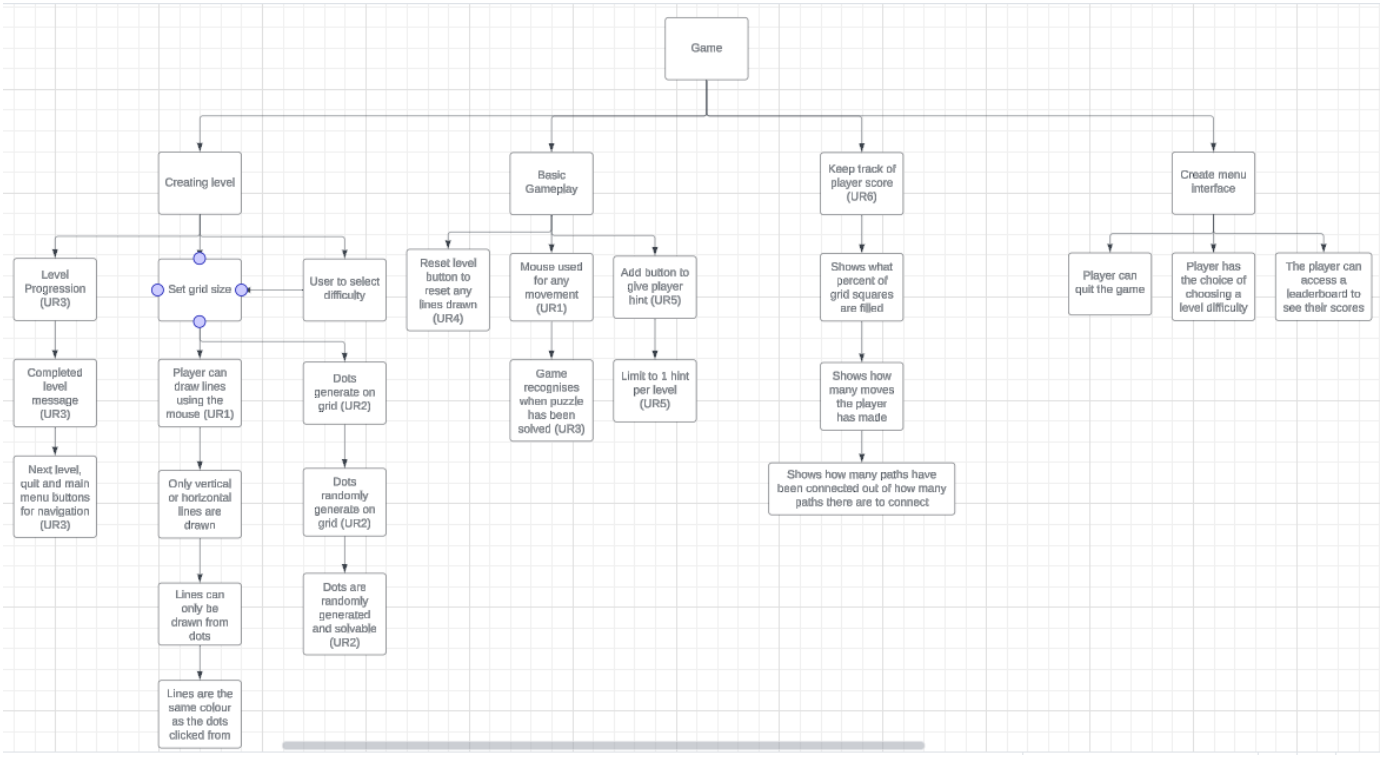
added. However once basic gameplay is made and it runs efficiently, it will be easier to determine which features can be added within the time limit.

User Requirements

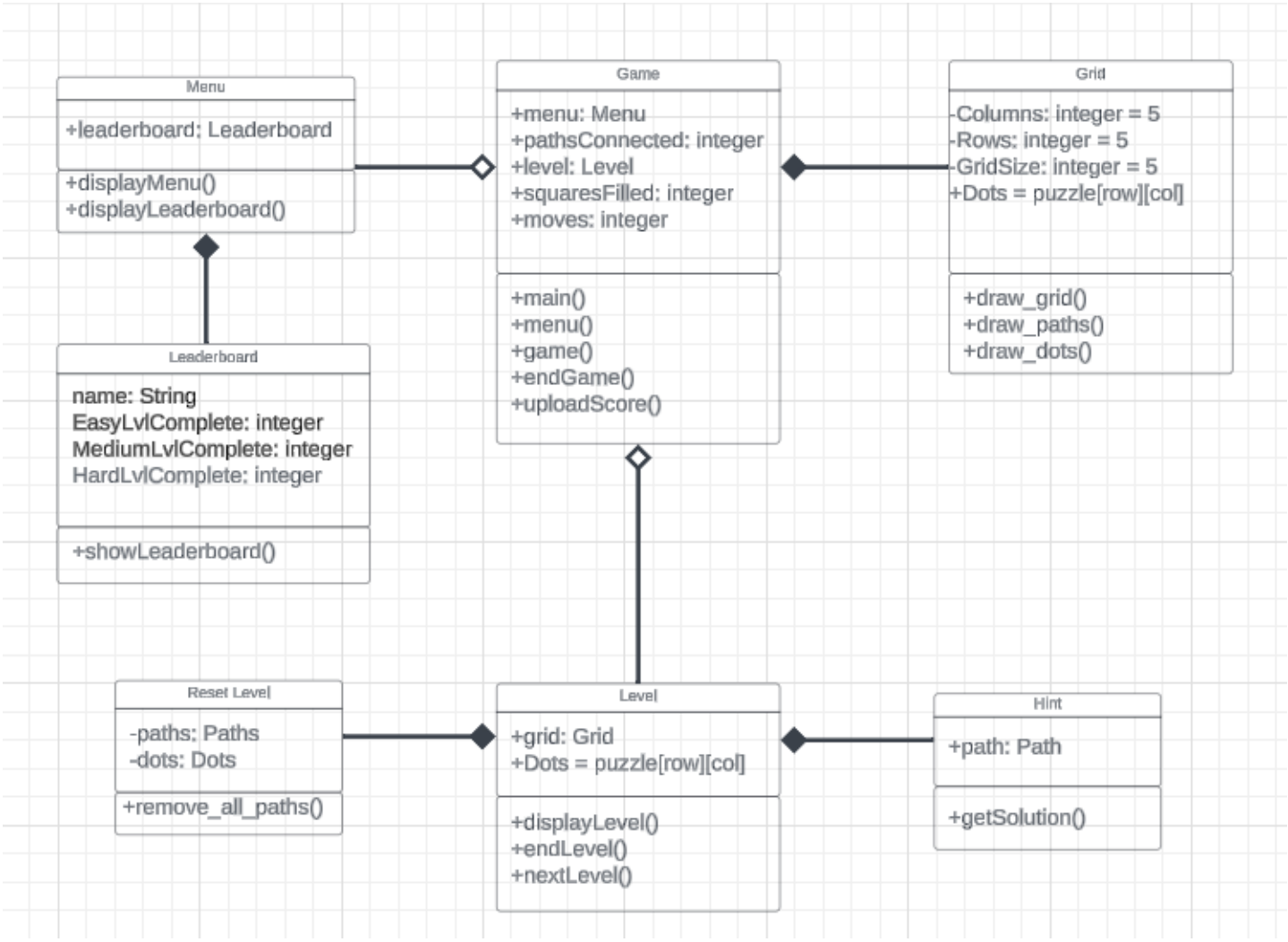
1. The user will need to be able to interact with the game using a mouse tracking algorithm so that when they click using the mouse the program can recognise when this happens, I will do this by using the in-built pygame function pygame.event.get().
2. Each level will have to be randomly generated but will also have to differ from other levels so that puzzles aren't repeated.
3. The program will need to recognise when the user has completed the puzzle, this would be when all corresponding dots have been connected and when every square in the grid is filled out and therefore show an appropriate message.
4. The program will also need to recognise if the user resets the level using the reset button which will remove any moves made.
5. The program will also need to recognise if the user asks for a hint and will therefore have to give the user 1 single completed solution.
6. The program will also need to be able to keep track of how many moves the player has made.
7. There will need to be a menu that the player can navigate before solving the puzzle.

Design

Overall plan for project



Initial class diagram

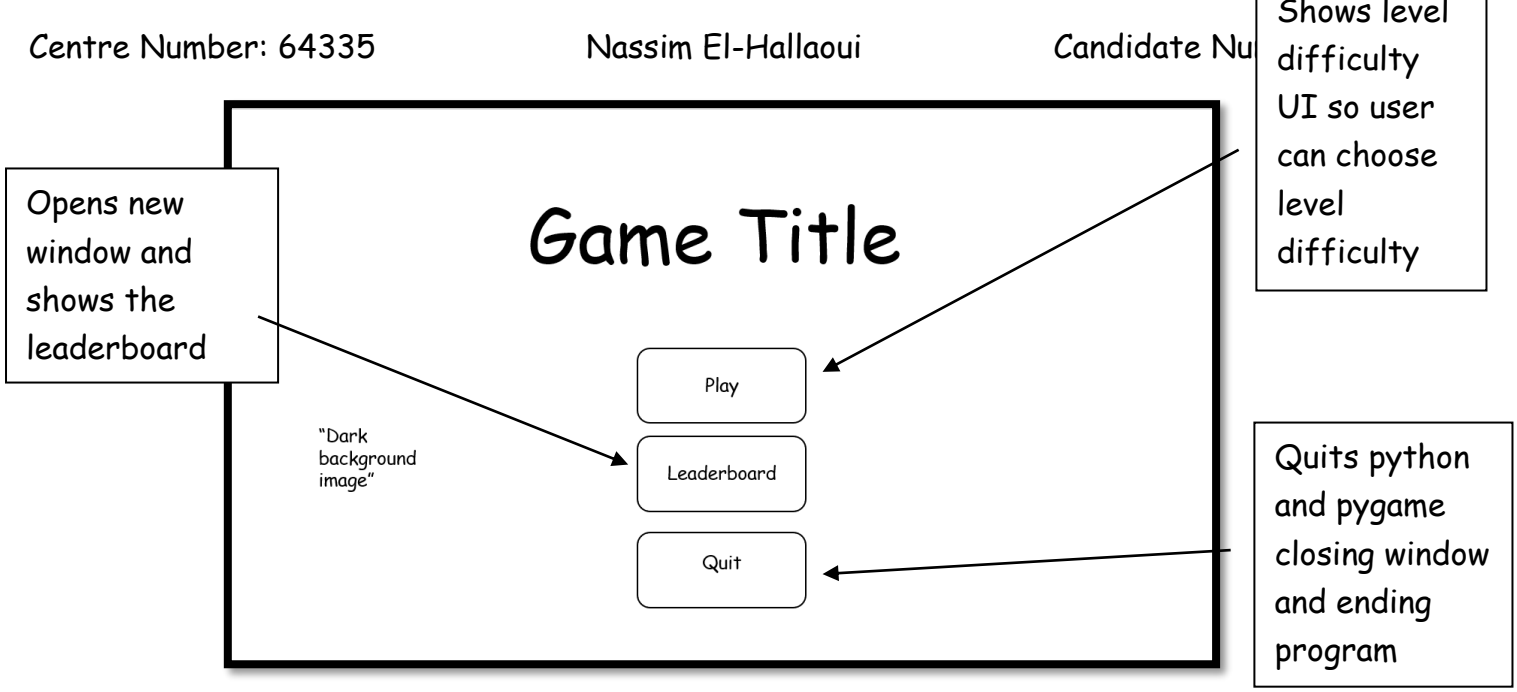


GUI Design

Main Menu

During my stakeholder interview, the stakeholders said that the interface should look simple and easy to navigate, therefore, the main menu will have a dark themed background and will have simple, clearly labelled text overlayed on top for easy navigation. In the centre there will be the name of the game, which I will ask my stakeholders as to what it should be. Below will be several buttons which will have different uses, one to play the game, another to display the game leader board, and another to quit the game therefore I believe this initial design meets the demands of the stakeholders.

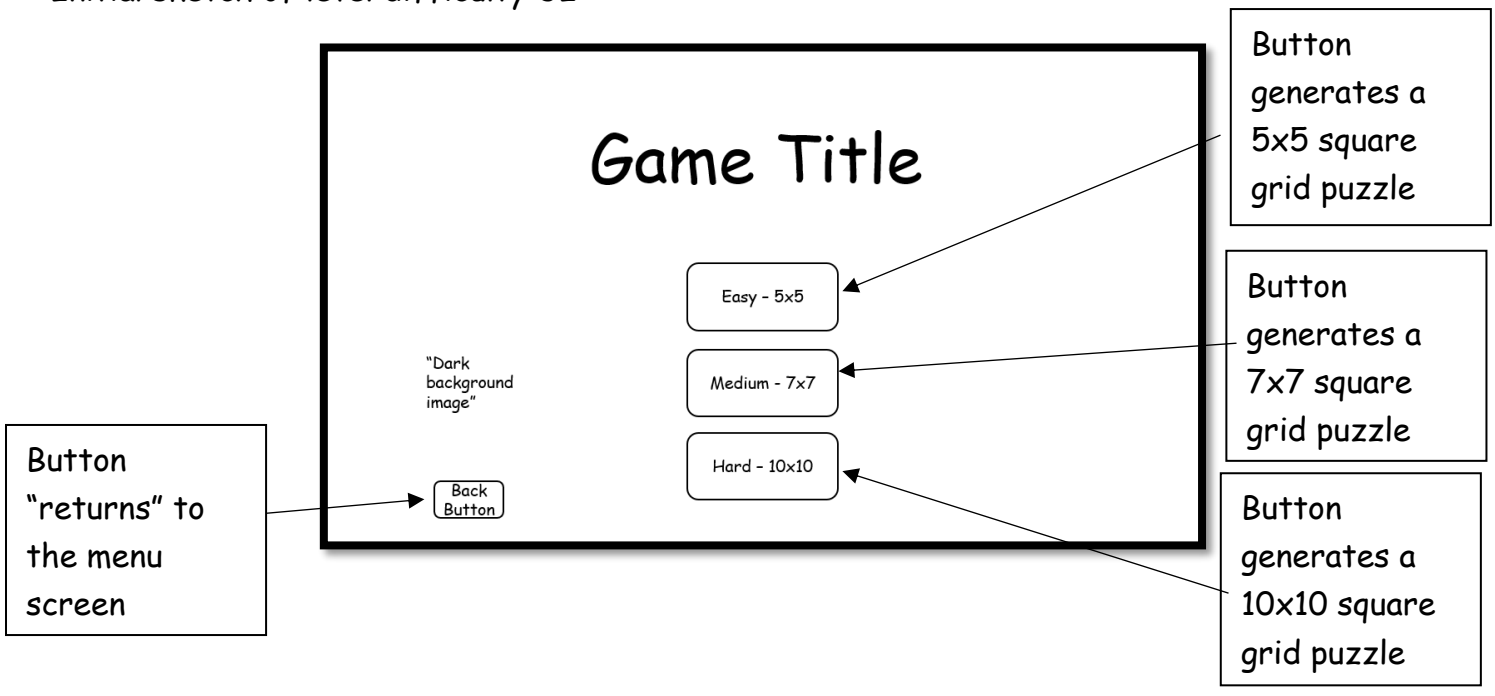
Initial design of main menu:



Level Difficulty

The level difficulty UI will comprise of 4 buttons allowing the user to choose what level difficulty they wish to attempt, easy which will generate a 5x5 square grid, medium which will generate a 7x7 square grid, and hard which will generate a 10x10 square grid, and finally a back button returning the user to the original menu.

Initial sketch of level difficulty UI:

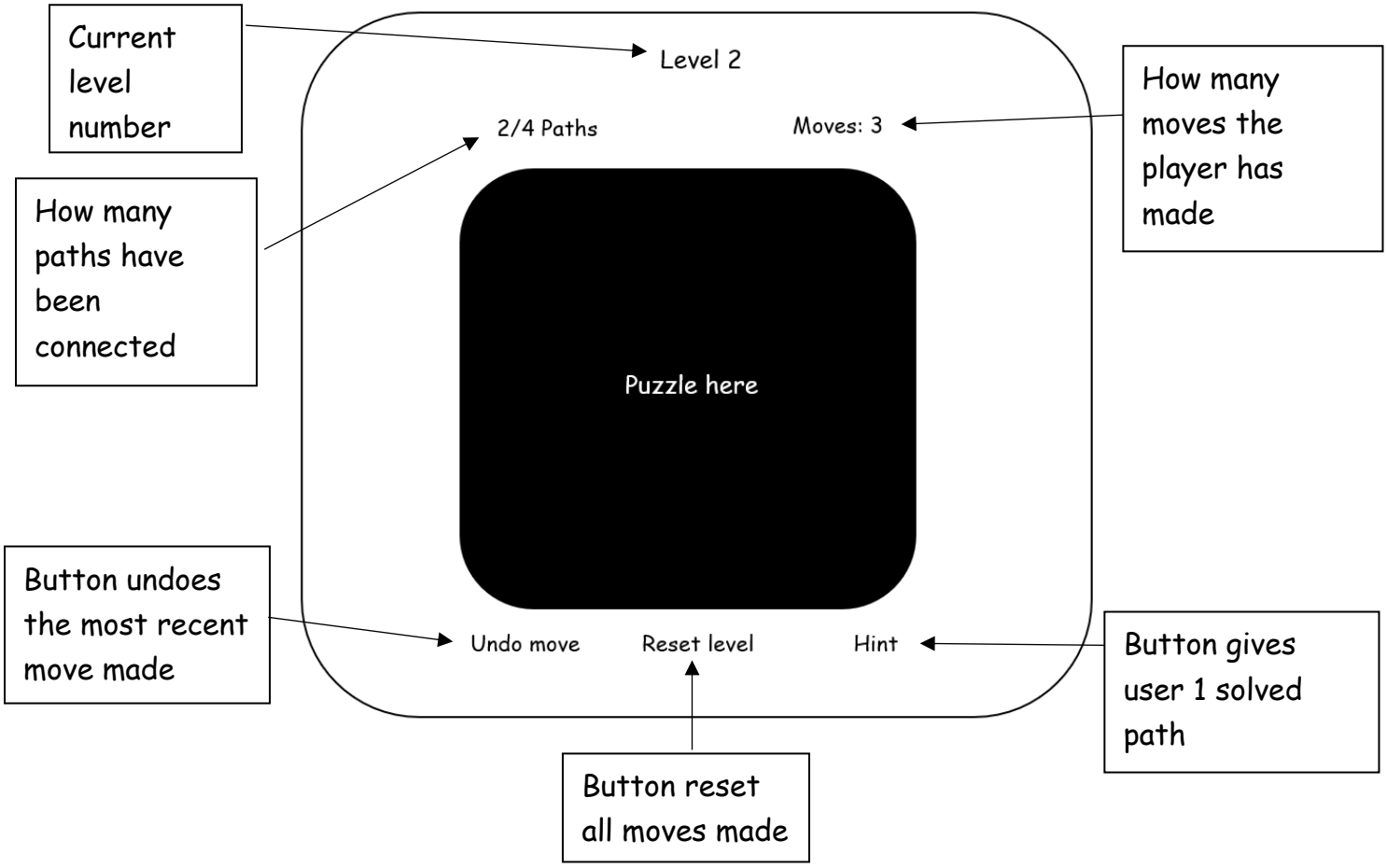


Level

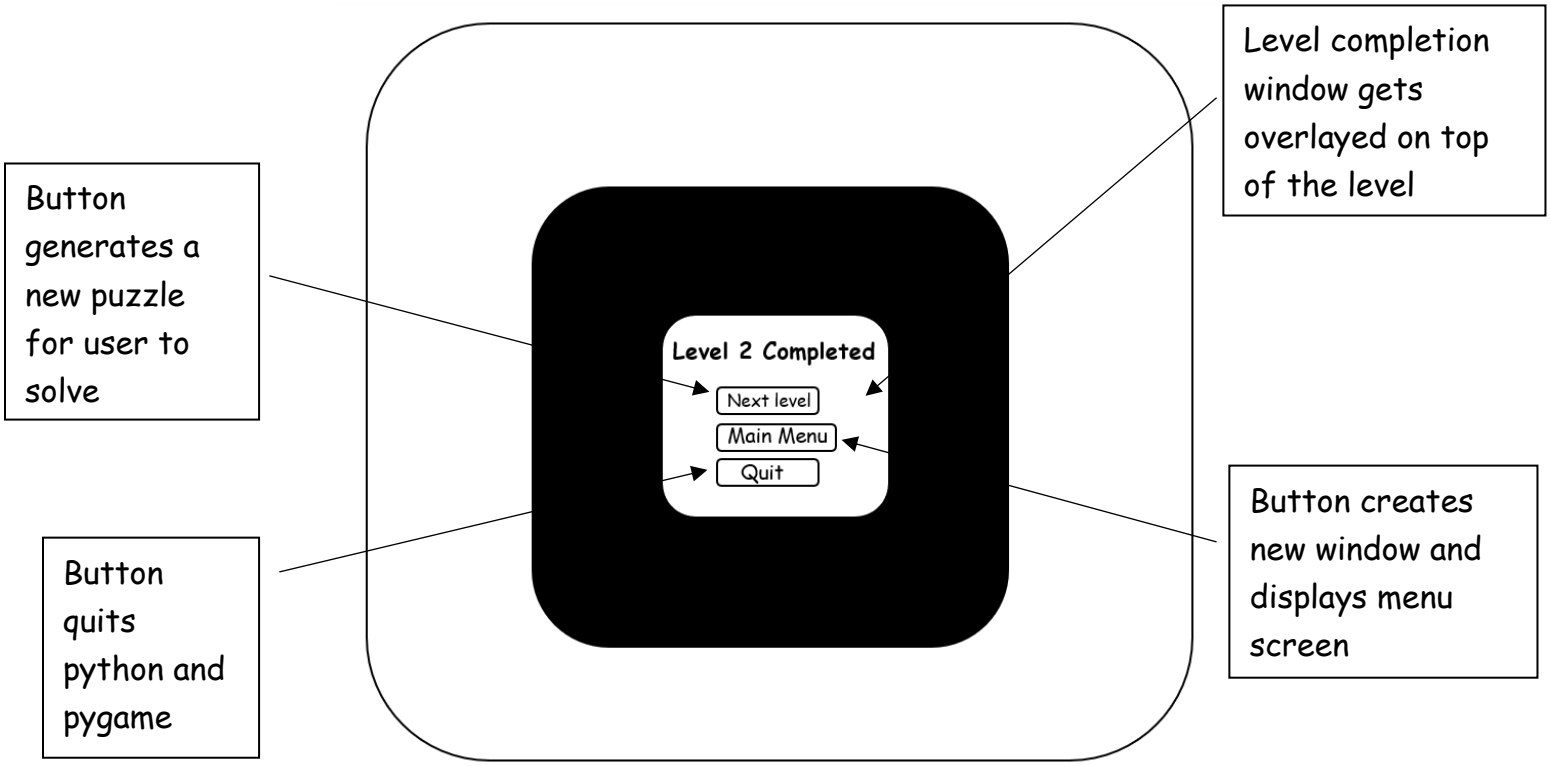
The level UI design will be simple and appealing as requested by my stakeholders, therefore I will follow a dark, modern colour scheme to not overcomplicate the visual aspect of the game. In the centre there will be a grid with a set size, determined by the user, with lines separating the squares. At the top in the centre of the window the program will display what level the user is on. And then at the top of the screen will be the 'statistics' about the level e.g. how many paths have been connected and how many moves the user has made. Then at the bottom of the

window will be different game mechanics such as an undo button, which will undo the move most recently made, a reset level button, which will remove all paths resetting the level, and a hint button, which will give the user one correct path per level.

Initial sketch of level UI:



Level End Screen



Inputs & Outputs

Main Menu

Input	Process	Output
Program is opened	newGame = Game() newGame.main()	Main menu UI is displayed
User presses "Play"	levelSelection()	Level difficulty UI is displayed
User presses "Leaderboard"	showLeaderboard()	Opens a new window with leaderboard
User presses "Quit"	quit()	Program closes

Level Difficulty

Input	Process	Output
User presses "Easy – 5x5"	easyLevel()	Program will generate a 5x5 square grid puzzle
User presses "Medium – 7x7"	mediumLevel()	Program will generate a 7x7 square grid puzzle
User presses "Hard – 10x10"	hardLevel()	Program will generate a 10x10 square grid puzzle
User presses "Back"	displayMenu()	Program returns to the main menu screen

Level

Input	Process	Output
A Dot is pressed	n/a	Nothing
The user drags while pressing a dot	drawPath()	A line of that colour dot is drawn
User presses the "hint" button	giveHint()	Program solves 1 single path
User presses the "reset level" button	resetLevel()	All paths are removed
User presses "undo" button	undoMove()	The move that was most recently made is undone

Level End Screen

Input	Process	Output
User presses "Next Level"	nextLevel()	A new level is generated
User presses "Level Selection"	levelSelection()	User is shown the Level Selection screen
User presses "Quit"	Quit()	Program closes

Program Decomposition

Sprint 1:

Create menu UI

When the game is opened, the game object will be created and then initialised. It will handle creating the game window and setting the size of the puzzle grid as well as other display surfaces. One of the first functions to be called will be the "game.main()" which will initialise the main game variables and the game window. It will then create the menu object and call the function "menu.displayMenu()", this will allow the user to navigate the game as show in the GUI designs above and then when the corresponding button is clicked the grid will be generated through the "level.displayLevel()" function then allowing the user to play the game.

Create level UI

When the play button is pressed, the game object will need to start level 1, to do this a new object called "level" will be created and a method called "level.displayLevel()" will create the level. The function will do this by drawing a grid of a set size chosen by the user and then will draw on circular dots in randomised locations on the grid so the user can solve the puzzle. The function will also need to draw on the game statistics as shown in the initial Level design above. The function will also need to recognise when there are changes in these values and will respond accordingly to keep the information correct.

Sprint 2:

Create algorithm to generate grid

When the level UI has been created, I can begin generating the grid. This will take place within the game surface "newGame.display" as this is where the grid will be displayed throughout the level and where all game logic will take place. The function will do this by drawing several lines depending on the level difficulty, if easy is pressed, the function should draw 5 rows and columns, if medium pressed, the function should draw 7 rows and columns etc.

Generate dots onto the grid in pre-determined places on the grid

Within the same function as the one above, the dots should be drawn in pre-determined positions. The position of the dots will be determined in a different function using multiple arrays to create a grid like array, where the indexes act as coordinates. The dots will be identified using numbers greater than 0 and there should be several pairs of dots, (the number should depend on the level difficulty, so harder levels have more pairs of dots) the number of the dot in the puzzle array will determine what colour the dot will be.

Create Grid UI

Once the grid and dots are drawn, the Grid UI should be implemented, this involves knowing when the user has pressed a button from "Level" UI or when the user wants to draw a path. The program should be able to recognise where the user is drawing a path based on the position of the mouse when the user is holding down the left mouse button whilst moving the mouse across the grid squares. This function should do this by using the "pygame.event.get()" method to determine what the user is doing with the mouse and therefore what actions need to be carried out.

Sprint 3:

Identify when the game is completed

Once all the Grid interface has been implemented, the focus should now be to detect for when the game is completed, this would be when all the grid squares are filled with a path, and when all dots have been connected to their corresponding dot of the same colour. When this is detected the "level end screen" UI should replace the "level" UI so the user can play another game, return to previous menu screen, or quit the game. By doing this it ensures that the game loop is breakable, and the game actually does something when completed.

Validation for the paths

Once the game can detect if the game is solved, path validation should be implemented next, this involves making sure that paths can only be drawn by starting from a dot, paths can only be drawn to the same colour dot, and if paths cross, the path is that has been crossed gets removed entirely. This should be done in the part of the code where the initial drawing algorithm is and should take place at the beginning, and end of that part of the code. To do this the coordinates of the paths and dots should be tracked as if drawing doesn't start at a coordinate a dot is held at, a path shouldn't be drawn. If the dot number at the first coordinate doesn't match the dot number at the second coordinate, a path shouldn't be drawn. And if coordinates of path overlap, the paths should be removed from the grid.

The buttons on the "Level" interface now have a function

Once the path validation has been implemented, the focus should be to give each of the "Level" buttons (Hint, Undo and Reset) a function. If a mouse click/ press is detected the program should check the coordinates of this, and if the coordinates are over one of these buttons, the corresponding action should occur. The "Hint" button should use pathfinding to give the user 1 solution/ solved path, this will help the user if they are struggling to complete the puzzle. The "Undo" button should work similar to the path validation, coordinates of the path drawn should be checked, and then the squares the paths are in should be reset so that the path is not in them. Finally, the "reset" button should remove all paths drawn by the user, this should be done by resetting all squares in the grid reverting them to what they were when the "drawGrid" was first called.

Sprint 4:

Random generation of dots in solvable positions

The main focus of this sprint should be to make sure that dots are generated randomly but do this whilst making sure that the puzzle can be solved, this is because this is the most important aspect of the game as if the program can always generate a solvable puzzle, it guarantees that the user can play for long periods of time without getting bored. This should be done using a pathfinding algorithm to determine all possible routes the set of dots generated and to see if corresponding dots connect to each other while making sure they do not overlap, if a puzzle is solvable the game should draw that puzzle for the user to solve, if the puzzle is not solvable, the game should generate a puzzle until a puzzle that can be solved is found.

Storing to database

When the user presses "Quit" after solving a puzzle, a new Tkinter window will pop up requesting they input a username to be stored in the database. At this point in time I am unsure as to what the scoring system will be, whether it be time based, point based or level

completion based, but I will deliberate with my stakeholders to see what their preferences are. Once the user has entered a username, the program will check to see if the username is valid or not and will confirm to the user if they have been successful or not.

Leaderboard screen

When the "Leaderboard" button is pressed on the menu screen, a new Tkinter window will be opened and will display the Leaderboard. It will contain a title and several buttons allowing potential sorting methods on a statistic of their choice (statistics will be decided after confirming with stakeholders). Once the window is opened, the program would user SQL to retrieve the results from the table and will display the top results in the desired category.

Sprint 1

Aim of this sprint

My aim for this sprint is to create the game, menu and level classes and to create a user interface for the game's menu and the level selection interface. I am starting with this as I would believe it would be more efficient to do the GUIs before any game mechanics, this will provide the game with a good foundation on which I can implement the game logic in the next few sprints.

Success Criteria

After this sprint I hope to achieve:

- Main menu interface is completed
- Level selection interface is completed
- Main menu and level selection buttons do the correct process when pressed

Pseudocode

```
Class Game
    Public title
    Public level
    public gridSize
    public cellSize
    public pathWidth
    public pathColour
    public windowWidth
    public windowHeight
    public gridColour
    public bgColour
    public bgImage = “bg.png”


    public procedure main()
        menu = self.Menu()
        menu.displayMenu()

    public procedure quit()
        exit()


Class menu
    Public leaderboard
    Public gameStart
    Public buttons
    public bigButtonHeight
    public bigButtonWidth
    public smallButtonHeight
    public smallButtonWidth

    Public procedure displayMenu()
        display “background image”
        drawTitle()
        createButtons()
        while game is running
            if button is clicked
                if button is start
                    gameStart()
                if button is leaderboard
                    leaderboard()
                if button is quit
                    quit()
            end if
        end while

    Public procedure drawTitle()
        display “title”

    Public procedure createButtons()
        display “start button”
        display “leaderboard button”
        display “quit button”
```

```
    Public procedure gameStart()
        print "Game started"

    Public procedure leaderboard()
        print "Leaderboard displayed"

Class button
    Public text
    Public x
    Public y
    Public width
    Public height
    Public activeColour
    Public inactiveColour
    Public action

    Public procedure drawButton()
        if mouse is over button
            display activeColour
            if mouse is clicked
                action()
            else
                display inactiveColour
        else
            display inactiveColour
        display text

Class levelSelection
    Public levels
    Public level
    Public buttons
    Public levelButtonHeight
    Public levelButtonWidth

    Public procedure displayLevels()
        display “background image”
        drawTitle()
        createButtons()
        while game is running
            if button is clicked
                if button is "Easy 5x5"
                    easyLevelGame()  'Fuction to start easy level game to be defined in future sprint'
                else if button is "Medium 7x7"
                    mediumLevelGame() 'Fuction to start medium level game to be defined in future sprint'
                else if button is "Hard 10x10"
                    hardLevelGame() ' Fuction to start hard level game to be defined in future sprint'
                else if button is "Back"
                    menu.displayMenu()
            end if
        end while

    Public procedure drawTitle()
        display “title”

    Public procedure createButtons()
        display “Easy- 5x5”
        display “Medium- 7x7”
        display “Hard- 10x10”
        display “Back”

game = new Game()
game.main()
```

Sprint 1 Code

```
#Colours
black = (0, 0, 0)
white = (255, 255, 255)
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
yellow = (255, 255, 0)
purple = (128, 0, 128)
orange = (255, 165, 0)
grey = (141, 144, 147)
forGreen = (34, 139, 34)
```

First, I started the program by setting the colour constants using RGB codes, I chose to do it this way instead of something like hex codes as it is easier to use and implement.

```
# Game class
class Game:
    def __init__(self):
        self.title = "Connect The Dots"
        self.running = True
        self.gridSize = 5
        self.cellSize = 100
        self.pathWidth = 20
        self.gridColour = (255, 255, 255)
        self.pathColour = (0, 0, 255)
        self.windowWidth = 1102.5
        self.windowHeight = 735
        #loads the background image and scales bg image to fit the window
        self.morph = pygame.image.load("background6.png")
        self.bgImage = pygame.transform.scale(self.morph, (self.windowWidth, self.windowHeight))

        self.display = pygame.display.set_mode((self.windowWidth,self.windowHeight))
        pygame.display.set_caption(self.title)

    def main(self): #creates menu class and displays menu
        self.menu = Menu()
        self.menu.displayMenu()

    def quit(self): #quits the game
        pygame.quit()
        quit()
```

```
# Create an instance of the Game class and call the main method
newGame = Game()
newGame.main()
```

I then created a game class, with the following attributes, title, running, gridSize, cellSize, pathWidth, pathColour, windowWidth, windowHeight, morph, bgImage and display. Despite not yet using gridSize, cellSize, pathWidth, pathColour and gridColour in this sprint, I still chose to have them as attributes for this class so that I can use them in this class and a future "Grid" class for future sprints.

The method "main(self)" calls the "Menu" class and runs the "displayMenu()" method withing that class, this will do this immediately once the program is opened.

The method "quit(self)" will simply quit pygame and then exit the tab when called.

Attribute	*What it does/ will do*
"self.title"	Will be used to set the window caption
"self.running"	Used to repeat the main loop until a condition is met
"self.gridSize"	Will be used to set the grid size depending on the level difficulty selected by the user e.g 5x5, 7x7 etc
"self.cellSize"	Will set the size of each individual square on the grid
"self.pathWidth"	Will set the width of the path when the user attempts to connect dots
"self.pathColour"	Will set the colour of the path
"self.windowWidth"	Sets the width of the window
"self.windowHeight"	Sets the height of the window
"self.morph"	Loads the background image
"self.bgImage"	Scales the background image to the windows width and height
"self.display"	Sets the window caption as whatever "self.title" is set as

```
# Button class
class Button:
    #creates buttons with text, x, y, width, height, colour, action
    def __init__(self,text, x, y, w, h, inactive_colour, active_colour, action):
        self._text = text
        self._x = x                     #coordinates of top left corner of button
        self._y = y
        self._w = w                     #button dimensions
        self._h = h
        self._inactive_colour = inactive_colour
        self._active_colour = active_colour
        self._action = action           #function to be called when button is clicked


    def drawButton(self):
        mouse = pygame.mouse.get_pos() #gets mouse position
        click = pygame.mouse.get_pressed() #gets mouse click

        if self._x + self._w > mouse[0] > self._x and self._y + self._h > mouse[1] > self._y: #checks if mouse is over button
            pygame.draw.rect(newGame.display, self._active_colour, (self._x, self._y, self._w, self._h)) #draws button in active colour
            if click[0] == 1 and self._action != None: #checks if button is clicked
                self._action() #calls the function associated with the button
        else:
            pygame.draw.rect(newGame.display, self._inactive_colour, (self._x, self._y, self._w, self._h)) #otherwise draws button in inactive colour

        smallText = pygame.font.Font("freesansbold.ttf", 20) #creates font, text and shape for button
        textSurf = smallText.render(self._text, True, black)
        textRect = textSurf.get_rect()

        textRect.center = ((self._x + (self._w/2)), (self._y +(self._h/2))) #displays and centers text on button
        newGame.display.blit(textSurf, textRect)
```

This class is used for generating buttons, I chose to do have this as a class as this will allow me to create buttons anywhere in the program with relative ease. The "drawButton" method draws buttons on the screen as well as handles its interaction with the mouse. It uses the attributes

from the class to determine, the text on the button, the position of the button on the games display, the width and height of the button, the colours of the button depending on if the mouse is hovering over the button or not.

Attribute	*What it does*
"self.text"	Sets what the button will say/ display
"self.x"	Sets button x coordinate on the game window
"self.y"	Sets button y coordinate on the game window
"self.w"	Sets the button width
"self.h"	Sets the button height
"self.inactive_colour"	Sets what colour the button will be when the mouse is not hovering over it
"self.active_colour"	Sets what colour the button will be when the mouse is hovering over it
"self.action"	Will set what the button will do when it is clicked

```
# Menu class
class Menu:
    def __init__(self): #initialise variables
        self.leaderboard = None
        self.gameStart = False
        self.buttons = None
        self.bigButtonHeight = 60
        self.bigButtonWidth = 120
        self.smallButtonHeight = 30
        self.smallButtonWidth = 100

    def text_objects(self, text, font):
        textSurface = font.render(text, True, grey)
        return textSurface, textSurface.get_rect()

    def displayMenu(self):
        newGame.display.blit(newGame.bgImage,(0,0)) #display background image
        self.drawTitle()    #draw title
        self.createButtons()    #create buttons

        while newGame.running: #main loop for menu
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.quit()
                    pygame.quit()
                for button in self.buttons:
                    button.drawButton() # draws button and checks if pressed
            pygame.display.update()
            newGame.clock.tick(30)
        newGame.playGame() # starts game when loop ends
```

```
def createButtons(self):
    self.buttons = [] #creates an array for the buttons to be stored in
    #Play button added to the array
    self.buttons.append(Button("Play", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, forGreen, green, self.levelSelection))
    #Leaderboard button added to the array
    self.buttons.append(Button("Leaderboard", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 400, self.bigButtonWidth, self.bigButtonHeight, forGreen, green, self.showLeaderboard))
    #Quit button added to the array
    self.buttons.append(Button("Quit", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, forGreen, green, newGame.quit))

def createButtons2(self):
    self.buttons = [] #sets another an array for the buttons to be stored in
    #Easy level button added to the array
    self.buttons.append(Button("Easy- 5x5", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, forGreen, green, self.easyLevel))
    #Medium level button added to the array
    self.buttons.append(Button("Medium- 7x7", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 400, self.bigButtonWidth, self.bigButtonHeight, forGreen, green, self.mediumLevel))
    #Hard level button added to the array
    self.buttons.append(Button("Hard- 10x10", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, forGreen, green, self.hardLevel))
    #Back button added to the array
    self.buttons.append(Button("Back", 150, 600, self.bigButtonWidth, self.smallButtonHeight, forGreen, green, self.displayMenu))
```

```
def drawTitle(self): #draws the title
    titleFont = pygame.font.Font("freesansbold.ttf", 100)
    titleSurf, titleRect = self.text_objects("Connect The Dots", titleFont)
    titleRect.center = ((newGame.windowWidth/2), (newGame.windowHeight/7))
    newGame.display.blit(titleSurf, titleRect)

def startGame(self):
    print("Play button pressed")
    self.gameStart = True #sets gameStart to true

def showLeaderboard(self):
    print("Leaderboard button pressed")

def easyLevel(self):
    print("Easy Level works")

def mediumLevel(self):
    print("Medium Level works")

def hardLevel(self):
    print("Hard Level works")

def levelSelection(self): #displays level selection menu
    newGame.display.blit(newGame.bgImage,(0,0))
    self.drawTitle()
    self.createButtons2()
    pygame.display.update()
    levelSelect = True

    while levelSelect: #main loop for level selection
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                newGame.quit()
                pygame.quit()
        for button in self.buttons: #draws button and checks if pressed
            button.drawButton()
        pygame.display.update()
        newGame.clock.tick(30)
```

Attribute	*What it does/ will do*
"self.leaderboard"	Will allow the user to see a leaderboard in a future sprint
"self.gameStart"	Used in a loop to show that game has not yet started
"self.buttons"	Used to loop so the button array can print out the buttons created by the class
"self.bigButtonHeight"	Sets the height of a "big button"
"self.bigButtonWidth"	Sets the width of a "big button"

"self.smallButtonHeight"	Sets the height of a "small button"
"self.smallButtonWidth"	Sets the width of a "small button"

I then chose to create a "Menu" class instead of having a "Menu" method in the "Game" class as there are many of features which are easier to implement when in a class rather than when being in a single larger method. I chose to set the button size attributes in this class as it this is where most of the menu buttons will be so they will only be accessed through this class. The "self.buttons" attribute is used in a for loop when the buttons are drawn, the buttons are generated in a separate method within the "Menu" class where they are appended to an empty array. Then they are drawn 1 after another in the for loop using the "drawButton()" method of the "Button" class. I chose to implement the buttons in this way as it is easier to keep track of what buttons are used in which screen/ part of the game as buttons are created in different arrays and draw using a for loop when needed.

The "text_objects" method creates and returns a rendered text surface and its corresponding area. The main use of this method is drawing the title on the menu and "levelSelection" screen, I used this method as it renders the text in few lines of code.

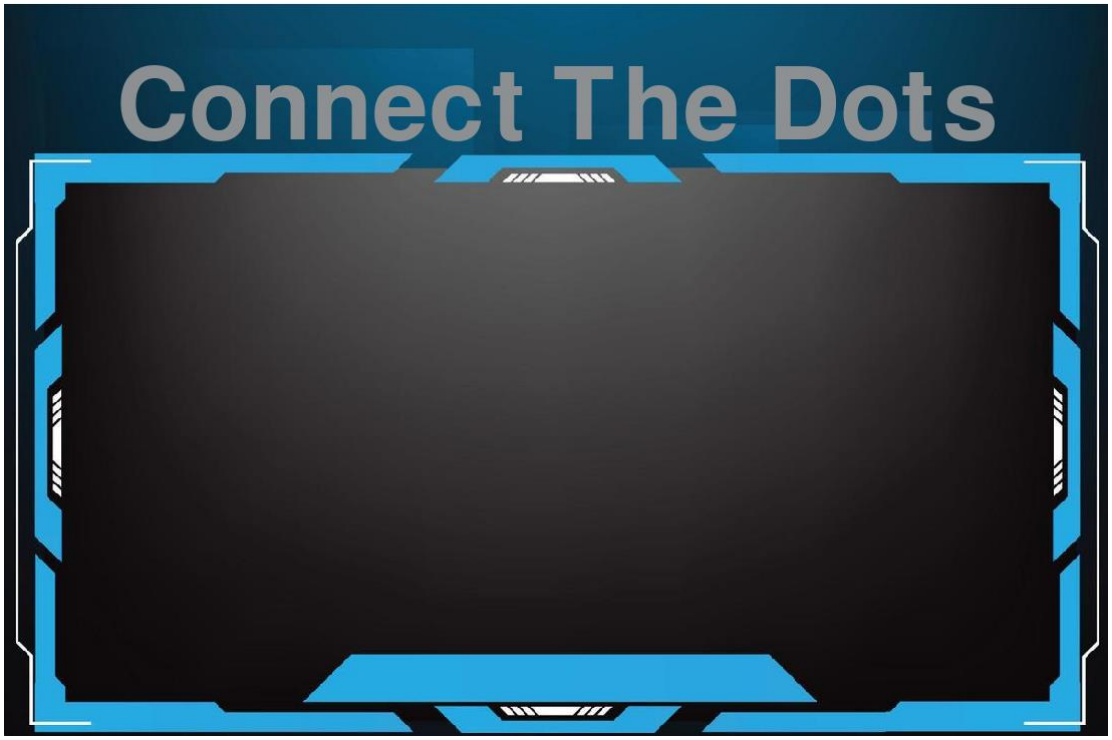
At the time of finishing this sprint the leader board and different difficulty levels have not been developed so when the corresponding buttons are clicked, a simple print statement is output instead of a visual change to the game.

Problems I encountered while coding

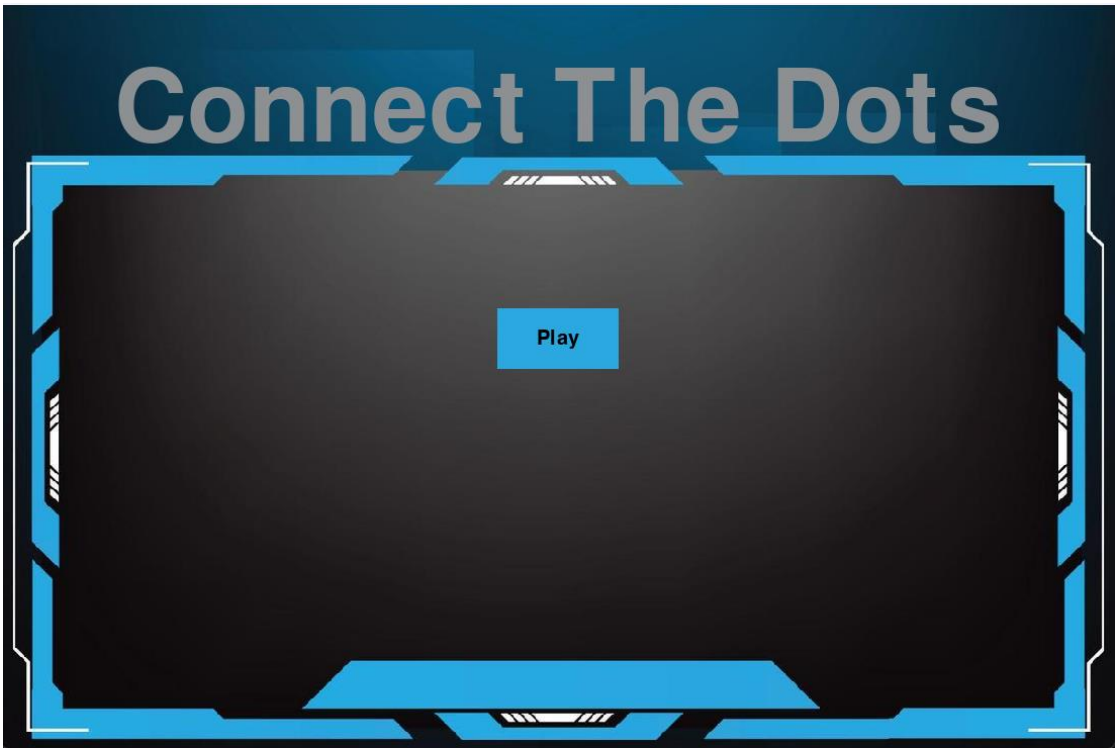
The only major problem I ran into this sprint was that upon running the program, the buttons would not appear until the user hovered the mouse over them. This was a problem as unless the player knows where the buttons are beforehand, it would be time consuming to find where the buttons are, although being an interesting problem to encounter this is not how I wanted the button mechanics to work. To solve this, I looked over the "Button" class and its methods and saw that the "else" statement was indented as so:

```
if self._x + self._w > mouse[0] > self._x and self._y + self._h > mouse[1] > self._y: #checks if mouse is over button
    pygame.draw.rect(newGame.display, self._active_colour, (self._x, self._y, self._w, self._h)) #draws button in active colour
    if click[0] == 1 and self._action != None: #checks if button is clicked
        self._action() #calls the function associated with the button
    else:
        pygame.draw.rect(newGame.display, self._inactive_colour, (self._x, self._y, self._w, self._h)) #otherwise draws button in inactive colour

    smallText = pygame.font.Font("freesansbold.ttf", 20) #creates font, text and shape for button
    textSurf = smallText.render(self._text, True, black)
    textRect = textSurf.get_rect()

    textRect.center = ((self._x + (self._w/2)), (self._y +(self._h/2))) #displays and centers text on button
    newGame.display.blit(textSurf, textRect)
```

What it looked like upon running the program:



What it looked like when mouse crosses the button:

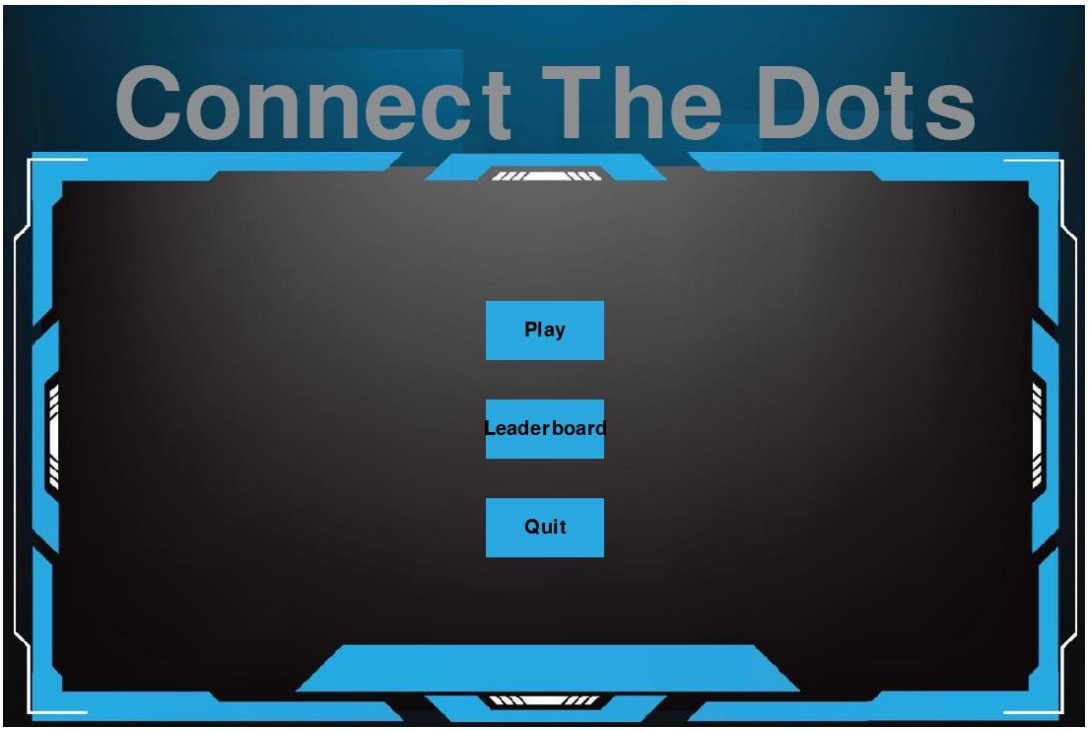


This was a problem as the else statement is used to draw the button in an inactive colour when the mouse is not over the button. The indentation of the else statement is crucial because it determines whether the code inside the else block will be executed as part of the else statement or not. Because of this there was no alternative action for the program to take until the mouse was over the button as it had no else statement to check.

I solved this by indenting the else statement in the correct block like so:

```
    if self._x + self._w > mouse[0] > self._x and self._y + self._h > mouse[1] > self._y: #checks if mouse is over button
        pygame.draw.rect(newGame.display, self._active_colour, (self._x, self._y, self._w, self._h)) #draws button in active colour
        if click[0] == 1 and self._action != None: #checks if button is clicked
            self._action() #calls the function associated with the button
    else:
        pygame.draw.rect(newGame.display, self._inactive_colour, (self._x, self._y, self._w, self._h)) #otherwise draws button in inactive colour

    smallText = pygame.font.Font("freesansbold.ttf", 20) #creates font, text and shape for button
    textSurf = smallText.render(self._text, True, black)
    textRect = textSurf.get_rect()

    textRect.center = ((self._x + (self._w/2)), (self._y +(self._h/2))) #displays and centers text on button
    newGame.display.blit(textSurf, textRect)
```

What it looks like when indented correctly:



Test Plan

Main menu

Action	Reason	Expected outcome
Open game	Make sure game initialises correctly	Menu appears with background and buttons in correct places
Press "Play"	Check "levelSelection" function works correctly	Level selection UI replaces Menu UI
Press "Leaderboard"	Checks "showLeaderboard" function works correctly	New window opens while game window remains open in the background

Press "Quit"	Checks "quit" function works correctly	Window closes and program stops running

Level Selection

Action	Reason	Expected Outcome
Press "Easy 5x5"	Checks "easyLevel" function works correctly	A 5x5 square grid is generated
Press "Medium 7x7"	Checks "mediumLevel" function works correctly	A 5x5 square grid is generated
Press "Hard 10x10"	Checks "hardLevel" function works correctly	A 5x5 square grid is generated
Press "Back button"	Checks "displayMenu" function works correctly	Game returns to menu screen

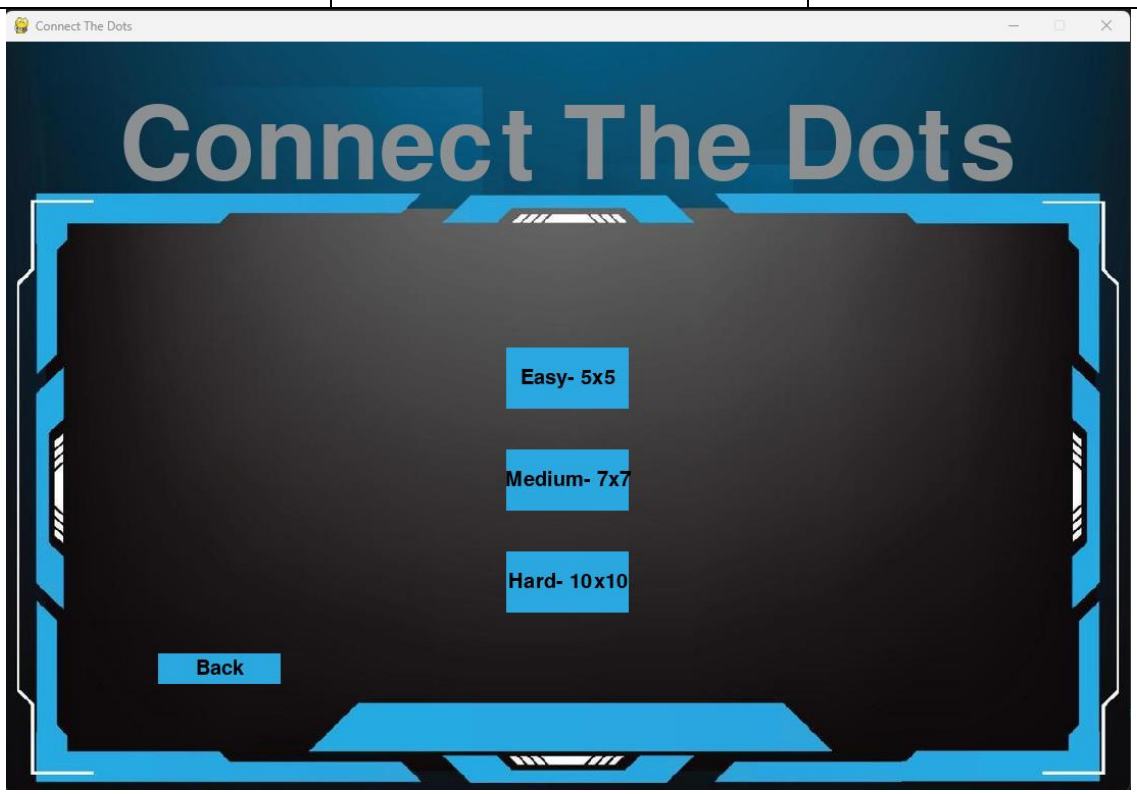
Formal Testing

Normal Data

Action	Expected Outcome	Actual Outcome
Game opened	Game title, background image and 3 buttons appear	As expected

Press "Play"	Menu interface replaced by level selection interface- background image, game title,	As expected

	3 central buttons and a back button	



Press "Leaderboard"	Button press confirmed in python shell. Menu doesn't change.	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Leaderboard button pressed
```

Press "Quit"	Window closes and program stops running	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Leaderboard button pressed
PS C:\Users\nasel\OneDrive\Documents\Howard stuff\A Level\Computing\A LEVEL PROJECT>
```

Press "Easy- 5x5"	Button press confirmed in python shell. Menu doesn't change.	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Easy Level works
```

Press "Medium- 7x7"	Button press confirmed in python shell. Menu doesn't change.	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Easy Level works
Medium Level works
```

Press "Hard- 10x10"	Button press confirmed in python shell. Menu doesn't change.	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Easy Level works
Medium Level works
Hard Level works
```

Press "Back"	Level selection interface is replaced with menu interface- game title, background image and 3 buttons	As expected

Boundary Data

There is no boundary data for this sprint.

Erroneous Data

Press anywhere on the game window	Nothing	As expected

Evaluation

Success Criteria	Completed?
Main menu interface is completed	Yes
Level selection interface is completed	Yes

Main menu and level selection buttons do the correct process when pressed	Yes

I have successfully achieved what I aimed to completed during this sprint. I have set considerable groundwork which will enable me to build on the current program with the addition of more classes and other logic.

Next Sprint

In the next sprint I aim to achieve:

- Create algorithm to generate grid
- Generate dots onto the grid in pre-determined places on the grid
- Create grid UI
- Create buttons for "level" from GUI design

Sprint 2

Aim of this sprint

My aim for this sprint is to create an algorithm that will generate a grid that corresponds to the level difficulty depending on what difficulty button the user pressed. Upon doing this the program should then generate several pairs of dots, at this point I will have the dots spawn in set locations on the grid and will get them to randomly generate in a later sprint. I will then attempt to implement the grid UI which will mean that the user can click on a dot a draw a line around the grid squares. And then to end the sprint I will create the buttons from the GUI design "level" and have then print out a statement when pressed.

Success Criteria

In this sprint I aim to achieve:

- Create algorithm to generate grid
- Generate dots onto the grid in pre-determined places on the grid
- Create grid UI
- Create buttons for "level" from GUI design

Pseudocode

```
Class Game
    Public title
    Public level
    public gridSize
    public cellSize
    public pathWidth
    public pathColour
    public windowWidth
    public windowHeight
    public gridColour
    public bgColour
    public bgImage = “bg.png”


    public procedure main()
        menu = self.Menu()
        menu.displayMenu()

    public procedure quit()
        exit()


Class menu
    Public leaderboard
    Public gameStart
    Public buttons
    public bigButtonHeight
    public bigButtonWidth
    public smallButtonHeight
    public smallButtonWidth

    Public procedure displayMenu()
        display “background image”
        drawTitle()
        createButtons()
        while game is running
            if button is clicked
                if button is start
                    gameStart()
                if button is leaderboard
                    leaderboard()
                if button is quit
                    quit()
            end if
        end while

    Public procedure drawTitle(self)
        display “title”

    Public procedure createButtons(self)
        display “start button”
        display “leaderboard button”
        display “quit button”
```

```
    Public procedure gameStart(self)
        print "Game started"

    Public procedure leaderboard(self)
        print "Leaderboard displayed"

    Public procedure text(self, text, font)
        surface = font.render(text, True, (0, 0, 0))
        return surface, surface.get_rect()

Class button
    Public text
    Public x
    Public y
    Public width
    Public height
    Public activeColour
    Public inactiveColour
    Public action

    Public procedure drawButton()
        if mouse is over button
            display activeColour
            if mouse is clicked
                action()
            else
                display inactiveColour
        else
            display inactiveColour
        display text
```

```
Class levelSelection
    Public levels
    Public level
    Public buttons
    Public levelButtonHeight
    Public levelButtonWidth

    Public procedure displayLevels()
        display “background image”
        drawTitle()
        createButtons2()
        while game is running
            if button is clicked
                if button is "Easy 5x5"
                    newEasyLevel = level()
                    newEasyLevel.easyLevelGame()
                else if button is "Medium 7x7"
                    newMediumLevel = level()
                    newMediumLevel.mediumLevelGame()
                else if button is "Hard 10x10"
                    newHardLevel = level()
                    newHardLevel.hardLevelGame()
                else if button is "Back"
                    menu.displayMenu()
            end if
        end while

    Public procedure drawTitle()
        display “title”

    Public procedure createButtons2()
        display “Easy- 5x5”
        display “Medium- 7x7”
        display “Hard- 10x10”
        display “Back”
```

```
Class Grid
    self.rows = rows
    self.cols = cols
    self.cellSize = cellSize
    self.pathWidth = pathWidth
    self.grid = []
    self.paths = []



    Public procedure drawDots()
        self.grid = []
        self.grid = [[1,0,0,0,1]]
        self.grid.append([2,0,0,0,2])
        self.grid.append([3,0,0,0,3])
        self.grid.append([4,0,0,0,4])
        self.grid.append([5,0,0,0,5])

    Public procedure drawGrid()
        for row in range(self.rows)
            for col in range(self.cols)
                drawRectangle(col * cellSize, row * cellSize, cellSize, cellSize, gridColour)
                self.drawDots()
                if self.grid != 0
                circle_center = (col * cellSize + cellSize // 2, row * cellSize + cellSize // 2)
                    if self.grid[row][col] == 1
                        drawCircle(display, red, circle_center, circle_radius)
                    else if self.grid[row][col] == 2
                        drawCircle(display, blue, circle_center, circle_radius)
                    else if self.grid[row][col] == 3
                        drawCircle(display, green, circle_center, circle_radius)
                    else if self.grid[row][col] == 4
                        drawCircle(display, yellow, circle_center, circle_radius)
                    else if self.grid[row][col] == 5
                        drawCircle(display, orange, circle_center, circle_radius)
                    end if
                end if
            end for
        end for

    Public procedure getNumber()
        return self.grid[row][col]
```

```
Public procedure drawPath()
    for path in paths
        if length(path) > 1
            for i = 0 to length(path) - 1
                start = path[i]
                end = path[i + 1]
                if start[0] == end[0] or start[1] == end[1]
                    number = self.getNumber(path[i][1], path[i][0])
                    pathColour = 0
                    if number == 1:
                        pathColour = red
                    else if number == 2:
                        pathColour = blue
                    else if number == 3:
                        pathColour = green
                    else if number == 4:
                        pathColour = yellow
                    else if number == 5:
                        pathColour = orange

                    drawLine(display, pathColour, start, end, self.pathWidth )
                else:
                    number = self.getNumber(path[i][1], path[i][0])
                    pathColour = 0
                    if number == 1:
                        pathColour = red
                    else if number == 2:
                        pathColour = blue
                    else if number == 3:
                        pathColour = green
                    else if number == 4:
                        pathColour = yellow
                    else if number == 5:
                        pathColour = orange
                    controlPoint = (end[0], start[1])
                    drawLine(display, pathColour, start, controlPoint, end, self.pathWidth)
```

```
Class level
    Public levelNum
    Public gridSize
    Public cellSize
    Public pathWidth
    Public pathColour
    Public gridColour
    Public bgColour
    Public bgImage = “bg.png”

    Public procedure main()
        levelSelection = self.levelSelection()
        levelSelection.displayLevels()

    Public procedure easyLevelGame()
        gridSize = 5
        cellSize = 50
        pathWidth = 10
        gridColour = white



    Public procedure mediumLevelGame()
        print "Medium level game started"

    Public procedure hardLevelGame()
        print "Hard level game started"


game = new Game()
game.main()
```

Sprint 2 Code

The following code is used within the “Menu” class, it creates an array of buttons for the level interface as seen in the GUI Design- Level. The buttons don’t have affect the game/ puzzle at the time of this sprint, but at a later sprint a corresponding function will be implemented to each button. I chose to implement the buttons in this way as it is easier to keep track of what buttons are used in which screen/ part of the game as buttons are created in different arrays and draw using a for loop when needed.

```
    def createButtons3(self):
        self.buttons = [] #sets another an array for the buttons to be stored in
        #Undo button added to the array
        self.buttons.append(Button("Undo", (newGame.windowWidth/2) - (75/2) - 160, 650, 75, 30, bgBlue, inactiveBlue, self.undoMove))
        #Hint button added to the array
        self.buttons.append(Button("Hint", (newGame.windowWidth/2) - (75/2) + 160, 650, 75, 30, bgBlue, inactiveBlue, self.giveHint))
        #Reset level button added to the array
        self.buttons.append(Button("Reset", (newGame.windowWidth/2) - (75/2), 650, 75, 30, bgBlue, inactiveBlue, self.resetLevel))

    def undoMove(self):
        print("Undo Move")

    def giveHint(self):
        print("Hint")

    def resetLevel(self):
        print("Reset Level")
```

The following code is also used within the "Menu" class, these methods are responsible for showing the game statistics, how many moves made, and how many paths successfully connected. However, I decided to put these stats in the top left-hand corner instead of being above the grid due to the limited space between the border of the background image and the edge of the grid.

```
    def gameStats(self): #displays the number of moves made by the player (value will be able to change in future sprint)
        movesFont = pygame.font.Font("freesansbold.ttf", 25)
        movesSurf = movesFont.render("Moves: " + str(newGame.moves), True, white)
        movesRect = movesSurf.get_rect()
        movesRect.topleft = (100, 200)
        newGame.display.blit(movesSurf, movesRect)


    def connectedPaths(self): #displays the number of connected paths (value will be able to change in future sprint)
        pathsFont = pygame.font.Font("freesansbold.ttf", 25)
        pathsSurf = pathsFont.render(str(newGame.joinedPaths) + " /5 Paths"  , True, white)
        pathsRect = pathsSurf.get_rect()
        pathsRect.topleft = (100, 230)
        newGame.display.blit(pathsSurf, pathsRect)
```

The following code is the main loop allowing the game to run, it starts by drawing all the game statistics and game title, it will then create then grid and later outputs it onto the display through a method inside the "Grid" class, it also checks for any mouse inputs and performs the corresponding action when detected.

```
    def easyLevel(self): #the main block of code for the game
        newGame.display.blit(newGame.bgImage,(0,0))
        self.drawTitle()
        self.gameStats() #displays the number of moves in the top left corner
        self.connectedPaths() #displays the number of connected paths in the top left corner
        self.createButtons3()
        newGrid = Grid(5, 5, 90, 20) #creates a grid object with 5 rows, 5 columns, cell size of 90 and path width of 20
        self.drawing = False
        self.currentPath = []

        while newGame.running:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.running = False

                elif event.type == pygame.MOUSEBUTTONDOWN: #checks if the mouse is clicked
                    self.drawing = True
                    mousePos = pygame.mouse.get_pos() #gets the mouse position and then converts it to a grid position
                    gridPos = newGrid.getCellFromMouse(mousePos)
                    self.currentPath.append(gridPos) #appends the grid position to the current path

                elif event.type == pygame.MOUSEBUTTONUP: #checks if the mouse is released
                    self.drawing = False
                    self.currentPath = []
                    if self.currentPath: #if the current path is not empty
                        newGrid.paths.append(self.currentPath) #once the mouse is released, the current path is added to the paths array

                elif event.type == pygame.MOUSEMOTION and self.drawing: #checks if the mouse is moving
                    mousePos = pygame.mouse.get_pos()
                    gridPos = newGrid.getCellFromMouse(mousePos)
                    if gridPos != self.currentPath[-1]: #if the grid position is not the same as the last grid position
                        self.currentPath.append(gridPos)

                for button in self.buttons: #draws the buttons
                    button.drawButton()

            pygame.display.flip() #updates the display
            newGrid.createGrid()
            newGrid.drawPath()
            if self.drawing and self.currentPath: #will only draw the path if the player is drawing and there is a path to draw
                for i in range(len(self.currentPath)-1): #loops through the current path
                    pathColour = 0 #sets the path colour to 0 initially
                    dotNum = newGrid.getDotColour(gridPos[1], gridPos[0])
                    if dotNum == 1:
                        pathColour = red
                    elif dotNum == 2:
                        pathColour = blue
                    elif dotNum == 3:
                        pathColour = green
                    elif dotNum == 4:
                        pathColour = yellow
                    elif dotNum == 5:
                        pathColour = orange
```

```
startPos = ((self.currentPath[i][0] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.xOffset, (self.currentPath[i][1] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.yOffset)
#stores the start position of line
endPos = ((self.currentPath[i+1][0] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.xOffset, (self.currentPath[i+1][1] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.yOffset)
#stores the end position of line
pygame.draw.line(newGame.display, pathColour, startPos, endPos, newGrid.pathWidth) #draws the line
```

In later sprints I will implement different difficulties and grid sizes.

```
    def mediumLevel(self):#this wont work unless puzzle has 7 rows and columns
        print("Medium Level works")

    def hardLevel(self): # this wont work unless puzzle has 10 rows and columns
        print("Hard Level works")
```

The following code is the part of the program that creates the grid, the drawing of paths, and creating the puzzle for the user to solve. It does this by creating an array and using numbers to store the position of the dots, in the screenshot below, the dots are already in set positions

and depending on the number, will determine what colour the dots (circles) will be. In future sprints I will attempt to implement random generation, so each puzzle is different and solvable.

The method "getDotColour" returns a single number, this number is from the puzzle to identify which dot is pressed if a dot is pressed.

```
class Grid:
    def __init__(self, rows, cols, cellSize, pathWidth):
        self.rows = rows
        self.cols = cols
        self.cellSize = cellSize
        self.pathWidth = pathWidth
        self.paths = [] #stores the paths drawn by the player
        self.xOffset = (newGame.windowWidth - self.cols * self.cellSize) // 2
        self.yOffset = (newGame.windowHeight - self.rows * self.cellSize) // 2 + 50 # +50 to offset, to account for the change in grid position

    def makePuzzle(self): #will add different puzzles for different levels later
        self.puzzle = []
        self.puzzle = [[1,0,0,0,1]]
        self.puzzle.append([2,0,0,0,2])
        self.puzzle.append([3,0,0,0,3])
        self.puzzle.append([4,0,0,0,4])
        self.puzzle.append([5,0,0,0,5])

    def createGrid(self):
        for row in range(self.rows):#draws grid
            for col in range(self.cols):
                rect = pygame.Rect(self.xOffset + col * self.cellSize, (self.yOffset + row * self.cellSize) , self.cellSize, self.cellSize) #creates rectangle for grid
                pygame.draw.rect(newGame.display, white, rect, 1) #draws grid lines
                self.makePuzzle() #draws dots
                if self.puzzle[row][col] != 0: #if the dot label is not 0, draw a dot
                    circle_centre = (self.xOffset + col * self.cellSize + self.cellSize // 2, ((self.yOffset + row * self.cellSize) ) + self.cellSize // 2)
                    circle_radius = 15
                    if self.puzzle[row][col] == 1: #if dot label is 1, draw red dot, if 2 draw blue dot, etc.
                        pygame.draw.circle(newGame.display, red, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 2:
                        pygame.draw.circle(newGame.display, blue, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 3:
                        pygame.draw.circle(newGame.display, green, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 4:
                        pygame.draw.circle(newGame.display, yellow, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 5:
                        pygame.draw.circle(newGame.display, orange, circle_centre, circle_radius)

    def getDotColour(self, row, col): #returns the number of the dot when looking at the puzzle
        return self.puzzle[row][col]
```

The following code is also part of the "Grid" class and will return a grid position depending on where the mouse is pressed on the window.

```
    def getCellFromMouse(self, pos): #returns the cell position of the mouse
        x, y = pos
        self.xOffset = int(self.xOffset) #sets xOffset to an integer as the window dimensions are floats
        row = (y - self.yOffset) // self.cellSize
        col = (x - self.xOffset) // self.cellSize
        if row < 0 or col < 0 or row > 4 or col > 4:
            return None
        return (col, row) #returns the cell position
```

Changes to Pseudocode

Level class:

While programming I chose not to have a "Level" class at this stage as I am only implementing one level for the "Easy" difficulty, however in future sprints, I may decide to change this for easier implantation of levels of different difficulty.

Level Selection class:

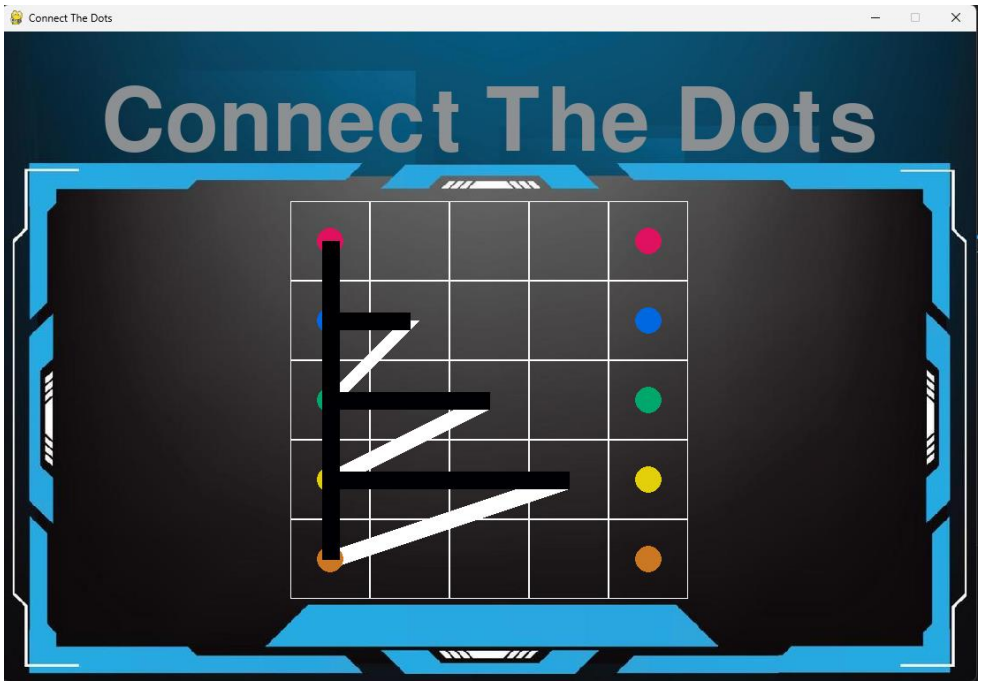
While programming I chose not to have a "Level Selection" class as the I believe that the "Level Selection" screen won't be used enough for the game to benefit from this additional feature.

Draw path method:

While programming I initially added the a "drawPath" method to the "Grid" class but upon testing I saw that the method doesn't affect the drawing of paths or the validation of the colour because this is done in the "easyLevel" method where the grid is first initialised.

Problems I encountered while coding

One of the problems for this sprint was that the drawing algorithm was not doing as I wanted it to, instead of drawing wherever the player dragged the mouse, it had drawn several lines as shown below. In this screenshot, I attempted to draw a horizontal line to connect the red dots but was shown these lines instead:



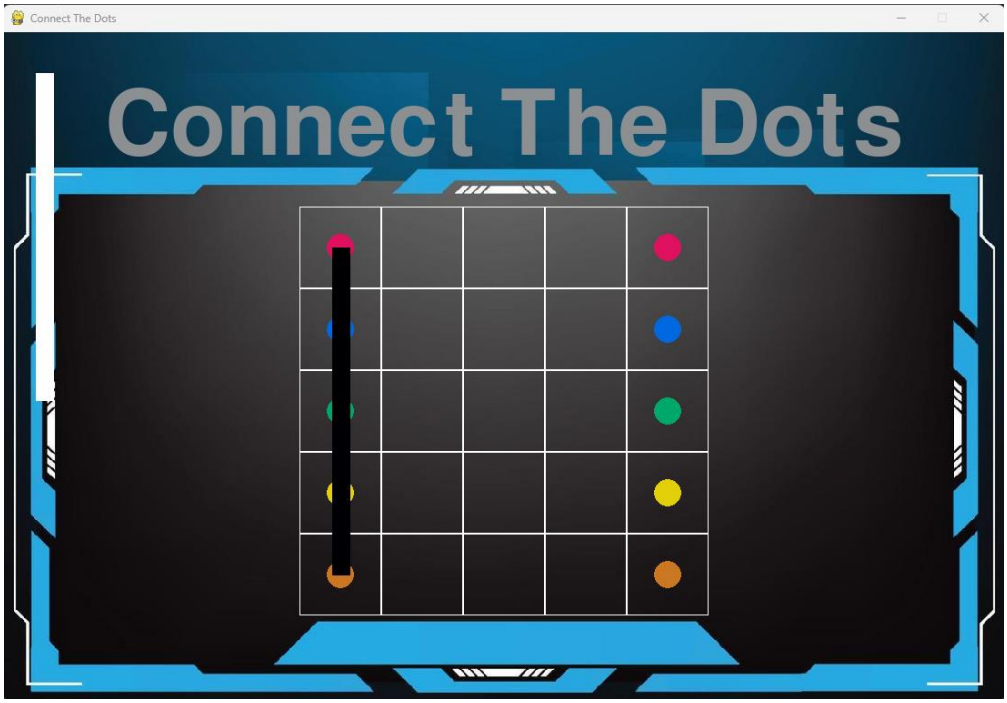
The blocks of code that controlled this aspect of the program looked like this:

```
def drawPath(self):
    for path in self.paths:
        if len(path) > 1:
            for i in range(len(path)-1):
                startPos = (path[i][1] * self.cellSize + self.cellSize // 2 + self.xOffset, path[i][1] * self.cellSize + self.cellSize // 2 + self.yOffset)
                endPos = (path[i+1][0] * self.cellSize + self.cellSize // 2 + self.xOffset, path[i+1][1] * self.cellSize + self.cellSize // 2 + self.yOffset)
                if startPos[0] == endPos[0] or startPos[1] == endPos[1]:
                    dotColour = self.puzzle[path[i][1]][path[i][0]]
                    pygame.draw.line(newGame.display, dotColour, startPos, endPos, self.pathWidth)
                else:
                    controlPoint = (endPos[0], startPos[1])
                    dotColour = self.puzzle[path[i][1]][path[i][0]]
                    pygame.draw.line(newGame.display, dotColour, startPos, controlPoint, self.pathWidth)
                    pygame.draw.line(newGame.display, dotColour, controlPoint, endPos, self.pathWidth)
```

The screenshot above shows the method that controlled the actual drawing of the line. What I found wrong with this method was that the "startPos" variable had "path[i][1]" which was the reason the diagonal lines were being drawn.

The code "path[i][1]" is used to access the second element of the "i"-th sub list within the "path" list. The part "path[i]" accesses the "i"-th sub list within the "path" list. The "[1]" indicates that we want the second item in the sub list. However, in Python, list indexing starts at 0, so setting "startPos" at "path[i][1]" would cause the program to start at the second item in the sub list. Therefore, changing this code to "path[i][0]" will give the desired outcome as it will access the correct item in the sub list.

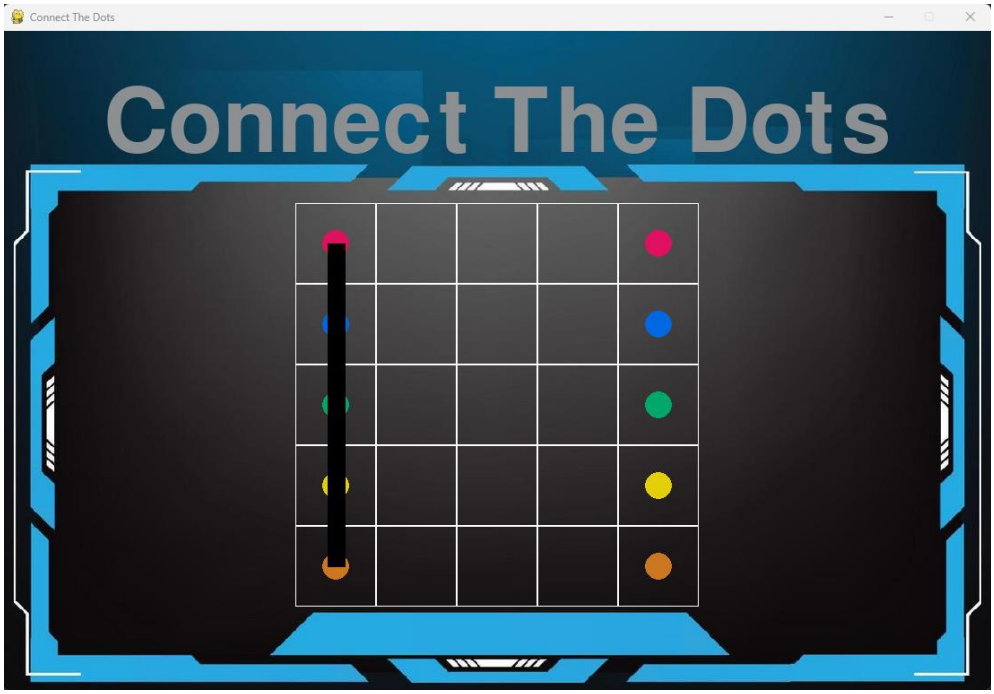
However, when corrected another undesired outcome occurred as shown below:



The method below was a part of the "Menu" class and would run when the user pressed the "Easy- 5x5" button, the only issue with this method was that at the end of the method the "startPos" and "endPos" variables were different to the ones in the "drawPath" method, in the "Grid" class. In this screenshot I had attempted to connect the red dots through a horizontal path. As a result of the variables being different values, the program would draw 2 different lines in the incorrect position being vertical. The reason for this is because the "drawPath" method had accounted for the x and y offset of the grid being centred and would draw a line in the correct/ desired starting point, whereas the "easyLevel" method did not account for the offset and would draw the line starting from the first "hidden" grid square top left of the window. To fix this problem I made sure the "startPos" and "endPos" variables were the same in both methods and that they accounted for the x and y offset of the grid.

```
def easyLevel(self):
    newGame.display.blit(newGame.bgImage,(0,0))
    self.drawTitle()
    newGrid = Grid(5, 5, 90, 20)
    self.drawing = False
    self.currentPath = []

    while newGame.running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                newGame.running = False

            elif event.type == pygame.MOUSEBUTTONDOWN:
                self.drawing = True
                mousePos = pygame.mouse.get_pos()
                gridPos = newGrid.getCellFromMouse(mousePos)
                self.currentPath.append(gridPos)
                print("Mouse position: ", gridPos)

            elif event.type == pygame.MOUSEBUTTONUP:
                self.drawing = False
                if self.currentPath:
                    newGrid.paths.append(self.currentPath)

            elif event.type == pygame.MOUSEMOTION and self.drawing:
                mousePos = pygame.mouse.get_pos()
                gridPos = newGrid.getCellFromMouse(mousePos)
                if gridPos != self.currentPath[-1]:
                    self.currentPath.append(gridPos)


        newGrid.createGrid()#and this
        newGrid.drawPath()#and this
        if self.drawing and self.currentPath:
            for i in range(len(self.currentPath)-1):
                startPos = (self.currentPath[i][0] * newGrid.cellSize + newGrid.cellSize // 2, self.currentPath[i][1] * newGrid.cellSize + newGrid.cellSize // 2)
                endPos = (self.currentPath[i+1][0] * newGrid.cellSize + newGrid.cellSize // 2, self.currentPath[i+1][1] * newGrid.cellSize + newGrid.cellSize // 2)
                pygame.draw.line(newGame.display, newGrid.pathColour, startPos, endPos, newGrid.pathWidth)
        pygame.display.flip()
```

The final problem I encountered was that a vertical line was being drawn instead of a horizontal line and a horizontal line was being drawn instead of a vertical line, as shown below:



I found that this was because a method used to determine grid position from where the mouse was, was returning the "row" and "col" values in the incorrect order, as seen below. To correct the error, I simply had the method return the values in the correct order being "(col, row)",

This fixed the problem, and the lines would now draw where the user drags the mouse.

```
def getCellFromMouse(self, pos):
    x, y = pos
    self.xOffset = int(self.xOffset)
    row = (y - self.yOffset) // self.cellSize
    col = (x - self.xOffset) // self.cellSize
    if row < 0 or col < 0 or row > 4 or col > 4:
        return None
    return (row, col)
```

Test Plan

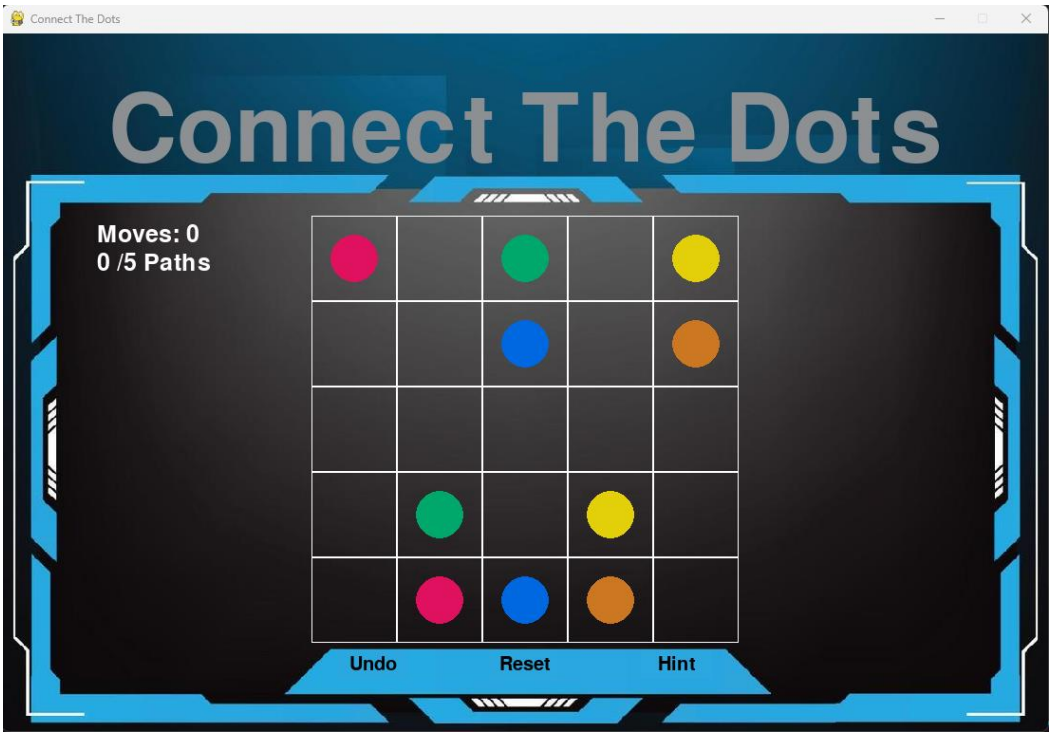
Level

Action	Reason	Expected Outcome
Press "Easy- 5x5" button	To check "easyLevel" function works correctly	A 5x5 square grid is output show with dots generated on it as well as 3 buttons and game statistics text
Presses a dot on the puzzle	Checks to see if anything unexpected happens	Nothing
Drags on dot while holding mouse button	Checks "easyLevel" function works correctly, this being the part that identifies when a path is being drawn	A line of the same colour as the dot is drawn
Connects a path to corresponding dot	Checks "easyLevel" function works correctly, this being the part that identifies when a path is being drawn	The line will be the same colour as the dot drawn from
Press "Hint"	Checks "giveHint" function works correctly	"Hint" is outputted
Press "Reset level"	Checks "resetLevel" function works correctly	"Reset level" is outputted
Press "Undo move"	Checks "undoMove" function works correctly	"Undo move" is outputted

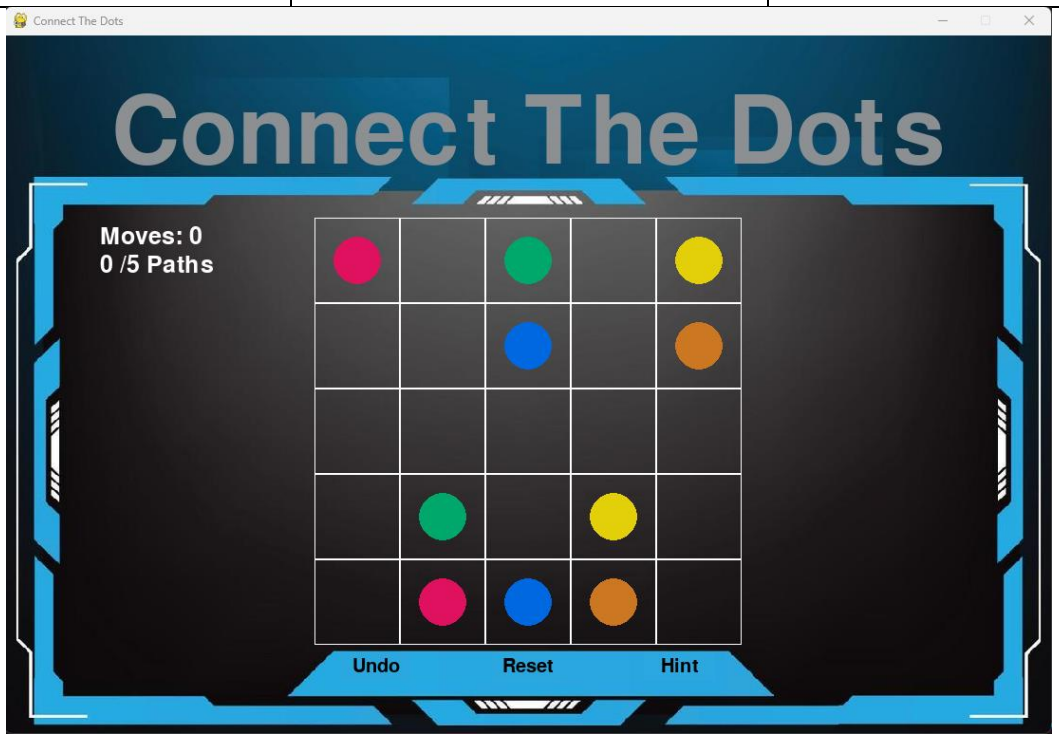
Formal Testing

Normal Data

Action	Expected Outcome	Actual Outcome
User presses "Easy- 5x5" button	A 5x5 square grid is shown with dots in pre-determined positions, 3 buttons are at the bottom of the window and 2 game statistics texts are in the top left corner	As expected



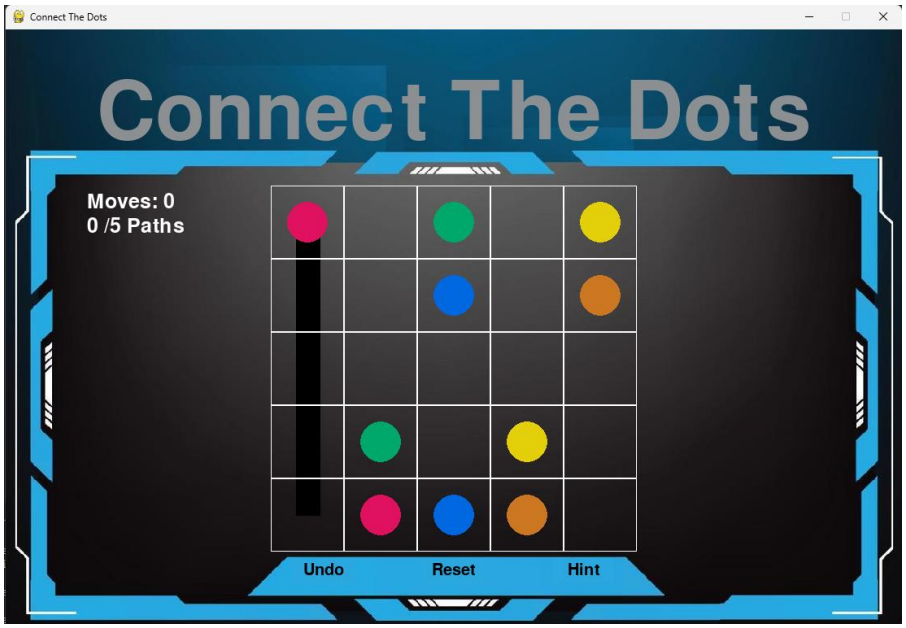
User presses a dot using the mouse	Nothing	As expected




```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

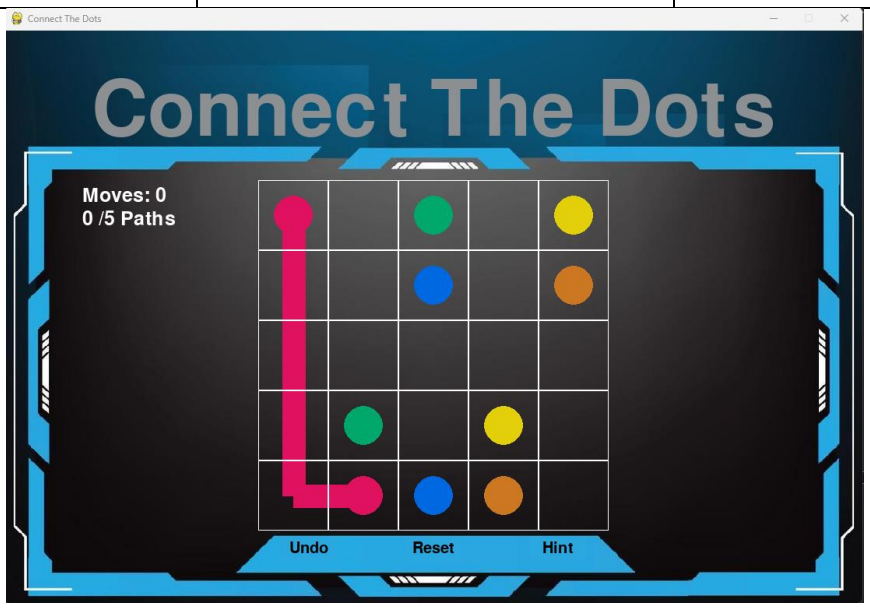
This is shown as there are no changes to the display or the system.

User drags on dot while holding mouse button	A line of the same colour as the dot is drawn	A black line is initially drawn



This was not an intended feature, and I will aim to fix it in future sprints.

User connects the path to the corresponding colour dot	The path will turn the same colour as the dot it is connected to.	As expected



User presses "Hint" button	"Hint" is outputted	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Hint
```

User presses "Reset" button	"Reset Level" is outputted	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Hint
Reset Level
```

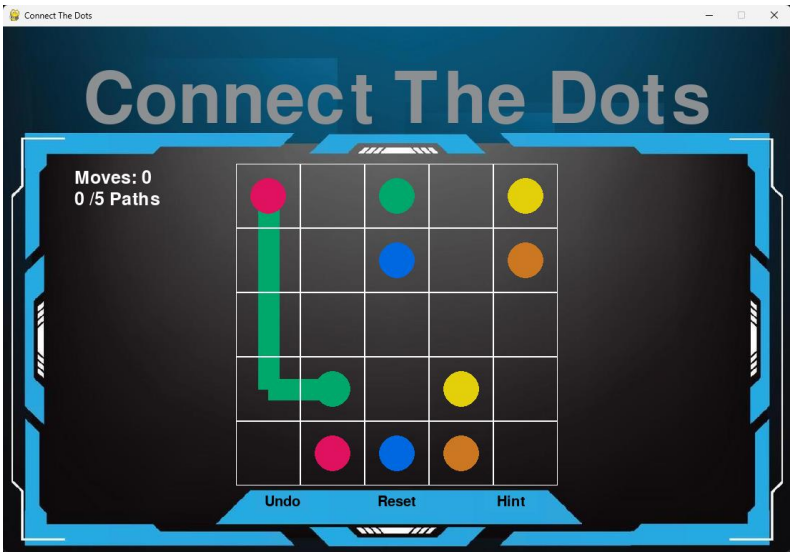
User presses "Undo" button	"Undo Move" is outputted	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Hint
Reset Level
Undo Move
```

Boundary Data

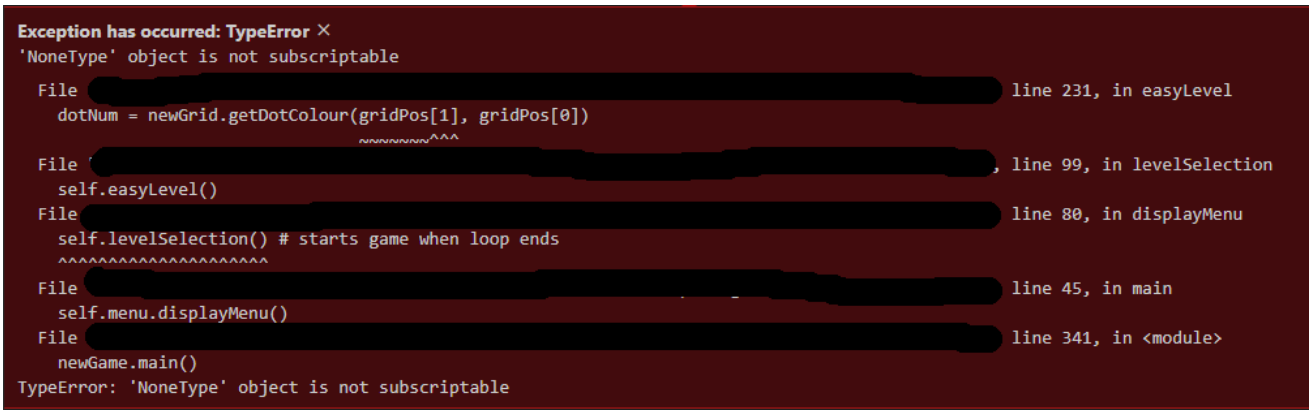
Action	Expected Outcome	Actual Outcome
Joining a path to a different colour dot	Path will turn the colour of the dot it is connected to	As expected

In this case I joined the red dot to the green dot causing the path to turn green.



Erroneous Data

Action	Expected Outcome	Actual Outcome
Press anywhere on the game window	Nothing	As expected
Drag a dot path outside the grid borders	Nothing	The program errors



Evaluation

Success Criteria	Completed?
Create algorithm to generate grid	Yes
Generate dots onto the grid in pre-determined places on the grid	Yes
Create grid UI	Yes
Create buttons for "level" from GUI design	Yes

In this sprint I have successfully achieved what I aimed to complete during this sprint, with some minor errors that will be fixed in the next few sprints, I can now focus on the main aspects of the game being the random generation and identifying when the game is completed.

Next Sprint

In the next sprint I aim to achieve:

- Identify when the game is completed
- Validation for the paths
  - Paths can only be drawn by starting from a dot
  - Paths can only be drawn to the same colour dot
  - If paths cross, the path is that has been crossed gets removed entirely
- Previous errors/ bugs are fixed
- The buttons on the "Level" interface now have a function

Sprint 3

Aim of this Sprint

My aim of this sprint is to further develop the features of the grid and level development to allow for a smoother and more efficient game, I plan to do this by creating an algorithm which will check when the game is completed (when all paths have been connected to the correct dots, and when all squares in the grid are filled). I also plan to do this by adding validation for when

drawing the paths, this is so that the chances of any mis-inputs by the user are ignored which will allow for a better experience when playing the game. Therefore, I also plan to add functions to the buttons that are in the "Level" interface to assist the player in completing the game. And to finish the sprint off I will attempt to fix the errors/bugs that I encountered in the previous sprint.

Success Criteria

In this sprint I aim to achieve:

- Identify when the game is completed
- Validation for the paths
    - Paths can only be drawn by starting from a dot
    - Paths can only be drawn to the same colour dot
    - If paths cross, the path is that has been crossed gets removed entirely
- Previous errors/ bugs are fixed
- The buttons on the "Level" interface now have a function

Pseudocode

```
Class Grid
    self.rows = rows
    self.cols = cols
    self.cellSize = cellSize
    self.pathWidth = pathWidth
    self.grid = []
    self.paths = []

    Public procedure drawDots()
        self.grid = []
        self.grid = [[1,0,0,0,1]]
        self.grid.append([2,0,0,0,2])
        self.grid.append([3,0,0,0,3])
        self.grid.append([4,0,0,0,4])
        self.grid.append([5,0,0,0,5])

    Public procedure drawGrid()
        for row in range(self.rows)
            for col in range(self.cols)
                drawRectangle(col * cellSize, row * cellSize, cellSize, cellSize, gridColour)
                self.drawDots()
                if self.grid != 0
                circle_center = (col * cellSize + cellSize // 2, row * cellSize + cellSize // 2)
                    if self.grid[row][col] == 1
                        drawCircle(display, red, circle_center, circle_radius)
                    else if self.grid[row][col] == 2
                        drawCircle(display, blue, circle_center, circle_radius)
                    else if self.grid[row][col] == 3
                        drawCircle(display, green, circle_center, circle_radius)
                    else if self.grid[row][col] == 4
                        drawCircle(display, yellow, circle_center, circle_radius)
                    else if self.grid[row][col] == 5
                        drawCircle(display, orange, circle_center, circle_radius)
                    end if
                end if
            end for
        end for

    Public procedure getNumber()
        return self.grid[row][col]

    Public procedure getCellPos(pos):
        x,y = pos
        if row < 0 or col < 0 or row > 4 or col > 4:
            return None
        return (col, row)
```

```
Class level
    Public levelNum
    Public gridSize
    Public cellSize
    Public pathWidth
    Public pathColour
    Public gridColour
    Public bgColour
    Public bgImage = “bg.png”

    Public procedure main()
        levelSelection = self.levelSelection()
        levelSelection.displayLevels()

    Public procedure createButtons3(self)
        display "Undo"
        display "Hint"
        display "Reset"
        display "Quit"

    Public procedure createButtons4(self)
        display "Next Level"
        display "Level Selection"
        display "Quit"

    Public procedure resetLvl(self)
        self.easyLevelGame

    Public procedure undoMove(self)
        if self.undo == 1:
            removed = len(self.paths[-1])
            self.total -= removed
            self.paths.pop()
            self.undo -= 1
        else:
            print "no undo"

    Public procedure hint(self):
        print "hint"
```

```
Public procedure easyLevelGame(self)
    newGrid = Grid(newGrid.rows, newGrid.cols, newGame.cellSize, newGrid.pathWidth)
    self.drawing = False
    self.total = 0
    self.length = 0
    self.moves = 0
    self.undo = 1
    self.level = True
    self.paths = []
    self.createButtons3

    while newGame.running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                newGame.running = False

            elif event.type == pygame.MOUSEBUTTONDOWN:
                self.drawing = True
                mousePos = pygame.mouse.getCellPos()
                self.paths.append(mousePos)

            elif event.type == pygame.MOUSEBUTTONUP:
                self.drawing = False
                self.total += self.length
                self.moves += 1

            elif event.type == pygame.MOUSEMOTION and self.drawing
                mousePos = pygame.mouse.getCellPos()
                self.length = len(self.paths)

            for button in self.buttons:
                button.brawButton()

        if self.drawing:
            pathColour = 0
            if mousePos == None:
                self.drawing = False
            else:
                dotNum = newGrid.getNumber(mousePos[1], mousePos[2])
                if dotNum == 1:
                    pathColour = red
                elif dotNum == 2:
                    path colour = blue
                elif dotNum == 3:
                    path colour = green
                elif dotNum == 4:
                    path colour = yellow
                elif dotNum == 5:
                    path colour = orange

                startPos = ((self.paths[i][0] * newGrid.cellSize + newGrid.cellSize//2), self.paths[i][1] * newGrid.cellSize + newGrid.cellSize // 2)
```

```
                endPos = ((self.paths[i + 1][0] * newGrid.cellSize + newGrid.cellSize//2), self.paths[i + 1][1] * newGrid.cellSize + newGrid.cellSize // 2)

            if self.total == 25:
                newGame.running = False
                self.solved()

    Public procedure solved(self):
        self.createButtons4()
        while not newGame.levelSolved:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.quit()

                for button in self.buttons:
                    button.drawButton()
            self.total = 0
            self.moves = 0

    Public procedure mediumLevelGame()
        print "Medium level game started"

    Public procedure hardLevelGame()
        print "Hard level game started"



game = new Game()
game.main()
```

Sprint 3 Code

The following code is found in the "Game" class and the only change to this class was the addition of the attribute "self.moves", it is set to the value of -1 because upon pressing the "Easy- 5x5" button, the program will recognise that as a MOUSEBUTTONUP event in the within the "easyLevel" method, therefore the program increments the value by 1. This means that the

value will be accurate for the player when the game starts, and they can play the game with no drawbacks.

```
class Game:
    def __init__(self):
        self.title = "Connect The Dots"
        self.clock = pygame.time.Clock()
        self.running = True
        self.menu_started = False
        self.game_started = False
        self.level_solved = False
        self.moves = -1
        self.squareFilled = 0
        self.gridSize = 5
        self.cellSize = 100
        self.pathWidth = 20
        self.bgColour = (0, 0, 0)
        self.gridColour = (255, 255, 255)
        self.windowWidth = 1102.5
        self.windowHeight = 735
        #loads the background image and scales bg image to fit the window
        self.morph = pygame.image.load("background6.png")
        self.bgImage = pygame.transform.scale(self.morph, (self.windowWidth, self.windowHeight))

        self.display = pygame.display.set_mode((self.windowWidth,self.windowHeight))
        pygame.display.set_caption(self.title)

    def main(self): #creates menu class and displays menu
        self.menu = Menu()
        self.menu.displayMenu()

    def quit(self): #quits the game
        pygame.quit()
        quit()
```

The following code can be found in the "Menu" class, the first method in this image creates buttons and adds them to an array and will be displayed once the level has been solved.

The "giveHint" method does not yet have a set function, I plan to give this method a function in a future sprint as once random generation of dots is implemented, I hope use pathfinding to allow the game to find a solution for a set of dots to assist the player in competing the game.

The "resetLevel" method will reset any paths drawn by redrawing the entire display on top of the previous one, I chose to do it this way as it was the easiest method to implement to get the desired result, however in future a sprint where random generation is implemented, this method will likely be modified to make it more efficient and relevant to the adaptations made.

The "undoMove" method would've removed the most recent path drawn by the user however due to be limited by time, I was unable to fully implement an undo feature. The method starts off by checking if the attribute "self.undo" is 1, as the user should only be able to use undo once, if it is 1 it will then find the length of the path drawn and subtract it from the total, this is because when the user presses on the "undo" button the program recognised the event as a MOUSEBUTTONUP and would have added on the value of "self.length" to "self.total" making the number of squares filled inaccurate. The program also subtracts 1 from "newGame.moves" as the program also recognises a MOUSEBUTTONUP event and would therefore increment the value of "newGame.moves" which is not a desired feature. The program will then find the length of the second to last element in the array "self.paths", the second to last value is found as the last value in the array is always "None" so by finding the second to last element the program is given a numerical value to subtract from the total squares that have been filled by a path. After this the last 2 elements in "self.paths" are removed using the built in "pop()" function and "self.undo" is subtracted so that the value is now 0 so it cannot be used again for the current level. If the user attempts to use the undo mechanic again no changes to the game occur. Upon

pressing the "undo" button, the method will output several print statements to verify that the button is working correctly. The first statement is to show the array "self.paths" before the "pop" occurs, the second statement is to show the array "self.paths" after the "pop" occurs where there should be a change in the array. The third statement is used when the user does not have an "undo" and will output a statement to verify this.

Due to limitations of time I was unable to visually remove the most recent path that was drawn by the user, so even after pressing the "button" the path remains visible even though the value of "self.total" is adjusted correctly. If I had more time, I would have implemented this by finding the coordinates of the squares that the path was drawn in and reset the individual squares so that they would be empty to enable the user to draw in them again.

```
def createButtons4(self):
    self.buttons = []
    #Button the will generate a new level/ problem in a future sprint
    self.buttons.append(Button("Next Level", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.nextLvl))
    #Button that will take the user to the level selection screen
    self.buttons.append(Button("Level Selection", (newGame.windowWidth/2) - (self.bigButtonWidth/2) - 20, 400, 160, self.bigButtonHeight, bgBlue, inactiveBlue, self.levelSelection))
    #Button that will quit the program
    self.buttons.append(Button("Quit", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, newGame.quit))

def giveHint(self): #then this is 34566787889
    print("Hint")

def resetLevel(self):
    self.easyLevel()

def undoMove(self):
    if self.undo == 1:
        self.total -= self.length #Subtracts the length so that when "Undo" is pressed, the squares covered remains the same
        newGame.moves -= 1 #Subtracts 1 from the "self.moves" variable so that the value of it does not increment when not wanted
        print("Before", self.paths)

        pathRemoved = len(self.paths[-2]) #stores the length of the path that is 2nd to last in the list
        self.total -= pathRemoved
        self.paths.pop() #removes the last 2 elements in the list
        self.paths.pop()

        print("after", self.paths)
        self.undo -= 1
    else:
        self.total -= self.length
        newGame.moves -= 1
        print("No undo")
```

I altered the method "gameStats" from the previous sprint and combined both "gameStats" and "connectedPaths" so that they were under a single method.

After discussing with my stakeholders, they mentioned that they would rather a feature that shows how many squares were filled as a percentage compared to my initial design of showing how many paths were connected, therefore I made this change as well as implementing both "moves" and "squaresFilled" so that when the user draws paths on the grid, they can see these values change in real time. I did this by using the "blit" function in pygame to blit them onto the display on top of a rectangle separate to the background image of the game. I chose to do it this way as previously when running the program, the text would overlap on each other when the user would draw a path and you could not determine what the values were just by looking at them, so to combat this I added a box directly below the values so that after the user stops drawing, the values are updated and both the text and the box are drawn on top of the previous box and text making it much easier to determine what the values were.

```
def gameStats(self, moves): #displays the number of moves made by the player and the percentage of squares filled
    pygame.draw.rect(newGame.display, bgBlue, (90, 195, 165, 70))
    Font = pygame.font.Font("freesansbold.ttf", 25)
    movesSurf = Font.render("Moves: " + str(moves), True, black)
    pathsSurf = Font.render("Filled: "+ str(newGame.squareFilled) + "%"  , True, black)
    movesRect = movesSurf.get_rect()
    pathsRect = pathsSurf.get_rect()
    movesRect.topleft = (100, 200)
    pathsRect.topleft = (100, 230)
    newGame.display.blit(movesSurf, movesRect)
    newGame.display.blit(pathsSurf, pathsRect)
```

For the "easyLevel" method several changes were made compared to last sprint to achieve my success criteria. First "self.paths" was set as an array to store all paths that were drawn, I chose to do it this way as in an array it is easy to access items at specific indexes and it will store items in order they are received making I easier to access them.

Then, I implemented both "self.total" and "self.length" by setting them to 0, this is so that their values can be modified when the user draws a path, I chose to implement these values in the "easyLevel" method as they would be modified only in this method and are only used in the "menu" class.

Next the "self.undo" attribute was implemented holding the numerical value of 1 so that it can be used to determine whether the player still has access to the "undo move" mechanic, this value can change. I chose to implement "self.undo" in this method and class instead of the "Grid" class because the "easyLevel" method is responsible for drawing the paths and the majority of game logic.

The next attribute in this method is "self.level" which holds the Boolean value "True", this attribute is used for a loop. Initially I did not have this attribute in my program, however when creating the level solved screen, I encountered an error which would still overlay the grid even when the display had swapped, (more detail in the "Problems I encountered" section), therefore, I added the "self.level" loop with the function that will create the grid inside it but outside the "newGame.running" loop which solved the issue.

The next thing I added in this sprint was appending the current path to the array "self.paths" which was used to determine the length of paths in a previous method shown.

The next thing added was using "self.length" to create a total squares filled using the attribute "self.total" this is used to determine when the game is completed and used to find out the percentage of squares filled.

The next thing added was the incrementation of "self.moves" whenever the event MOUSEBUTTONUP was detected, I chose to implement it in this way as it was a simple way to determine when the user made a move and if "self.moves" increments when it is not supposed to, it can be easily fixed by subtracting from "self.moves" so that the correct value is shown.

The next thing I implemented was an equation that would determine the percentage of squares filled, found using "self.total" and multiplying it by 100, then dividing by the number of rows and columns using "DIV" so only a whole number is returned.

```
def easyLevel(self): #the main block of code for the game
    newGame.display.blit(newGame.bgImage,(0,0))
    self.drawTitle()
    self.createButtons3()
    newGrid = Grid(5, 5, 90, 30) #creates a grid object with 5 rows, 5 columns, cell size of 90, path width of 30
    self.drawing = False
    self.currentPath = []
    self.paths = []
    self.total = 0
    self.length = 0
    self.undo = 1
    self.level = True

    while self.level:
        newGrid.createGrid()
        while newGame.running: #main loop for the game
            self.gameStats(newGame.moves)
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.running = False

                elif event.type == pygame.MOUSEBUTTONDOWN: #checks if the mouse is clicked
                    self.drawing = True
                    mousePos = pygame.mouse.get_pos() #gets the mouse position and then converts it to a grid position
                    gridPos = newGrid.getCellFromMouse(mousePos)
                    self.currentPath.append(gridPos) #appends the grid position to the current path
                    self.paths.append(self.currentPath)

                elif event.type == pygame.MOUSEBUTTONUP: #checks if the mouse is released
                    self.drawing = False
                    self.currentPath = []
                    self.total += self.length
                    print(self.total)
                    newGame.moves += 1
                    newGame.squareFilled = ((self.total * 100) // (newGrid.rows * newGrid.cols)) #returns a precentage value on how many squares are filled

                elif event.type == pygame.MOUSEMOTION and self.drawing: #checks if the mouse is moving
                    mousePos = pygame.mouse.get_pos()
                    gridPos = newGrid.getCellFromMouse(mousePos)
                    if gridPos != self.currentPath[-1]: #if the grid position is not the same as the last grid position
                        self.currentPath.append(gridPos)
                    self.length = len(self.currentPath)

                for button in self.buttons:
                    button.drawButton()

            pygame.display.flip() #updates the display
```

In the previous sprint there was an error where if you draw a line outside of the grid, the game would error and stop working, so to fix this if implemented an if statement before the part of the program that would draw the path to say that if the variable "gridPos" holds the value of "None" the message "Invalid move" would be printed, this is because individual grid squares will hold a numerical value between 0 and 4 and anywhere else on the screen would not hold a value therefore would assume the value of None.

The final thing I implemented in the "easyLevel" method was using "self.total" to determine if the game has been solved, this was done by multiplying the number of rows and columns and comparing that value to the number of squares filled, and if it is equal to that value (which is 25) it then breaks out of both game loops and runs then calls the "Solved()" method where a new screen will be displayed. By setting the flags "newGame.running" and "newGame.menu.level" as False, the program stops running both loops as the condition they need to run is no longer met. I chose to implement it this way as I wanted the program to check to see if the game has been solved after each iteration at the end of the program instead of checking it at the start of the program.

```
        #This will block of code is responsible for drawing the path
        if self.drawing and self.currentPath: #will only draw the path if the player is drawing and there is a path to draw
            for i in range(len(self.currentPath)-1): #loops through the current path
                pathColour = 0 #sets the path colour to 0 initially
                if gridPos == None:
                    self.drawing = False
                    print("Invalid move")
                    newGame.menu.total -= self.length
                    newGame.moves -= 1
                else:
                    dotNum = newGrid.getDotColour(gridPos[1], gridPos[0])
                    if dotNum == 1:
                        pathColour = red
                    elif dotNum == 2:
                        pathColour = blue
                    elif dotNum == 3:
                        pathColour = green
                    elif dotNum == 4:
                        pathColour = yellow
                    elif dotNum == 5:
                        pathColour = orange

                    startPos = ((self.currentPath[i][0] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.xOffset, (self.currentPath[i][1] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.yOffset)
                    #stores the start position of line
                    endPos = ((self.currentPath[i+1][0] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.xOffset, (self.currentPath[i+1][1] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.yOffset)
                    #stores the end position of
                    pygame.draw.line(newGame.display, pathColour, startPos, endPos, newGrid.pathWidth) #draws the line

        if self.total == (newGrid.rows * newGrid.cols): #if the total is equal to the number of squares in the grid
            newGame.running = False
            newGame.menu.level = False

            self.solved()
```

This method creates an array for buttons to be stored in, and then creates 3 different buttons and appends them into the array to be stored to be called on later in the program. I chose to implement the buttons in this way as it is easier to keep track of what buttons are used in which screen/ part of the game as buttons are created in different arrays and draw using a for loop when needed.

```
    def createButtons4(self):
        self.buttons = []
        #Button the will generate a new level/ problem in a future sprint
        self.buttons.append(Button("Next Level", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.nextLvl))
        #Button that will take the user to the level selection screen
        self.buttons.append(Button("Level Selection", (newGame.windowWidth/2) - (self.bigButtonWidth/2) - 20, 400, 160, self.bigButtonHeight, bgBlue, inactiveBlue, self.levelSelection))
        #Button that will quit the program
        self.buttons.append(Button("Quit", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, newGame.quit))
```

This method is found within the "Menu" class, this method is called when the total number of squares filled is equal to the number of rows multiplied by the number of columns. The methods function is to display text confirming that the level was solve as well as to display a screen with 3 buttons where the user will be able to navigate the game with the use of the buttons from the "self.createButtons4()" method. The method then runs a loop while a condition is met, in this case while the variable "newGame.level_solved" is not true. In this loop, it will change the Boolean value of several flags so that if the user chooses to navigate to a previous menu, they are able to do so and are shown the correct display. I chose to implement the flags in this way as it was simple to implement and understand what was happening when looking at the code.

```
def solved(self): #displays the level solved screen
    newGame.display.blit(newGame.bgImage, (0,0))
    self.drawTitle()
    self.createButtons4()
    font = pygame.font.Font(None, 52)
    text = font.render("Level Solved!", True, white)
    newGame.display.blit(text, (450, 200))
    pygame.display.update()
    while not newGame.level_solved:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                newGame.quit()
                pygame.quit()

            newGame.game_started = False #sets this flag as False so the player can go back to previous screens without error
            newGame.running = True #sets these two flags as True so the player can play again
            newGame.menu.level = True

            for button in self.buttons:
                button.drawButton()

        pygame.display.update()
```

This method does not have a set function yet other that printing the message "next level" when called, this is because in this sprint I have yet to implement random dot generation as dots are generated in the same position each time the program is run. If I have time at the end of the project, I plan to make this method generate a new random puzzle when called so the user can continue to solve puzzles without having to exit the program and re-run it or to potentially play the same puzzle each time.

```
def nextLvl(self):
    print("next level")
```

Changes to Pseudocode

Level Class:

As I chose to not to include a level class in my previous sprint, I decided to move several methods into the "Menu" class instead as at this stage I am still yet to implement multiple level difficulties.

Undo Mechanic

In my pseudocode I initially had the "undo" method find the length of the last data at the last index rather that the second to last index. I changed this while testing the mechanism as the array would always end in the data type "None", so a length was not found causing an error.

Number of Moves

In my pseudocode for the "easyLevelGame" I had set the value of "self.moves" as 0, as it is logical to think that upon immediately starting the game the player would not have made any

moves, however while testing it came to my attention that the number of moves would increment to one upon pressing the "Easy- 5x5" button, therefore I changed the value of this variable to "-1" in the actual program.

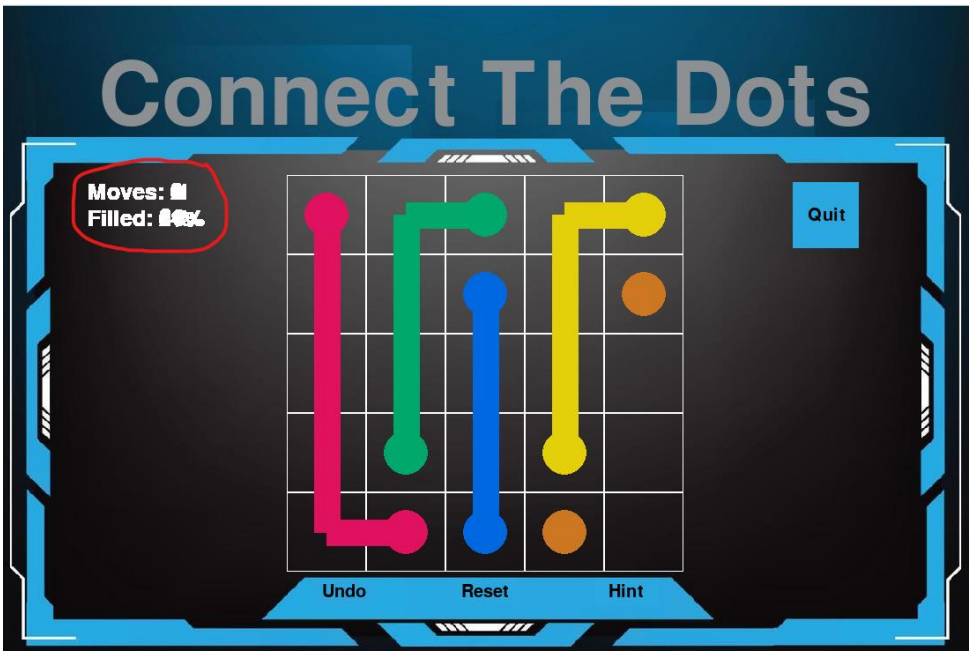
Problems I Encountered While Coding

The first problem I encountered was in the "gameStats" method, the text would overlap when the player would draw a path, rather than the numbers incrementing. This was due to the program re-drawing the text in the same position every time the required event is detected, without removing the previous text, which caused it to overlap making it difficult to determine what the values were.

Error code:

```
def gameStats(self, moves): #displays the number of moves made by the player (value will
    Font = pygame.font.Font("freesansbold.ttf", 25)
    movesSurf = Font.render("Moves: " + str(moves), True, white)
    pathsSurf = Font.render("Filled: "+ str(newGame.squareFilled) + "%"  , True, white)
    movesRect = movesSurf.get_rect()
    pathsRect = pathsSurf.get_rect()
    movesRect.topleft = (100, 200)
    pathsRect.topleft = (100, 230)
    newGame.display.blit(movesSurf, movesRect)
    newGame.display.blit(pathsSurf, pathsRect)
```

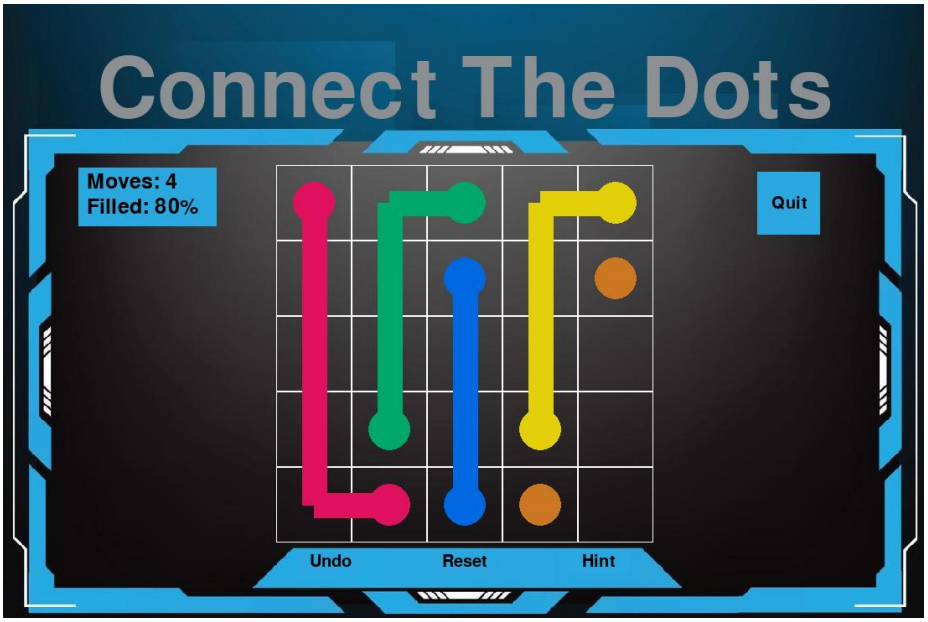
Error code run:



Fixed code:

```
def gameStats(self, moves): #displays the number of moves made by the player (value will
    pygame.draw.rect(newGame.display, bgBlue, (90, 195, 165, 70))
    Font = pygame.font.Font("freesansbold.ttf", 25)
    movesSurf = Font.render("Moves: " + str(moves), True, black)
    pathsSurf = Font.render("Filled: "+ str(newGame.squareFilled) + "%"  , True, black)
    movesRect = movesSurf.get_rect()
    pathsRect = pathsSurf.get_rect()
    movesRect.topleft = (100, 200)
    pathsRect.topleft = (100, 230)
    newGame.display.blit(movesSurf, movesRect)
    newGame.display.blit(pathsSurf, pathsRect)
```

Fixed code run:

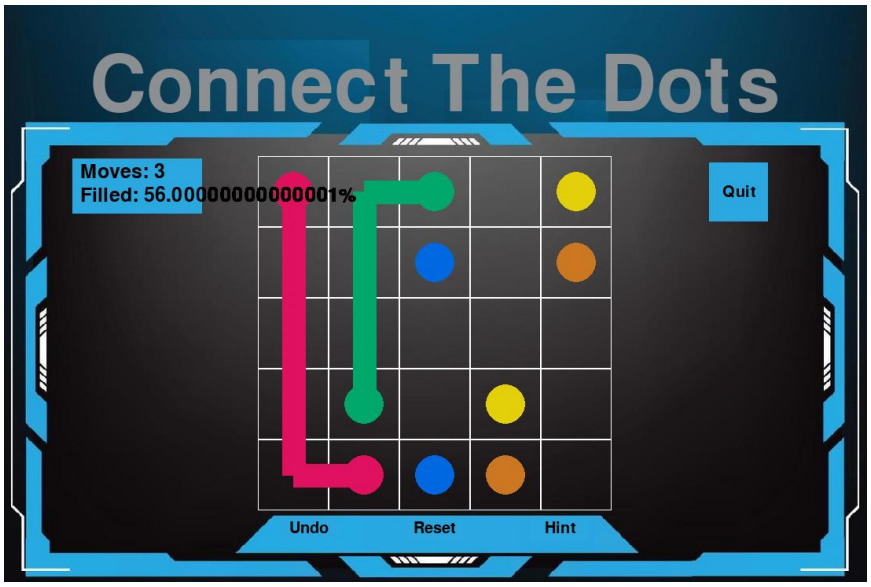


Another problem I encountered was when calculating the percentage of squares filled, the program would occasionally display a long decimal number which looked out of place (Error 1). This would occur when the player would undo a move and then draw a path after. To fix this I decided to use the "DIV" equivalent in python which is "//" however due to the number of squares filled being less than 25 for the majority of game, it would return 0 which would not give accurate information for the user (Error 2). To solve this problem, I multiplied the number of squares filled by 100 to give a larger number to divide by which solved the issue.

Error 1 code:

```
newGame.squareFilled = ((self.total) / (newGrid.rows * newGrid.cols)) * 100
```

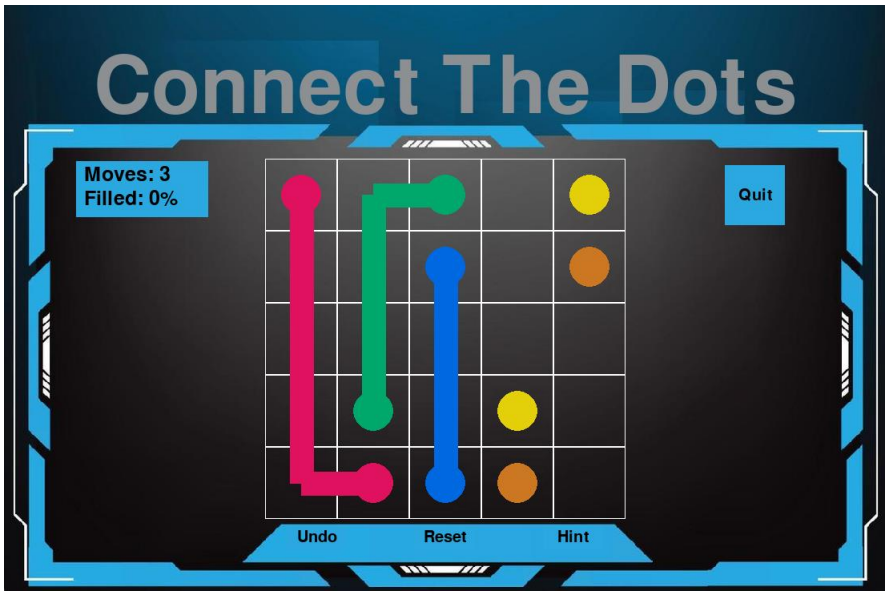
Error 1 run:



Error 2 code:

```
newGame.squareFilled = ((self.total) // (newGrid.rows * newGrid.cols)) * 100
```

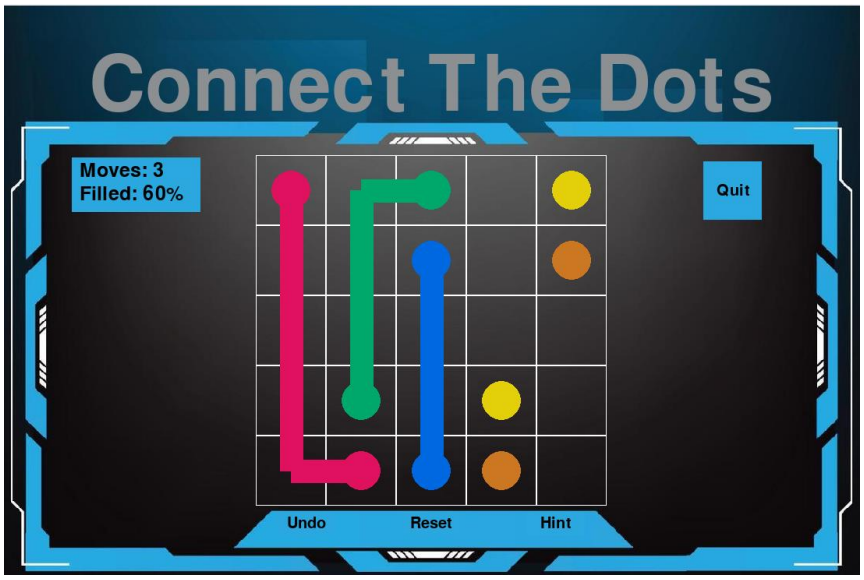
Error 2 run:



Fixed code:

```
newGame.squareFilled = ((self.total * 100) // (newGrid.rows * newGrid.cols))
```

Fixed code run:

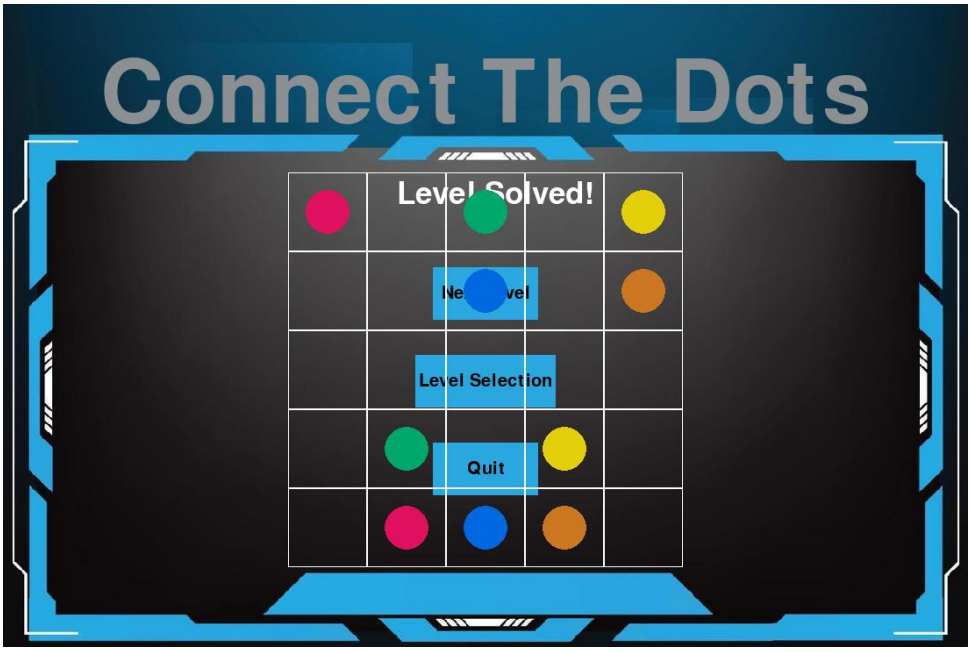


The next problem I encountered was that upon solving the game, the grid lines would not disappear causing an overlap of both displays. This was because initially the program wouldn't break out of the "newGame.running" loop meaning that the program would run both functions simultaneously causing them to display on top of one another. To solved this problem I created a new loop called "self.level" inside the "easyLevel()" method and a loop inside the "solved()" method so that the methods could be easily separated to ensure that they don't overlap.

Error code:

```
        while newGame.running:
            for event in pygame.event.get():
                if event.type == pygame.QUIT: ···

                elif event.type == pygame.MOUSEBUTTONDOWN: #checks if th

                elif event.type == pygame.MOUSEBUTTONUP: #checks if the

                elif event.type == pygame.MOUSEMOTION and self.drawing:

                for button in self.buttons: #draws the buttons ···

            #while self.level:
            pygame.display.flip() #updates the display
            newGrid.createGrid()

            if self.drawing and self.currentPath: #will only draw the pa

            if self.total == 25:
                self.solved()


    def solved(self): #displays the level solved screen
            newGame.display.blit(newGame.bgImage, (0,0))
            self.createButtons4()
            self.drawTitle()
            font = pygame.font.Font(None, 52)
            text = font.render("Level Solved!", True, white)
            newGame.display.blit(text, (450, 200))
            pygame.display.update()
            for button in self.buttons:
                button.drawButton()

            self.total = 0
            newGame.moves = -1
```

Error code run:



Fixed code:

```
        while self.level:
            newGrid.createGrid()
            #self.gameStats(newGame.moves) #THIS DOES NOT INCREMENT THE MOVES EXPLAINNN
            pygame.display.update()
            while newGame.running: #main loop for the game
                self.gameStats(newGame.moves) #THIS DOES NOT REMOVE THE BOX WITH THIS E
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:···

                    elif event.type == pygame.MOUSEBUTTONDOWN: #checks if the mouse is

                    elif event.type == pygame.MOUSEBUTTONUP: #checks if the mouse is re

                    elif event.type == pygame.MOUSEMOTION and self.drawing: #checks if

                    for button in self.buttons:
                        button.drawButton()

                pygame.display.flip() #updates the display

                #This will block of code is responsible for drawing the path
                if self.drawing and self.currentPath: #will only draw the path if the p

                if self.total == (newGrid.rows * newGrid.cols): #if the total is equal
                    newGame.running = False
                    newGame.menu.level = False

                    self.solved()
```

```
    def solved(self): #displays the level solved screen
        newGame.display.blit(newGame.bgImage, (0,0))
        self.drawTitle()
        self.createButtons4()
        font = pygame.font.Font(None, 52)
        text = font.render("Level Solved!", True, white)
        newGame.display.blit(text, (450, 200))
        pygame.display.update()
        while not newGame.level_solved:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.quit()
                    pygame.quit()

                newGame.game_started = False #set these 3 flags as true so t
                newGame.running = True
                newGame.menu.level = True

                for button in self.buttons:
                    button.drawButton()

            pygame.display.update()

            self.total = 0
            newGame.moves = -1
```

Fixed code run:

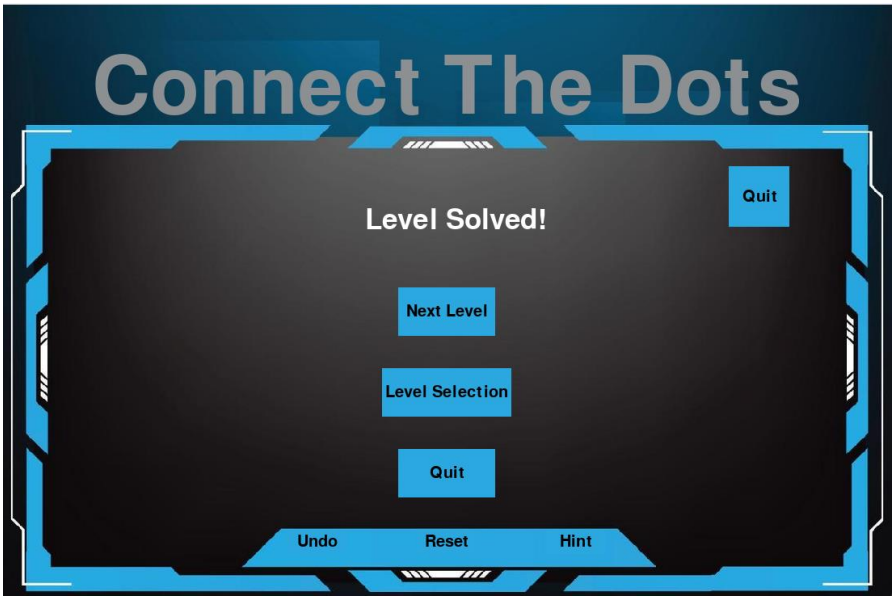
The screen is now shown as it should without the grid in the overlapped with it.

For this problem it occurred at the same time as the error previously explained. The error was that the buttons from the game screen would not disappear after solving the game when the screen would change. This was occurring as the flag for one of the while loops would not break out of said loop so that the grid would not be displayed in the next screen, but the other loop was still running which would display the buttons.

Error code:

```
    while newGame.running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT: ···

            elif event.type == pygame.MOUSEBUTTONDOWN: #checks if th

            elif event.type == pygame.MOUSEBUTTONUP: #checks if the

            elif event.type == pygame.MOUSEMOTION and self.drawing:

            for button in self.buttons: #draws the buttons ···

        #while self.level:
        pygame.display.flip() #updates the display
        newGrid.createGrid()

        if self.drawing and self.currentPath: #will only draw the pa

        if self.total == 25:
            self.solved()


def solved(self): #displays the level solved screen
        newGame.display.blit(newGame.bgImage, (0,0))
        self.createButtons4()
        self.drawTitle()
        font = pygame.font.Font(None, 52)
        text = font.render("Level Solved!", True, white)
        newGame.display.blit(text, (450, 200))
        pygame.display.update()
        for button in self.buttons:
            button.drawButton()

        self.total = 0
        newGame.moves = -1
```

Error code run:

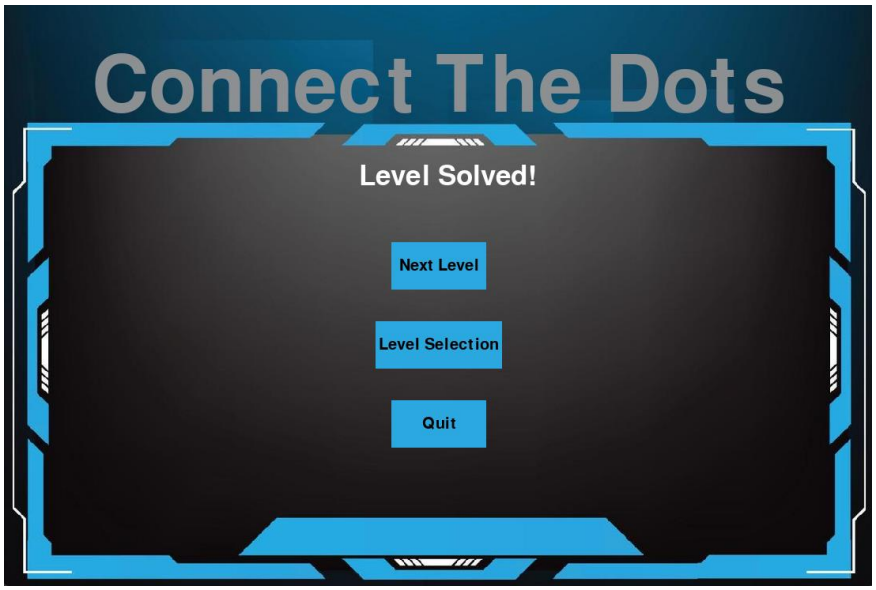


Fixed code:

```
while self.level:
    newGrid.createGrid()
    #self.gameStats(newGame.moves) #THIS DOES NOT INCREMENT THE MOVES EXPLAINNN
    pygame.display.update()
    while newGame.running: #main loop for the game
        self.gameStats(newGame.moves) #THIS DOES NOT REMOVE THE BOX WITH THIS E
        for event in pygame.event.get():
            if event.type == pygame.QUIT: ···

            elif event.type == pygame.MOUSEBUTTONDOWN: #checks if the mouse is

            elif event.type == pygame.MOUSEBUTTONUP: #checks if the mouse is re

            elif event.type == pygame.MOUSEMOTION and self.drawing: #checks if

            for button in self.buttons:
                button.drawButton()

        pygame.display.flip() #updates the display

        #This will block of code is responsible for drawing the path
        if self.drawing and self.currentPath: #will only draw the path if the p

        if self.total == (newGrid.rows * newGrid.cols): #if the total is equal
            newGame.running = False
            newGame.menu.level = False

            self.solved()
```

```
def solved(self): #displays the level solved screen
    newGame.display.blit(newGame.bgImage, (0,0))
    self.drawTitle()
    self.createButtons4()
    font = pygame.font.Font(None, 52)
    text = font.render("Level Solved!", True, white)
    newGame.display.blit(text, (450, 200))
    pygame.display.update()
    while not newGame.level_solved:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                newGame.quit()
                pygame.quit()

            newGame.game_started = False #set these 3 flags as true so t
            newGame.running = True
            newGame.menu.level = True

            for button in self.buttons:
                button.drawButton()

        pygame.display.update()

        self.total = 0
        newGame.moves = -1
```

Fixed code run:



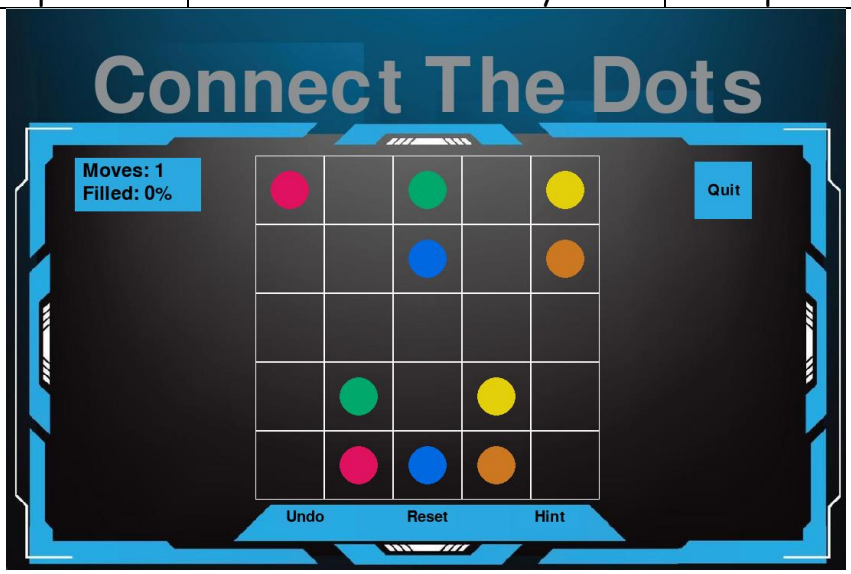
Test Plan

Action	Reason	Expected Outcome
Presses a dot on the puzzle	Checks to see if anything unexpected happens	Moves will increment by 1
Connects a path to corresponding dot	Checks "easyLevel" function works correctly, this being the part that identifies when a path is being drawn	The line will be the same colour as the dot drawn from, then moves and squares filled values are adjusted
Users attempts to draw a path outside the grid	Checks to see if the program still errors or if it has been fixed	"Invalid move" is output in terminal
Press "Hint"	Checks "giveHint" function works correctly	"Hint" is output in terminal
Press "Reset" button	Checks "resetLevel" function works correctly	All paths are removed
Press "Undo" button	Checks "undoMove" function works correctly	Correct messages are outputted depending on if the player has an "undo" to use or not
User fills all squares with a path so that "squaresFilled" is equal to 25	To check if "solved" function works correctly	A new window is displayed saying "Level solved" with 3 buttons
User presses "Next Level"	Checks to see if "nextLvl" function works correctly	"Next level" is output in terminal
User presses "Level Selection"	Checks to see if the user can successfully navigate back to a previous menu screen	User is taken back to the level selection screen so they can choose a different difficulty setting
User presses on the screen on the win screen	Check to see if anything unexpected happens	Nothing

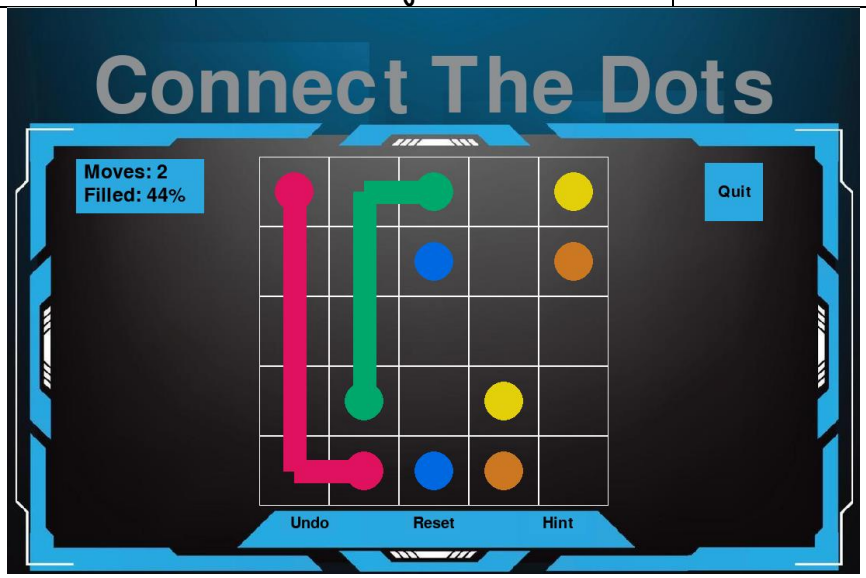
Formal Testing

Normal Data

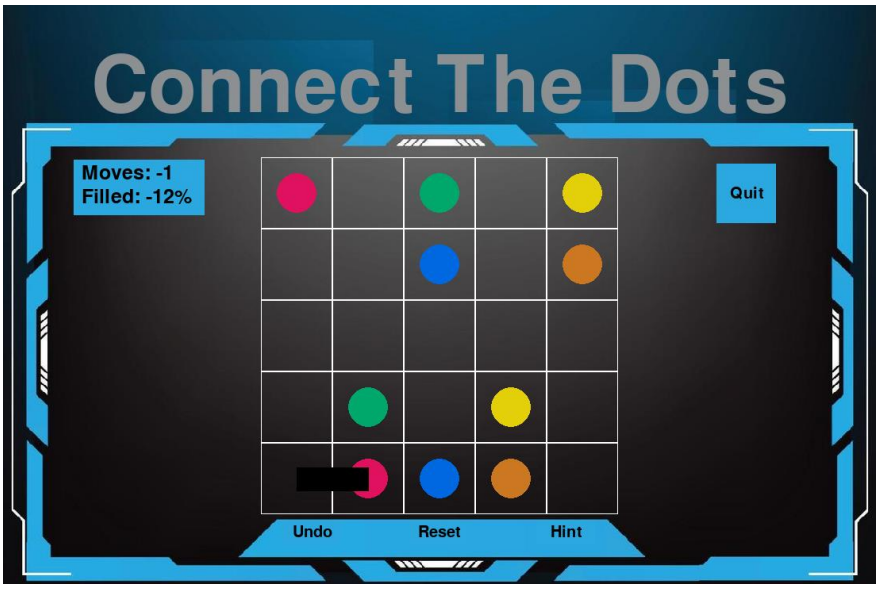
Action	Expected Outcome	Actual Outcome
Presses a dot on the puzzle	Moves will increment by 1	As expected



Connects a path to corresponding dot	The line will be the same colour as the dot drawn from, then moves and squares filled values are adjusted	As expected



Users attempts to draw a path outside the grid	"Invalid move" is outputted in python terminal	As expected



(Drawn from bottom left red dot directly left outside the grid line)

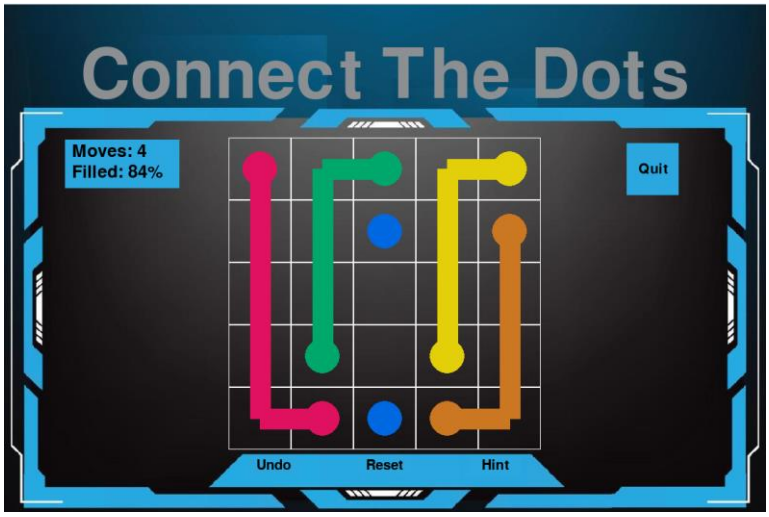
```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Invalid move
Invalid move
```

User presses "Hint"	"Hint" is output in python terminal	As expected

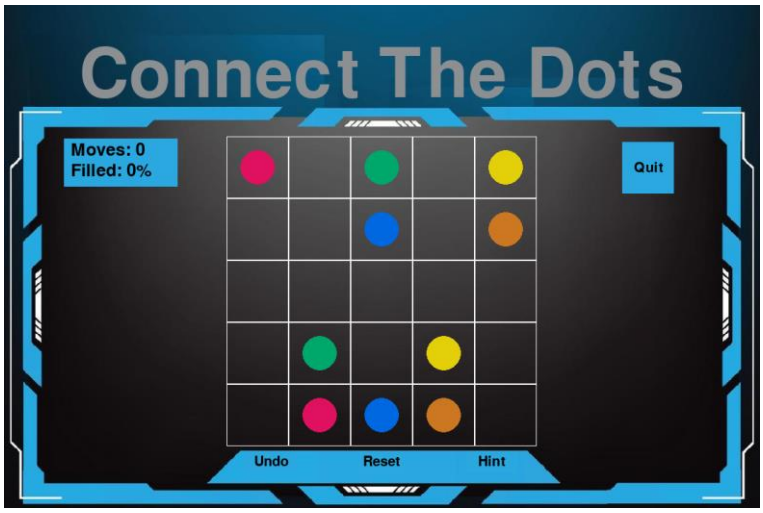
```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Hint
```

Press "Reset level"	All paths are removed	As expected

Before:



After:



Press "Undo" button	Correct messages are output in terminal depending on if the player has an "undo" to use or not	As expected

If the player has "undo":

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Before [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)], [(1, 3), (1, 2), (1, 1), (1, 0), (2, 0)], [None]]
after [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)]]
```

If the player does not have "undo":

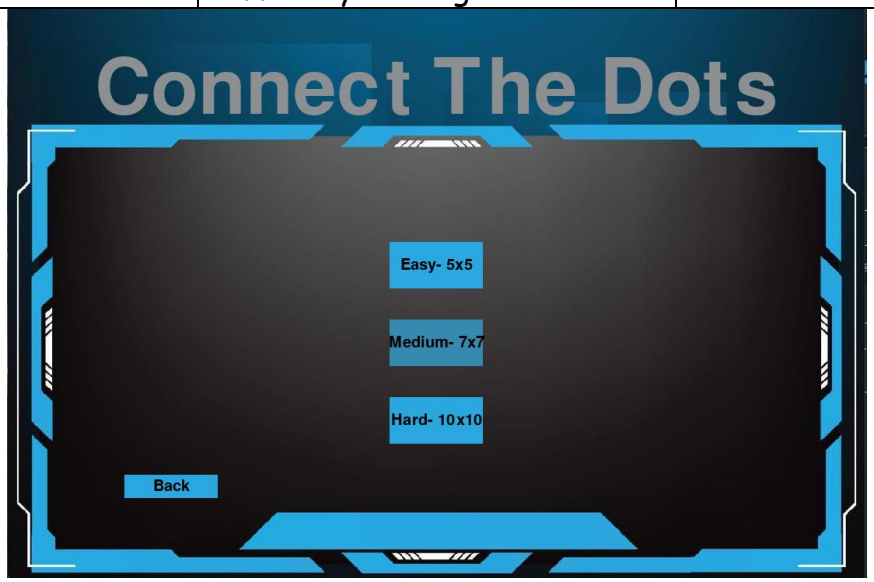
```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Before [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)], [(1, 3), (1, 2), (1, 1), (1, 0), (2, 0)], [None]]
after [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)]]
No hint
```

User fills all squares with a path so that "squaresFilled" is equal to 25	A new window is displayed saying "Level solved" with 3 buttons	As expected

User presses "Next Level"	"Next level" is output in terminal	As expected

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
next level
```

User presses "Level Selection"	User is taken back to the level selection screen so they can choose a different difficulty setting	As expected

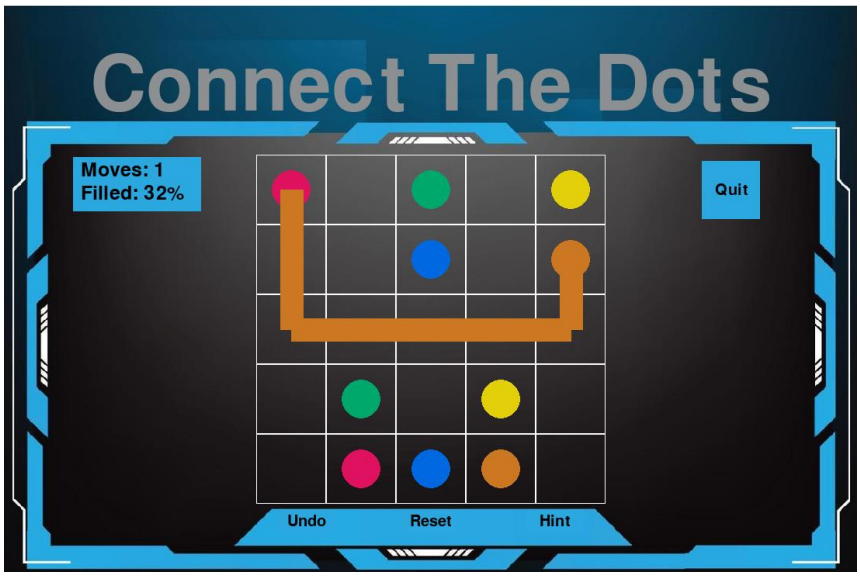


User presses on the screen on the win screen	Nothing	As expected

Boundary Data

Action	Expected Outcome	Actual Outcome
Joining a path to a different colour dot	Path will turn the colour of the dot it is connected to	As expected

In this case I connected the red dot to an orange one.



Erroneous Data

There was no erroneous data for this sprint.

Evaluation

Success Criteria	Completed?
Identify when the game is completed	Yes
Validation for the paths	No
Previous errors/ bugs are fixed	Yes
The buttons on the "Level" interface now have a function	Partially

In this sprint I was not able to complete everything on my success criteria. Due to the limitations of time, I focused on the main parts of my success criteria so the game can be solved, this included identifying when the game was solved, which I successfully implemented.

I was unable to implement path validation due to the limitation of time, if I was to implement this in a future sprint, first I would make it so that paths can only be drawn from dots, this would be done by checking the "dotNum" of a paths start position, if it is not equal to 0 a dot will be held at that location so the program will draw a path, but if it is equal to 0, a path should not be drawn as there is not a dot at that location. Then I would make it so that paths can only be connected to the same colour dots, I would do this make checking the "dotNum" of a paths end position and then compare it to the "dotNum" of a paths end position, if they are the same the program should continue to draw, however if they are different the program should set the variable "drawing" to false so they do not connect. And then finally I would check to see if paths overlap, and if they do, to remove both paths completely, I would do this by checking the coordinates of each path, and if have the same coordinates, both paths should be removed so the player can redraw them.

I also managed to fix the errors that were occurring in the previous sprint meaning the player can have a better experience when playing the game.

I was partially able to complete implementing functions for buttons in "Level" interface, this was because I successfully got the "Reset" level mechanic working and got half of the "Undo" mechanic working, however I was unable to implement a "hint" mechanic and hope to be able to implement it in a future sprint.

Next Sprint

In my next sprint I aim to achieve:

- Random generation of dots, but also so that they can be solved despite being in random places on the grid.
- And if there is time:
    - Implement path validation
    - Get both hint and undo mechanics working

Sprint 4

Aim of this Sprint

For this sprint I hope to implement random generation of dots so the user can solve a variety of levels to make the game more engaging. Due to a limitation of time, this will be the main focus of this sprint as it is an important feature to get working and takes priority over the others in my success criteria.

Success Criteria

- Random generation of dots, but also so that they can be solved despite being in random places on the grid.
- And if there is time:
    - Implement path validation
    - Get both hint and undo mechanics working

Pseudocode

```
    Public procedure easyLevelGame(self)
        newGrid = Grid(newGrid.rows, newGrid.cols, newGame.cellSize, newGrid.pathWidth)
        self.drawing = False
        self.total = 0
        self.length = 0
        self.moves = 0
        self.undo = 1
        self.level = True
        self.paths = []
        self.createButtons3

        while newGame.running:
            if newGrid.solvePuzzle(newGrid.puzzle) is True:
                print "solved"
            else:
                print "not solved"
```

```
Class Grid
    self.rows = rows
    self.cols = cols
    self.cellSize = cellSize
    self.pathWidth = pathWidth
    self.grid = []
    self.paths = []

    self.startNodes = {}
    self.endNodes = {}
    self.allNodes = []
    self.distances = {}
    self.sortedNodes = []
    self.pathFindPuzzle = []

    Public procedure drawDots()
        self.grid = [[0,0,0,0,0]]
        self.grid.append([0,0,0,0,0])
        self.grid.append([0,0,0,0,0])
        self.grid.append([0,0,0,0,0])
        self.grid.append([0,0,0,0,0])

        for i in range(0,5):
            x1 = random.randint(0,4)
            y1 = random.randint(0,4)
            x2 = random.randint(0,4)
            y2 = random.randint(0,4)

            self.grid[x1][y1] = i
            self.grid[x2][y2] = i
```

```
    Public procedure solvePuzzle(grid):
        if checkGrid(grid) is False:
            Return False

        if returnSolved(grid) is True:
            Return True

        for dotNum in sortedNodes
            start = startNodes[dotNum]
            end = endNodes[dotNum]

            if distance between start and end > 1:
                directions = EMPTY LIST

                if value to the RIGHT of start = 0:
                    if end is to the RIGHT of start:
                        INSERT "right" at the beginning of directions
                    else:
                        ADD "right" to the end of directions

                if value to the LEFT of start = 0:
                    if end is to the LEFT of start:
                        INSERT "left" at the beginning of directions
                    else:
                        ADD "left" to the end of directions

                if value BELOW start = 0:
                    if end is BELOW start:
                        INSERT "down" at the beginning of directions
                    else:
                        ADD "down" to the end of directions

                if value ABOVE start = 0:
                    if end is ABOVE start:
                        INSERT "up" at the beginning of directions
                    else:
                        ADD "up" to the end of directions

                if directions is EMPTY:
                    Return False
```

```
                for direction in directions:
                    if direction = "right":
                        MOVE start RIGHT
                        grid[start] = dotNum
                        if solvePuzzle(grid) is True:
                            Return True
                        else:
                            RESET start and grid

                    else if direction = "left":
                        MOVE start LEFT
                        grid[start] = dotNum
                        if solvePuzzle(grid) is True:
                            Return True
                        else:
                            RESET start and grid

                    else if direction = "up":
                        MOVE start UP
                        grid[start] = dotNum
                        if solvePuzzle(grid) is True:
                            Return True
                        else:
                            RESET start and grid

                    else if direction = "down":
                        MOVE start DOWN
                        grid[start] = dotNum
                        if solvePuzzle(grid) is True:
                            Return True
                        else:
                            RESET start and grid
            Return False
```

```
    Public procedure checkGrid(grid)
        for row in range(self.rows)
            for col in range(self.cols)
                if grid[row][col] > 0:
                    dotNum = grid[row][col]

                    if value BELOW > 0 AND value BELOW != dotNum:
                        if value ABOVE > 0 AND value ABOVE != dotNum:
                            if value RIGHT > 0 AND value RIGHT != dotNum:
                                if value LEFT > 0 AND value LEFT != dotNum:
                                    Return False

                    if value BELOW = dotNum AND value ABOVE = dotNum AND value RIGHT = dotNum:
                        Return False

                    else if value BELOW = dotNum AND value ABOVE = dotNum AND value LEFT = dotNum:
                        Return False

                    else if value BELOW = dotNum AND value RIGHT = dotNum AND value LEFT = dotNum:
                        Return False

                    else if value ABOVE = dotNum AND value RIGHT = dotNum AND value LEFT = dotNum:
                        Return False

        Return True
```

```
    Public procedure returnSolved(total):
        if total == 25:
            return True
        else:
            return False

    Public procedure abs(a , b):
        if a > b:
            return a - b
        else:
            return b - a

    Public procedure drawGrid()
        for row in range(self.rows)
            for col in range(self.cols)

                if puzzle[row][col] > 0:
                    if puzzle[row][col] IS IN startNodes:
                        endNodes[puzzle[row][col]] = (row, col)
                        distances[puzzle[row][col]] = abs(row - startNodes[puzzle[row][col]][0]) + abs(col -
startNodes[puzzle[row][col]][1])
                    else:
                        startNodes[puzzle[row][col]] = (row, col)

                    ADD (row, col) TO allNodes

                drawRectangle(col * cellSize, row * cellSize, cellSize, cellSize, gridColour)
                self.drawDots()
                if self.grid != 0
                circle_center = (col * cellSize + cellSize // 2, row * cellSize + cellSize // 2)
                    if self.grid[row][col] == 1
                        drawCircle(display, red, circle_center, circle_radius)
                    else if self.grid[row][col] == 2
                        drawCircle(display, blue, circle_center, circle_radius)
                    else if self.grid[row][col] == 3
                        drawCircle(display, green, circle_center, circle_radius)
                    else if self.grid[row][col] == 4
                        drawCircle(display, yellow, circle_center, circle_radius)
                    else if self.grid[row][col] == 5
                        drawCircle(display, orange, circle_center, circle_radius)
                    end if
                end if
            end for
        end for
```

Sprint 4 Code

The following code is from the "Menu" class and is called in the level end screen menu, it will cause a new grid to be generated so the player can solve a different puzzle.

```
    def nextLvl(self):
        self.easyLevel()
```

The following code is from the "easyLevel" method and the only change to this method was that the program would run the "solvePuzzle" method from the "Grid" class to check to see if the puzzle stored in the variable "newGrid.pathFindPuzzle" is solvable or not, if the puzzle is solvable the Boolean value "True" will be returned causing "Solvable" to be printed in the terminal, otherwise "Not solved" will be printed.

```
def easyLevel(self): #the main block of code for the game
    newGame.display.blit(newGame.bgImage,(0,0))
    self.drawTitle()
    self.createButtons3()
    newGrid = Grid(5, 5, 90, 30) #creates a grid object with 5 rows, 5 columns, cell size of 90, path width of 30
    self.drawing = False
    self.currentPath = []
    self.paths = []
    self.total = 0
    self.length = 0
    newGame.moves = -1
    self.undo = 1
    self.level = True


    while self.level:
        newGrid.createGrid()

        if newGrid.solvePuzzle(newGrid.pathFindPuzzle): #runs the solvePuzzle function on the pathFindPuzzle and checks if it is solvable
            print("Solved")
        else:
            print("Not solved")
```

The following code is found in the "Grid" class, I created several arrays and dictionaries to help the program with store the node locations, so that a backtracking algorithm can be used to navigate between them. I chose to implement "startNodes", "endNodes" and "distances" as dictionaries as it allows for a value to be specifically assigned to another value, e.g., the node number 2 having the start position of (2,2). I chose to implement "allNodes" and "sortedNodes" as an array as it I simply need to store a list of all generated nodes and nothing else. As seen below.

```
class Grid:
    def __init__(self, rows, cols, cellSize, pathWidth):
        self.rows = rows
        self.cols = cols
        self.cellSize = cellSize
        self.pathWidth = pathWidth
        self.xOffset = (newGame.windowWidth - self.cols * self.cellSize) // 2
        self.yOffset = (newGame.windowHeight - self.rows * self.cellSize) // 2 + 50 # +50 to offset, to account for the change in grid position

        self.puzzle = []
        self.startNodes = {} #creates a dictionary for the start nodes
        self.endNodes = {} #creates a dictionary for the end nodes
        self.allNodes = [] #creates an array for all the nodes
        self.distances = {} #creates a dictionary for the distances
        self.sortedNodes = [] #creates an array for the sorted nodes
        self.pathFindPuzzle = [] #creates a path finding puzzle
```

```
pygame 2.6.0 (SDL 2.28.4, Python 3.12.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
sorted nodes []
start nodes {6: (0, 1), 5: (0, 4), 4: (1, 4), 3: (2, 1), 2: (2, 2)}
end nodes {6: (2, 0), 3: (3, 0), 2: (3, 1), 4: (3, 4), 5: (4, 2)}
all nodes [(0, 1), (0, 4), (1, 4), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 4), (4, 2)]
distances {6: 3, 3: 2, 2: 2, 4: 2, 5: 6}
```

The following code is also found in the "Grid" class and is responsible for generating coordinates for the dots to be drawn in, it does this by randomly choosing a coordinate from an array from all possible coordinates from the 5x5 square grid and then removing that value from the array, so it isn't chosen again. I chose to implement it this way as ensures that dots won't overlap/ be drawn in the same position as the coordinates are removed once they have been chosen once. The program will then use the coordinates to then insert a number between 2-6, which is generated from the "for" loop, compared to having it as 1-5 in previous sprints, I made this change as the puzzle will get surrounded by 1's acting as walls, so by changing the "i" value it ensures that any indexes labelled as "1" won't generate a dot when it is not supposed to.

The method then will create a "pathFindPuzzle" using the puzzle generated previously which essentially surrounds the puzzle in 1's so that it walls are created to limit where the program will search to join nodes together. I chose to do it this way as it means the program has a limited/ enclosed area to search rather than searching in an area where there are no limits.

```
def makePuzzle(self):
    self.puzzle = [[0,0,0,0,0]] #creates a puzzle with 5 rows and 5 columns that gets random values assigned to it
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])

    #generates random coordinates for the dots to be draw in from the list of coordinates
    coords = [(0,0), (0,1), (0,2), (0,3), (0,4), (1,0), (1,1), (1,2), (1,3), (1,4), (2,0), (2,1), (2,2), (2,3), (2,4), (3,0), (3,1), (3,2), (3,3), (3,4), (4,0), (4,1), (4,2), (4,3), (4,4)]
    for i in range(2,7):
        coord1 = random.choice(coords) #chooses a random coordinate from "coords" then removes it from the array
        x,y = coord1 # x and y are set to the values of the coordinate
        coords.remove(coord1)

        coord2 = random.choice(coords)
        w,z = coord2
        coords.remove(coord2)

        self.puzzle[y][x] = i #changes the value held at the y,x postion to a sigular value so that the dots can be drawn
        self.puzzle[z][w] = i

    self.pathFindPuzzle = [[1,1,1,1,1,1,1]] #creates a path finding puzzle by adding 1's to the puzzle so that the there is a "barrier" around the grid
    self.pathFindPuzzle.append([1, self.puzzle[0], 1])
    self.pathFindPuzzle.append([1, self.puzzle[1], 1])
    self.pathFindPuzzle.append([1, self.puzzle[2], 1])
    self.pathFindPuzzle.append([1, self.puzzle[3], 1])
    self.pathFindPuzzle.append([1, self.puzzle[4], 1])
    self.pathFindPuzzle.append([1,1,1,1,1,1,1])
```

This method is found in the "Grid" class and will return the absolute value of the difference between 2 numbers where a positive value is returned.

```
def abs(self, a , b): #returns the absolute value of the difference between two numbers
    if a > b:
        return a - b
    else:
        return b - a
```

The next method is responsible for drawing the grid and dots on the screen. I added a block of code that is designed to sort the keys of the "self.distances" dictionary based on their values in ascending order. It repeatedly finds the minimum value, appends the corresponding key to "self.sortedNodes", and removes the key from "self.distances" until the dictionary is empty. I

chose to implement it this way as it ensures that "self.sortedNodes" contains the keys sorted by their associated values.

The next block of code is responsible for identifying and categorizing nodes in the puzzle, calculating distances between start and end nodes, and maintaining a list of all processed nodes. This is commonly used in algorithms that involve pathfinding, graph traversal, or puzzle-solving, where nodes and their relationships need to be systematically tracked and analysed, in this case figuring out where the nodes are and the distance between them.

The only change made to the final block of code was that I had to change the numbers for the dots and what colour they are drawn to account for the surrounding 1's in the "self.pathFindPuzzle". I had to implement it this way as it made sure that extra indexes labelled as 1 would not cause extra dots to be drawn.

```
def createGrid(self): #will create the grid and draw the dots
    self.makePuzzle() #creates the puzzle

    #sorts the nodes by distance so that the shortest distance is first
    keys = list(self.distances)
    for i in range(0, len(self.distances)):
        min = self.distances[keys[0]]
        for key in self.distances:
            if self.distances[key] < min:
                min = self.distances[key]
        self.sortedNodes.append(key)
        self.distances.pop(key)

    for row in range(self.rows):
        for col in range(self.cols):

            #finds the start and end nodes that need to be connected
            if self.puzzle[row][col] > 0:
                if self.puzzle[row][col] in self.startNodes:
                    self.endNodes[self.puzzle[row][col]] = (row, col)
                    self.distances[self.puzzle[row][col]] = abs(row - self.startNodes[self.puzzle[row][col]][0]) + abs(col - self.startNodes[self.puzzle[row][col]][1])
                else:
                    self.startNodes[self.puzzle[row][col]] = (row, col)
                self.allNodes.append((row, col))


            #draws grid and dots with colours
            rect = pygame.Rect(self.xOffset + col * self.cellSize, (self.yOffset + row * self.cellSize) , self.cellSize, self.cellSize) #creates rectangle for grid
            pygame.draw.rect(newGame.display, white, rect, 1) #draws grid lines
            if self.puzzle[row][col] != 0 or 1: #if the dot label is not 0, draw a dot
                circle_centre = (self.xOffset + col * self.cellSize + self.cellSize // 2, ((self.yOffset + row * self.cellSize) ) + self.cellSize // 2)
                circle_radius = 25
                if self.puzzle[row][col] == 2: #if dot label is 1, draw red dot, if 2 draw blue dot, etc.
                    pygame.draw.circle(newGame.display, red, circle_centre, circle_radius)
                elif self.puzzle[row][col] == 3:
                    pygame.draw.circle(newGame.display, blue, circle_centre, circle_radius)
                elif self.puzzle[row][col] == 4:
                    pygame.draw.circle(newGame.display, green, circle_centre, circle_radius)
                elif self.puzzle[row][col] == 5:
                    pygame.draw.circle(newGame.display, yellow, circle_centre, circle_radius)
                elif self.puzzle[row][col] == 6:
                    pygame.draw.circle(newGame.display, orange, circle_centre, circle_radius)
```

The next method is found in the "Grid" class, the "checkGrid" method systematically validates the grid by checking each cell and its adjacent cells against specific conditions. This ensures that the grid meets the required criteria, and any deviations from these criteria result in the grid being marked as invalid. I chose to implement this method as when checking to see if the puzzle is solvable, time can be saved by checking to see if the dots are surrounded by one another or if there are empty spaces surrounded by dots for example.

```
def checkGrid(self, grid): #checks if the grid is valid
    for row in range(self.rows):
        for col in range(self.cols):
            if grid[row][col] > 0: #checks if the value at the grid position is greater than 0
                dotNum = grid[row][col]
                if grid[row + 1][col] > 0 and grid[row + 1][col] != dotNum: #checks if the value below the current grid position is greater than 0 and not equal to the dot number
                    if grid[row - 1][col] > 0 and grid[row - 1][col] != dotNum: #checks if the value above the current grid position is greater than 0 and not equal to the dot number
                        if grid[row][col + 1] > 0 and grid[row][col + 1] != dotNum: #checks if the value to the right of the current grid position is greater than 0 and not equal to the dot number
                            if grid[row][col - 1] > 0 and grid[row][col - 1] != dotNum: #checks if the value to the left of the current grid position is greater than 0 and not equal to the dot number
                                return False
                if grid[row + 1][col] == dotNum and grid[row - 1][col] == dotNum and grid[row][col + 1] == dotNum: #checks if the value below, above and to the right of the current grid position is equal to the dot number
                    return False
                elif grid[row + 1][col] == dotNum and grid[row - 1][col] == dotNum and grid[row][col - 1] == dotNum: #checks if the value below, above and to the left of the current grid position is equal to the dot number
                    return False
                elif grid[row + 1][col] == dotNum and grid[row][col + 1] == dotNum and grid[row][col - 1] == dotNum: #checks if the value below, to the right and to the left of the current grid position is equal to the dot number
                    return False
                elif grid[row - 1][col] == dotNum and grid[row][col + 1] == dotNum and grid[row][col - 1] == dotNum: #checks if the value above, to the right and to the left of the current grid position is equal to the dot number
                    return False
    return True
```

The “solvePuzzle” method is also found in the “Grid” class. The “solvePuzzle” function is designed to solve a puzzle using a recursive backtracking algorithm. It begins by checking if the grid is valid using the “checkGrid” function. If the grid is not valid, the function returns False. Next, it checks if the puzzle is already solved using the “returnSolved” function. If the puzzle is solved, it returns True.

The function then iterates through the sorted nodes, represented by “self.sortedNodes”. For each node, it retrieves the start and end positions from “self.startNodes” and “self.endNodes”, respectively. It calculates the Manhattan distance between the start and end nodes using the “abs” function. If the distance is greater than 1, it initializes an empty list called directions to store possible movement directions.

The function checks the four possible directions (right, left, down, up) to see if moving in that direction is valid (i.e., the grid value in that direction is 0). Depending on the relative position of the end node to the start node, it either inserts the direction at the beginning or appends it to the end of the directions list.

If no valid directions are found, the function returns False. Otherwise, it iterates through the directions list and attempts to move in each direction. For each direction, it updates the start node's position and sets the grid value at the new position to the current dot number. It then recursively calls “solvePuzzle” with the updated grid. If the recursive call returns True, the function returns True. If not, it backtracks by resetting the grid value and the start node's position to their previous states.

Overall, the “solvePuzzle” function uses a combination of grid validation, direction checking, and recursive backtracking to solve the puzzle.

```
def solvePuzzle(self, grid): #solves the puzzle using a recursive backtracking algorithm
    if self.checkGrid(grid) == False: #checks if the grid is valid
        return False

    for dotNum in self.sortedNodes: #loops through the sorted nodes
        start = self.startNodes[dotNum]
        end = self.endNodes[dotNum]

        if (abs(end[0], start[0]) + abs(end[1], start[1])) > 1: #checks if the distance between the start and end nodes is greater than 1
            directions = []
            if grid[start[0]][start[1] + 1] == 0: #checks if the value to the right of the start node is 0
                if end[1] > start[1]: #checks if the end node is to the right of the start node
                    directions.insert(0, "right") #inserts "right" at the start of the directions array
                else:
                    directions.append("right")  #appends "right" to the end of the directions array

            if grid[start[0]][start[1] - 1] == 0: #checks if the value to the left of the start node is 0
                if end[1] < start[1]: #checks if the end node is to the left of the start node
                    directions.insert(0, "left") #inserts "left" at the start of the directions array
                else:
                    directions.append("left") #appends "left" to the end of the directions array

            if grid[start[0] + 1][start[1]] == 0: #checks if the value below the start node is 0
                if end[0] > start[0]:
                    directions.insert(0, "down")
                else:
                    directions.append("down")

            if grid[start[0] - 1][start[1]] == 0: #checks if the value above the start node is 0
                if end[0] < start[0]:
                    directions.insert(0, "up")
                else:
                    directions.append("up")

            if len(directions) == 0: #checks if the length of the directions array is 0
                return False
```

```
            for direction in directions:
                if direction == "right": #checks if the direction is right
                    start[1] += 1 #increments the x value of the start node
                    grid[start[0]][start[1]] = dotNum #sets the value at the start node to the dot number
                    if self.solvePuzzle(grid) == True: #recursively calls the solvePuzzle function
                        return True
                    else:
                        grid[start[0]][start[1]] = 0 #sets the value at the start node to 0
                        start[1] -= 1 #decrements the x value of the start node

                elif direction == "left": #checks if the direction is left
                    start[1] -= 1 #decrements the x value of the start node
                    grid[start[0]][start[1]] = dotNum #sets the value at the start node to the dot number
                    if self.solvePuzzle(grid) == True: #recursively calls the solvePuzzle function
                        return True
                    else:
                        grid[start[0]][start[1]] = 0 #sets the value at the start node to 0
                        start[1] += 1 #increments the x value of the start node

                elif direction == "up": #checks if the direction is up
                    start[0] -= 1
                    grid[start[0]][start[1]] = dotNum
                    if self.solvePuzzle(grid) == True:
                        return True
                    else:
                        grid[start[0]][start[1]] = 0
                        start[0] += 1

                elif direction == "down": #checks if the direction is down
                    start[0] += 1
                    grid[start[0]][start[1]] = dotNum
                    if self.solvePuzzle(grid) == True:
                        return True
                    else:
                        grid[start[0]][start[1]] = 0
                        start[0] -= 1
            return False
```

References

For this sprint I implemented a puzzle checker to see if the randomly generated puzzle is solvable, to do this I took inspiration from https://www.101computing.net/connect-flow-backtracking-algorithm/.

Changes to Pseudocode

Draw Dots Method

For this method, I changed how the coordinates were generated. This was because the way the coordinates were generated in the pseudocode, would regularly generate duplicate coordinates, which caused dots to overlap in the same grid square meaning the game could not be solved. So, to combat this I decided to generate dots by creating a list with all possible coordinates and then randomly choosing one of those coordinates for a dot to be drawn in.

I also had to change the x and y positions from "self.grid[x1][y1]" in the pseudocode to "self.puzzle[y][x]" in the python program so that the position the dots are in, are accurate.

I also added a new puzzle to be passed into the "solvePuzzle" method. I did this as the new puzzle is simply the old puzzle but surrounded in 1's, this is so that there are limits as to where the "solvePuzzle" algorithm can path find to.

Draw Grid Method

For the "drawGrid" method I added a block of code in that will calculate the distances between the start and end nodes as well as sorting them in ascending order. This helps with the pathfinding method as the relationships of nodes are systematically tracked and analysed ultimately speeding up the pathfinding process.

Problems I Encountered While Coding

The first problem I encountered was that an error would occur with the "checkGrid" function when searching to see if the grid was valid as, saying that the list index was out of range.

```
Exception has occurred: IndexError ×
list index out of range
  File "[redacted]", line 542, in checkGrid
    if grid[row][col] > 0: #checks if the value at the grid position is greater than 0
       ~~~~^^^^^
  File [redacted] line 411, in solvePuzzle
    if self.checkGrid(grid) == False: #checks if the grid is valid
       ^^^^^^^^^^^^^^^^^^^
  File [redacted] line 221, in easyLevel
    if newGrid.solvePuzzle(newGrid.pathFindPuzzle): #runs the solvePuzzle function on the pathFindPuzzle and checks if it is solvable
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File [redacted], line 102, in levelSelection
    self.easyLevel()
  File [redacted] line 82, in displayMenu
    self.levelSelection() # starts game when loop ends
    ^^^^^^^^^^^^^^^^^^^^
```

The code below was responsible for creating the puzzle:

```
def makePuzzle(self):
    self.puzzle = [[0,0,0,0,0]] #creates
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])

    for i in range(1,6):
        x = random.randint(0,4)
        y = random.randint(0,4)
        w = random.randint(0,4)
        z = random.randint(0,4)
        self.puzzle[y][x] = i
        self.puzzle[z][w] = i
```

And the code below was where the error was occurring:

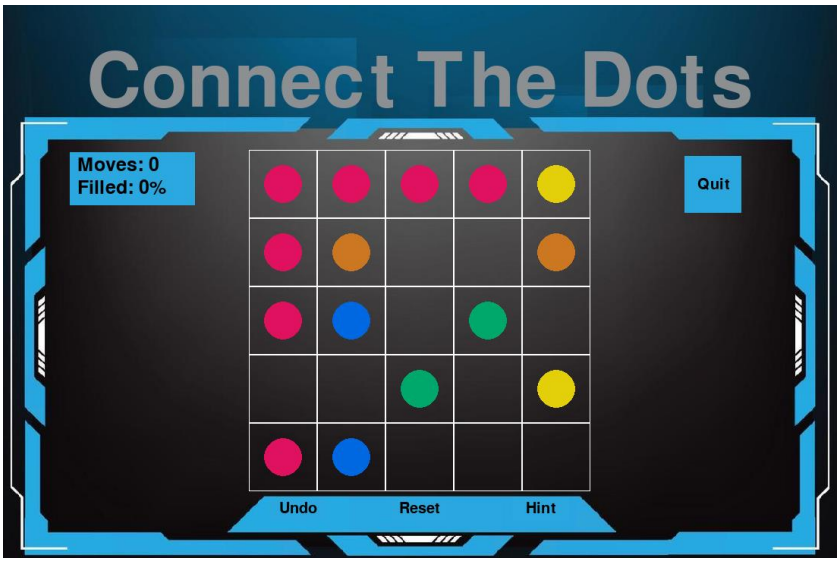
```
def checkGrid(self, grid): #checks if the grid is valid
    for row in range(self.rows):
        for col in range(self.cols):
            if grid[row][col] > 0: #checks if the value at the grid position is greater than 0
                dotNum = grid[row][col]
                if grid[row + 1][col] > 0 and grid[row + 1][col] != dotNum: #checks if the value below the current grid position is greater than 0 and not equal to the dot number
                    if grid[row - 1][col] > 0 and grid[row - 1][col] != dotNum: #checks if the value above the current grid position is greater than 0 and not equal to the dot number
                        if grid[row][col + 1] > 0 and grid[row][col + 1] != dotNum: #checks if the value to the right of the current grid position is greater than 0 and not equal to the dot number
                            if grid[row][col - 1] > 0 and grid[row][col - 1] != dotNum: #checks if the value to the left of the current grid position is greater than 0 and not equal to the dot number
                                return False
                if grid[row + 1][col] == dotNum and grid[row - 1][col] == dotNum and grid[row][col + 1] == dotNum: #checks if the value below, above and to the right of the current grid position is equal to the dot number
                    return False
                elif grid[row + 1][col] == dotNum and grid[row - 1][col] == dotNum and grid[row][col - 1] == dotNum: #checks if the value below, above and to the left of the current grid position is equal to the dot number
                    return False
                elif grid[row + 1][col] == dotNum and grid[row][col + 1] == dotNum and grid[row][col - 1] == dotNum: #checks if the value below, to the right and to the left of the current grid position is equal to the dot number
                    return False
                elif grid[row - 1][col] == dotNum and grid[row][col + 1] == dotNum and grid[row][col - 1] == dotNum: #checks if the value above, to the right and to the left of the current grid position is equal to the dot number
                    return False
    return True
```

So to fix this error, I chose to create a new array which was simply the original array "self.puzzle" surrounded in ones to create walls to limit where the program would search. I did this by appending the "self.puzzle" inside of a new array, where it would be stored at index "1", where the "wall of 1's" would be stored at indexes 0 and 2.

Shown below:

```
def makePuzzle(self):
    self.puzzle = [[0,0,0,0,0]]
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle = [[1,1,1,1,1,1,1]]
    self.puzzle.append([1,0,0,0,0,0,1])
    self.puzzle.append([1,0,0,0,0,0,1])
    self.puzzle.append([1,0,0,0,0,0,1])
    self.puzzle.append([1,0,0,0,0,0,1])
    self.puzzle.append([1,0,0,0,0,0,1])
    self.puzzle.append([1,1,1,1,1,1,1])

    for i in range(1,6):
        x = random.randint(0,4)
        y = random.randint(0,4)
        w = random.randint(0,4)
        z = random.randint(0,4)
        self.puzzle[y][x] = i
        self.puzzle[z][w] = i
```

However I initially appended the "self.puzzle" to the same array called "self.puzzle", this caused an error where the wall of 1's would be drawn as red dots, shown below:



To combat this error I simply changed the name of the new array so that the program would not draw the dots in the new array, I also append chose to append the "self.puzzle" array into the "self.pathFindPuzzle" so the random puzzle is kept.

The fixed code:

```
def makePuzzle(self):
    self.puzzle = [[0,0,0,0,0]] #creates a puzzle with 5 rows
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])

    for i in range(1,6):
        x = random.randint(0,4)
        y = random.randint(0,4)
        w = random.randint(0,4)
        z = random.randint(0,4)
        self.puzzle[y][x] = i
        self.puzzle[z][w] = i
    self.pathFindPuzzle = [[1,1,1,1,1,1,1]]
    self.pathFindPuzzle.append([1, self.puzzle[0], 1])
    self.pathFindPuzzle.append([1, self.puzzle[1], 1])
    self.pathFindPuzzle.append([1, self.puzzle[2], 1])
    self.pathFindPuzzle.append([1, self.puzzle[3], 1])
    self.pathFindPuzzle.append([1, self.puzzle[4], 1])
    self.pathFindPuzzle.append([1,1,1,1,1,1,1])
```

The next problem I encountered in this sprint was when randomly generating dots as shown below, there was a chance that dots would be generated in the same position due to the way they were being generated. To combat this, I decided to create a list of all possible coordinates for the grid, and then to randomly choose 2 coordinates for a pair of dots from this list, and then remove these coordinates from the list to ensure that they cannot be chosen again.

The code before:

```
def makePuzzle(self):
    self.puzzle = [[0,0,0,0,0]] #creates a puzzle with 5 rows
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])

    for i in range(2,7):
        x = random.randint(0,4)
        y = random.randint(0,4)
        w = random.randint(0,4)
        z = random.randint(0,4)
        self.puzzle[y][x] = i
        self.puzzle[z][w] = i
    self.pathFindPuzzle = [[1,1,1,1,1,1,1]]
    self.pathFindPuzzle.append([1, self.puzzle[0], 1])
    self.pathFindPuzzle.append([1, self.puzzle[1], 1])
    self.pathFindPuzzle.append([1, self.puzzle[2], 1])
    self.pathFindPuzzle.append([1, self.puzzle[3], 1])
    self.pathFindPuzzle.append([1, self.puzzle[4], 1])
    self.pathFindPuzzle.append([1,1,1,1,1,1,1])
```

The code after:

```
def makePuzzle(self):
    self.puzzle = [[0,0,0,0,0]] #creates a puzzle with 5 rows and 5 columns that gets random values assigned to it
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])
    self.puzzle.append([0,0,0,0,0])

    #generates random coordinates for the dots to be draw in from the list of coordinates
    coords = [(0,0), (0,1), (0,2), (0,3), (0,4), (1,0), (1,1), (1,2), (1,3), (1,4), (2,0), (2,1), (2,2), (2,3), (2,4), (3,0), (3,1), (3,2), (3,3), (3,4), (4,0), (4,1), (4,2), (4,3), (4,4)]
    for i in range(2,7):
        coord1 = random.choice(coords) #chooses a random coordinate from "coords" then removes it from the array
        x,y = coord1 # x and y are set to the values of the coordinate
        coords.remove(coord1)

        coord2 = random.choice(coords)
        w,z = coord2
        coords.remove(coord2)

        self.puzzle[y][x] = i #changes the value held at the y,x postion to a sigular value so that the dots can be drawn
        self.puzzle[z][w] = i

    self.pathFindPuzzle = [[1,1,1,1,1,1,1]] #creates a path finding puzzle by adding 1's to the puzzle so that the there is a "barrier" around the grid
    self.pathFindPuzzle.append([1, self.puzzle[0], 1])
    self.pathFindPuzzle.append([1, self.puzzle[1], 1])
    self.pathFindPuzzle.append([1, self.puzzle[2], 1])
    self.pathFindPuzzle.append([1, self.puzzle[3], 1])
    self.pathFindPuzzle.append([1, self.puzzle[4], 1])
    self.pathFindPuzzle.append([1,1,1,1,1,1,1])
```

Test Plan

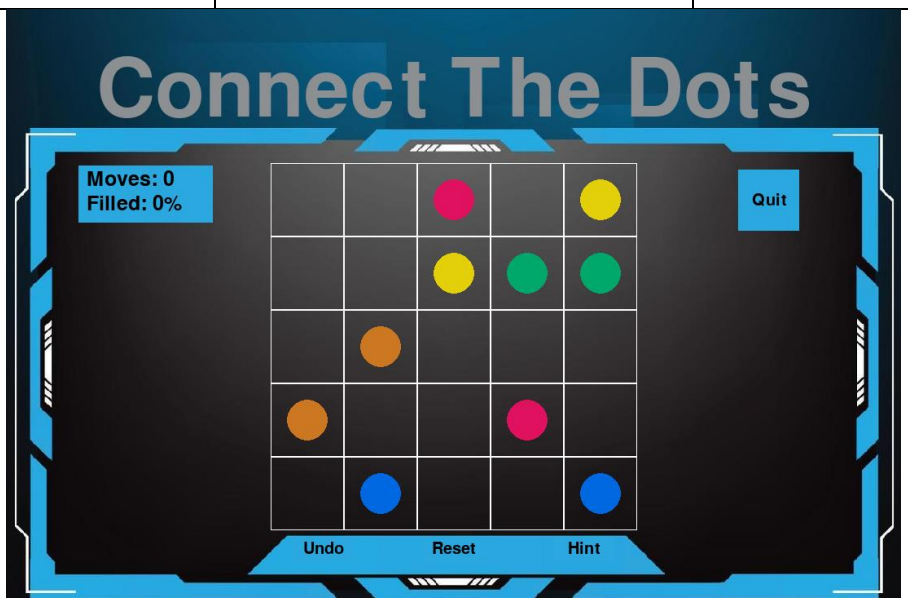
Action	Reason	Expected Outcome
Presses a dot on the puzzle	Checks to see if anything unexpected happens	Moves will increment by 1
Connects a path to corresponding dot	Checks "easyLevel" function works correctly, this being the part that identifies when a path is being drawn	The line will be the same colour as the dot drawn from, then moves and squares filled values are adjusted
Users attempts to draw a path outside the grid	Checks to see if the program still errors or if it has been fixed	"Invalid move" is output in terminal
Press "Hint"	Checks "giveHint" function works correctly	"Hint" is output in terminal
Press "Reset" button	Checks "resetLevel" function works correctly	All paths are removed

Press "Undo" button	Checks "undoMove" function works correctly	Correct messages are outputted depending on if the player has an "undo" to use or not
User fills all squares with a path so that "squaresFilled" is equal to 25	To check if "solved" function works correctly	A new window is displayed saying "Level solved" with 3 buttons
User presses "Next Level"	Checks to see if "nextLvl" function works correctly	"Next level" is output in terminal
User presses "Level Selection"	Checks to see if the user can successfully navigate back to a previous menu screen	User is taken back to the level selection screen so they can choose a different difficulty setting
User presses on the screen on the win screen	Check to see if anything unexpected happens	Nothing

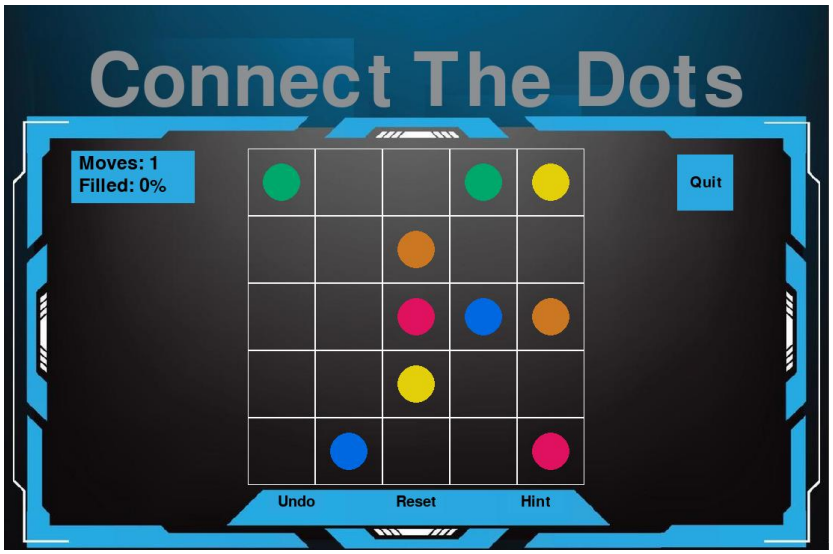
Formal Testing

Normal Data

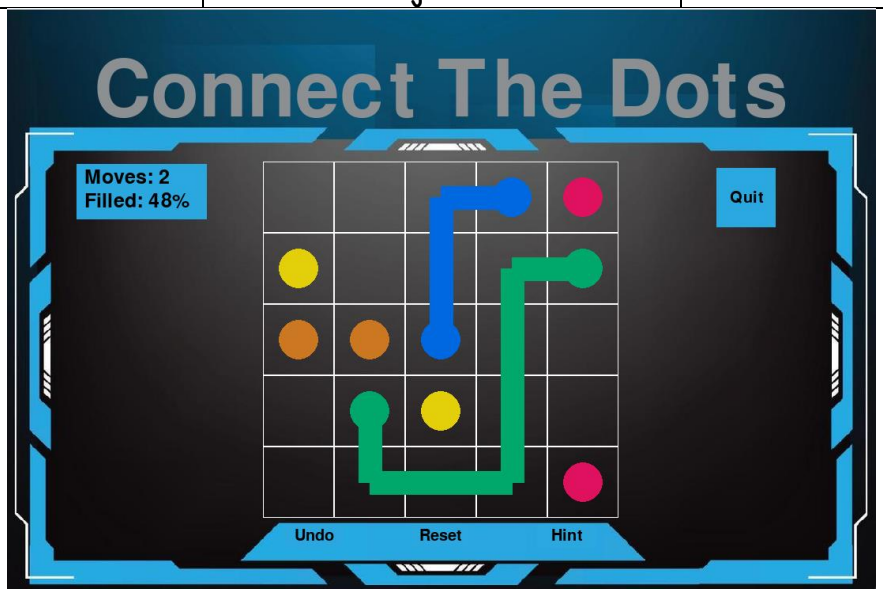
Action	Expected Outcome	Actual Outcome
User presses "Easy -5x5" button	A randomly generated puzzle is created	As expected



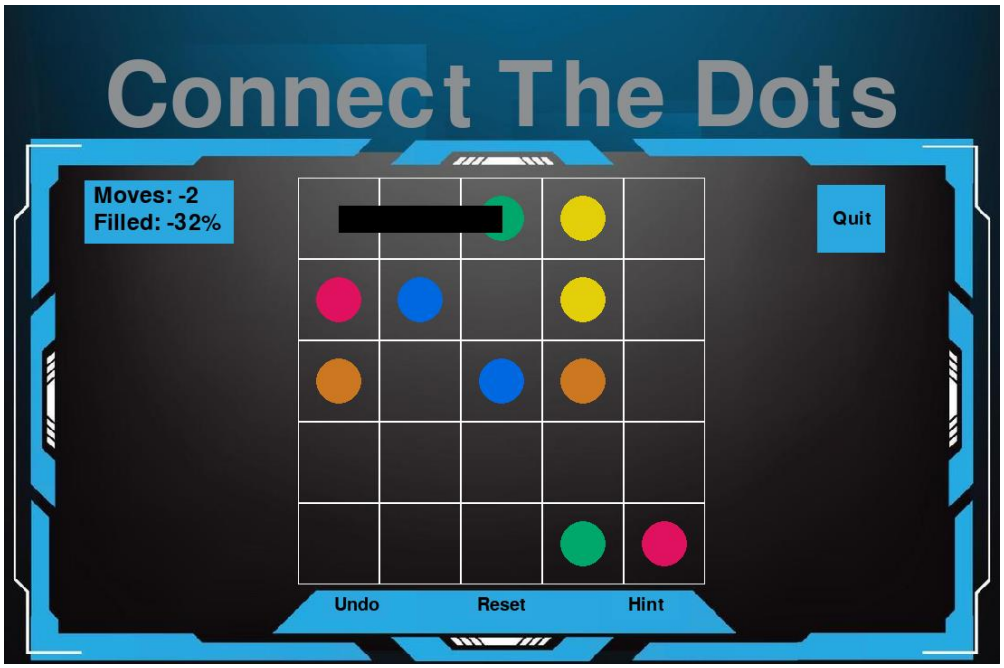
Presses a dot on the puzzle	Moves will increment by 1	As expected



Connects a path to corresponding dot	The line will be the same colour as the dot drawn from, then moves and squares filled values are adjusted	As expected



Users attempts to draw a path outside the grid	"Invalid move" is output in terminal	As expected



(Drawn from top middle green dot directly left outside the grid line)

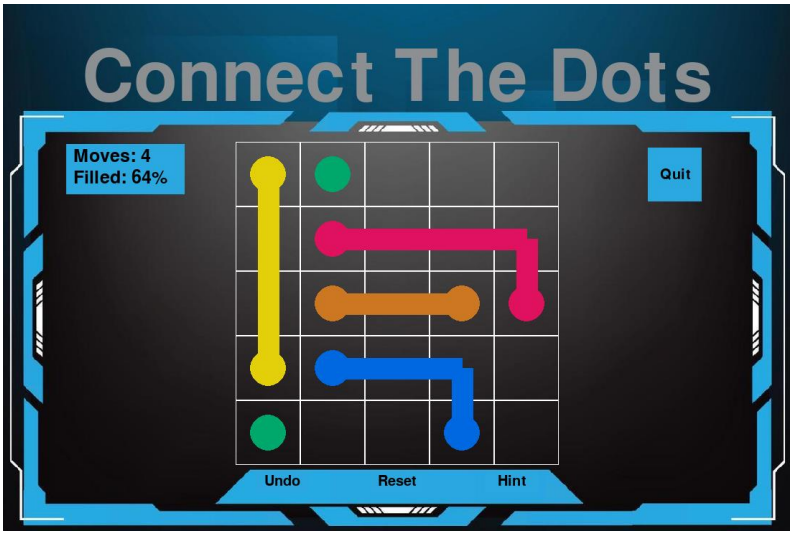
```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Invalid move
Invalid move
```

Press "Hint"	"Hint" is output in terminal	As expected

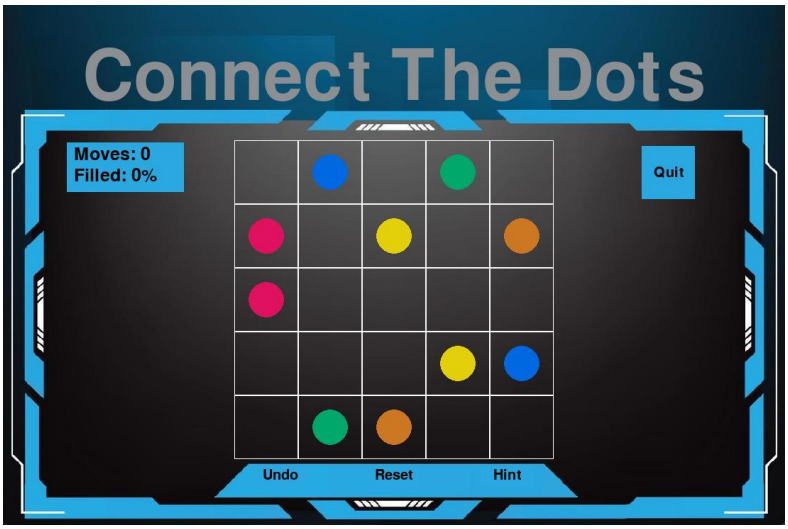
```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Hint
```

Press "Reset" button	All paths are removed	The entire puzzle is reset/ a new puzzle is drawn

Before:



After:



This occurs as the "resetLevel" method draws a new grid with no paths connected over the top of the old grid, but this causes the old puzzle to be removed causing the "reset" button to not work as intended.

Press "Undo" button	Correct messages are outputted depending on if the player has an "undo" to use or not	As expected

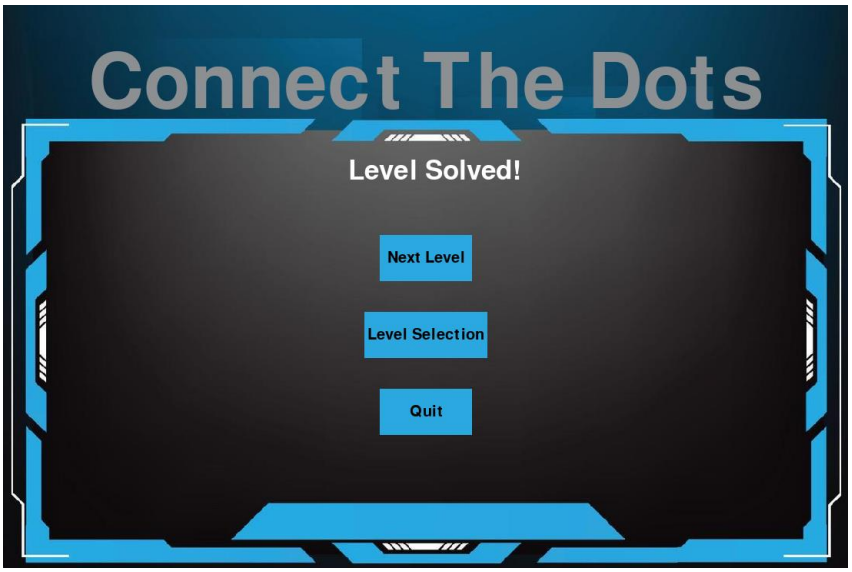
If the player has "undo":

```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Before [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)], [(1, 3), (1, 2), (1, 1), (1, 0), (2, 0)], [None]]
after [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)]]
```

If the player does not have "undo":

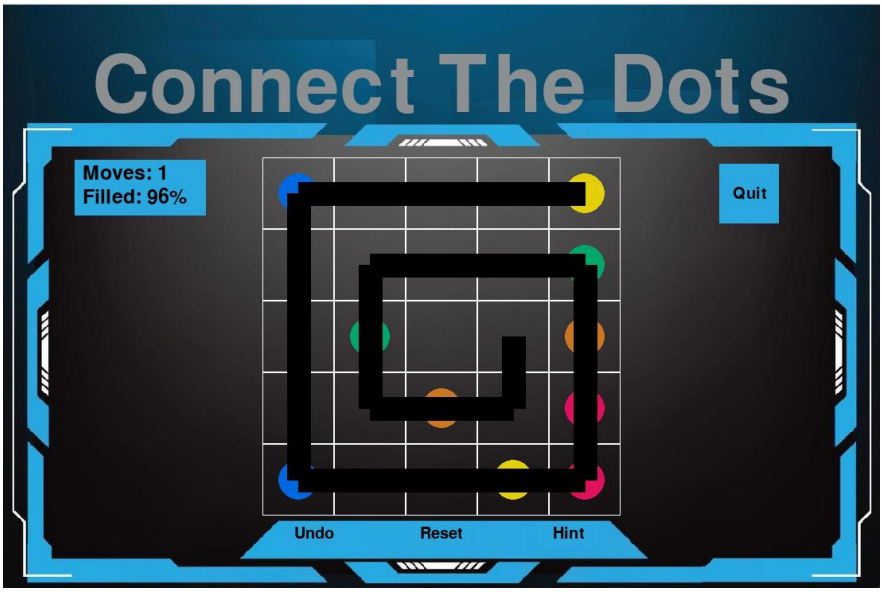
```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Before [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)], [(1, 3), (1, 2), (1, 1), (1, 0), (2, 0)], [None]]
after [[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4)]]
No hint
```

User fills all squares with a path so that "squaresFilled" is equal to 25	A new window is displayed saying "Level solved" with 3 buttons	As expected

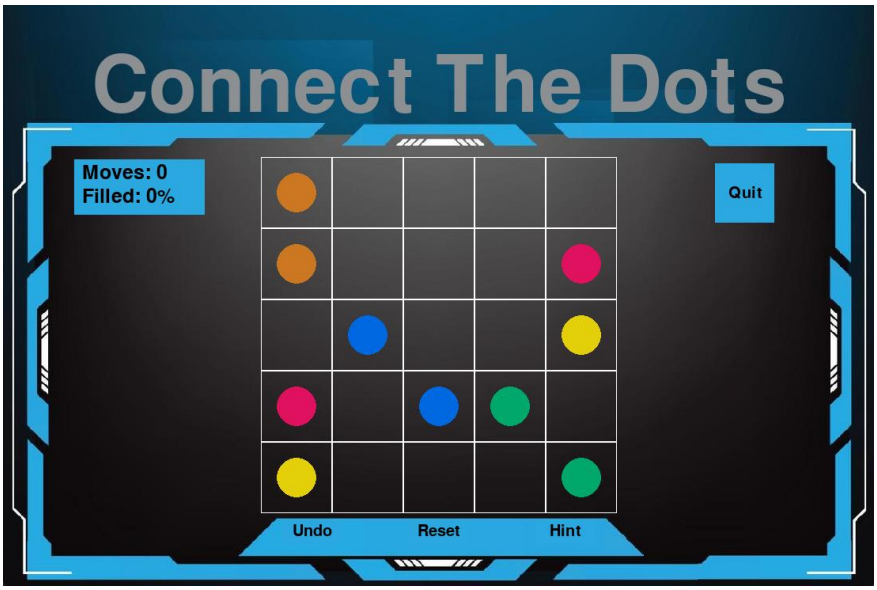


While this part of the game works, it only works as the user is able to draw over dots and other paths. If I had more time at the end of the project, I would have made it so that the user cannot do these things, making it more accurate for how the game should be played.

Error shown below:

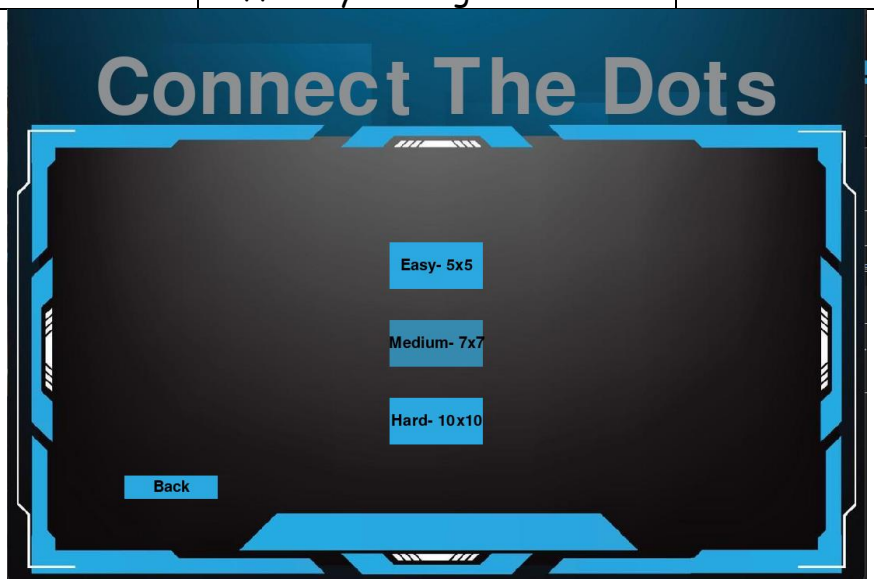


User presses "Next Level"	A new puzzle is generated and drawn, and "next level" is printed in terminal	As expected




```
pygame 2.3.0 (SDL 2.24.2, Python 3.11.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
next level
```

User presses "Level Selection"	User is taken back to the level selection screen so they can choose a different difficulty setting	As expected

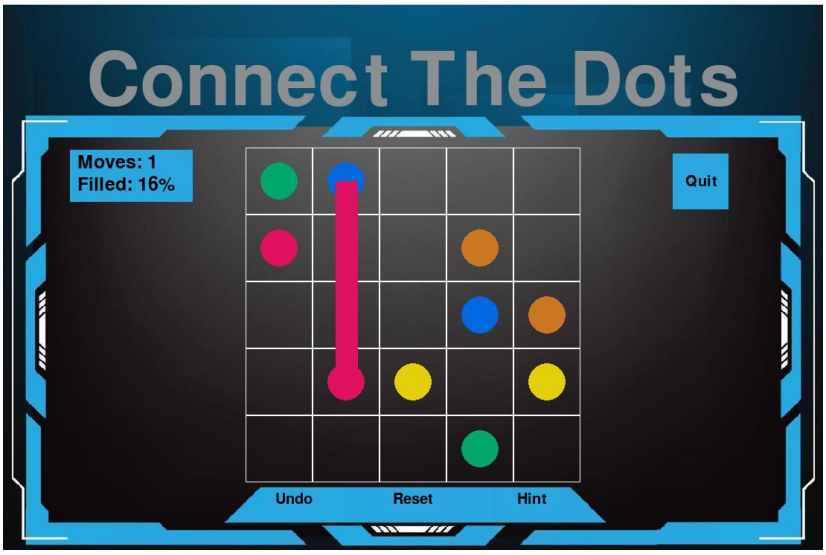


User presses on the screen on the win screen	Nothing	As expected

Boundary Data

Action	Expected Outcome	Actual Outcome
Joining a path to a different colour dot	Path will turn the colour of the dot it is connected to	As expected

In this case I attempted to draw the blue dot to the red dot.



Erroneous Data

There was no erroneous data for this sprint.

Evaluation

Success Criteria	Completed?
Random generation of dots, but also so that they can be solved despite being in random places on the grid	Partially
Implement path validation	No
Get both hint and undo mechanics working	No

In this sprint I was not able to fully accomplish my success criteria. However, I was able to implement random generation of dots which was successful, but despite creating a method to detect if the random puzzle was solvable, I was unable to successfully get it to work hence why the first success criteria point is partially completed.

I was also unable to implement path validation again due to limitations of time, this meant that the player could draw over any colour dot as well as overlap paths, creating an unrealistic game scenario.

I was also unable to get both "hint" and "undo" mechanics working, and due to the random generation, the "reset" mechanic does not work as myself and the stakeholders want it to. If I had more time, I would correct the methods so that the corresponding mechanics work as desired.

Evaluation

Post Development Testing

User Requirement	Test	Expected Outcome	Actual Outcome
1. The user will need to be able to interact with the game using a mouse tracking algorithm so that when they click using the mouse the program can recognise when this happens, I will do this by using the in-built pygame function pygame.event.get().	1st menu screen	When the game runs, it opens with 3 buttons on the screen and when pressed, they perform the corresponding action	As expected, Testing evidence UR1 - 1
	2nd menu screen (level selection)	When the user presses "play" button, a new screen will be displayed over	As expected, Testing evidence UR1- 2
	Play game	When user presses "Easy- 5x5" button a new screen will be displayed with a grid and dots inside in random positions	As expected, Testing evidence UR1 - 3
	Use buttons	Program recognises that the buttons have been pressed	As expected, Testing evidence UR1 - 4
	Drawing paths	When holding and dragging the mouse, a path will be drawn, if connected to a dot the path will take the colour of that dot, otherwise the path will be black	As expected, Testing evidence UR1 – 5 and 6
2. Each level will have to be randomly generated but will also have to differ from other levels so that puzzles aren't repeated.	Press "Easy 5x5" to generate level	A grid is drawn with dots in random positions	As expected, Testing evidence UR2 – 1
	Press "next level" button to generate another level	A new grid is drawn with dots in different random positions	As expected, Testing evidence UR2 - 2

3. The program will need to recognise when the user has completed the puzzle, this would be when all corresponding dots have been connected and when every square in the grid is filled out and therefore show an appropriate message.	User draws paths so that every square is filled	A new display is drawn with 3 buttons and "Level Solved" text confirming that the level was solved	As expected, Testing evidence UR3

4. The program will also need to recognise if the user resets the level using the reset button which will remove any moves made.	User presses "Reset" button	Grid will reset and dots are drawn in new positions	As expected, Testing evidence UR4

5. The program will also need to recognise if the user asks for a hint and will therefore have to give the user 1 single completed solution.	User presses "Hint" button	"Hint" will be outputted in the python terminal	As expected, Testing evidence UR5

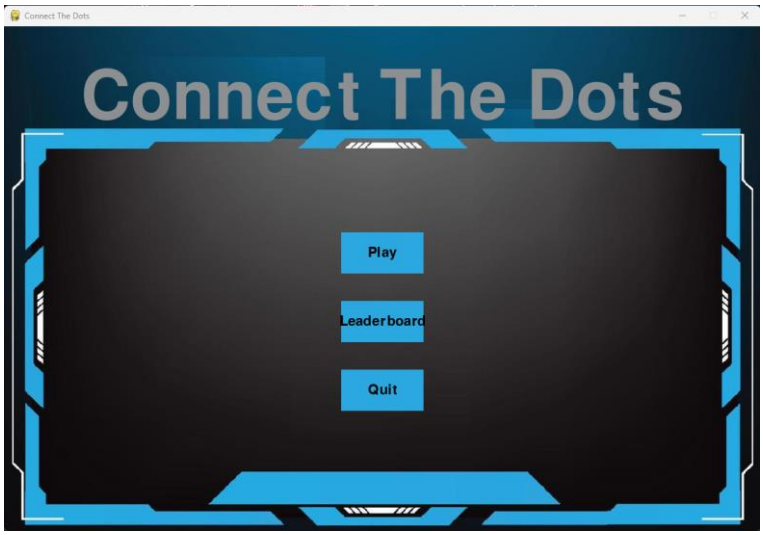
6. The program will also need to be able to keep track of how many moves the player has made.	User clicks on the puzzle/ draws a path	"Moves" in the top left of the window will be incremented by one	As expected, Testing evidence UR6

7. There will need to be a menu that the player can navigate before solving the puzzle.	The user can easily navigate between the "Menu" screen and the "Level Selection" screen using the buttons	Buttons work accordingly so that the user can navigate between the "Menu" and "Level Selection" screen	As expected, Testing evidence UR7

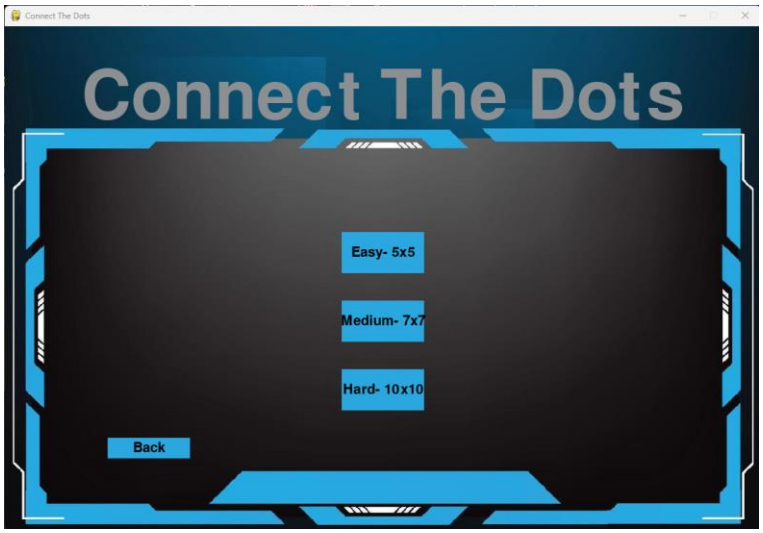
Testing evidence

UR1

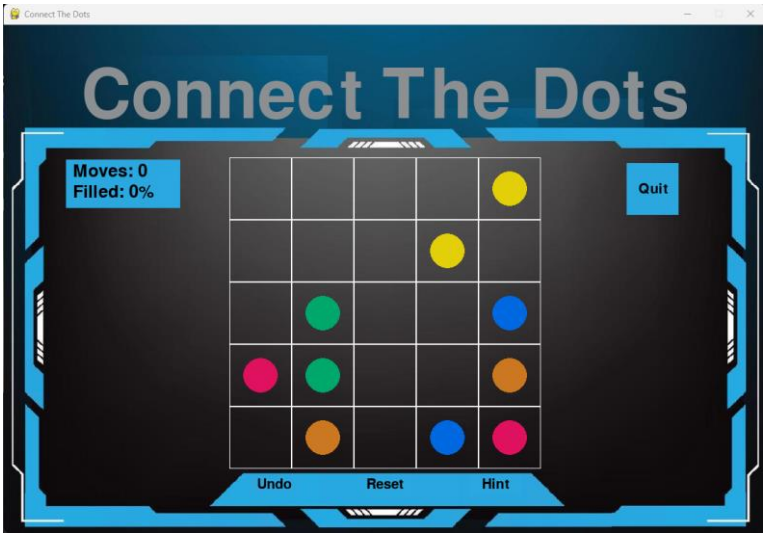
1:



2:



3:



4:

See sprint 1 page 37 for "Leaderboard" button

See Evaluation UR1 – 3 for "Easy- 5x5" button

See Sprint 1 page 37, 38 for "Medium" and "Hard" buttons

See Sprint 1 page 38 for "Back" button

See Sprint 1 page 37 for "Quit" button

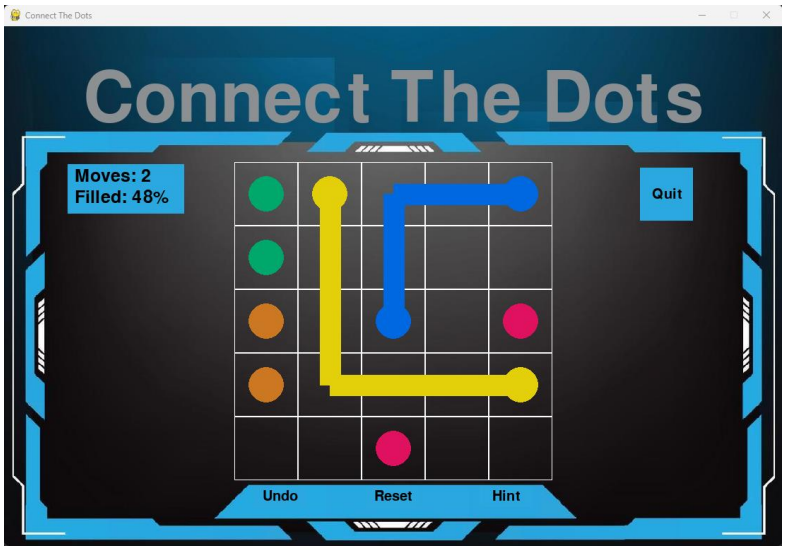
See Sprint 4 page 93, 94 for "Hint", "Undo" and "Reset" buttons

See Sprint 4 page 95, 96 for "Next Level" button

See Sprint 4 page 96 for "Level Selection" button

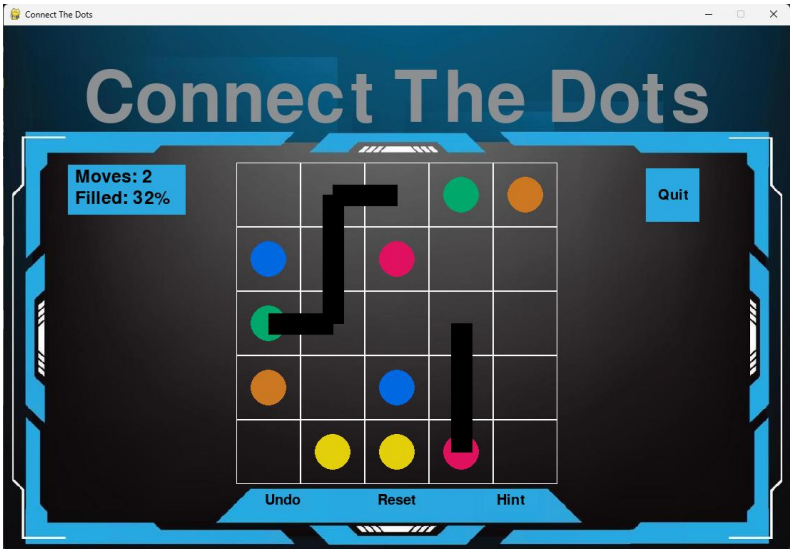
5:

Connecting dots when drawing:



6:

Not connecting dots when drawing:



UR2

1:

See Evaluation UR1 – 3 for "Easy- 5x5" button puzzle generation

2:

See Sprint 4 page 95, 96 for "Next Level" button puzzle generation

UR3

See Sprint 4 page 94, 95 to test to see if the game has been completed

UR4

See Sprint 4 page 93 for "Reset" button

UR5

See Sprint 4 page 93 for "Hint" button

UR6

See Sprint 4 page 91, 92 for the "Moves" mechanic

UR7

See Evaluation UR1 – 2 for "Play" button

See Evaluation UR1 – 3 for "Easy– 5x5" button

See Sprint 1 page 38 for "Back" button

Stakeholder Feedback

Stakeholder 1- Finley Whetstone,

Stakeholder 2- Ethan Venton,

Stakeholder 3- James Hatswell

Stakeholder 4- Evie-May Apted

What do you think of the game's appearance?

Stakeholder 1: *It's simple to understand and very intuitive*

Stakeholder 2: *It's very concise it's very simplistic/ uncomplex, overall, a solid display*

Stakeholder 3: *It's has a minimalistic vibe*

Stakeholder 4: *It looks very clean, nice and simple as well*

Would you say the game is intuitive and easy to understand and use?

Stakeholder 1: *Yes, it is easy as it is already an existing game*

Stakeholder 2: *It's alright*

Stakeholder 3: *It's easy to navigate so I'm happy about that*

Stakeholder 4: *Yes, not too complex to understand*

Do you think the gameplay is engaging and enjoyable?

Stakeholder 1: *Not really as you can't really solve the game as compared to other games*

Stakeholder 2: *I think if the game was fully implemented and working correctly, it would be really enjoyable*

Stakeholder 3: *If you were to get a puzzle that you can solve, I'm sure it would be more enjoyable*

Stakeholder 4: *When the puzzle is solvable then yes, otherwise not particularly*

What would you like to see working in the game?

Stakeholder 1: *The game mechanics actually have a function intended*

Stakeholder 2: *In the future I would like to see that paths can't overlap one another, a scoreboard so for example you get a certain amount of points after completing a level, and different level difficulties to choose from*

Stakeholder 3: *Make it so the random puzzles can be solved*

Stakeholder 4: *You can use the buttons when playing the game (reset, hint, undo)*

What specific tasks do you expect the program to do that it currently does not?

Stakeholder 1: *I would expect the larger grids to work*

Stakeholder 2: *I would expect it to not let paths overlap one another*

Stakeholder 3: *I would expect puzzles to always be solvable*

Stakeholder 4: *I would expect the game to be able to make multiple levels without having to close the game*

How well do you think the current version reflects the original project goals?

Stakeholder 1: *Not too bad, it looks like the original idea, but it doesn't work fully so that's the only thing letting the game down*

Stakeholder 2: *It reflects well, what you intended it to do, but it doesn't execute it the best so that is an area to improve*

Stakeholder 3: *I think it could reflect better as you can't really play the game as puzzles aren't likely to be solvable*

Stakeholder 4: *Pretty good but could be better*

Do you like the game information/ statistics the game shows you

Stakeholder 1: *Yeah, it's nice to see how many moves you've made and how many squares you've filled*

Stakeholder 2: *I like that you've shown how many moves the user has made. You can see how many squares you've filled in so that seems a bit unnecessary*

Stakeholder 3: *Yeah, it's cool to see*

Stakeholder 4: Yes, it's a nice feature

Would you like to see any other game information/ statistics mechanics be implemented, if so, what would you like to see:

Stakeholder 1: *Not really, as long as the main part of the game works, I'm not really fussed*

Stakeholder 2: *Yes, perhaps you could show the player how many paths they have connected as you initially intended to*

Stakeholder 3: *Maybe a timer to see how long it takes to complete a level*

Stakeholder 4: *Maybe a different shaped grid*

Evaluation of User Requirements

1. The user will need to be able to interact with the game using a mouse tracking algorithm so that when they click using the mouse the program can recognise when this happens, I will do this by using the in-built pygame function pygame.event.get().

This user requirement has been fully met. My post development testing clearly indicates that the user is able to interact with the game, this was done as described, using pygame with the built in function "pygame.event.get()" and is used constantly throughout the duration of the program.

2. Each level will have to be randomly generated but will also have to differ from other levels so that puzzles aren't repeated.

This user requirement was partially completed. While levels are randomly generated and each puzzle is different, the levels are unlikely to be solvable. To ensure that levels are solvable, I would make it so that the program will create a random puzzle, and then run the "solvePuzzle" method on that puzzle, and if it is solvable, to return True and draw that puzzle for the user to solve, however if False is returned, the program should then create a new randomly generated puzzle and test to see if that one is solvable, this process would be repeated until a solvable puzzle is found. This would be done using a while loop to ensure that it will loop until a certain condition is met to break out of the loop.

3. The program will need to recognise when the user has completed the puzzle, this would be when all corresponding dots have been connected and when every square in the grid is filled out and therefore show an appropriate message.

This user requirement was partially met. The program is able to recognise that the puzzle is solved, but this only involves checking to see if every grid square is filled, not checking to see if all dots have been connected to their corresponding pair. To implement this feature, I would make it so that the program can identify the dot number of the dot the user starts drawing from, I would then check the dot number of the dot where the user stops drawing, if they match the path can be drawn, if they don't match, the program should remove the path and adjust the appropriate values if needed.

4. The program will also need to recognise if the user resets the level using the reset button which will remove any moves made.

This user requirement was partially met. While this user requirement was initially met in earlier sprints, in later sprints, where random generation was implemented, this feature did not work as intended. To meet this user requirement, I would check the initial puzzle generated, copy that puzzle and store it in a new variable/ array and when the "Reset" feature is called, draw the puzzle stored in the new variable/ array so that the original puzzle is not lost.

5. The program will also need to recognise if the user asks for a hint and will therefore have to give the user 1 single completed solution.

This user requirement was not met. Due to limitations of time, I focused on more important aspects of the game to implement, if I had more time to implement this feature, I would do it using a pathfinding method similar to the "solvePuzzle" method. The program will choose 1 dot at random, that has not already been solved, it will then locate the other dot that it needs to connect to, and path find to it so that it does not negatively affect other dots (e.g it blocks the path for another path of dots), it should then automatically draw this path for the player confirming that the "Hint" mechanic was used/ detected.

6. The program will also need to be able to keep track of how many moves the player has made.

This user requirement has been fully met. My post development testing clearly indicates that this feature works. This is done using the "pygame.event.get()" method to check when the mouse is clicked and when it is released, feedback from my stakeholders indicated that they liked this feature and would potentially like other features similar to this in the future.

7. There will need to be a menu that the player can navigate before solving the puzzle.

This user requirement has been fully met. The user is able to get back and forth between menus, this was done using a "Back" button which will return the user to the "menu" screen from the "level selection" screen. This was also done in the "level end screen" where the user is able to return to the "level selection" screen and if they choose from this screen, go back to the "menu" screen.

Maintenance and Potential Issues

One issue that may arise is that the user may wish to update their version of Python or any of the independent libraries used e.g., Pygame. The game works on the current version of Python, but it is hard to say whether future changes or improvements may cause any problems.

Another issue that may arise could be that if the device the program is being run on, doesn't have the background image in the same file as the program. This would cause the program to error as it would be looking for an image that isn't in the computers files.

I have tested the game on a number of Windows machines as well as a single Linux machine and the game worked fine on them. On these devices Pygame was already installed, but if on a device the library is not installed the program would error on running meaning the user wouldn't be able to play the game. To combat this, I would recommend going into the devices terminal and using PIP to install Pygame, however for this step the user would need access to the internet.

Potential Improvements

One of my stakeholders mentioned that a leader board feature would be good to implement. If I was to implement a leader board, I would do this using a flat file database to store how many puzzles a player solves. This means that each time the game is played, a new row will be added to the database with the appropriate game details, this way, even if players enter the same identifier, the program will store both scores rather than the lower score being replaced by a higher score. I would choose to use the SQLite module to create and access the database in Python.

Another thing I would look to improve would be the random generation of dots as this was what the majority of stakeholders requested to be fixed. To see how I would do this please see Evaluation of User Requirements – 2.

Another part of the game I would look to improve would be the buttons part of the game interface, (Hint, Reset, Undo). To see how these would implemented please see Evaluation of User Requirements – 4 and 5 for Reset and Hint buttons. To implement the undo mechanic, I would determine the coordinates of the last path drawn, then using these coordinates, reset

the individual squares that the path is drawn in so that they are empty, by doing it this way, it ensures that only the most recent path is removed and keeps any other paths drawn.

Appendix

Final Code

```
import pygame
import time
import random

pygame.init()

#Colours
black = (0, 0, 0)
white = (255, 255, 255)
red = (224, 17, 95)#(255, 0, 0)
green = (0, 168, 107)#(0, 255, 0)
blue = (0, 105, 225)#(0, 0, 255)
yellow = (228, 208, 10)#(255, 255, 0)
orange = (204, 119, 34)#(255, 165, 0)
grey = (141, 144, 147) #for title
bgBlue = (42,168,223) #for buttons
inactiveBlue = (54,138,175) #for buttons

# Game class
class Game:
    def __init__(self):
        self.title = "Connect The Dots"
        self.clock = pygame.time.Clock()
        self.running = True
        self.menu_started = False
        self.game_started = False
        self.level_solved = False
        self.moves = -1
        self.squareFilled = 0
        self.gridSize = 5
        self.cellSize = 100
        self.pathWidth = 20
        self.bgColour = (0, 0, 0)
        self.gridColour = (255, 255, 255)
        self.windowWidth = 1102.5
        self.windowHeight = 735
        #loads the background image and scales bg image to fit the window
        self.morph = pygame.image.load("background6.png")
        self.bgImage = pygame.transform.scale(self.morph, (self.windowWidth, self.windowHeight))

        self.display = pygame.display.set_mode((self.windowWidth,self.windowHeight))
        pygame.display.set_caption(self.title)

    def main(self): #creates menu class and displays menu
        self.menu = Menu()
        self.menu.displayMenu()

    def quit(self): #quits the game
        pygame.quit()
        quit()
```

```python

# Menu class
class Menu:
    def __init__(self): #initialise variables
        self.leaderboard = None
        self.gameStart = False
        self.buttons = None
        self.bigButtonHeight = 60
        self.bigButtonWidth = 120
        self.smallButtonHeight = 30
        self.smallButtonWidth = 100

    def text_objects(self, text, font):
        textSurface = font.render(text, True, grey)
        return textSurface, textSurface.get_rect()

    def displayMenu(self):
        newGame.display.blit(newGame.bgImage,(0,0)) #display background image
        self.drawTitle()    #draw title
        self.createButtons() #create buttons
        while not newGame.menu_started: #main loop for menu
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.quit()
                    pygame.quit()
                for button in self.buttons:
                    button.drawButton() # draws button and checks if pressed

            pygame.display.update()
            newGame.clock.tick(30)
        self.levelSelection() # starts game when loop ends

    def levelSelection(self): #displays level selection menu
        newGame.display.blit(newGame.bgImage,(0,0))
        self.drawTitle()
        self.createButtons2()
        pygame.display.update()
        time.sleep(0.1)

        while not newGame.game_started: #main loop for level selection
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.quit()
                    pygame.quit()
                for button in self.buttons:
                    button.drawButton() # draws button and checks if pressed

            pygame.display.update()
            newGame.clock.tick(30)
        self.easyLevel()
```

```python

    def createButtons(self):
        self.buttons = [] #creates an array for the buttons to be stored in
        #Play button added to the array
        self.buttons.append(Button("Play", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.startMenu))
        #Leaderboard button added to the array
        self.buttons.append(Button("Leaderboard", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 400, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.showLeaderboard))
        #Quit button added to the array
        self.buttons.append(Button("Quit", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, newGame.quit))

    def createButtons2(self):
        self.buttons = [] #sets another an array for the buttons to be stored in
        #Easy level button added to the array
        self.buttons.append(Button("Easy- 5x5", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.startGame))
        #Medium level button added to the array
        self.buttons.append(Button("Medium- 7x7", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 400, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.mediumLevel))
        #Hard level button added to the array
        self.buttons.append(Button("Hard- 10x10", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.hardLevel))
        #Back button added to the array
        self.buttons.append(Button("Back", 150, 600, self.bigButtonWidth, self.smallButtonHeight, bgBlue, inactiveBlue, self.backButton))

    def createButtons3(self):
        self.buttons = [] #sets another an array for the buttons to be stored in
        #Undo button added to the array
        self.buttons.append(Button("Undo", (newGame.windowWidth/2) - (75/2) - 160, 650, 75, 30, bgBlue, inactiveBlue, self.undoMove))
        #Hint button added to the array
        self.buttons.append(Button("Hint", (newGame.windowWidth/2) - (75/2) + 160, 650, 75, 30, bgBlue, inactiveBlue, self.giveHint))
        #Reset level button added to the array
        self.buttons.append(Button("Reset", (newGame.windowWidth/2) - (75/2), 650, 75, 30, bgBlue, inactiveBlue, self.resetLevel))
        self.buttons.append(Button("Quit", 900, 200, 75, 75, bgBlue, inactiveBlue, newGame.quit))

    def createButtons4(self):
        self.buttons = []
        #Button the will generate a new level
        self.buttons.append(Button("Next Level", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 300, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, self.nextLvl))
        #Button that will take the user to the level selection screen
        self.buttons.append(Button("Level Selection", (newGame.windowWidth/2) - (self.bigButtonWidth/2) - 20, 400, 160, self.bigButtonHeight, bgBlue, inactiveBlue, self.levelSelection))
        #Button that will quit the program
        self.buttons.append(Button("Quit", (newGame.windowWidth/2) - (self.bigButtonWidth/2), 500, self.bigButtonWidth, self.bigButtonHeight, bgBlue, inactiveBlue, newGame.quit))

    def giveHint(self):
        print("Hint")

    def resetLevel(self):
        self.easyLevel()
```

```
146
147     def undoMove(self):
148         if self.undo == 1:
149             self.total -= self.length #Subtracts the length so that when "Undo" is pressed, the squares covered remains the same
150             newGame.moves -= 1 #Subtracts 1 from the "self.moves" variable so that the value of it does not increment when not wanted
151             print("Before", self.paths)
152 
153             pathRemoved = len(self.paths[-2]) #stores the length of the path that is 2nd to last in the list
154             self.total -= pathRemoved
155             self.paths.pop() #removes the last 2 elements in the list
156             self.paths.pop()
157 
158             print("after", self.paths)
159             self.undo -= 1
160         else:
161             self.total -= self.length
162             newGame.moves -= 1
163             print("No hint")
164 
165 
166 
167     def gameStats(self, moves): #displays the number of moves made by the player and the percentage of squares filled
168         pygame.draw.rect(newGame.display, bgBlue, (90, 195, 165, 70))
169         Font = pygame.font.Font("freesansbold.ttf", 25)
170         movesSurf = Font.render("Moves: " + str(moves), True, black)
171         pathsSurf = Font.render("Filled: "+ str(newGame.squareFilled) + "%"  , True, black)
172         movesRect = movesSurf.get_rect()
173         pathsRect = pathsSurf.get_rect()
174         movesRect.topleft = (100, 200)
175         pathsRect.topleft = (100, 230)
176         newGame.display.blit(movesSurf, movesRect)
177         newGame.display.blit(pathsSurf, pathsRect)
178 
179     def drawTitle(self): #draws the title
180         titleFont = pygame.font.Font("freesansbold.ttf", 100)
181         titleSurf, titleRect = self.text_objects("Connect The Dots", titleFont)
182         titleRect.center = ((newGame.windowWidth/2), (newGame.windowHeight/7))
183         newGame.display.blit(titleSurf, titleRect)
184 
185     def startMenu(self): #breaks out of the initial menu loop
186         newGame.menu_started = True
187 
188     def startGame(self): #breaks out of the level selection loop
189         newGame.game_started = True #sets gameStart to true
190 
191     def backButton(self): #sets the program back to the inital menu screen
192         newGame.menu_started = False
193         self.displayMenu()
194 
195     def showLeaderboard(self):
196         print("Leaderboard button pressed")
197 
198     def nextLvl(self):
199         self.easyLevel()
```

```
200 
201     def easyLevel(self): #the main block of code for the game
202         newGame.display.blit(newGame.bgImage,(0,0))
203         self.drawTitle()
204         self.createButtons3()
205         newGrid = Grid(5, 5, 90, 30) #creates a grid object with 5 rows, 5 columns, cell size of 90, path width of 30
206         self.drawing = False
207         self.currentPath = []
208         self.paths = []
209         self.total = 0
210         self.length = 0
211         newGame.moves = -1
212         self.undo = 1
213         self.level = True
214 
215 
216         while self.level:
217             newGrid.createGrid()
218 
219             if newGrid.solvePuzzle(newGrid.pathFindPuzzle): #runs the solvePuzzle function on the pathFindPuzzle and checks if it is solvable
220                 print("Solved")
221             else:
222                 print("Not solved")
223 
224             pygame.display.update()
225             while newGame.running: #main loop for the game
226                 self.gameStats(newGame.moves)
227                 for event in pygame.event.get():
228                     if event.type == pygame.QUIT:
229                         newGame.running = False
230 
231                     elif event.type == pygame.MOUSEBUTTONDOWN: #checks if the mouse is clicked
232                         self.drawing = True
233                         mousePos = pygame.mouse.get_pos() #gets the mouse position and then converts it to a grid position
234                         gridPos = newGrid.getCellFromMouse(mousePos)
235                         self.currentPath.append(gridPos) #appends the grid position to the current path
236                         self.paths.append(self.currentPath)
237 
238                     elif event.type == pygame.MOUSEBUTTONUP: #checks if the mouse is released
239                         self.drawing = False
240                         self.currentPath = []
241                         self.total += self.length
242                         newGame.moves += 1
243                         newGame.squareFilled = ((self.total * 100) // (newGrid.rows * newGrid.cols)) #returns a precentage value on how many squares are filled
244 
245                     elif event.type == pygame.MOUSEMOTION and self.drawing: #checks if the mouse is moving
246                         mousePos = pygame.mouse.get_pos()
247                         gridPos = newGrid.getCellFromMouse(mousePos)
248                         if gridPos != self.currentPath[-1]: #if the grid position is not the same as the last grid position
249                             self.currentPath.append(gridPos)
250                         self.length = len(self.currentPath)
251 
252                     for button in self.buttons:
253                         button.drawButton()
254 
255                 pygame.display.flip() #updates the display
256 
```

```
                #This will block of code is responsible for drawing the path
                if self.drawing and self.currentPath: #will only draw the path if the player is drawing and there is a path to draw
                    for i in range(len(self.currentPath) - 1): #loops through the current path
                        pathColour = 0 #sets the path colour to 0 initially
                        if gridPos == None:
                            self.drawing = False
                            print("Invalid move")
                            newGame.menu.total -= self.length
                            newGame.moves -= 1
                        else:
                            dotNum = newGrid.getDotColour(gridPos[1], gridPos[0])
                            if dotNum == 2:
                                pathColour = red
                            elif dotNum == 3:
                                pathColour = blue
                            elif dotNum == 4:
                                pathColour = green
                            elif dotNum == 5:
                                pathColour = yellow
                            elif dotNum == 6:
                                pathColour = orange

                            startPos = ((self.currentPath[i][0] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.xOffset, (self.currentPath[i][1] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.yOffset)
                            # stores the start position of line
                            endPos = ((self.currentPath[i+1][0] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.xOffset, (self.currentPath[i+1][1] * newGrid.cellSize + newGrid.cellSize // 2) + newGrid.yOffset)
                            # stores the end position of line
                            pygame.draw.line(newGame.display, pathColour, startPos, endPos, newGrid.pathWidth) # draws the line

                if self.total == (newGrid.rows * newGrid.cols): #if the total is equal to the number of squares in the grid
                    newGame.running = False
                    self.level = False #set the level to false so the player can move on to the next level

                    self.solved()

    def solved(self): #displays the level solved screen
        newGame.display.blit(newGame.bgImage, (0,0))
        self.drawTitle()
        self.createButtons4()
        font = pygame.font.Font(None, 52)
        text = font.render("Level Solved!", True, white)
        newGame.display.blit(text, (450, 200))
        pygame.display.update()
        while not newGame.level_solved:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    newGame.quit()
                    pygame.quit()

                newGame.game_started = False #set these 3 flags as true so the player can go back to previous screens without error
                newGame.running = True
                newGame.menu.level = True

                for button in self.buttons:
                    button.drawButton()

            pygame.display.update()
```

```
            self.total = 0
            newGame.moves = -1

    def mediumLevel(self):#this wont work unless puzzle has 7 rows and columns
        print("Medium Level works")

    def hardLevel(self):
        print("Hard Level works")
```

```
# Button class
class Button:
    #creates buttons with text, x, y, width, height, colour, action
    def __init__(self,text, x, y, w, h, inactive_colour, active_colour, action):
        self._text = text
        self._x = x                     #coordinates of top left corner of button
        self._y = y
        self._w = w                     #button dimensions
        self._h = h
        self._inactive_colour = inactive_colour
        self._active_colour = active_colour
        self._action = action           #function to be called when button is clicked

    def drawButton(self):
        mouse = pygame.mouse.get_pos() #gets mouse position
        click = pygame.mouse.get_pressed() #gets mouse click

        if self._x + self._w > mouse[0] > self._x and self._y + self._h > mouse[1] > self._y: #checks if mouse is over button
            pygame.draw.rect(newGame.display, self._active_colour, (self._x, self._y, self._w, self._h)) #draws button in active colour
            if click[0] == 1 and self._action != None: #checks if button is clicked
                self._action() #calls the function associated with the button
        else:
            pygame.draw.rect(newGame.display, self._inactive_colour, (self._x, self._y, self._w, self._h)) #otherwise draws button in inactive colour

        smallText = pygame.font.Font("freesansbold.ttf", 20) #creates font, text and shape for button
        textSurf = smallText.render(self._text, True, black)
        textRect = textSurf.get_rect()

        textRect.center = ((self._x + (self._w/2)), (self._y +(self._h/2))) #displays and centers text on button
        newGame.display.blit(textSurf, textRect)

#Grid class
class Grid:
    def __init__(self, rows, cols, cellSize, pathWidth):
        self.rows = rows
        self.cols = cols
        self.cellSize = cellSize
        self.pathWidth = pathWidth
        self.xOffset = (newGame.windowWidth - self.cols * self.cellSize) // 2
        self.yOffset = (newGame.windowHeight - self.rows * self.cellSize) // 2 + 50 # +50 to offset, to account for the change in grid position

        self.puzzle = []
        self.startNodes = {} #creates a dictionary for the start nodes
        self.endNodes = {} #creates a dictionary for the end nodes
        self.allNodes = [] #creates an array for all the nodes
        self.distances = {} #creates a dictionary for the distances
        self.sortedNodes = [] #creates an array for the sorted nodes
        self.pathFindPuzzle = [] #creates a path finding puzzle
```

```
371
372     def makePuzzle(self):
373         self.puzzle = [[0,0,0,0,0]] #creates a puzzle with 5 rows and 5 columns that gets random values assigned to it
374         self.puzzle.append([0,0,0,0,0])
375         self.puzzle.append([0,0,0,0,0])
376         self.puzzle.append([0,0,0,0,0])
377         self.puzzle.append([0,0,0,0,0])
378 
379         #generates random coordinates for the dots to be draw in from the list of coordinates
380         coords = [(0,0), (0,1), (0,2), (0,3), (0,4), (1,0), (1,1), (1,2), (1,3), (1,4), (2,0), (2,1), (2,2), (2,3), (2,4), (3,0), (3,1), (3,2), (3,3), (3,4), (4,0), (4,1), (4,2), (4,3), (4,4)]
381         for i in range(2,7):
382             coord1 = random.choice(coords) #chooses a random coordinate from "coords" then removes it from the array
383             x,y = coord1 # x and y are set to the values of the coordinate
384             coords.remove(coord1)
385 
386             coord2 = random.choice(coords)
387             w,z = coord2
388             coords.remove(coord2)
389 
390             self.puzzle[y][x] = i #changes the value held at the y,x postion to a sigular value so that the dots can be drawn
391             self.puzzle[z][w] = i
392 
393         self.pathFindPuzzle = [[1,1,1,1,1,1,1]] #creates a path finding puzzle by adding 1's to the puzzle so that the there is a "barrier" around the grid
394         self.pathFindPuzzle.append([1, self.puzzle[0], 1])
395         self.pathFindPuzzle.append([1, self.puzzle[1], 1])
396         self.pathFindPuzzle.append([1, self.puzzle[2], 1])
397         self.pathFindPuzzle.append([1, self.puzzle[3], 1])
398         self.pathFindPuzzle.append([1, self.puzzle[4], 1])
399         self.pathFindPuzzle.append([1,1,1,1,1,1,1])
400 
401 
402 
403     def returnSolved(self, total): #checks if the puzzle is solved and returns a boolean value
404         if total == 25:
405             return True
406         else:
407             return False
408 
409     def abs(self, a , b): #returns the absolute value of the difference between two numbers
410         if a > b:
411             return a - b
412         else:
413             return b - a
```

```
415     def solvePuzzle(self, grid): #solves the puzzle using a recursive backtracking algorithm
416         if self.checkGrid(grid) == False: #checks if the grid is valid
417             return False
418 
419         if self.returnSolved(grid): #checks if the puzzle is solved
420             return True
421 
422         for dotNum in self.sortedNodes: #loops through the sorted nodes
423             start = self.startNodes[dotNum]
424             end = self.endNodes[dotNum]
425 
426             if (abs(end[0], start[0]) + abs(end[1], start[1])) > 1: #checks if the distance between the start and end nodes is greater than 1
427                 directions = []
428                 if grid[start[0]][start[1] + 1] == 0: #checks if the value to the right of the start node is 0
429                     if end[1] > start[1]: #checks if the end node is to the right of the start node
430                         directions.insert(0, "right") #inserts "right" at the start of the directions array
431                     else:
432                         directions.append("right")  #appends "right" to the end of the directions array
433 
434                 if grid[start[0]][start[1] - 1] == 0: #checks if the value to the left of the start node is 0
435                     if end[1] < start[1]: #checks if the end node is to the left of the start node
436                         directions.insert(0, "left") #inserts "left" at the start of the directions array
437                     else:
438                         directions.append("left") #appends "left" to the end of the directions array
439 
440                 if grid[start[0] + 1][start[1]] == 0: #checks if the value below the start node is 0
441                     if end[0] > start[0]:
442                         directions.insert(0, "down")
443                     else:
444                         directions.append("down")
445 
446                 if grid[start[0] - 1][start[1]] == 0: #checks if the value above the start node is 0
447                     if end[0] < start[0]:
448                         directions.insert(0, "up")
449                     else:
450                         directions.append("up")
451 
452                 if len(directions) == 0: #checks if the length of the directions array is 0
453                     return False
454 
```

```
                for direction in directions:
                    if direction == "right": #checks if the direction is right
                        start[1] += 1 #increments the x value of the start node
                        grid[start[0]][start[1]] = dotNum #sets the value at the start node to the dot number
                        if self.solvePuzzle(grid) == True: #recursively calls the solvePuzzle function
                            return True
                        else:
                            grid[start[0]][start[1]] = 0 #sets the value at the start node to 0
                            start[1] -= 1 #decrements the x value of the start node

                    elif direction == "left": #checks if the direction is left
                        start[1] -= 1 #decrements the x value of the start node
                        grid[start[0]][start[1]] = dotNum #sets the value at the start node to the dot number
                        if self.solvePuzzle(grid) == True: #recursively calls the solvePuzzle function
                            return True
                        else:
                            grid[start[0]][start[1]] = 0 #sets the value at the start node to 0
                            start[1] += 1 #increments the x value of the start node

                    elif direction == "up": #checks if the direction is up
                        start[0] -= 1
                        grid[start[0]][start[1]] = dotNum
                        if self.solvePuzzle(grid) == True:
                            return True
                        else:
                            grid[start[0]][start[1]] = 0
                            start[0] += 1

                    elif direction == "down": #checks if the direction is down
                        start[0] += 1
                        grid[start[0]][start[1]] = dotNum
                        if self.solvePuzzle(grid) == True:
                            return True
                        else:
                            grid[start[0]][start[1]] = 0
                            start[0] -= 1
                return False

    def createGrid(self): #will create the grid and draw the dots
        self.makePuzzle() #creates the puzzle

        #sorts the nodes by distance so that the shortest distance is first
        keys = list(self.distances)
        for i in range(0, len(self.distances)):
            min = self.distances[keys[0]]
            for key in self.distances:
                if self.distances[key] < min:
                    min = self.distances[key]
            self.sortedNodes.append(key)
            self.distances.pop(key)

        for row in range(self.rows):
            for col in range(self.cols):

                #finds the start and end nodes that need to be connected
                if self.puzzle[row][col] > 0:
                    if self.puzzle[row][col] in self.startNodes:
                        self.endNodes[self.puzzle[row][col]] = (row, col)
                        self.distances[self.puzzle[row][col]] = abs(row - self.startNodes[self.puzzle[row][col]][0]) + abs(col - self.startNodes[self.puzzle[row][col]][1])
                    else:
                        self.startNodes[self.puzzle[row][col]] = (row, col)
                    self.allNodes.append((row, col))


                #draws grid and dots with colours
                rect = pygame.Rect(self.xOffset + col * self.cellSize, (self.yOffset + row * self.cellSize) , self.cellSize, self.cellSize) #creates rectangle for grid
                pygame.draw.rect(newGame.display, white, rect, 1) #draws grid lines
                if self.puzzle[row][col] != 0 or 1: #if the dot label is not 0, draw a dot
                    circle_centre = (self.xOffset + col * self.cellSize + self.cellSize // 2, ((self.yOffset + row * self.cellSize) ) + self.cellSize // 2)
                    circle_radius = 25
                    if self.puzzle[row][col] == 2: #if dot label is 1, draw red dot, if 2 draw blue dot, etc.
                        pygame.draw.circle(newGame.display, red, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 3:
                        pygame.draw.circle(newGame.display, blue, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 4:
                        pygame.draw.circle(newGame.display, green, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 5:
                        pygame.draw.circle(newGame.display, yellow, circle_centre, circle_radius)
                    elif self.puzzle[row][col] == 6:
                        pygame.draw.circle(newGame.display, orange, circle_centre, circle_radius)
```

```
536
537     def checkGrid(self, grid): #checks if the grid is valid
538         for row in range(self.rows):
539             for col in range(self.cols):
540                 if grid[row][col] > 0: #checks if the value at the grid position is greater than 0
541                     dotNum = grid[row][col]
542                     if grid[row + 1][col] > 0 and grid[row + 1][col] != dotNum: #checks if the value below the current grid position is greater than 0 and not equal to the dot number
543                         if grid[row - 1][col] > 0 and grid[row - 1][col] != dotNum: #checks if the value above the current grid position is greater than 0 and not equal to the dot number
544                             if grid[row][col + 1] > 0 and grid[row][col + 1] != dotNum: #checks if the value to the right of the current grid position is greater than 0 and not equal to the dot number
545                                 if grid[row][col - 1] > 0 and grid[row][col - 1] != dotNum: #checks if the value to the left of the current grid position is greater than 0 and not equal to the dot number
546                                     return False
547                     if grid[row + 1][col] == dotNum and grid[row - 1][col] == dotNum and grid[row][col + 1] == dotNum: #checks if the value below, above and to the right of the current grid position is equal to the dot number
548                         return False
549                     elif grid[row + 1][col] == dotNum and grid[row - 1][col] == dotNum and grid[row][col - 1] == dotNum: #checks if the value below, above and to the left of the current grid position is equal to the dot number
550                         return False
551                     elif grid[row + 1][col] == dotNum and grid[row][col + 1] == dotNum and grid[row][col - 1] == dotNum: #checks if the value below, to the right and to the left of the current grid position is equal to the dot number
552                         return False
553                     elif grid[row - 1][col] == dotNum and grid[row][col + 1] == dotNum and grid[row][col - 1] == dotNum: #checks if the value above, to the right and to the left of the current grid position is equal to the dot number
554                         return False
555         return True
556
557     def getDotColour(self, row, col): #returns the number of the dot when looking at the puzzle
558         return self.puzzle[row][col]
559
560     def getCellFromMouse(self, pos): #returns the cell position of the mouse
561         x, y = pos
562         self.xOffset = int(self.xOffset) #sets xOffset to an integer as the window dimensions are floats
563         row = (y - self.yOffset) // self.cellSize
564         col = (x - self.xOffset) // self.cellSize
565         if row < 0 or col < 0 or row > 4 or col > 4:
566             return None
567         return (col, row) #returns the cell position
568
569
570
571
572
573 # Create an instance of the Game class and call the main method
574 newGame = Game()
575 newGame.main()
```