An Empirical Study of Online Graph Coloring Coloring Algorithms

Group Members: Anmol Arora (40172251), Jay Dhanani (40232469), Pragya Tomar (40197757), Vamsi Krishna Reddy (40232641), Venkata Srikar Vishnu Datta (40236936)

Abstract:

Online graph coloring algorithms have gained more attention in recent years due to their wide range of uses in industries like computer networking, distributed systems, map coloring, and even in games such as sudoku. An online graph is a structure $G^{\prec}$ (V, E, $\prec$), where G=(V,E) is a graph and $\prec$ is a linear order on V. In other words, the graph is built incrementally in an online fashion, where one vertex together with its edges with the existing vertices are revealed. In an online graph coloring algorithm, a function assigns a color to each vertex upon its arrival in such a way that the no adjacent vertices share the same color. Additionally, once a color is assigned to a vertex, it cannot be modified afterwards. The goal of the online coloring algorithm here is to minimize the number of colors and use as least number of colors as possible. This report presents an empirical study of two online graph coloring greedy algorithms namely First Fit and CBIP algorithm on the basis their average competitive ratio. The competitive ratio of an online graph coloring algorithm 'A' is a measure of how well an algorithm perform as compare to the chromatic number of a graph where the chromatic number is defined as the minimum number of colors needed for coloring the graph. Overall, this empirical study examines the results to determine the pattern by testing the algorithms on a collection of randomly generated graphs.

Problem Description:

Given an online graph $G^{\prec}$ (V, E, $\prec$), where G=(V,E) is a graph and $\prec$ is a linear order on V with V as the set of vertices and E as the set of edges. The online graph coloring algorithm aims to minimize the number of colors needed for coloring the graph to its chromatic number χ, subject to the requirement that no two neighbouring vertices have the same color.

In this report, the empirical study is performed on two greedy algorithms and the results are analysed to observe how the average competitive ratio depends on the n (number of vertices in the graph). The project report aims to address the following research questions by demonstrating the two greedy online graph coloring algorithms:

- For different graph sizes and degree distributions, how many colors will be used by the First Fit and CBIP algorithm?
- What is the competitive ratio i.e. the ratio of the coloring produced by the algorithm to the optimal coloring for two greedy algorithms?
- Identify if there is any pattern and analyze the results based on the data gathered through testing the greedy algorithms.
- Described why CBIP is successful for k=2, but fails for k=3, 4.

A coloring algorithm is online if, given graph G on vertices v1, . . . , vn, it assigns (irrevocably) for each $k \in \{1, \ldots, n\}$ a color to vk that depends only on the subgraph of G induced by seen vertices$\{vj \mid 1 \leq j \leq k\}$. The two algorithms that are evaluated against 100 graphs of varying degrees are First Fit and CBIP. First-fit is the online graph coloring algorithm that considers vertices one at a time in the same order and assigns each vertex the least positive integer not used already on a neighbor.

On the other hand, CBIP rather than greedily coloring each node v, the algorithm just avoids using the same color in the connected component in which v initially occurs. Hence, when a vertex v arrives, CBIP computes the connected component C_v to which v belongs and computes a partition of C_v into two blocks based on which the color is assigned to the vertex. Lastly, depending on the total number of colours used by each method, the competitive ratio is determined for both algorithms to identify if any pattern emerge.

Implementation:

An on-line coloring of graph G is an algorithm that properly colors G by receiving its vertices in some order $v_1, \ldots v_n$ with the constraint that adjacent vertices have different colors. Without looking ahead or altering the colours that have already been given, an online graph colouring method automatically colours the vertices chosen from the list. The First Fit and CBIP algorithms are two on-line graph colouring techniques that are presented in this study. The first fit algorithm is the most basic online colouring method. It operates by giving the graph's current vertex the smallest available integer as colour. On the other hand, with the partial graph that is currently available, CBIP computes a full connected component CC to which a vertex v belongs when it appears. Additionally, the implementation consists of a graph generator algorithm in order to generate the random graphs with the help of which the output graphs are passed to both the algorithms (CBIP & First Fit) in an online fashion and a GUI logic to build an interactive interface.

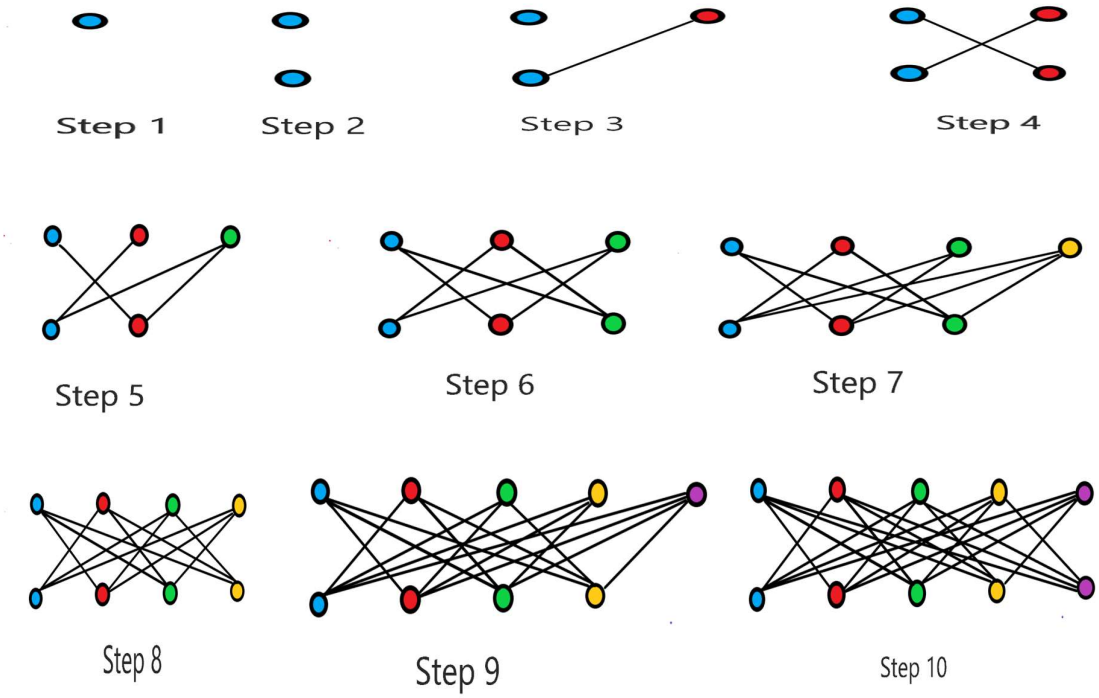
1) First Fit :

The first method that comes to mind when using a limited number of colors is First fit. First fit is a naturally greedy method that assigns the lowest integer as color to the current vertex v of the graph, with the condition that the color should not occur in the pre-neighborhood of vertex v. Hence, for each node first-fit algorithm color it with the smallest non-conflicting code by avoiding using the same color in the connected component in which vertex v initially occurs. The first-fit algorithm is shown below .

Algorithm 1 The *FirstFit* algorithm for online coloring.

procedure *FirstFit*
- $i \leftarrow 1$
- $c \leftarrow \emptyset$ ▷ map that stores the coloring
- **while** $i \leq n$ **do**
  - A new vertex $v = \sigma(i)$ arrives with its pre-neighborhood $N^-(v)$
  - $c(v) \leftarrow \min(\mathbb{N} \setminus c(N^-(v)))$
  - $i \leftarrow i + 1$
- **return** c

First, in the provided pseudocode, the dictionary 'c' used to hold the colors is initialized, and the index of the current vertex being colored, denoted by the letter 'i' is set to 1. Next, there is a loop traversing through the set of vertices 'n' where it will select the next vertex 'v' in the graph and find the pre-neighborhood and assign the smallest unused color to it. The dictionary with the final color assignments for each vertex is returned at the end.



The basic implementation of the First Fit graph coloring algorithm is shown in the image below, which specifies the class "FirstFit" that counts the number of distinct colors that are assigned to the vertex in the graph. The following are the several methods used in implementation:-

- **__init__(self):** The variable "color count" and the dictionary "colours" are initialised to their default values in this constructor method.
- **add_vertex(self, vertex, sub_graph):** When a new vertex is introduced to the graph that isn't in the list of "used colors," this method is used to assign the lowest colour feasible to it. Additionally, if no such colour is discovered, a new colour will be assigned and added to the "colours" dictionary.
- **get_unique_colors(self):** This method keeps track of the unique colors used by creating a dictionary ‘unique_used_colors’ whose length is increased by 1 whenever it adds a color to the new vertex. The length of the dictionary ‘unique_colors_used’ is then returned.
- **get_colors(self):** The 'colors' dictionary, which translates each vertex to its corresponding colors, is returned using this function.

final.py 3 ×

C: > Users > pragy > Downloads > final.py > ...

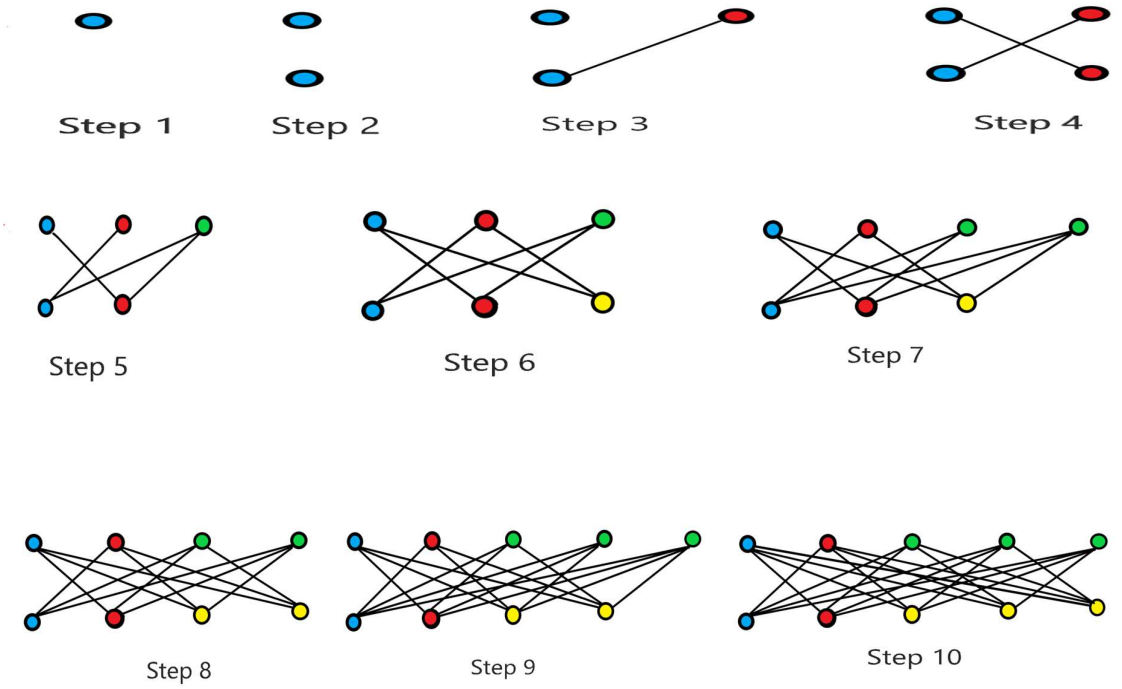
```
48  ########## ------------------------- Graph Generator [END] ----------------------- #############
49  #////////////////////////////////////////////////////////////////////////////////////////////////
50  ########## -------------------------    FirstFit [START]   ----------------------- #############
51  class FirstFit:
52      colors = {}
53      color_count = 1
54      def __init__(self):
55          self.colors = {}
56          self.color_count = 1
57      def add_vertex(self, vertex, sub_graph):
58          used_colors = {self.colors[neighbor] for neighbor in sub_graph[vertex] if neighbor in self.colors}
59          for i in range(self.color_count):
60              if i not in used_colors:
61                  self.colors[vertex] = i
62                  # print("vertex = " + str(vertex) + " color = " + str(self.colors[vertex]))
63                  break
64          if vertex not in self.colors:
65              self.colors[vertex] = self.color_count
66              # print("vertex = " + str(vertex) + " color = " + str(self.colors[vertex]))
67              self.color_count += 1
68      def get_unique_colors(self):
69          unique_used_colors = {}
70          for v in self.colors:
71              unique_used_colors[self.colors[v]] = 1
72          return len(unique_used_colors)
73      def get_colors(self):
74          return self.colors
75  ##########  -------------------------     FirstFit [END]    ----------------------- #############
```

2) **CBIP:** The graph coloring technique known as CBIP stands for Coloring based On Independent Partition, uses the independent set concept to color an online graph. On arrival of vertex 'v', CBIP first divides the graph into two independent sets of vertices, assigning colors to each independent set while making sure that no two adjacent vertices have the same color. The CBIP algorithm is shown below.

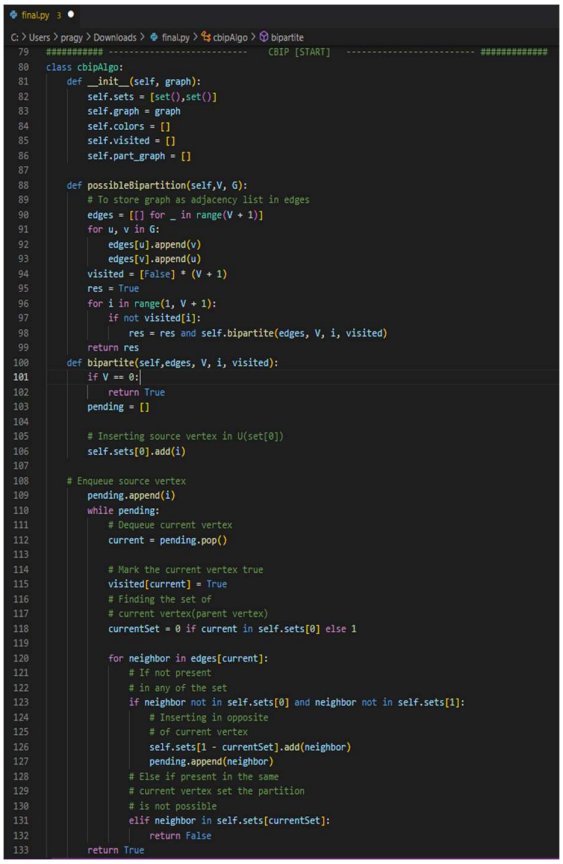
Algorithm 2 The *CBIP* algorithm for online coloring of bipartite graphs.

procedure *CBIP*
  $i \leftarrow 1$
  $c \leftarrow \emptyset$ ▷ map that stores the coloring
  while $i \leq n$ **do**
    A new vertex $v = \sigma(i)$ arrives with its pre-neighborhood $N^-(v)$
    $CC \leftarrow$ connected component of v in $G \cap \sigma([i])$
    Partition vertices of CC into A and B such that all edges go between A and B only and $v \in A$
    $c(v) \leftarrow \min(\mathbb{N} \setminus c(B))$
    $i \leftarrow i + 1$
  return c

In the CBIP algorithm, the color of each vertex is denoted by 'I' which is initialized to 1 along with an empty map 'c' to store color of each vertex. Next, a while loop will traverse through set of n vertices in the graph. First, a new vertex arrives with its pre-neighborhood enters the loop. Next, it will compute the linked component of vertex v and divide the vertices into two independent sets based on which assigns the smallest color to the vertex 'v' and increment 'i' by 1. Finally, after traversing through the loop, the algorithm returns the map 'c'.

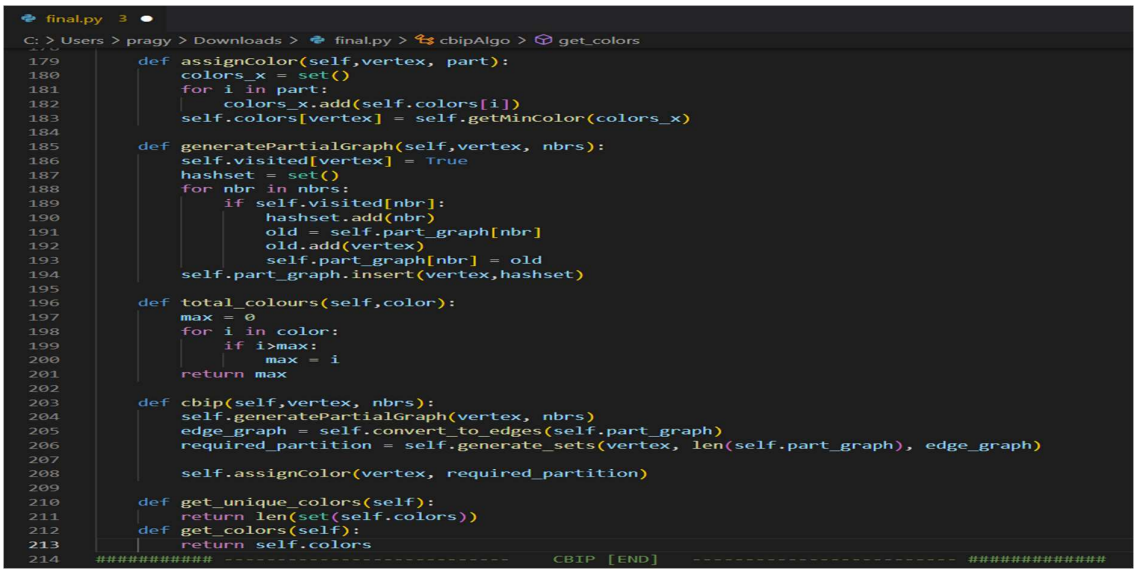


The basic implementation of the CBIP(Coloring Bipartite) graph coloring algorithm is shown in the image below, which specifies the class "cbipAlgo" that counts the number of unique colors that are assigned to the vertex in the graph. The following methods are used in the CBIP algorithm implementation:-



```
########## ------------------------- CBIP [START] ------------------------- ##########
class cbipAlgo:
    def __init__(self, graph):
        self.sets = [set(),set()]
        self.graph = graph
        self.colors = []
        self.visited = []
        self.part_graph = []

    def possibleBipartition(self,V, G):
        # To store graph as adjacency list in edges
        edges = [[] for _ in range(V + 1)]
        for u, v in G:
            edges[u].append(v)
            edges[v].append(u)
        visited = [False] * (V + 1)
        res = True
        for i in range(1, V + 1):
            if not visited[i]:
                res = res and self.bipartite(edges, V, i, visited)
        return res
    def bipartite(self,edges, V, i, visited):
        if V == 0:
            return True
        pending = []

        # Inserting source vertex in U(set[0])
        self.sets[0].add(i)

    # Enqueue source vertex
        pending.append(i)
        while pending:
            # Dequeue current vertex
            current = pending.pop()

            # Mark the current vertex true
            visited[current] = True
            # Finding the set of
            # current vertex(parent vertex)
            currentSet = 0 if current in self.sets[0] else 1

            for neighbor in edges[current]:
                # If not present
                # in any of the set
                if neighbor not in self.sets[0] and neighbor not in self.sets[1]:
                    # Inserting in opposite
                    # of current vertex
                    self.sets[1 - currentSet].add(neighbor)
                    pending.append(neighbor)
                # Else if present in the same
                # current vertex set the partition
                # is not possible
                elif neighbor in self.sets[currentSet]:
                    return False
        return True
```

```
    def generate_sets(self,vertex, size, edge_graph):
        self.sets[0] = set()
        self.sets[1] = set()
        flag = 1
        result = []

        if self.possibleBipartition(len(self.part_graph), edge_graph):
            new_set = set()
            for elem in self.sets[0]:
                new_set.add(elem - 1)

            result.append(new_set)

            new_set = set()

            for s in self.sets[1]:
                new_set.add(s - 1)
                if s - 1 == vertex: flag = 0
            result.append(new_set)

        return result[flag]

    def convert_to_edges(self,part_graph):
        edge_graph = []
        for u in range(len(part_graph)):
            for v in part_graph[u]:
                if u <= v:
                    edge_graph.append([u + 1, v + 1])
        return edge_graph

    def setup(self,vertices):
        self.colors = list(range(vertices))

        for i in range(0, vertices):
            self.colors[i] = float('inf')
        self.visited = [False] * vertices

    def getMinColor(self,color):
        n = len(color)
        for i in range(1, n + 2):
            if i not in color:
                return i
        return -1
```

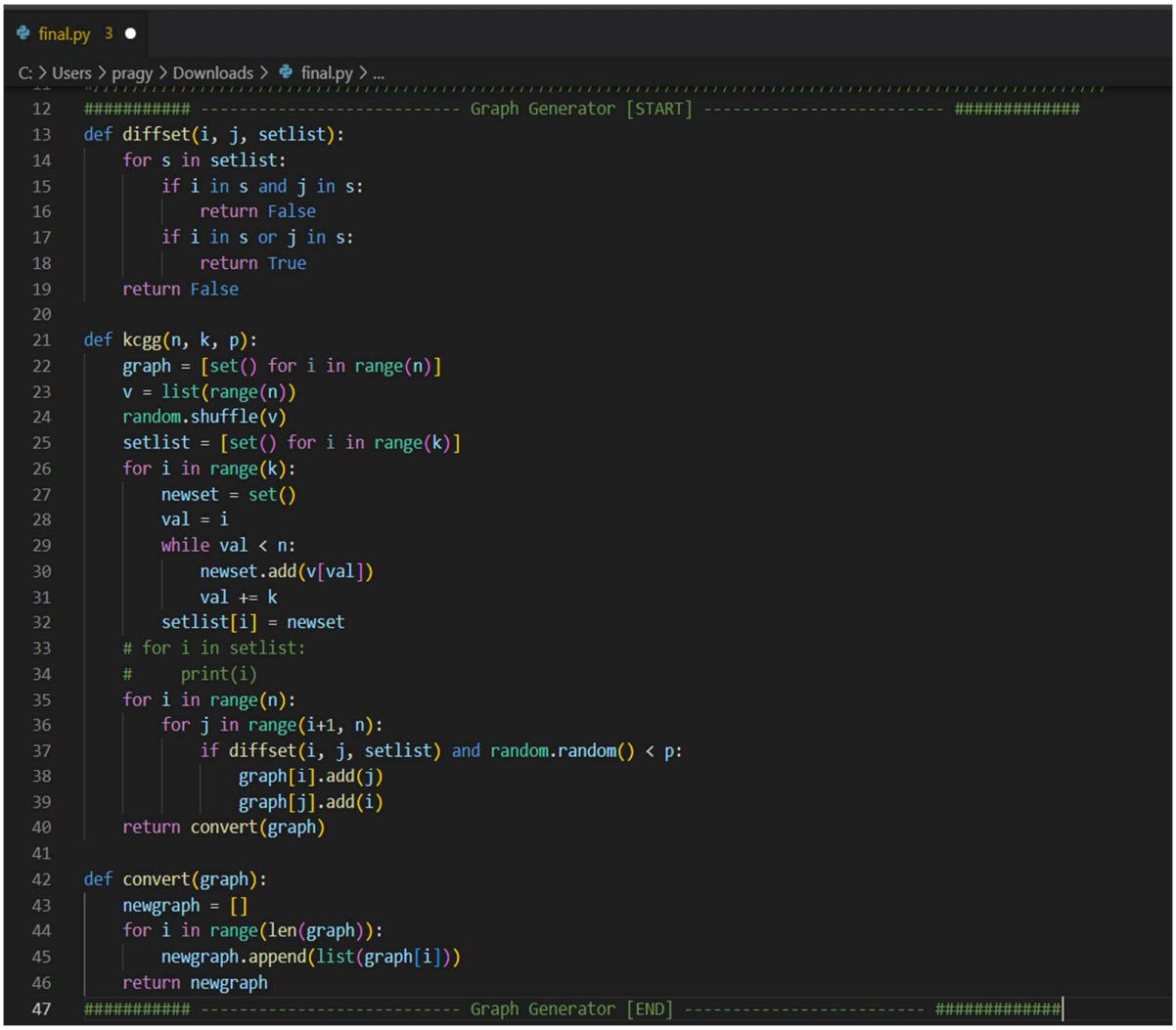


```
        def assignColor(self,vertex, part):
            colors_x = set()
            for i in part:
                colors_x.add(self.colors[i])
            self.colors[vertex] = self.getMinColor(colors_x)

        def generatePartialGraph(self,vertex, nbrs):
            self.visited[vertex] = True
            hashset = set()
            for nbr in nbrs:
                if self.visited[nbr]:
                    hashset.add(nbr)
                    old = self.part_graph[nbr]
                    old.add(vertex)
                    self.part_graph[nbr] = old
            self.part_graph.insert(vertex,hashset)

        def total_colours(self,color):
            max = 0
            for i in color:
                if i>max:
                    max = i
            return max

        def cbip(self,vertex, nbrs):
            self.generatePartialGraph(vertex, nbrs)
            edge_graph = self.convert_to_edges(self.part_graph)
            required_partition = self.generate_sets(vertex, len(self.part_graph), edge_graph)

            self.assignColor(vertex, required_partition)

        def get_unique_colors(self):
            return len(set(self.colors))
        def get_colors(self):
            return self.colors
    ############ ------------------------- CBIP [END] ------------------------- ##############
```

- **__init__(self, graph):** The method is used to initialize the cbipAlgo object with a graph represented as an adjacency list.
- **possibleBipartition(self, V, G):** Determines if the graph can be partitioned into two sets such that each vertex in a set has no adjacent vertices in the same set.
- **bipartite(self,edges, V, i, visited):** This function is used as a helper for 'possibleBipartition' function that determines if a given subgraph is bipartite.
- **generate_sets(self, vertex, size, edge_graph):** It generates a set of vertices that can be colored using the same color as the given vertex. This method is used to determine which set of vertices can be colored with the same color as the given vertex.
- **convert_to_edges(self, part_graph):** It is used to convert a partial graph represented as an adjacency list to a list of edges.
- **setup(self, vertices):** It initializes the cbipAlgo object with a list of vertices.
- **getMinColor(self, color):** It is used to determine the minimum color that has not been used to color adjacent vertices.
- **assignColor(self, vertex, part):** It assigns a color to a vertex based on the colors of adjacent set of vertices.
- **generatePartialGraph(self, vertex, nbrs):** The function is used to generate a partial graph based on the given vertex and its neighbors.
- **total_colours(self, color):** It determines the total number of colors used to color the graph.
- **cbip(self, vertex, nbrs):** This function runs the CBIP algorithm to color the given graph.
- **get_unique_colors(self):** It returns the total number of unique colors used to color the graph.
- **get_colors(self):** Returns a list of colors assigned to each vertex in the graph.

3) **Graph Generator:** The graph generator algorithm works based on the three parameters, 'n' which represent the number of vertices in the graph , 'k' which indicates the chromatic number and 'p' is the probability of having an edge between two vertices.
   Initially, the algorithm splits the n vertices into k independent sets. For example, if k=2 and n=100, then the first 50 elements are inserted into set-1 and the rest are inserted into set-2. If k=3 and n=100, then the first set has 33 vertices, the second set has 33 vertices and the third set has 34 vertices and so on. Each vertex in a set does not have edges with any other vertex in that set. This indicates that an entire set can be colored with one color since they do not have any edge between them. For any vertices 'i' and 'j' in iteration, the algorithm

generates a random probability between 0 and 1, and checks if the calculated probability is less than given probability ‘p’. If yes, then it adds an edge between two vertices ‘i’ and ‘j’, only if they belong to a different set. This step ensures that the generated graph is not a full graph and ensures randomness. The basic implementation of the Graph Generator algorithm is shown in the image below which basically consists of three methods namely ‘diffset’ to check if ‘i’ and ‘j’ are both elements of any set in ‘setlist’ based on which it returns a boolean value. Another function used in graph generator algorithm is ‘kcgg’ which creates a list of sets called ‘setlist’ and an empty ‘graph’ with ‘n’ nodes. Also, using the ‘diffset’ function it checks for each pair whether ‘i’ and ‘j’ belong to the different sets in ‘setlist’ and based on the output assigns an undirected edge with probability ‘p’. At the end, using the function ‘convert’ , it converts a graph into an adjacency list representation and returns the output.



Results:

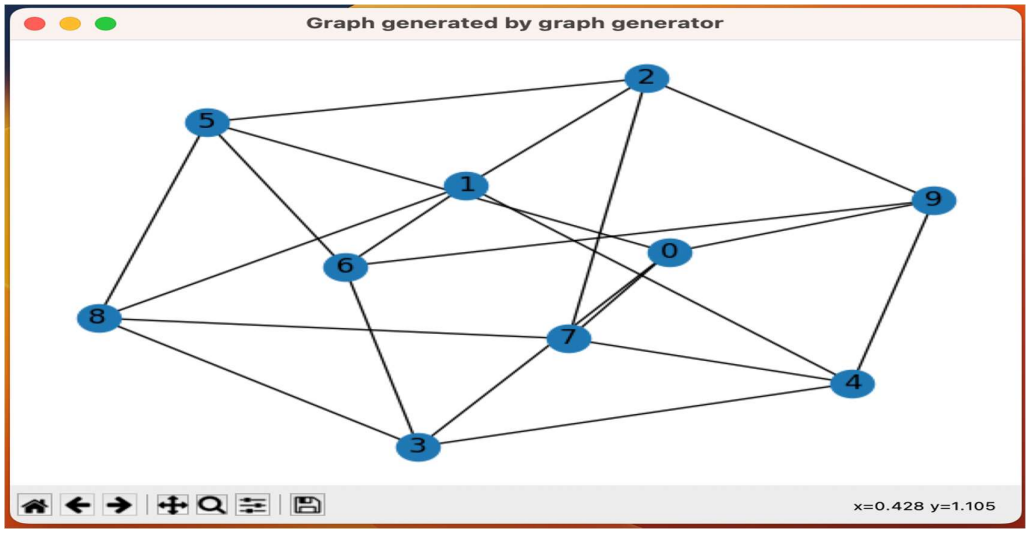
The main objective of the project report is to empirically study the behavior of the First Fit and CBIP algorithm based on the competitive ratio and how the competitive ratio depends on the number of vertices in the graph. The competitive ratio[6] of an algorithm 'A' on an online graph G(V,E) is defined as follows:

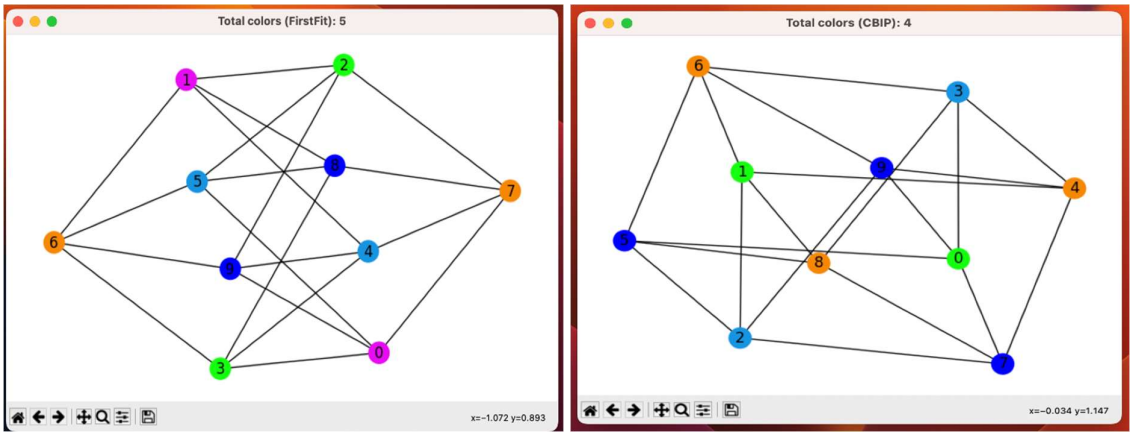
$$\rho(\mathcal{A}, G) = \frac{\text{number of colors used by } \mathcal{A} \text{ on } G}{\chi(G)},$$

where χ(G) denotes the chromatic number of Graph G. After successfully implementing the algorithms, the empirical study is then carried out for both the algorithms on the average competitive ratio using the following steps:

- Two parameters are chosen: N, which represents the number of instances of the online graphs and n is the number of vertices in each online graph.
- Next, N number of instances are created of k-colorable online graph on n-vertices.
- On each of the N instances of the k-colorable graph, the algorithm is then implemented.
- Finally, to evaluate and analyze the trend, 'n' is raised, and the algorithm is implemented again to see how 'n' affects the average competitive ratio.

The application of the algorithm yields the following result for the previously mentioned example, demonstrating that the First Fit algorithm and CBIP algorithm, respectively, use 5 and 4 colors on a randomly generated graph by the graph generator program.





Next, empirical research is carried out for both the algorithms, starting first with the First Fit and followed by the CBIP online graph coloring algorithm for k-colorable online graphs.

First Fit:

Case 1: k=2, N=100

k=2, N=100, n=	50	100	200	400	800	1600
Avg. Competitive Ratio	1.5	1.67	1.82	2.02	2.21	2.29

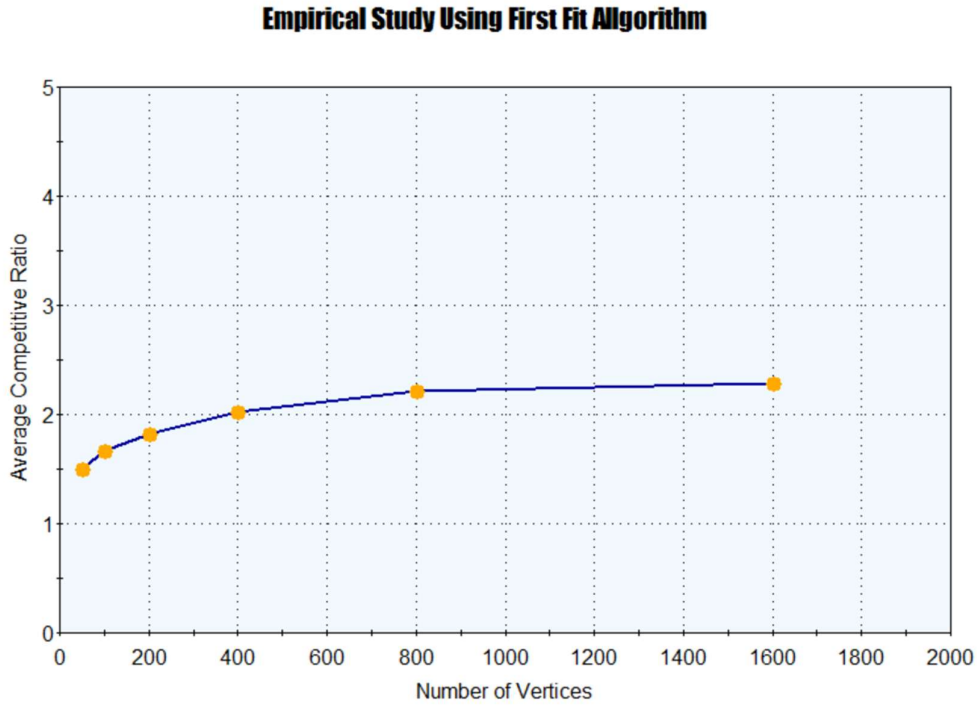
Case 2: k=3, N=100

k=3, N=100, n=	50	100	200	400	800	1600
Avg. Competitive Ratio	1.57	1.85	2.40	2.78	3.49	3.79

Case 3: k=4, N=100

k= 4, N=100, n=	50	100	200	400	800	1600
Avg. Competitive Ratio	1.63	2.05	2.46	3.03	3.57	4.81

Overall, conducting empirical study on the above data using First-Fit algorithm provides valuable insights into the performance of algorithm in real-world scenarios. As a result, after plotting the data, First-Fit follows **'log n'** trend for the average competitive ratio because it demonstrates that as the number of vertices 'n' in the online graph increases then the average competitive ratio of first fit algorithm increases logarithmically. Consequently, it can manage more instances of the online graph with a relatively slight increase in the average competitive ratio.



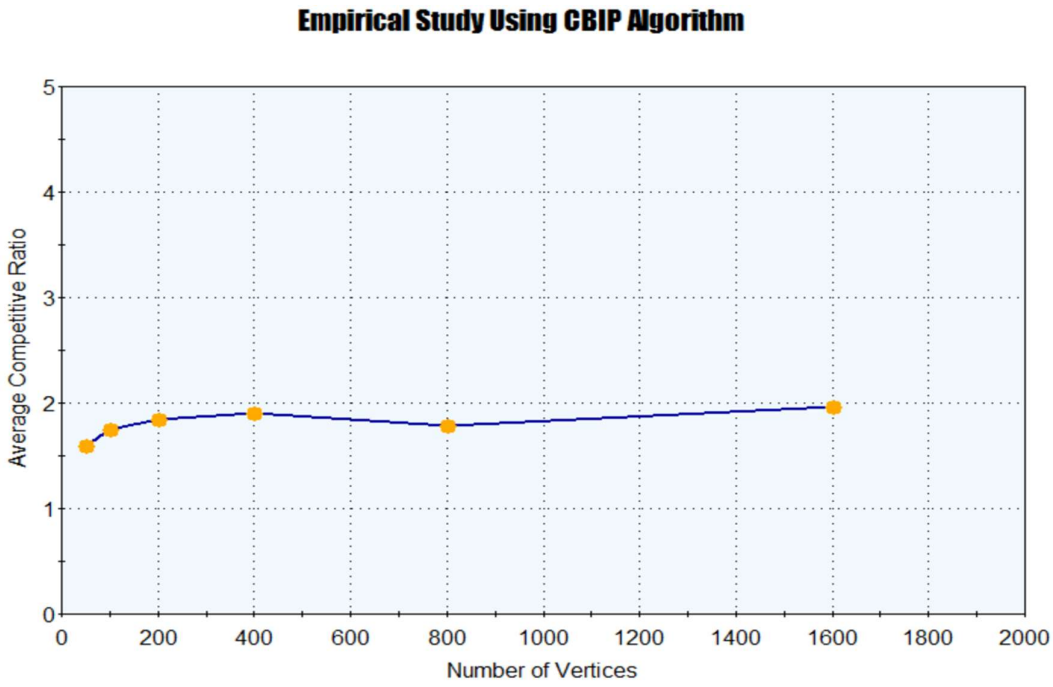
CBIP:

Case 1: k=2, N=100

k=2, N=100, n=	50	100	200	400	800	1600
Avg. Competitive Ratio	1.59	1.75	1.85	1.91	1.79	1.97

Case 2: k=3 and k=4

For this case, CBIP is considered as NP-hard problem. CBIP algorithm can be used to color the graphs when the value of k is equal to 2 but it eventually becomes impractical in terms of computational tractability for the values of k greater than 2 due to the exponential growth of partition possibilities with the increase in the size of the graph as well as the value of k and the accuracy of the coloring in the graph in reasonable time period.



As a result, after doing empirical research on the dataset using the CBIP algorithm, it is discovered that for k=2, it is successfully implemented, and the trend it follows is comparable to First-fit, or more specifically, it is $\leq$ **logn**. However, as previously stated, it is demonstrated to be an NP-hard problem for k=3 and k=4.

Conclusion and Future Work:

In conclusion, the study conducted an empirical analysis of two online graph coloring algorithms, namely First Fit and CBIP Algorithm. Online graph coloring involves revealing the vertices in online fashion i.e., one vertex at a time adding connecting edges to the already existing vertices. The objective is to utilize the fewest colors possible to color the entire graph with the restriction that the algorithm should assign colors without first knowing the full network.

As a result, two approaches: First Fit and CBIP are utilized to tackle the problem. First Fit works by coloring the arrived vertex by the smallest natural number that has not been used by its existing neighbors. But on the other side, upon arrival of the vertex, CBIP finds the two distinct sets of vertices and assigns the colors while making sure that no two neighboring vertices have the same color.

The study involves implementing both the algorithms as well as graph generator algorithm to generate random graphs. The algorithms were then tested to evaluate the performance by creating 100 instances of the problem and varying the number of vertices in each case. After finding the average competitive ratio, it is discovered that First Fit performs well for k=2,3,4 and has a log n trend for increasing values of n. On the other hand, conducting an empirical study on the average competitive ratio using CBIP algorithm revealed that it follows almost same trend as First Fit for k=2 i.e., less than or equal to log(n) but is computationally impractical for k=3, 4.The problem becomes more complex and is called as NP-hard for larger values of k in case of CBIP algorithm. Additionally, the implementation considered the project's bonus challenge, leading us to create an interactive interface for the experiment.

For future work, empirical studies can be conducted on larger instances with a greater number of vertices and larger values of chromatic number in order to analyze the results and further evaluate the performance of the algorithms. In addition to this, the study found that finding the average competitive ratio trend with k values of 3 and 4 is computationally difficult for CBIP algorithm . Therefore, for future purpose various methods can be examined to make this problem computationally feasible.

Team Work Distribution:

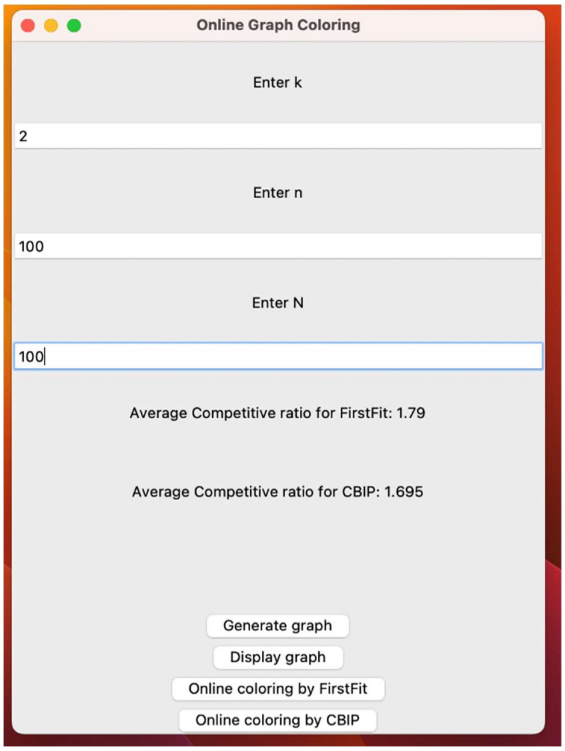
For this project, our team worked together to implement the algorithms and evaluate their performance based on the empirical study of the average competitive ratio. The team consists of five members, each of whom was assigned specific roles and responsibilities. To complete the project successfully, the team was divided with the following roles:

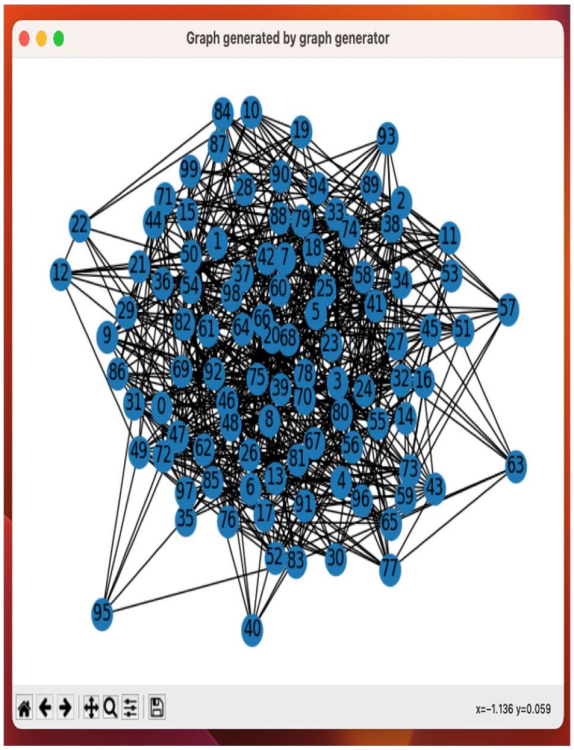
Tasks Assigned	Student Name	Student Id
CBIP Algorithm, Graph Generator Algorithms	Pragya Tomar	40197757
First Fit Algorithm, Graph Generator Algorithms	Anmol Arora	40172251
First Fit Algorithm, CBIP Algorithm	Jay Dhanani	40232469
First Fit Algorithm, CBIP Algorithm	Vamsi Krishna Reddy Munnangi	40232641
CBIP Algorithm, Graph Generator Algorithms	Venkata Srikar Vishnu Datta Akella	40236936

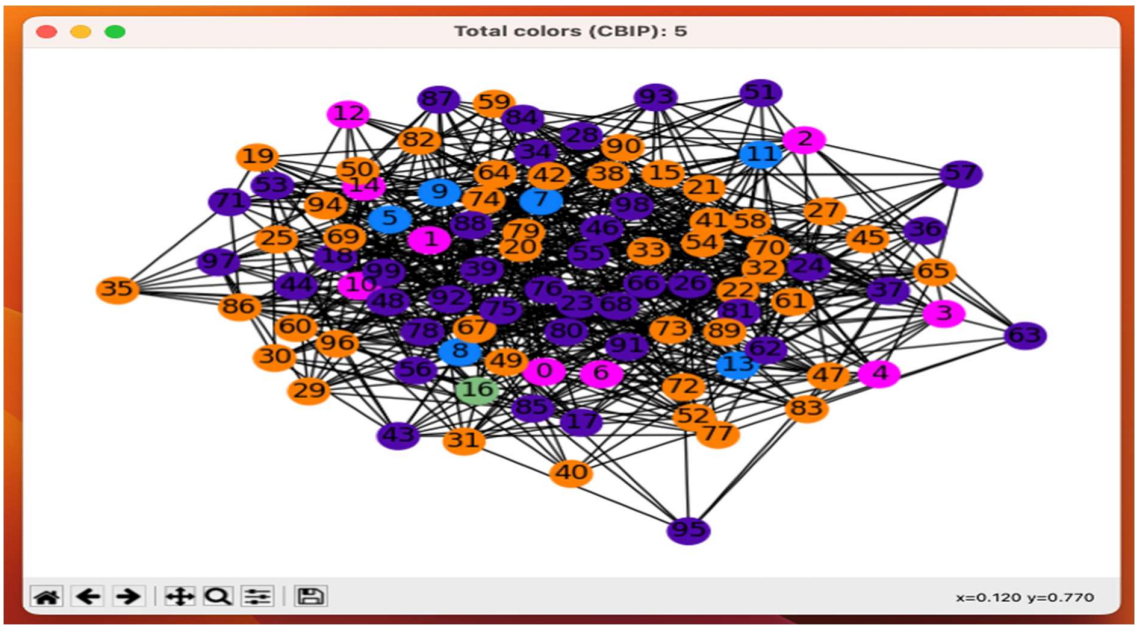
- The project scope was made at the beginning of the project to get an insight of the goals. The efforts were estimated to evenly distribute the tasks considering the availability of each person.
- Errors and other technical challenges were discussed and resolved collaboratively in the group meetings along with the planning and setting up next milestones to be achieved before the following meeting.
- Communications among the team were done effectively both online and in-person. Regular Meetings were held to keep track of the progress of the tasks assigned to each member of the team.
- All team members collaborated on the **project report and bonus problem** to demonstrate the assessment and outcomes.

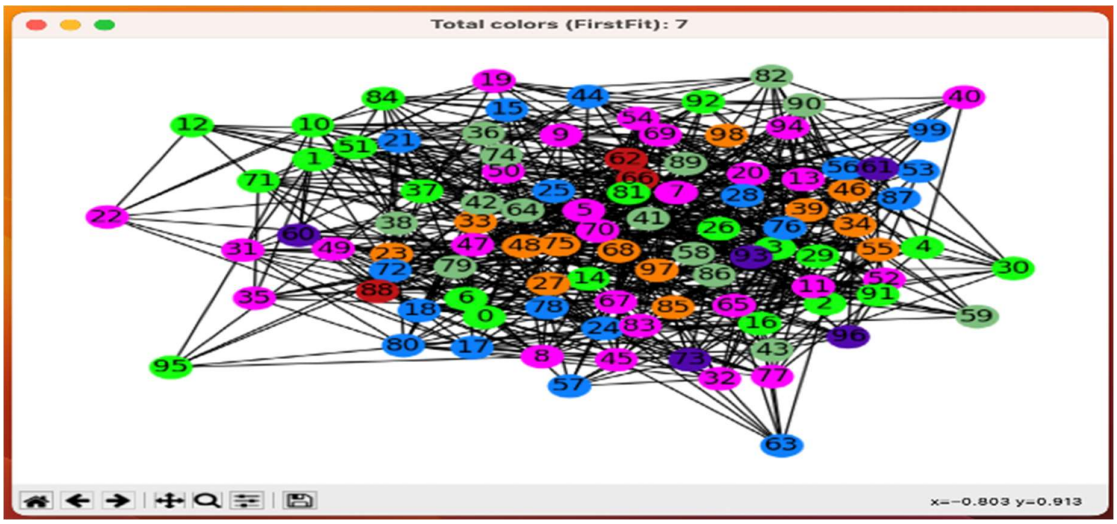
Bonus Problem Attempt:

The team successfully completed the bonus problem given in the project in which all the team members worked collaboratively. An interactive interface is built for the problem which consists of three input boxes and two buttons to compute the average competitive ratio using First Fit and CBIP graph coloring algorithms. The examples for the same are shown in the images below:









Moreover, to run the code the steps are as follows:

1) Installation

- install python 3.11
- install following dependencies using pip
  - pip install tk
  - pip install networkx
  - pip install matplotlib
  - pip install distinctipy

2) Run following command:

- python final.py

3) After running python command a new GUI window will open

- Enter the value of n, k, and N
- Click on Generate graph button
- Click on Display Graph button (This will display graph in separate windows)

  (Only first instance of graphs will be considered here)

- Click on Online coloring by FirstFit (This will display Competitive ratio for FirstFit)

- Click on Online coloring by CBIP (This will display Competitive ratio for CBIP)

References:

1. http://www.cs.toronto.edu/~bor/2421f19/papers/kierstead-survey.pdf Coloring Graphs On-line by Hal A. Kierstead
2. https://reader.elsevier.com/reader/sd/pii/S0195669815001328?token=19650712F32F14443E55C2F67FF0CDE424497D3D4F5ABF70074BCAB19142B780AC1834A842A418B7CE1A8FC5D9DFB79C&originRegion=us-east 1&originCreation=20230403232255 First-fit coloring on interval graphs has performance ratio at least 5 by H.A. Kierstead, David A. Smith, W.T. Trotter.
3. https://users.renyi.hu/~gyarfas/Cikkek/33_GyarfasLehel_OnLineAndFirstFitColoringsOfGraphs.pdf
4. https://bohr.wlu.ca/hfan/cp412/references/ChapterOne.pdf Garey, M. R., & Johnson, D. S. (1979). Computers and intractability: A guide to the theory of NP-completeness. W. H. Freeman.
5. https://arxiv.org/pdf/2005.10852 Online Coloring and a New Type of Adversary for Online Graph Problems

6. https://core.ac.uk/download/pdf/82082554.pdf Kubale, M. (2004). A note on the chromatic index problem for bipartite graphs. Discrete Mathematics, 285(1-3), 267-269.

7. https://www.cs.bgu.ac.il/~elkinm/fp136-barenboim.pdf Barenboim, L., & Elkin, M. (2016). Deterministic graph coloring in nearly linear time. Journal of the ACM, 63(6), 1-54.

8. https://arxiv.org/abs/2112.11831 Fekete, S. P., & Schepers, J. (2006). Online graph algorithms: Survey and bibliography. Algorithmica, 46(4), 239-292.

9. http://www.cs.toronto.edu/~bor/2421f19/papers/kierstead-survey.pdf Chen, J., & Lin, M. (2018). Online graph coloring with restricted conflicts. Journal of Computer Science and Technology, 33(5), 1015-1026.

10. Referred to Project Description, videos and study material provided by the professor in lectures and moodle.