Computing bounds on chromatic numbers using Java

Artur Oganyan, Rinat Ishmaev, Diana Grecu, Filippos Leonidas

Nathan Bouquet, Mikolaj Gawrys

(Group 23)

23 January, 2023

Abstract

Graphs are one of the foundations of mathematics and computing. Graphs can be used in almost all real-world applications, such as transport networks, media marketing and search engine algorithms. They may even be used to determine this project's examination because they provide structure in performance assessing, just showing their endless utilities. Given its significant role in applied mathematics and computing it is essential to be able to develop and understand the software and algorithms that are used to analyse and manipulate graphs. They also play a prominent role in the visualisation of data which helps to understand large amounts of data and their trends/relationships.

Graph[4] colouring is studied because it is an interesting field of graph theory and an optimization problem that can be applied to many aspects of everyday life. Indeed, it can be used for the guarding of an art gallery problem[1], the travelling salesman problem[1], or even scheduling a specific number of classes that cannot overlap for a certain number of students[1].

In phase 1 we studied the graph colouring problem[6] in which a graph is generated with an arbitrary number of vertices and edges (connections) and we were tasked with computing a chromatic number[7] for this graph under the condition that adjacent vertices (connected by one edge) are not coloured the same. In phase 2, we studied how to create a game with these chromatic number algorithms[7] and implemented this into the game[8]. In phase 3, we optimised our existing algorithms and we additionally created new ones capable of dealing with larger graphs.

The main outcomes of this study are learning as a team how to form solid algorithms to tackle the graph colouring problem[4], how to implement them into a game[8] to be played and how to optimise these algorithms for them to run efficiently.

The findings from this study leave us with multiple algorithms used for graph colouring implemented into a game[8]. The graph colouring algorithms[6] are very useful in multiple real-world scenarios such as scheduling[5], data mining[1], clustering[5] and networking[1] as it allows resources used to be optimized in order to ensure a more efficient working system[1]. The application of the theory (algorithms) to real-life (interactive games) shows exactly how useful these techniques are for real-world problems.

Introduction

A graph is a mathematical structure used to display numerical data, statistics, and relationships between objects. They consist of a set of vertices and a set of edges connecting them. The vertices represent the objects and the edges the bonds between them. Graphs are useful in visually modelling a vast range of real-world systems (e.g. school schedules, transportation routes, interdependencies between different tasks and social connections). In the subject of graph theory, the graph colouring[4] problem is considered one of the most famous problems in that field. The problem entails that for any graph given, how can the vertices be coloured so that no adjacent (connected by the same edge) vertices are coloured with the same colour and that the number of colours used is mitigated (chromatic number[7]).

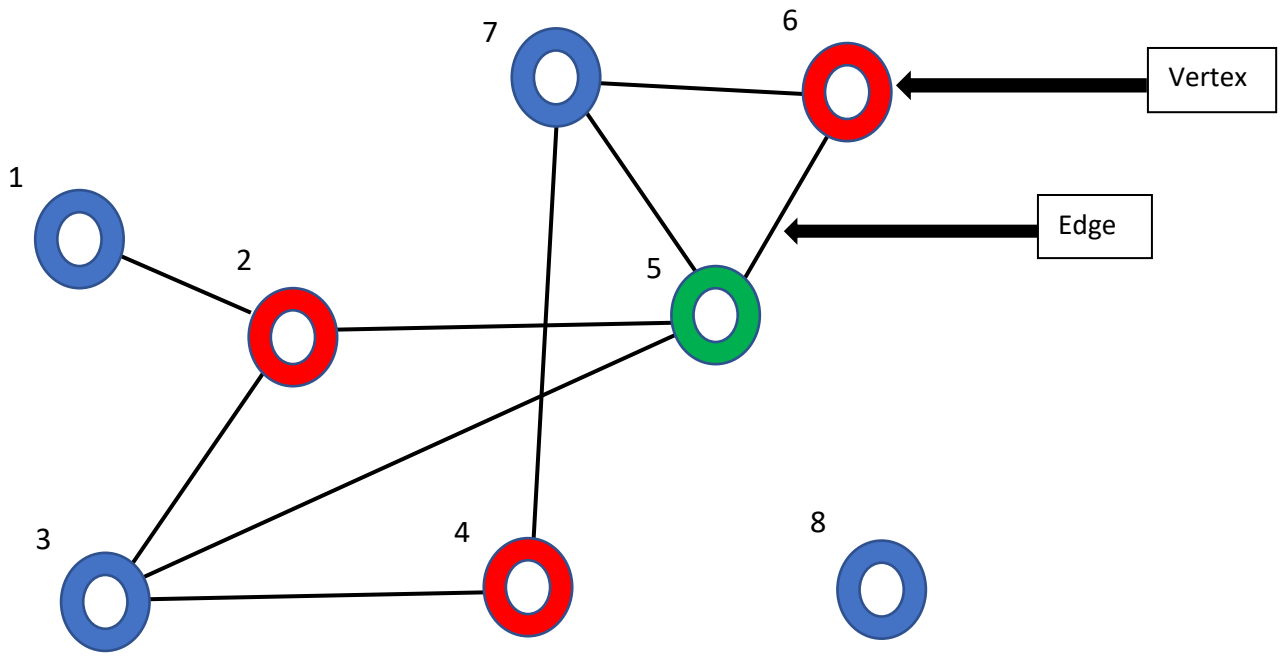


Figure 1

In the figure above (Fig. 1) there is a graph with 7 vertices (hollow circles) and 9 edges (lines connecting the hollow circles). This is an example of how to optimally colour a graph using only 3 different colours. This solution is one of "proper" colouring as all the adjacent vertices are coloured differently.

Vertices 5, 6 and 7, and vertices 2, 3 and 5, are both cliques. A clique is a subset of vertices where every two vertices are connected by an edge (fully connected subgraph). Cliques are found in undirected graphs (where the edges do not have a directional character) and unweighted graphs (where the data structure of the graph are not associated with any weight/value). Cliques are useful in efficiently calculating the upper and lower bounds for the chromatic number because the maximum clique contains the maximum number of colours required to optimally colour the graph. This means that less computing power is required to calculate these values as only this clique is analysed instead of iterating through all the other vertices in the graph.

The act of colouring a graph optimally has very little to do with just colouring vertices, it has more to do with using this optimization technique[6] to solve other problems where there are limited resources and restrictions. The colours represent the variable we are trying to optimize and mitigate wasted resources.

In phase 1 the questions we wanted to answer were what type of brute force technique we wanted to use and whether the chromatic number[7] could be bigger than the size of the maximum clique. For phase 2 we asked ourselves how we could display a graph to mitigate the overlapping of the edges between vertices, how we could make the game more accessible for visually impaired players and how to make the game feel engaging and realistic. In phase 3 we asked ourselves how we could optimize the brute force and the lower bound algorithm[12] and introduce a technique to make computing these values more efficient.

As a starting point, research has been conducted, in order to explore possible algorithms suitable for our problem. Those algorithms include DSATUR, Bron-Kerbosch[9] algorithm, Greedy algorithm and Brook's Theorem. We chose the Bron-Kerbosch and Greedy[11] algorithm to start with, since they are simple to implement and tweak to preference.

The report includes sections such as: methods and implementation, where the reader can understand what is behind the approach and how was it carried out throughout the research. In the results section, the reader will find tables and comparisons for various computations executed by the algorithms. In the discussion and conclusion sections, one may find analysis of the results and the questions that follow, along with the summary of the research respectively.

Methods

Phase 1 algorithms

Lower bound algorithm –
Our lower bound algorithm[12] finds the number of vertices of in the maximum clique[10] which would therefore give us the lower bound. We used a recursive method that given a two-dimensional array and the number of edges and vertices of that graph would give us the number of vertices of every clique. Our algorithm looks for the biggest connection that a vertex has with all the other vertices.

Upper Bound algorithm –
The upper bound[13] algorithm locates the vertex with the highest degree of edges in a maximal clique (Fig. 2)[10]. This means that the chromatic number for that clique and subsequently the rest of the graph was the number of edges plus one to account for the original vertex chosen.

(Upper Bound = d+1, where d is the number of edges from a given vertex)

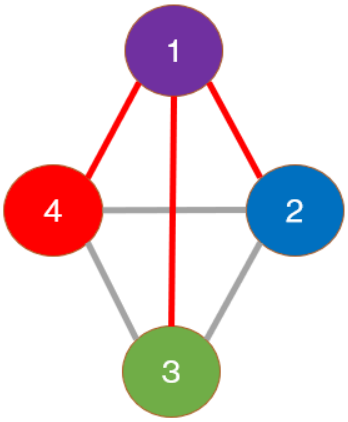


Figure 2 – This figure shows a clique where the upper bound is 4 and the highest degree of edges is 3.

```
Program to generate the Upper Bound
Initialize: new arraylist for numbers that repeat
        New Hashset for numbers in connections
Start of the first For Loop:
    Start of the second For Loop:
        If the element in equivalent to k Then the counter increases by 1
        Add counter to the array
    End of the second For Loop
    Return the maximum value+1 within the array
End of the first For Loop
```

Brute Force –
The brute force[16] algorithm solves problems through exhaustion. The number of operations taken to complete a task generally correlates with the input size. Exhaustive algorithms tend to be consistent and straightforward. Our algorithm goes through all possible colour variations for the vertices, testing whether 'n' number of colours is suitable to colour the graph optimally (making sure no two adjacent vertices have the same colour. The value 'n' starts from 1 and is tested up until n equals the chromatic number. The algorithm first randomly picks a vertex in the graph, and then assigns a colour to that chosen array of vertices. The algorithm returns true if the next vertex can share the same colour and false if not, it will then pass through the whole graph and output the chromatic number when all vertices are true.

```
Program to generate the Brute Force

Start Method to create a 2D Array

Initialize: new Arraylist for partitions

Start For Loop for every numbers that traverse through the list at numbers+2 increments

        Add a sublist from numbers index to either the minimum values of either the index+2 or the size
        of the numbers list

End For Loop

Initialize: new 2D Integer Array with the amount of its rows equaling the partitions size

Start For Loop for all partitions

        Initialize: Integer List with the length of the row equal at the index at partition's list

        If the array's element at index equals value of index at partitions list

        Then Initialize: new 1D Integer Array with dimension the size of the list row

End For Loop

End Method
```

```
Start Method to create a 2D Integer array graph

Initialize 2D integer array graph with the rows and columns equal to the amount of vertices

Start first For Loop with the vertices length

        Start second For Loop with the edges length

                graph rows equals to the vertices' s value at index

                graph columns equals to the edges' s value at index

        End second For Loop

End first For Loop

Start For Loop with the edges length

        graph at row the edges element at index 0 – 1 and column at edges element at index 1 – 1
        equals 1

        graph at row the edges element at index 1 – 1 and column at edges element at index 0 – 1
        equals 1

End For Loop

Return graph

End Method
```

```
Start Method for coloring the graph

Initialize: Array colors for the amount of colors used

//free pick color for first vertex

Return the number of colors used after the recursive call for checking the graph's vertices colors, the array being always updated

End Method
```

```
Method for helping the graph coloring

Start For Loop

        If it is possible to color the vertex with that color then we are adding the color to the array

                If vertex+1 is less than the length of colors

                        If not last vertex recursively checks the graph's colors for next vertex return true

                                If colors at index equals 0, then the color is not needed Else return true

End For Loop

Return false

End Method
```

```
Method that checks whether adjacent vertices share same color

Start For Loop with the amount of colors

        If graph at index is equivalent to 1 AND counter is equivalent to colors at index Then return false

        Else return true

End For Loop

End Method
```

Pruning unconnected vertices[14] - We concluded after testing the brute force algorithm[16] on the example graphs that the more vertices and edges the graph had, the more time it would take to compute chromatic[7] number from our brute force algorithm. Furthermore, it would also help for our lower[12] and upper bound algorithms[13]. To optimize these algorithms, we implemented a pruning method[2]. This method locates vertices with no edges as these vertices do not affect the chromatic number[7] and slow down the code's running time. Therefore, the selected vertices are eliminated, and the brute force algorithm doesn't have to take them into consideration as they don't affect the chromatic number[7]. This allows the code to run faster as it doesn't have to check if those vertices are to be coloured or not. The input for the algorithm is our adjacency matrix. The algorithm loops through it and checks if a complete line is filled with 0, which would mean this vertex is not connected to any other vertex. If it is, we take this line out of the 2D array. The output is the adjacency matrix pruned.

```
Program for Graph Pruning

Initialize counter 1 with 0

        counter 2 with 0

        length with the graph length-1

Start While Loop until the length is smaller than 0

    Start For Loop

        If the graph's index is equivalent to 0 then counter 1 increases by 1

    End For Loop

    Start If counter 1 is equivalent to the graph length then counter 2 increases by 1

        Start first For Loop from length to the graph length-1

            Start second For Loop

                If graph's element at index equals to the graph's element on the next line

                 Then length decreases by 1

            End second For Loop

        End first For Loop

    End If

End While

If counter 2 doesn't equal 0

Then initialize: new 2D integer Array with the rows and the columns  being the length of the graph –
counter 2

    Start first For Loop

        Start second For Loop

            If the first graph's element at the index is equal to the second graph's element
        at index Then return second graph
            Else return first graph

        End second For Loop

    End first For Loop

End If
```

Phase 3 algorithms

Bron-Kerbosch Algorithm[9] –
This algorithm lists all the maximal cliques[10] in graphs where the edges connecting the vertices have no directional character (undirected). A maximal clique is a subset of vertices where each pair of vertices is connected by an edge, where no additional vertices can be added to the subset, conserving its absolute connectivity.

Adjacency matrix algorithm[5] –
The new adjacency matrix is one method which loops over the entirety of the graph. We first create a two-dimensional byte array with both its length and height each equalling the number of vertices in the graph. To make the algorithm more efficient we used the data type byte, as it requires less computational power than integers. This array is then filled with zeros (0) when there are no connections between two vertices and a one (1) when there is a connection.

We loop through the graph, using a "for each loop" which increments each connection of the text graph file. When two indexes are connected, we add a one to the 2D array at position [edge(u) - 1][edges(v) - 1] and at position [edge(v)-1][edges(u) - 1] as the adjacency matrix is symmetric diagonally.
This algorithm is in our main method of our "ReadGraph" class. The input for the algorithm is the text file graph. And the output is a 2D byte array.

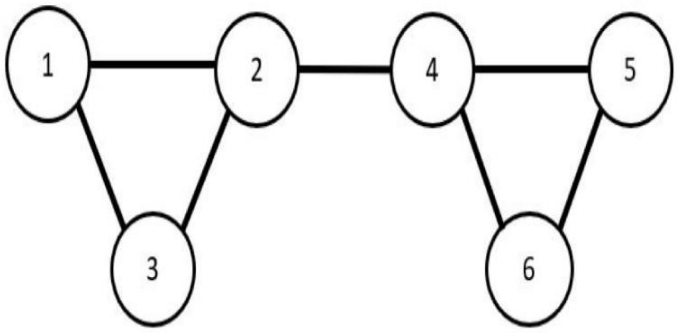


Figure 3

The adjacency matrix for this graph is:

Vertex	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	0	0
3	1	1	0	0	0	0
4	0	1	0	0	1	1
5	0	0	0	1	0	1
6	0	0	0	1	1	0

Figure 4

Brute force algorithm[16] –
1) To count how many edges are connected to each vertex of the graph, we created two loops which go through our adjacency matrix[5]. Each connection that vertices have are then added to a hash-set, called connections, which has two parameters: an integer, which is the number of the vertices, and an arraylist, where you can find the vertices connected to the first parameter (the number of a vertex). Then using the size method, we can count how many edges are connected to each vertex.

2) To calculate the chromatic number, we created the method "colour", which is a greedy algorithm[11]. The method has five parameters:

- an integer array named 'colours' which is the colour each vertex is
- an integer 'vertex' which is the current iteration of the vertices
- our hash-map connections with two parameters: an integer (the number of the vertices) and an arraylist, where you can find the vertices connected to the first parameter (the number of the vertex)
- an integer colour which is the colour we are trying to give a vertex
- and an integer ArrayList sorted which is the order from biggest to smallest number of connections each vertex has.

Our algorithm checks if we can colour the vertex using the 'CanColor' method which returns a boolean true or false indicating whether it is possible to be coloured with that specific colour selected.
Lower bound algorithm[12] –
To calculate a lower bound for the chromatic number[7], the algorithm must compute the size of the largest maximal clique in the graph. This is done by locating the vertex with the highest degree of edges. After this "starting vertex" is located, we then check if any of the other vertices in the graph are connected to it. We then iterate through all the vertices connected with the "starting vertex" and check if they are connected to the any other vertices also connected with the "starting vertex". If the vertices are connected, they are considered part of the clique[3], if not

they are disregarded. We then use a counter to sum up the number of vertices in this connected clique and use that as our lower bound.

Pruning chains[2] - The pruning algorithm removes vertices of the graph that have only one connection. We created a method prune which takes into account two parameters, our adjacency matrix[5] (2D byte array "adjacencyMatrix") and the number of vertices of the graph (integer "vertices"). The method checks firstly if the number of vertices is greater than 3, as chain graphs can be pruned till they have zero vertices. If it is, it loops through our 2D array, initialising two integer variables occurrences and vertex at 0, and checks if each index is equal to 1, which means there is a connection. If there is, it increments our variable occurrences by 1, and vertex is given the value of the second for loop. When the algorithm has gone through all the width of the array, it checks if the occurrences are equal to 1, which means the vertex has only one connection. We then change its value in the adjacency matrix to 0 at index [i][vertex] and [vertex][i], which means there is no connection and break.
We then call our method outside the for loops but in the first if statement until all the vertices connected by only one edge are pruned, but we decrement the number of vertices by 1.
The output is the adjacency matrix pruned.

GUI (Graphical User Interface) - Coded in JavaFX

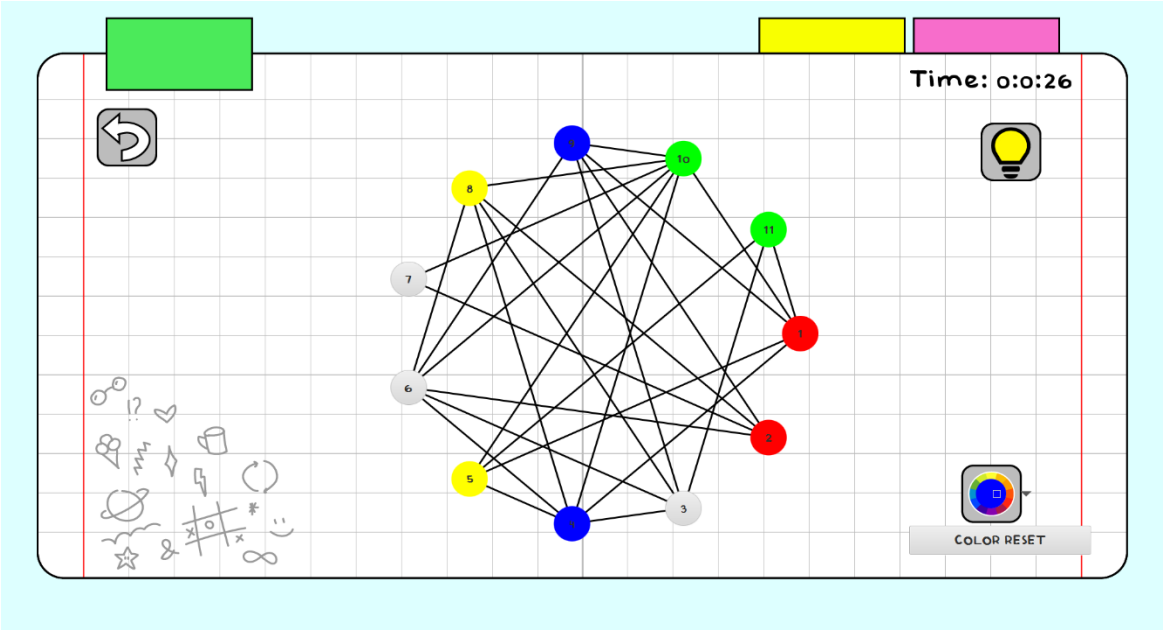


Figure 5

Random Graph generator - The random graph generator algorithm[15] works by firstly placing our centre coordinates in the middle of the screen, being x: 960 and y: 540. The graphs radius is calculated by multiplying the number of nodes by twenty (n*20). All the nodes are then positioned in a polygonal shape using cosine and sine functions to calculate their coordinates. Every vertex positioned, is given a number. Finally, using the agency matrix the edges are positioned between the nodes.

Hint button (Fig. 6) - The hint button's purpose is to aid the player when they are stuck. It also makes the game interactive. In all game modes, when pressing the hint button once, a post note pops up. It reveals the vertex to be coloured first, which is the vertex with the most connection. On the second click the user will either get told they are on track, if the colours used are less than or equal to the chromatic number[7], or they will get told to use less colours, if the number of colours used are more than the chromatic number. On the third and last click the chromatic number is revealed to the player.

Figure 6

Timer (Fig. 7) - The timer works by displaying the hours, minutes, and seconds in the top right corner of the game screen. The timer starts when a game mode button is selected. It works by increasing a variable by 1 as every new frame is created. The code adds a new frame every second so therefore every time the frame changes, the count (for seconds increases) increases. Every time the seconds reaches 60, the counter resets and adds 1 to the minute counter, and the same applies for the minutes and hours counters.

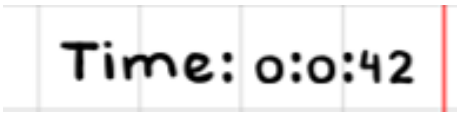


Figure 7

Game mode Selection (Fig. 8) - We have three buttons on the left of the screen which are labelled to the three different game modes possible to play. When one button is selected, it changes shade and adds a border to indicate selection, then the program deletes the other two buttons to remove them from the screen. When unselected, the button returns to its original shade and the other game mode buttons are re-created with the same functions.

In the game mode "to the bitter end" the player must complete the graph with the chromatic number, which is the least number of colours possible to colour the graph, in the least amount of time possible. The game finishes when the graph is completed with the chromatic number or when the player quits the game.

In the game mode "timerunner" the player must complete the graph with the best upper bound possible in a fixed time frame, which we set to 5 seconds per edge and 1 second per vertex in the graph.

Figure 8

In the game mode "colour queue" randomly orders the vertices (provided by the method *Collections.shuffle()* which randomly shuffles an arraylist of the number of vertices in the graph), and the player needs to fully colour the graph to finish the game. When the player colours a vertex they cannot change that colour, which is what makes this game mode engaging as it requires more intuition to complete.

Back Button (Fig. 9) - This button is intended for the user to click when they want to exit the game mode and go back to the previous window.

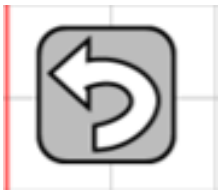


Figure 9

Colour Selector (Fig. 10) - The colour selector enables the user to colour the vertices using the class 'ColorPicker'. The chosen colour can then later be read from the 'ColorPicker'. When pressed, the colour palette pops up. Under the colour selector there is the reset button which enables the player to reset the graph colours to null, meaning they can now be recoloured.

Figure 10

Implementation

UML Diagram: The purpose of a Unified Modelling Language diagram is to visually represent the GUI system along with its main and abstract classes, algorithms, and actions. This allows for a better understanding when it comes to altering, maintaining the system. We also use this graph in our presentation as someone without prior knowledge of the subject can grasp how the system operates.

The UML diagram below (Fig. 11) represents the internal logic of our program. It underlines the most important sections and assists in the understanding of how all the components work simultaneously to deliver the GUI.

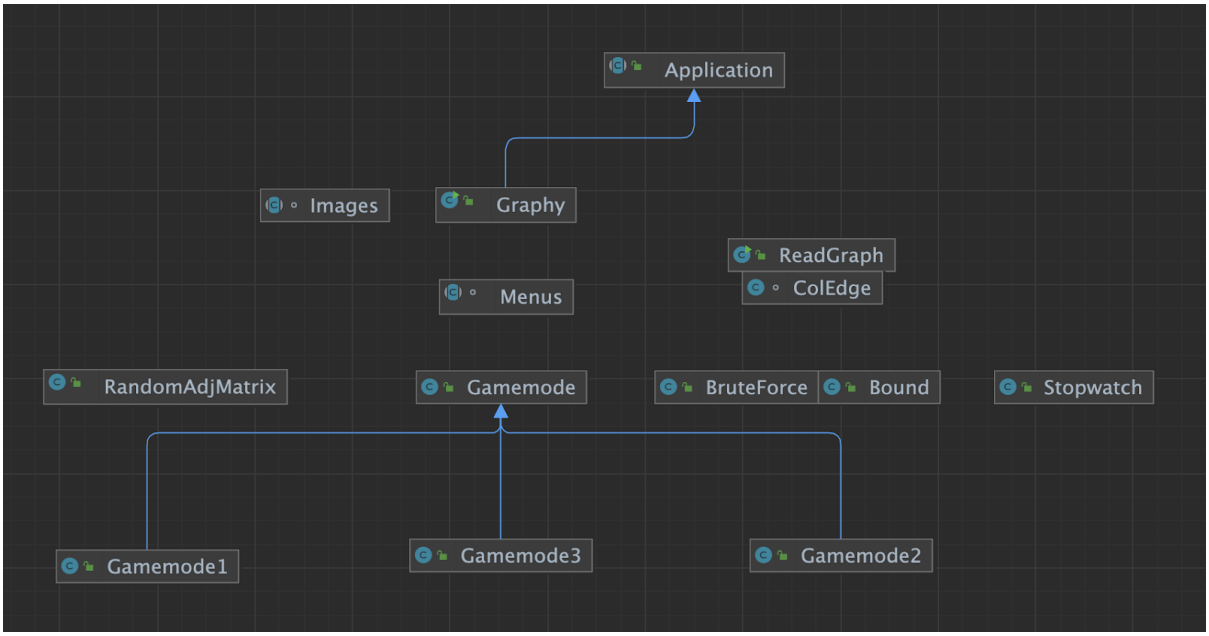


Figure 11

Adjacency matrix - The adjacency matrix[5] we used was a 2D array with the length and width equalling the number of nodes in the graph. When created the matrix is filled with 0, as there are no connections yet (connection between two nodes is represented by a 1). Then, we randomly create possible connections between randomly selected nodes, determined by the user input of the number of vertices and edges. The upper half of the adjacency matrix is mirrored onto the lower half and then the graph is printed onto the screen using the matrix to plot the connections.

Polygonal Point Placement - We use two polygonal placing[15] formulas (for X and Y) for the coordinates with the intention of making the vertices and connections easier for the user to see

by ceasing the chaotic connections made from random scattering. Using these two formulas (Fig. 12) we can calculate any n-sided graph (Fig. 13) without much computing power.

$$X = x + r * \cos\left(\frac{2 * \pi * i}{N}\right)$$

$$Y = y + r * \sin\left(\frac{2 * \pi * i}{N}\right)$$

Figure 12

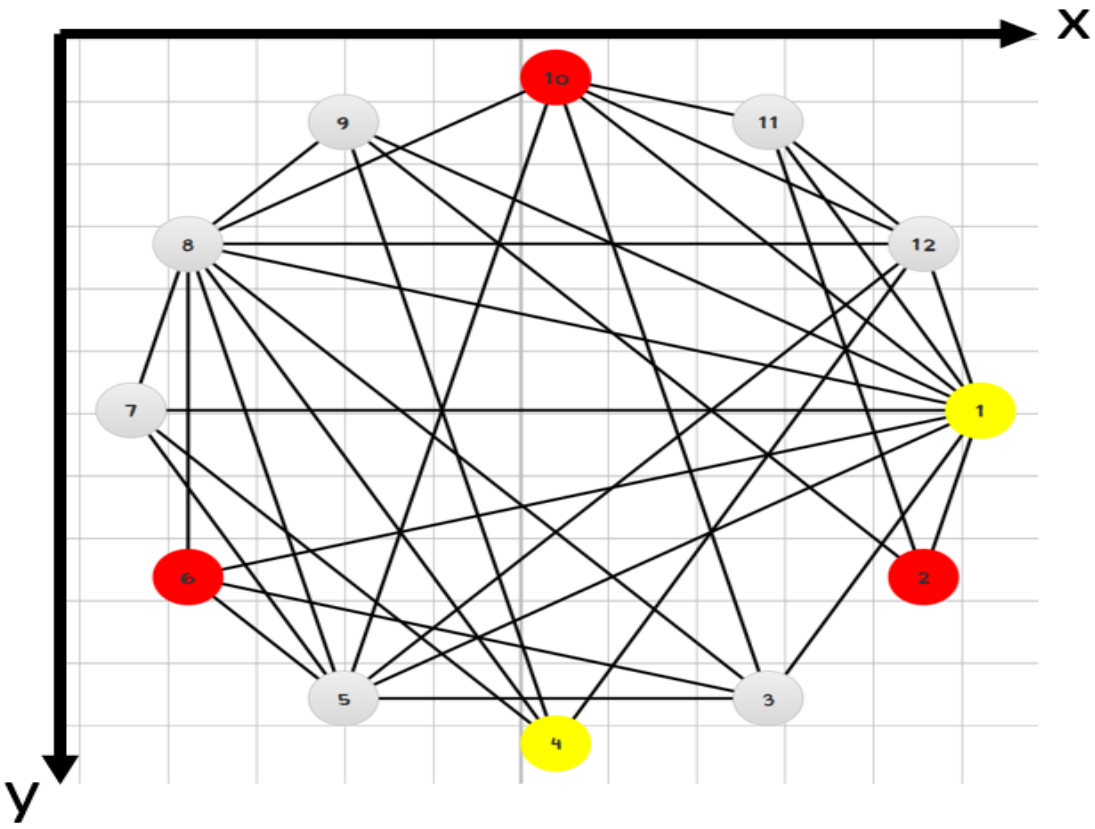


Figure 12 – Example of a generated graph

Graph Type Selection (Fig. 14) - There are three buttons each with a different way to generate a graph. The “Random” button is for a random generation where we have code that randomly selects the number of vertices and edges to be generated. The “Specified graph“ button allows the user to choose how many vertices and edges they want generated. The “Input Own Graph” button allows the user to import their own file with the number of edges and vertices already declared.

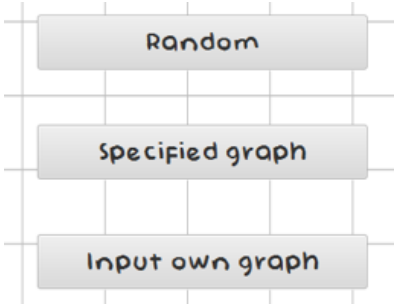


Figure 13

Experiments

Experiments are crucial when producing any type of product as they allow for the testing and validation of the design and desired deliverables. Through extensive experimentation, possible issues are identifiable which allows from changes to be made to help improve the product. Without experiments we would not know whether the product will meet performance standards, be reliable or identify opportunities to further optimize the product.

In phase 1 we were supplied with a set of graphs with the correct chromatic numbers to test our algorithms on to compare the accuracy of our chromatic numbers and bounds. In phase 3 we also used these graphs to test our new and improved algorithms on.

In phase 2 all project members performed exhaustive testing of the game on different devices to see if it was compatible using different software's. This was done to also ensure that there were no bugs in the game as we all played the game for combined total of at least multiple hours. After every new implementation of the graphical user interface, we played the game to ensure these new components were running well and by these probing tests we noticed that the random scattering of the vertices was very difficult to use when playing the game, so we introduced formulas to position the coordinates of the vertices in a polygonal shape. This allows for the user to clearly see every connection making for a better playing experience.

We wanted to create a table comparing the times of the original and improved algorithms but in phase 1 our algorithms produced bounds with a very large range and inaccurate chromatic numbers, meaning that they were very quick to run. Whereas in phase 3 our algorithms produced much more accurate results, resulting in them taking longer to compute. Therefore, we did not compare these results as it was a given that more complex and accurate programs took longer to run.

For all of the experiments, 20 simulations were run on each of the graphs from the provided dataset. Apart from that, simulations were run on self-made graphs, which can be found in the appendix. We ran our software on three different operating systems (Windows, MacOS, Linux), to ensure consistency.

Results

Phase 1 – Figure 15

Graph No.	Lower Bound	Upper Bound	Chromatic Number	LB UB Difference
03	2	8	4	6
04	2	5	3	3
05	2	5	2	3
10	2	14	N/A	12
13	2	22	2	20
16	2	503	N/A	501

Phase 3 – Figure 16

Graph no.	Lower Bound	Upper Bound	Chromatic Number	LB UB difference
1	6	8	N/A	2
2	2	3	N/A	1
3	3	6	N/A	3
5	5	10	N/A	5
7	3	3	3	0
9	8	12	N/A	4
16	98	98	98	0
17	15	15	15	0
19	8	8	8	0
20	2	3	N/A	1

Discussion

In phase 1, our brute force algorithm has found the chromatic number for 4 graphs, and with our new algorithms we have found the chromatic number for 14 graphs. This is due to an improved brute force method, in which we start colouring vertices that have the most connections, and due to our lower bound and upper bound algorithms that converge to the chromatic number. This is the case for graphs 1, 3, 4, 9, 11, 13, 16, 18 and 19.

In phase 3, we have found the chromatic numbers for 10 graphs of the newly provided dataset. In some cases, the lower and upper bound converge to the chromatic number, this is the case for graphs 2, 7, 13, 16, 17, 19 and 20.

Additionally, with our new lower bound algorithm we have found better lower bounds. Indeed, in phase 1, our lower bound algorithm gave us a lower bound of 2 for each graph, whereas our lower bound algorithm from phase 3 gives us a much tighter one. For graphs 2, 7 and 14, our new lower bounds are 14, 4 and 14 respectively. This is due to a different approach, which locates the size of the largest maximal clique in the graph, which is our lower bound.

Moreover, the conversion from a text file to an adjacency matrix in phase 3 is 20 times faster than the one from phase 1, making the results from this phase significantly better in all aspects than the ones from the previous phase.

One example is with the broadest range between upper and lower bound in phase 1 being 501 and in phase 3 being 6. This improvement of 98% accentuates exactly how well these new algorithms excel in computing the bounds.

The different types of brute force techniques that we tried to implement are:

- Exhaustive search: Generates all possible solutions and checks each one to see if it is a valid for the problem presented.
- Backtracking: Attempts different possibilities and backtracks when it reaches a dead end.
- Generate and Test: Generates multiple sets of possible solutions and tests them against a set of restrictions to see if they are applicable for the current problem.
- Recursive search: Breaks the problem down into smaller problems and solves them recursively (many successive executions).
- Combinatorial search: Generates all possible combinations of elements and checks each one to see if it can solve the current problem.
- Dynamic Programming: Breaks down the problem into smaller subproblems and solves them recursively, but also stores the solutions for other problems to avoid unnecessary computation.

The technique that best worked for us initially was the exhaustive search due to its simplicity. However, stepping into phase three, we decided to try a backtracking approach which, as seen in the results section, has significantly improved the outcomes.

We improved the gaming experience by adding custom drawings and fonts to fit the game style. This made the game feel more authentic and collective feel as it had a solid theme throughout and felt like a more fun experience than just a blank page with buttons and a graph. The game background is a sheet of paper, the differently coloured game modes are page markers at the top of the screen (the whole marker is revealed on the opposite end of the page, exactly like it

occurs in real life. These graphic details make our game unique in the sense that it makes the user feel as if they are drawing or colouring a graph, creating a more engaging and immersive experience when playing.

One of the real-world problems we tackled was making the game colour blind friendly. We solved this by adding numbers to each of the vertices and colours, so the user only needs to assign a set of numbers to different vertices rather than just colours allowing for someone who even sees no colour to play and complete the game.

We deducted that through extensive playing of the game from all our group members that we needed to somehow sort the vertices to clearly see the edge connections (as opposed to random node distribution). In order to do this, we used polygonal placement formulas to position the vertices in the form of a polygon which allowed us to distinctly see each different edge, making playing the game a lot easier especially in the game mode with a time constraint.

Despite having effective algorithms, we were not able to compute chromatic numbers for all the graphs provided in the datasets. This issue might be addressed in future studies, when we discover even more efficient ways to tackle the graph colouring problem.

Conclusion

After three phases of working on our Graph Colouring project, we created algorithms to determine the chromatic number, the lower and the upper bounds in phase 1. These algorithms were then implemented into a playable computer game in phase 2. In phase 3, the existing algorithms developed in phase 1, were modified to improve temporal and computational efficiency.

The algorithms we have implemented and then perfected throughout the duration of the project can in fact solve real world problems with acceptable accuracy as shown in the results section.

With that being said, there is room for improvement in future research, including the backtracking brute force algorithm itself, as well as the bounds that we calculate. We could achieve this by possibly exploring new structures in the graphs we did not see before. Graph colouring, being an NP-complete problem will continue to attract researchers and mathematicians alike to discover more and more optimal solutions.

References

1. Tosuni, B. (2015). Graph Colouring Problems in Modern Computer Science. Volume 1. Issue 2. https://revistia.org/files/articles/ejis_v1_i2_15/Besjana.pdf
2. Wikipedia contributors (2022). Decision tree pruning. Wikipedia. https://en.wikipedia.org/wiki/Decision_tree_pruning
3. Johnston, H. C. (1975). Cliques of a Graph--Variations on the Bron-Kerbosch Algorithm. International Journal of Computer and Information Sciences, Vol. 5, No. 3. https://link.springer.com/content/pdf/10.1007/BF00991836.pdf?pdf=inline%20link.
4. Wikipedia contributors. (2022). Graph coloring. Wikipedia. https://en.wikipedia.org/wiki/Graph_coloring
5. Lewis, R. M. R. (2015). Guide to graph colouring: algorithms and applications. Switzerland: Springer Cham. Guide to Graph Colouring: Algorithms and Applications | SpringerLink

6. Mostafaie, T., Khiyabani F. M., Navimipour, N. J. (2020). A systematic study on meta-heuristic approaches for solving the graph coloring problem. Computers and Operations Research v120 (202008). https://www-sciencedirect-com.mu.idm.oclc.org/science/article/pii/S0305054819302928

7. Heckel, A. (2018). The chromatic number of dense random graphs. Random Structures & Algorithms v53 n1, 140-182. https://onlinelibrary-wiley-com.mu.idm.oclc.org/doi/full/10.1002/rsa.20757?sid=worldcat.org

8. Edén, R. (2014). JMonkeyEngine 3.0 cookbook : over 80 practical recipes to expand and enrich your jMonkeyEngine skill set with a close focus on game development. Birmingham, UK: Packt. https://web.p.ebscohost.com/ehost/detail/detail?vid=0&sid=323b741f-bda1-415e-a43f-58ccc9e04d18%40redis&bdata=JnNpdGU9ZWhvc3QtbGl2ZSZzY29wZT1zaXRl#AN=830329&db=nlebk

9. Bron, C., Kerbosch, J. (1973). Algorithm 457: Finding all cliques of an undirected graph. Communication of ACM 16 (9), 575-577. https://dl.acm.org/doi/pdf/10.1145/362342.362367

10. Carraghan, R., Pardalos, P.M. (1990). An exact algorithm for the maximum clique problem. Operations Research Letters, 9(6), 375-382. https://www.sciencedirect.com/science/article/pii/016763779090057C

11. Torres-Jimenez, J., Perez-Torres, J. C. (2019). A greedy algorithm to construct covering arrays using a graph representation. Information Sciences v477 (201903): 234-245. https://www.sciencedirect.com/science/article/pii/S0020025518308648

12. GD (Symposium), Auber, D., Valtr, P. (2021). Graph drawing and network visualization : 28th international symposium, GD 2020, Vancouver, BD, Canada, September 16-18, 2020 : revised selected papers. Cham : Springer, 71-84. https://link-springer-com.mu.idm.oclc.org/book/10.1007/978-3-030-68766-3

13. Gouda, K., Arafa, M., Calders, T. (2018). A novel hierarchical-based framework for upper bound computation of graph edit distance. Pattern Recognition v80, 210-224. https://www-sciencedirect-com.mu.idm.oclc.org/science/article/pii/S0031320318301092

14. Brandner, F., Jordan, A. (2014). Refinement of worst-case execution time bounds by graph pruning. Article in Computer Languages, Systems & Structuresv40 n3-4 (201410): 155-170. https://www-sciencedirect-com.mu.idm.oclc.org/science/article/pii/S1477842414000359

15. Kostchka, A., Kratochvíl, J. (1995). Covering and coloring polygon-circle graphs. Discrete Mathematics v163 n1, 299-305. https://www-sciencedirect-com.mu.idm.oclc.org/science/article/pii/S0012365X96003445

16. Jakovac, M., Peterin, I. (2018). The b-chromatic number and related topics - A survey. Discrete Applied Mathematics v235 (20180130): 184-201. https://www.sciencedirect.com/science/article/pii/S0166218X17303943

Appendix

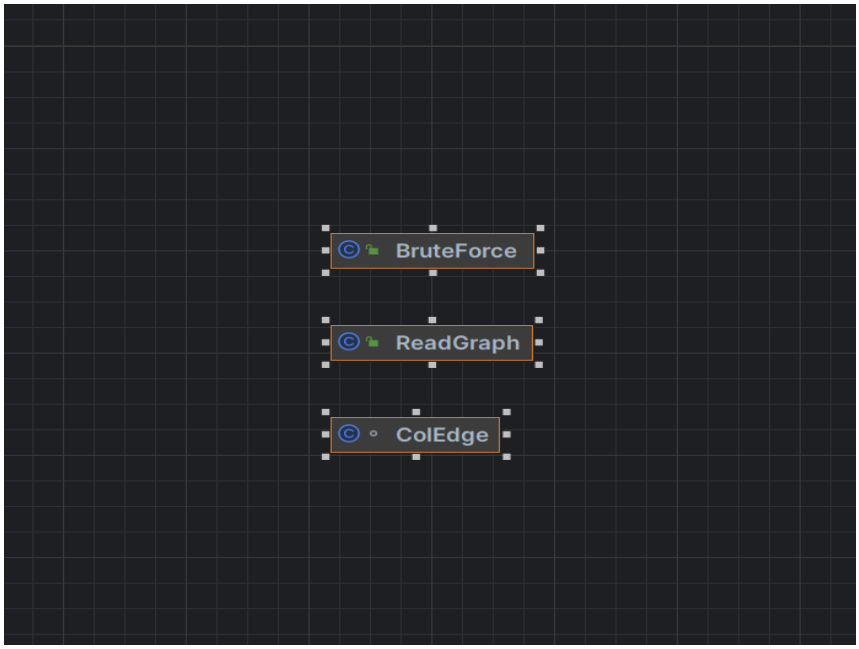
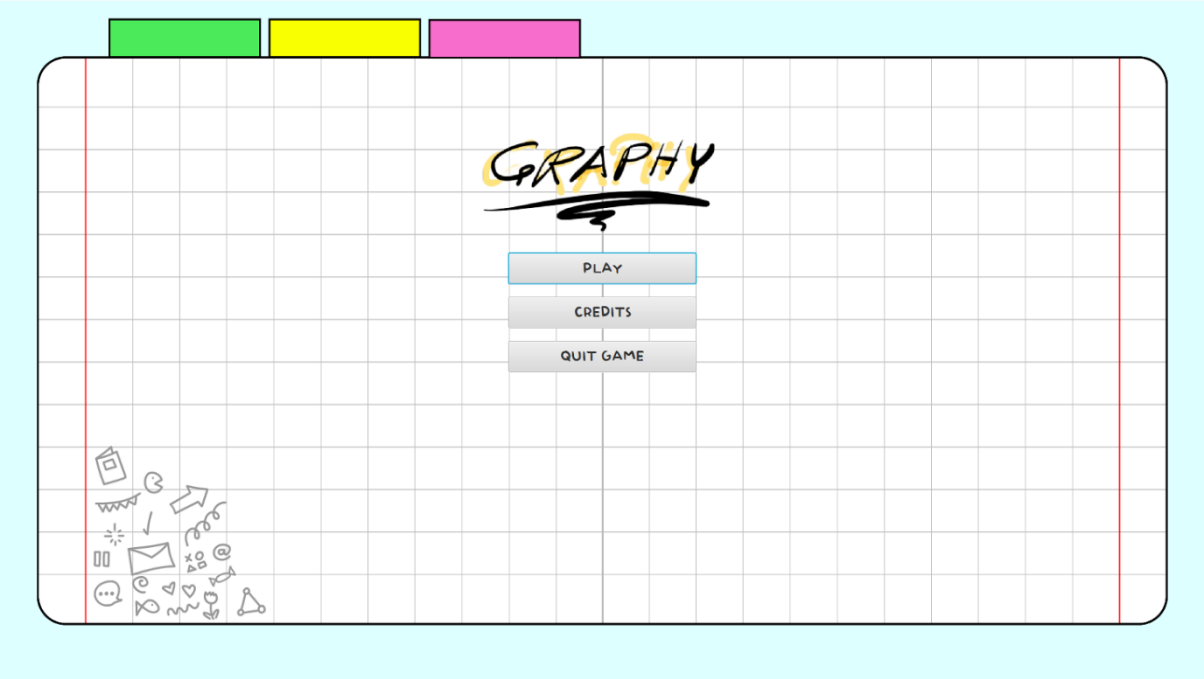


Figure 14 – UML diagram for phase 3

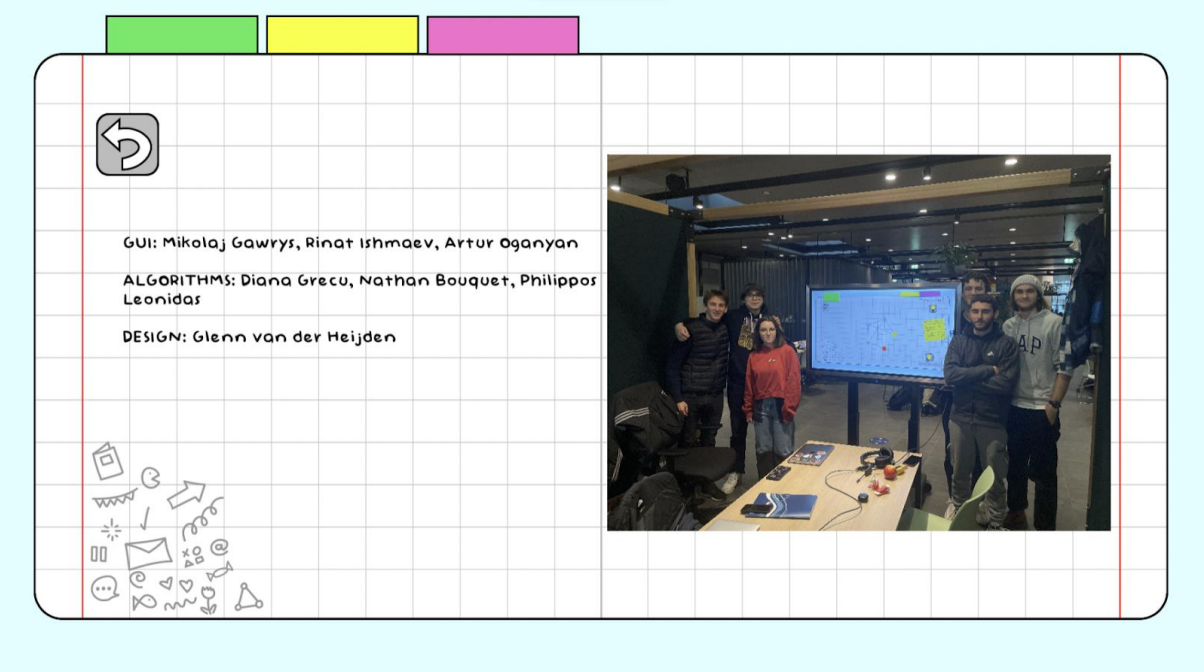
User Manual - Graphy:

Getting Started - Turn on your computer. Start Graphy. In a few moments, the Graphy main menu will appear. Our main menu includes three buttons: button play, which opens to our three game modes that the player can choose from, button credits and button quit game, which quits



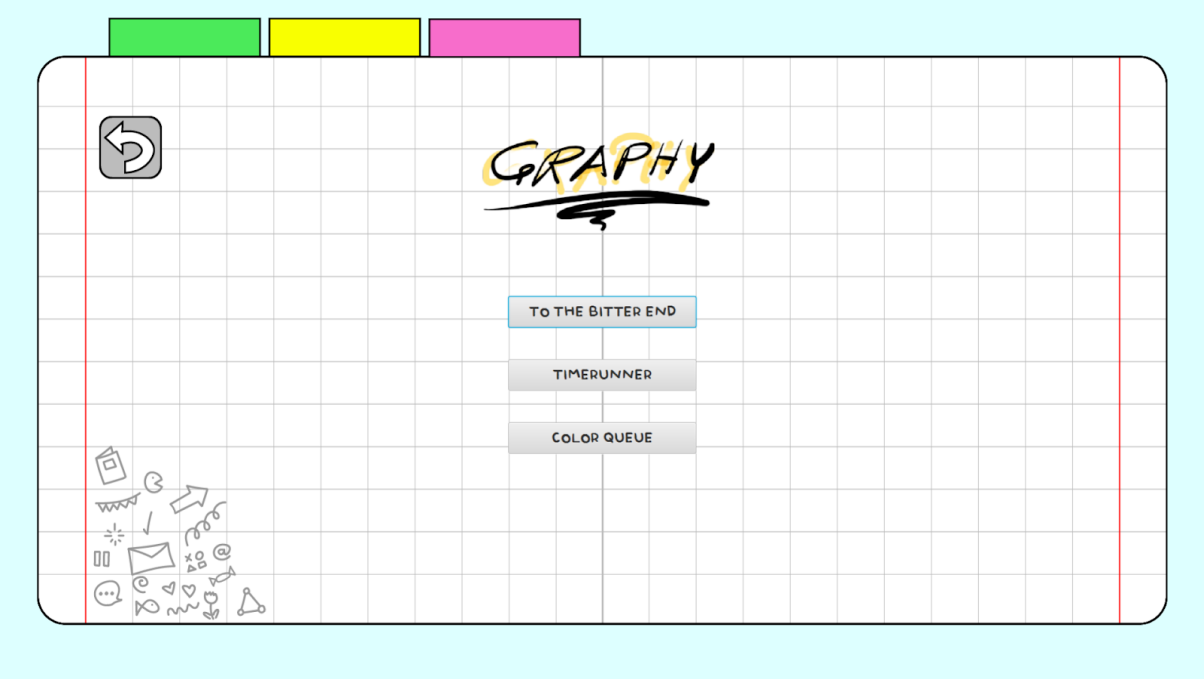
the game.

Control - Graphy uses a keyboard and a mouse to play. The mouse is primarily used to select each vertex the user wants to colour, and the keyboard is primarily used to enter the degree of vertices and edges depending on which graph type the user selects to play with.

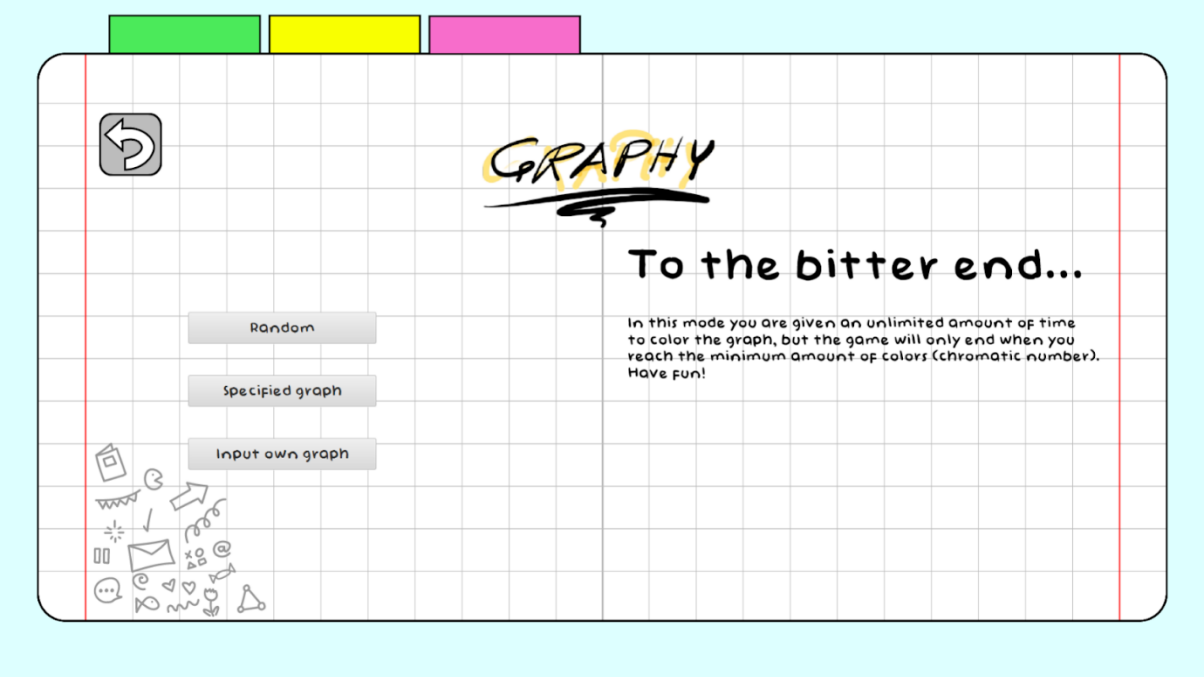


Credits screen - On our credits screen the player can find out who created Graphy. The name and role of each creator are on the left and a group picture on the right.

Pregame mode selection screen - The pregame mode selection screen shows the three game modes that the player can choose from.



Pregame menu - After choosing a game mode, our pregame menu opens where the player can choose which graph, they would like to play. There are three buttons the player can choose from: button random, which creates a random graph, button specified graph, where the player chooses how many vertices and edges, they want for the graph, and button input own graph, where the player can input their own txt file.



Rules - The rules for each game mode are different from each other are which all specified in our GUI.