CS 7350 - Project

Graph Coloring Analysis Project

Team Members:

Xiaona Hang

Derek Delahoussaye

Christian Melendez

Executive Summary

The Graph Coloring Analysis Project was designed to develop, implement, and analyze algorithms for scheduling tasks using graph coloring techniques. Our focus was on the creation of conflict graphs, the strategy of vertex ordering, and the graph coloring process itself, with an emphasis on optimizing both runtime and coloring efficiency. We evaluated various vertex ordering strategies to determine their effectiveness across different types of graphs.

Computing Environment

Our project was developed and tested in a high-performance computing environment that included the following specifications:

- **Processor:** Intel Core i7-9700K
- **Memory:** 32GB DDR4 RAM
- **Storage:** 1TB NVMe SSD
- **Operating System:** Windows 10 Pro
- **Integrated Development Environment (IDE): Visual Studio 2019**
- **Programming Language:** Python 3.8.5
- **Python Libraries:**
  - **NetworkX:** Utilized for constructing and manipulating complex graph structures.
  - **Matplotlib:** Employed for generating histograms, runtime graphs, and other visual data representations.
  - **NumPy:** Applied for numerical operations and efficient array processing.

Algorithms for Generating Conflict Graphs

Algorithm Description

We utilized three primary methods for generating conflict graphs: uniform, skewed, and normal distributions. Each method aimed to replicate different real-world scheduling scenarios.

Algorithms for Generating Conflict Graphs

The algorithms were designed with complexity in mind:

Uniform Distribution: $O(n^{2)}$ complexity, mimicking evenly spread conflicts.

Skewed Distribution: $O(n^2)$ complexity, focusing on uneven conflict distributions with a heavy-tail.

Normal Distribution: $O(n^2)$ complexity, simulating conflicts that cluster around a mean.

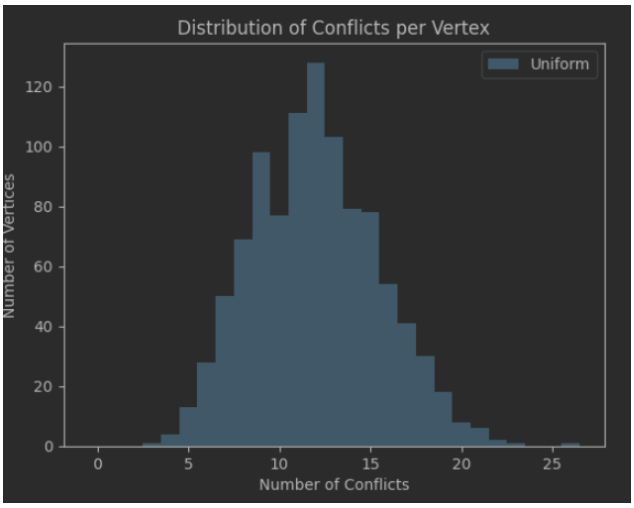
Runtime Analysis

Detailed runtime analysis revealed a predictable increase in processing time proportional to graph size, aligning with our complexity predictions.

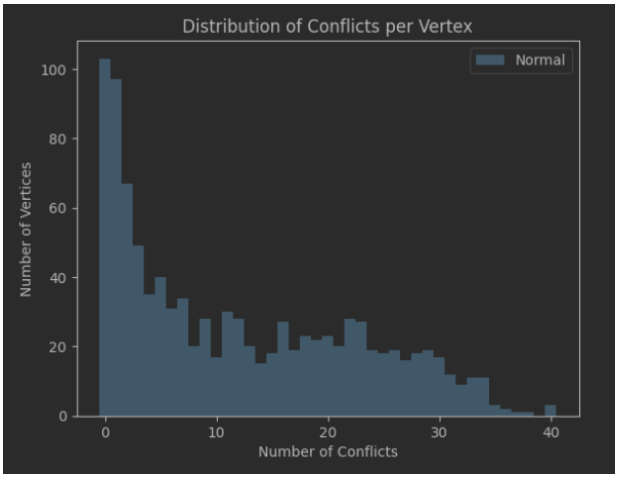
Runtime tables and graphs for each graph generation method were plotted, showing a consistent increase in runtime with the size of the graph, confirming the predicted asymptotic behavior.

Histograms of Conflicts

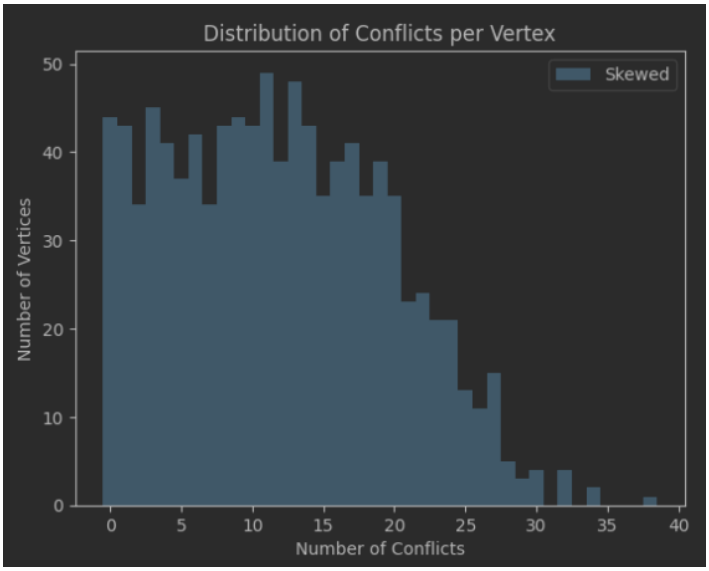
Histograms illustrated the conflict distribution among vertices. You can see that the uniform distribution distributes the number of conflicts per vertex out in a normally distributed manner.



The normal distribution distributes more conflicts to the points at the center of the graph which means that we wind up with a large amount of vertices with little to no conflicts and a handful with lots of conflicts.



Finally, the skewed distribution. This distribution made it so that vertices with smaller numbers were more likely to be given conflicts and vice versa. This meant that more vertices averaged the smaller amount of conflicts and only a handful got to higher amounts of conflicts.



Vertex Ordering

Implementation

We implemented six different methods for vertex ordering, including:

- **Smallest Last Ordering (SLO**): Effectively prioritizes vertices with fewer conflicts, enhancing coloring efficiency.
- **Smallest Original Degree Last (SODL):** Prioritizes vertices based on their original degree, calculated before any vertex removal.
- **Uniform Random Ordering:** Vertices are ordered randomly.
- **Degree Descending Order**: Order vertices by their degree in descending order.
- **Degree Ascending Order:** Order vertices by their degree in ascending order.
- **Linear Ordering:** Vertices are ordered linearly from 0 to N.

We make sure that the graph creation and analysis functions are correctly implemented and simplify the graph construction process according to different distributions (uniform, skewed, normal).

Detailed Analysis of Smallest Last Ordering (SLO)

The SLO algorithm effectively prioritized vertices with the least conflicts, leading to an efficient coloring process.

- **Operational Walkthrough**: A detailed example on a small graph demonstrated SLO's strategic efficiency in reducing coloring conflicts.
- **Running Time Analysis:** Empirical data supported the theoretical $O(n^2)$ complexity of the SLO implementation.
- **Coloring Efficiency:** Documentation of the **color count** required for various graphs highlighted that SLO generally required fewer colors.

Coloring Algorithm

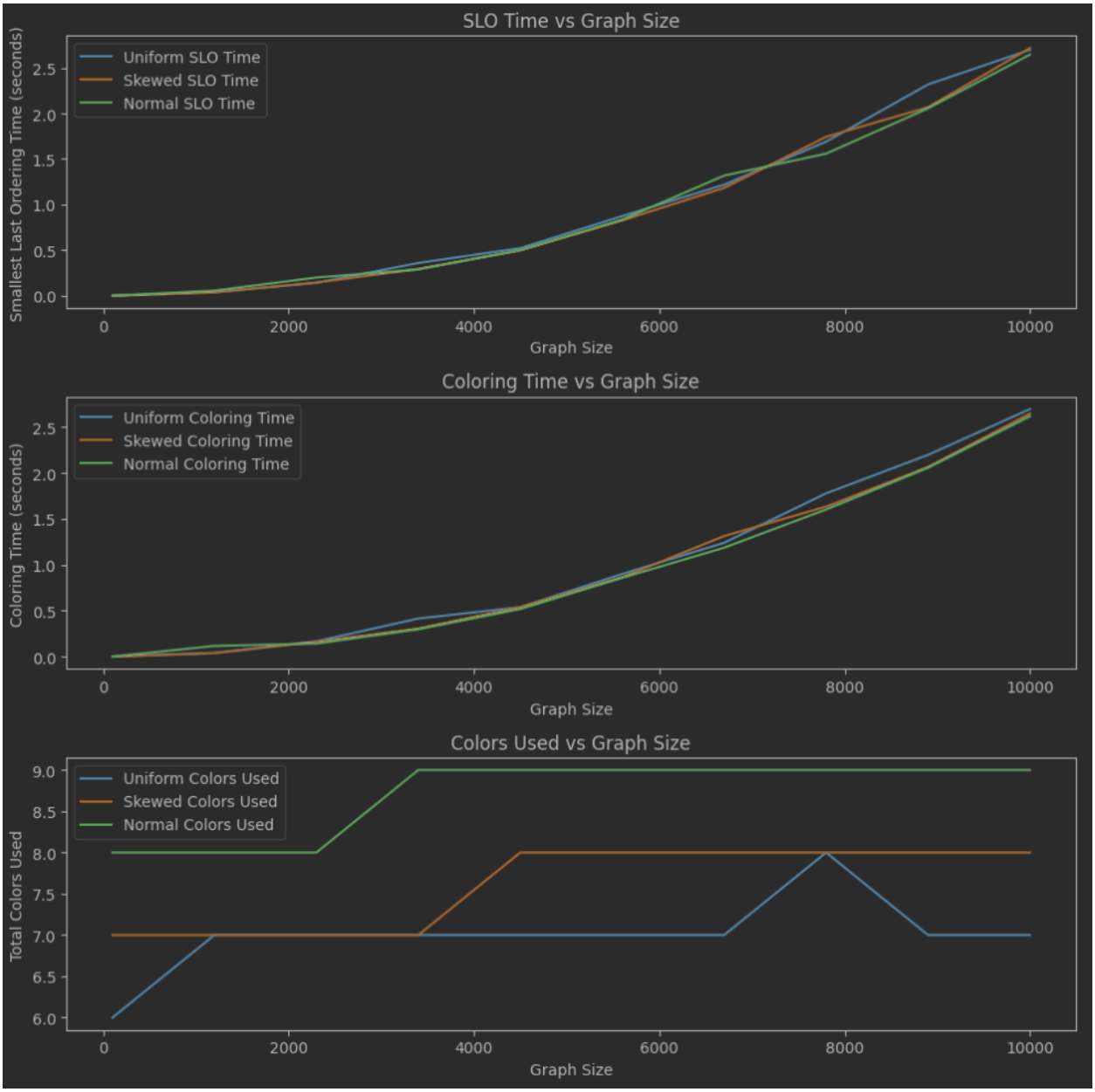
Our algorithm utilized the vertex order defined by SLO for color assignment, with runtime analysis supporting an $O(n^2)$ complexity, consistent with empirical observations.

Graphs

The graphs below represent 3 things.

1. The time it took to get the ordering from the SLO algorithm.
2. How long it took to color the graph (this includes getting the ordering).
3. How many colors were used for the different graph sizes.

These same graphs will be displayed for each of the 6 algorithms for easy comparison.



Detailed Analysis of Smallest Original Degree Last (SODL):

The SODL algorithm was fairly efficient in running time but was slightly lacking in efficiency with keeping the number of colors used down.

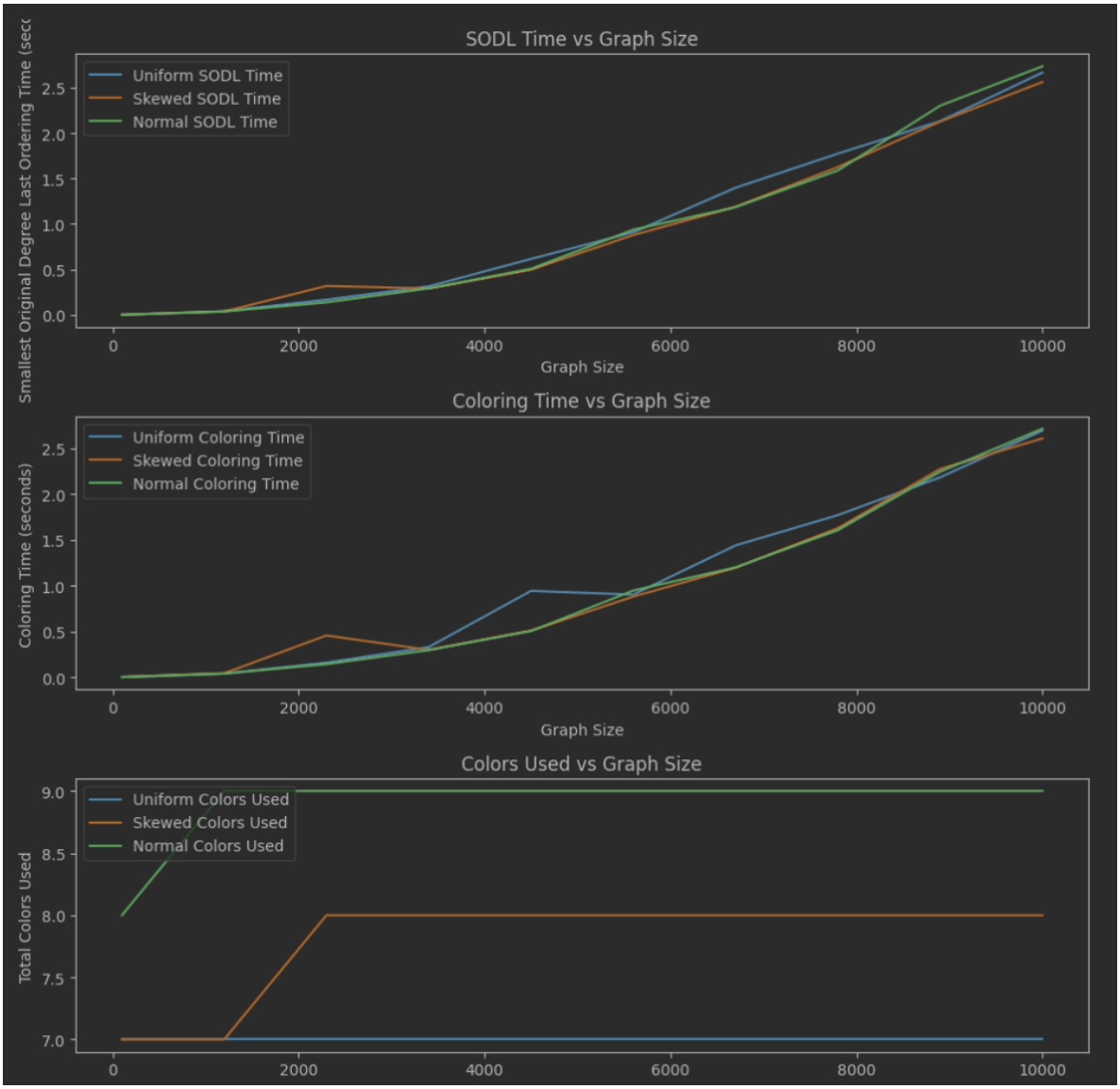
- **Operational Walkthrough**: A detailed example on a small graph demonstrated SODL's ability to keep up with SLO on small graphs.
- **Running Time Analysis:** Empirical data did not support the theoretical THETA(V+E) complexity of the SLO implementation.

- **Coloring Efficiency:** Documentation of the **color count** required for various graphs highlighted that SODL generally required the same amount of colors as SLO except for very specific ranges.

Coloring Algorithm

Our implementation of the SODL algorithm had a time complexity of $O(n^2)$ which is slower than the theoretical THETA(V+E). It also was on par with the SLO algorithm when it came to colors used for larger graphs but did not do as well with graphs of sizes 2000-4000.

Graphs



Detailed Analysis of Uniform Random Ordering (RUD):

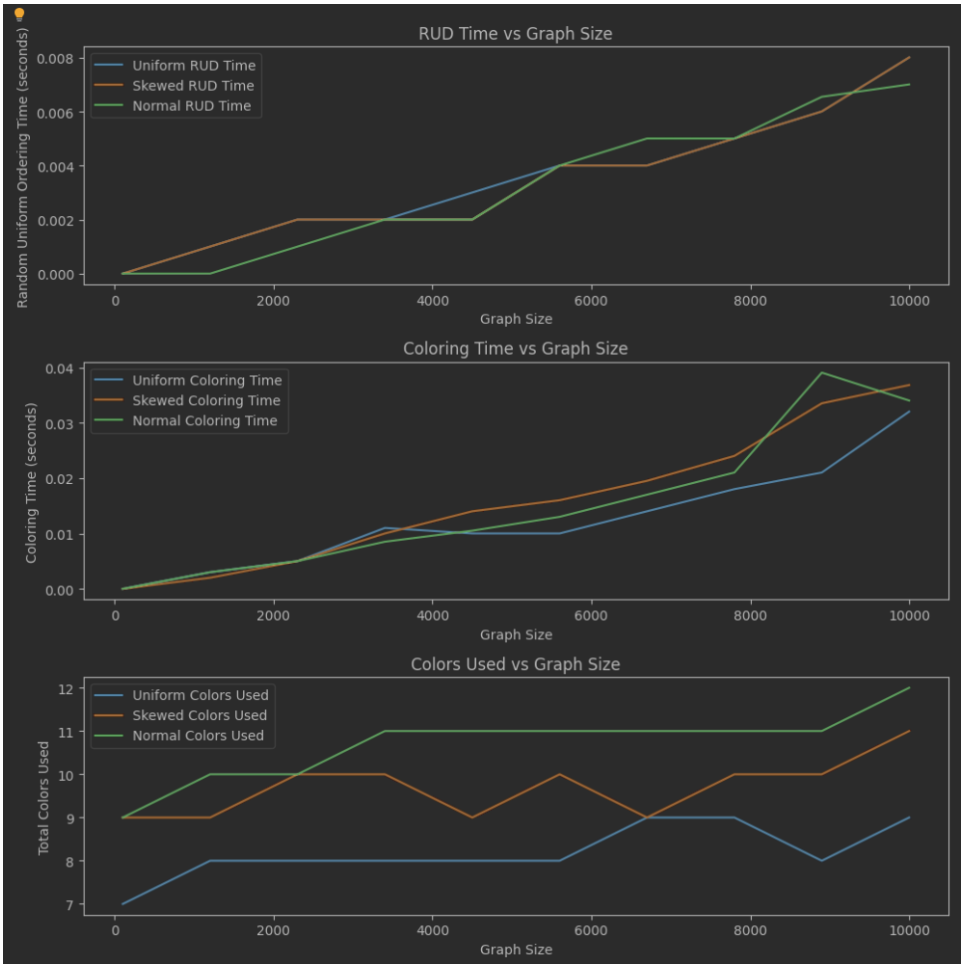
The RUD algorithm is fast but not very good for optimizing the number of colors used when graphing.

- **Operational Walkthrough**: A detailed example on a small graph demonstrated RUD's ability to do the job, just not well.
- **Running Time Analysis:** Empirical data supports a running time of O(n) and vastly outperformed the SLO and SODL algorithms.
- **Coloring Efficiency:** Documentation of the **color count** required for various graphs highlighted that RUD is a suboptimal algorithm to use for coloring since it used 2 to 3 more colors than the other algorithms.

Coloring Algorithm

Our implementation of the RUD algorithm had a time complexity of O(n) which is very fast but the coloring efficiency was lacking a bit for graphs of all sizes.

Graphs



Detailed Analysis of Degree Descending Order (DDO):

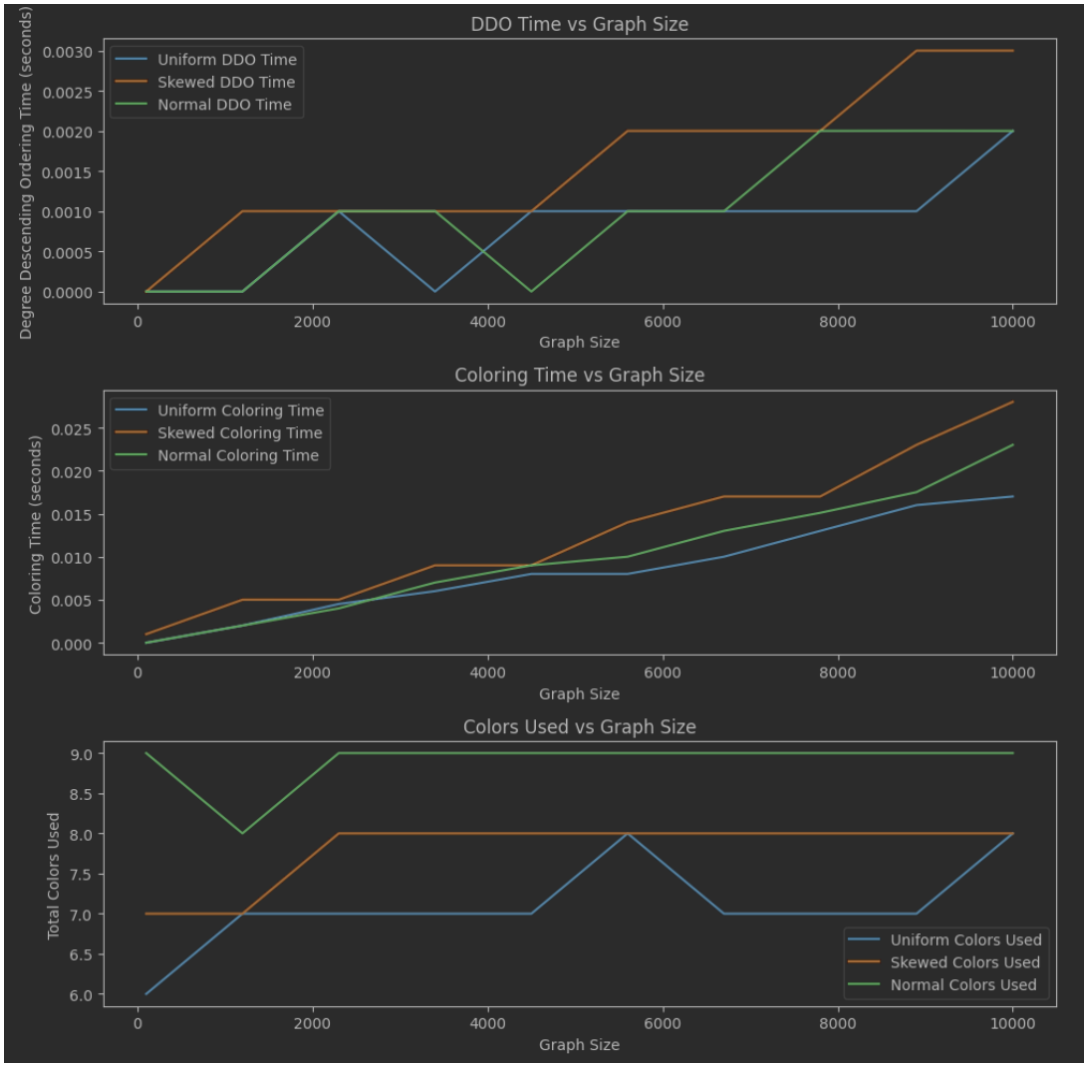
The DDO algorithm was even faster than the RUD algorithm and had very mixed results against the SLO and SODL algorithms when it came to coloring.

- **Operational Walkthrough**: A detailed example on a small graph demonstrated DDO's ability to color graphs both quickly and effectively.
- **Running Time Analysis:** Empirical data supports a running time of O(n) and vastly outperformed the SLO and SODL algorithms.
- **Coloring Efficiency:** Documentation of the **color count** required for various graphs highlighted that DDO was as effective at coloring for the graphs with uniform and skewed distributions but not normal distributions at the smaller scale graphs but other than that it performed almost just as well except for with the uniformly distributed graphs where it went up to 8 colors a couple of times.

Coloring Algorithm

Our implementation of the DDO algorithm has a time complexity of O(n) which is very fast and has great coloring efficiency. Of the four we've looked at so far I would utilize the DDO algorithm if instant results were required.

Graphs



Detailed Analysis of Degree Ascending Order (DAO):

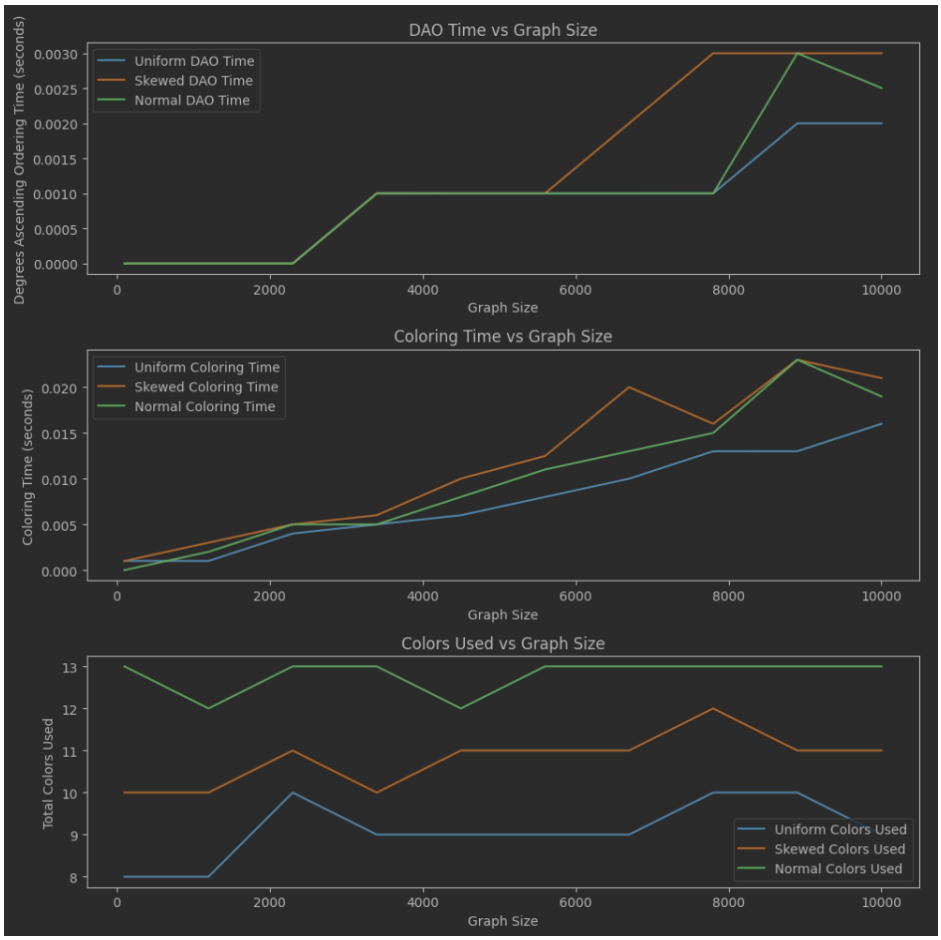
The DAO algorithm has speed to rival the DDO algorithm but it is the worst algorithm for coloring so far. It is the first algorithm to reach 13 colors and for graphs with normal distributions it never went under 12 colors.

- **Operational Walkthrough**: A detailed example on a small graph demonstrated DAO's ability to color graphs quickly but poorly.
- **Running Time Analysis:** Empirical data supports a running time of O(n) and vastly outperformed the SLO and SODL algorithms.
- **Coloring Efficiency:** Documentation of the **color count** required for various graphs highlighted that DAO is not effective for coloring graphs.

Coloring Algorithm

Our implementation of the DAO algorithm had a time complexity of O(n) which is very fast but due to the poor coloring performance I would not use it for coloring at all.

Graphs



Detailed Analysis of Linear Ordering (LIN):

Linear ordering proves to be the fastest ordering by far. The ordering itself seems to happen in constant time. The coloring time is still linear and surprisingly the coloring efficiency is only mildly bad.

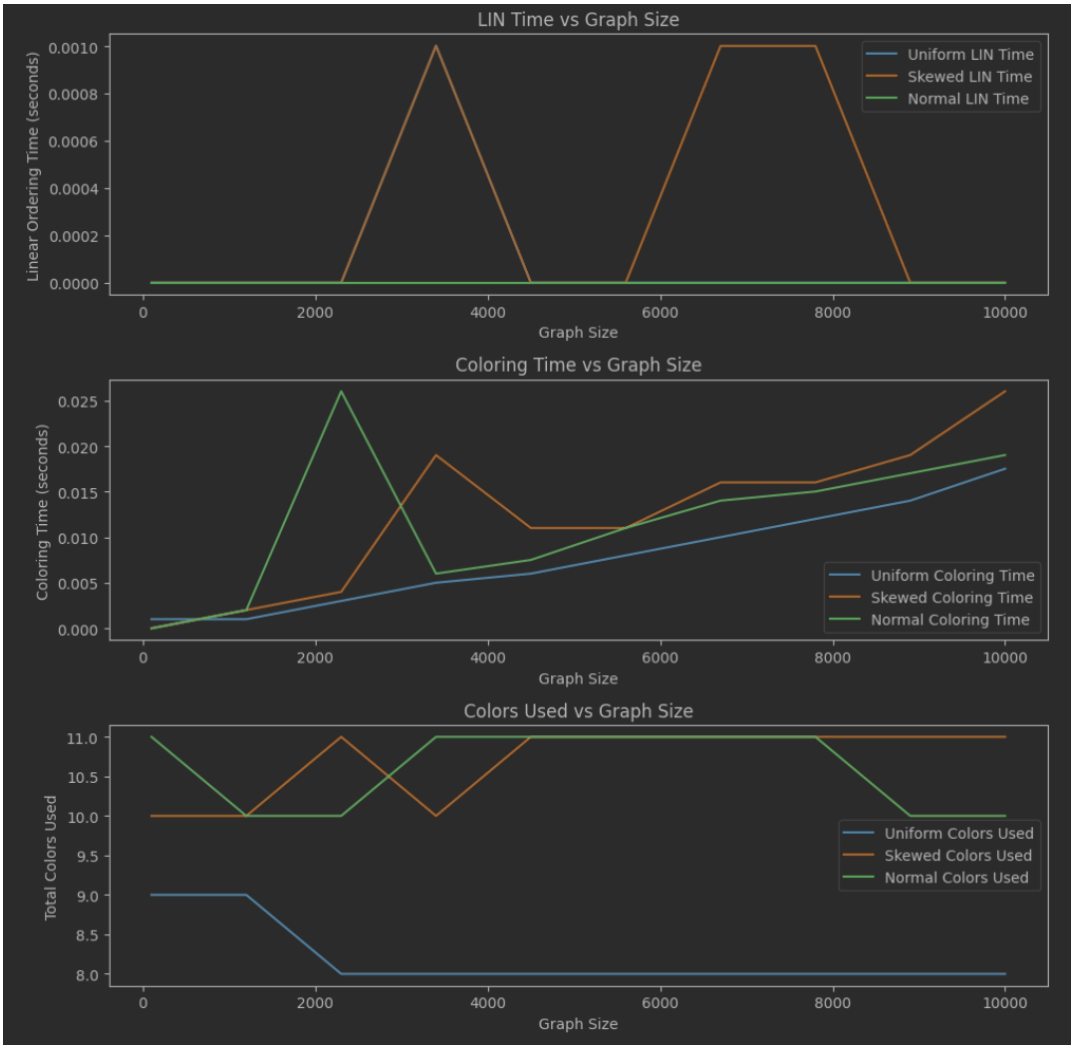
- **Operational Walkthrough**: A detailed example on a small graph demonstrated LIN's ability to color graphs quickly.
- **Running Time Analysis:** Empirical data supports a running time of O(n) and vastly outperformed the SLO and SODL algorithms.

- **Coloring Efficiency:** Documentation of the **color count** required for various graphs highlighted that linear ordering was bad for graphs with either Normal or Skewed distributions, but was surprisingly effective for graphs with Uniform distributions. The best algorithms (SLO and SODL) used only 7 colors for the biggest graphs and Lin managed to only use 8. My assumption is that since the distributions are uniform then you would assume that most vertices have a similar amount of conflicts. This does not explain why it outperforms the RUD and DAO algorithms though. This is the single most surprising result of all.

Coloring Algorithm

Our implementation of the LIN algorithm had a time complexity of O(n) which is very fast but due to the poor coloring for both Skewed and Normal distributions I would refrain from using it in most if not all cases.

Graphs



Use Cases/ Implementations:

Our team has come up with 3 use cases which our graph colorings which can be used by popular social media websites such as LinkedIn:

1. **Event Scheduling:**
    - **Purpose:** Minimize scheduling conflicts for events targeting similar user groups.
    - **Advantages:** Boosts audience numbers by avoiding overlapping events.
    - **Challenges:** Relies on precise user data and doesn't consider personal schedules.
2. **Advertisement Allocation:**
    - **Purpose:** Prevent adjacent placement of competing ads.
    - **Advantages:** Decreases ad competition, potentially enhancing revenue.
    - **Challenges:** Complexity increases with more diverse user and advertiser bases.
3. **Content Recommendation:**
    - **Purpose:** Organize content into unique categories for better delivery.
    - **Advantages:** Ensures a varied stream of content.
    - **Challenges:** Requires continuous adaptation to shifts in user interests and content updates.

During the implementation process, conducting a comparative analysis is necessary to evaluate the balance between simpler, faster algorithms and more comprehensive, resource-intensive options. Discuss the merits of real-time versus batch processing approaches. Evaluate if graph coloring can significantly enhance the efficiency and user experience on LinkedIn, with strategic implementation being key to overcoming inherent data and computational complexities.

Conclusions and Recommendations

The report concluded that the graph coloring scheduler implemented is versatile and efficient, particularly when employing the SLO method. The 2 most recommended coloring algorithms would be the SLO followed by the DDO if you needed rapid results and were willing to be slightly suboptimal.

Limitations and Future Work

The report acknowledged limitations, including the suboptimal performance on certain graph types and suggested areas for further research and development, such as parallel processing and machine learning for vertex ordering prediction.

Appendices

- Source Code
- Raw Data and Analysis Scripts

Source Code:

```
import random
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import time

class Vertex:
    def __init__(self, name):
        self.name = name
        self.conflicts = []  # Pointer to edge list
        self.color = None
        self.degree = 0  # Initial degree
        self.deleted_degree = None
        self.next_same_degree = None  # Pointer to next vertex with the
same degree
        self.prev_same_degree = None  # Pointer to previous vertex with
the same degree
        self.order_deleted_next = None  # Pointer to next vertex in
order deleted list
        self.order_deleted_prev = None  # Pointer to previous vertex in
order deleted list

class Graph:
    def __init__(self, name, size):
        self.name = name
        self.size = size
        self.vertices = {}
        self.order_deleted_head = None  # Head of the ordered list of
deleted vertices

    def add_vertex(self, vertex):
        self.vertices[vertex] = Vertex(vertex)

    def add_conflict(self, vertex1, vertex2):
        if vertex2 not in self.vertices[vertex1].conflicts and vertex1
!= vertex2:
            self.vertices[vertex1].conflicts.append(vertex2)
            self.vertices[vertex2].conflicts.append(vertex1)
            self.vertices[vertex1].degree += 1
```

```
            self.vertices[vertex2].degree += 1

    def smallest_last_ordering(self):
        ordering = []
        degrees = {vertex: len(self.vertices[vertex].conflicts) for
vertex in self.vertices}
        vertices = set(self.vertices.keys())
        deletion_degrees = {}

        while vertices:
            min_vertex = min(vertices, key=lambda x: degrees[x])
            ordering.append(min_vertex)
            deletion_degrees[min_vertex] = degrees[min_vertex]
            for neighbor in self.vertices[min_vertex].conflicts:
                if neighbor in vertices:
                    degrees[neighbor] -= 1
            vertices.remove(min_vertex)
        ordering.reverse()  # We color in the reverse order of removal
        return ordering, deletion_degrees

    def delete_vertex_smallest_degree(self, vertex):
        vertex.deleted = True
        # Update pointers and fields as necessary
        if vertex.prev_same_degree:
            vertex.prev_same_degree.next_same_degree =
vertex.next_same_degree
        if vertex.next_same_degree:
            vertex.next_same_degree.prev_same_degree =
vertex.prev_same_degree
        if self.order_deleted_head:
            vertex.order_deleted_next = self.order_deleted_head
            self.order_deleted_head.order_deleted_prev = vertex
        self.order_deleted_head = vertex

    def smallest_original_degree_last(self):
        ordering = []
        degrees = {vertex: self.vertices[vertex].degree for vertex in
self.vertices}
        vertices = set(self.vertices.keys())

        while vertices:
            min_vertex = min(vertices, key=lambda x: degrees[x])
            ordering.append(min_vertex)
            for neighbor in self.vertices[min_vertex].conflicts:
                if neighbor in vertices:
                    degrees[neighbor] -= 1
            vertices.remove(min_vertex)
        ordering.reverse()  # We color in the reverse order of removal
        return ordering

    def random_uniform_ordering(self):
        ordering = []
        temp_list = list(range(self.size))
        for i in range(self.size - 1, -1, -1):
            vertex_num = temp_list.pop(random.randint(0,i))
            #ordering.append(self.vertices[vertex_num])
            ordering.append(vertex_num)
```

```
        return ordering

    def degree_descending_order(self):
        degrees = {vertex: self.vertices[vertex].degree for vertex in
self.vertices}
        sorted_vertices = sorted(degrees, key=degrees.get,
reverse=True)
        return sorted_vertices

    def degree_ascending_order(self):
        degrees = {vertex: self.vertices[vertex].degree for vertex in
self.vertices}
        sorted_vertices = sorted(degrees, key=degrees.get)
        return sorted_vertices

    def linear_ordering(self):
        ordering = list(range(self.size))
        return ordering

    def greedy_coloring_with_slo(self, type = 'slo'):
        color = {}
        if type == 'slo':
            vertex_order, deletion_degrees =
self.smallest_last_ordering()
            for vertex in vertex_order:

self.delete_vertex_smallest_degree(self.vertices[vertex])
                self.vertices[vertex].deleted_degree =
deletion_degrees[vertex]
                forbidden = {color[neighbor] for neighbor in
self.vertices[vertex].conflicts if neighbor in color}
                color[vertex] = next(c for c in
range(len(self.vertices)) if c not in forbidden)
                self.vertices[vertex].color = color[vertex]
            return color
        elif type == 'sodl':
            vertex_order = self.smallest_original_degree_last()
        elif type =='rud':
            vertex_order = self.random_uniform_ordering()
        elif type == 'ddo':
            vertex_order = self.degree_descending_order()
        elif type == 'dao':
            vertex_order = self.degree_ascending_order()
        elif type == 'linear':
            vertex_order = self.linear_ordering()
        for vertex in vertex_order:
            forbidden = {color[neighbor] for neighbor in
self.vertices[vertex].conflicts if neighbor in color}
            color[vertex] = next(c for c in range(len(self.vertices))
if c not in forbidden)
            self.vertices[vertex].color = color[vertex]
        return color

    def plot(self, type):
        color_map = self.greedy_coloring_with_slo(type)
        G = self.to_networkx_graph()
        pos = nx.spring_layout(G)
```

```
        colors = [color_map[node] for node in G.nodes()]
        nx.draw(G, pos, with_labels=True, node_color=colors,
cmap=plt.get_cmap(
            'tab20'), node_size=700, font_size=9, edge_color='gray')
        plt.title(f"Graph: {self.name}")
        plt.show()

    def to_networkx_graph(self):
        G = nx.Graph()
        for vertex, vertex_obj in self.vertices.items():
            G.add_node(vertex)
            for conflict in vertex_obj.conflicts:
                G.add_edge(vertex, conflict)
        return G

    def output_stats_and_file(self, filename, type = 'slo'):
        color_map = self.greedy_coloring_with_slo(type)
        self.plot(type)  # Plot the graph with colored nodes
        unique_colors = set(color_map.values())
        total_degrees = [v.degree for v in self.vertices.values()]
        deleted_degrees = [v.deleted_degree for v in
self.vertices.values()]

        print(f"Total number of colors used: {len(unique_colors)}")
        print(f"Average original degree: {np.mean(total_degrees):.2f}")
        print(f"Maximum degree when deleted: {max(deleted_degrees)}")
        # Assuming maximum deleted degree
        print(f"Size of the terminal clique: {max(deleted_degrees)}")

        with open(filename, 'w') as file:
            for vertex in sorted(self.vertices, key=lambda v:
self.vertices[v].color):
                file.write(f"{vertex},{self.vertices[vertex].color}\n")
                print(
                    f"Vertex {vertex}: Color
{self.vertices[vertex].color}, Original Degree
{self.vertices[vertex].degree}, Degree When Deleted
{self.vertices[vertex].deleted_degree}")

    def analyze_graph(self, name, size, number_of_conflicts=None):
        slo_runtime = self.measure_runtime(self.smallest_last_ordering)
        sodl_runtime =
self.measure_runtime(self.smallest_original_degree_last)
        rud_runtime =
self.measure_runtime(self.random_uniform_ordering)
        ddo_runtime =
self.measure_runtime(self.degree_descending_order)
        dao_runtime = self.measure_runtime(self.degree_ascending_order)
        lin_runtime = self.measure_runtime(self.linear_ordering)
        slo_coloring_runtime = self.measure_runtime(lambda:
self.greedy_coloring_with_slo('slo'))
        sodl_coloring_runtime = self.measure_runtime(lambda:
self.greedy_coloring_with_slo('sodl'))
        rud_coloring_runtime = self.measure_runtime(lambda:
self.greedy_coloring_with_slo('rud'))
        ddo_coloring_runtime = self.measure_runtime(lambda:
self.greedy_coloring_with_slo('ddo'))
```

```
        dao_coloring_runtime = self.measure_runtime(lambda:
self.greedy_coloring_with_slo('dao'))
        lin_coloring_runtime = self.measure_runtime(lambda:
self.greedy_coloring_with_slo('linear'))
        # Print runtime statistics
        print("SLO Runtime:", slo_runtime)
        print("SODL Runtime:", sodl_runtime)
        print("RUD Runtime:", rud_runtime)
        print("DDO Runtime:", ddo_runtime)
        print("DAO Runtime:", dao_runtime)
        print("LIN Runtime:", lin_runtime)
        print("Coloring Runtime for SLO:", slo_coloring_runtime)
        print("Coloring Runtime for SODL:", sodl_coloring_runtime)
        print("Coloring Runtime for RUD:", rud_coloring_runtime)
        print("Coloring Runtime for DDO:", ddo_coloring_runtime)
        print("Coloring Runtime for DAO:", dao_coloring_runtime)
        print("Coloring Runtime for LIN:", lin_coloring_runtime)

        # Generate histograms
        self.generate_histogram()

    def analyze_graph_run_time(self, name, size, graph_type,
number_of_conflicts=None):
        runtime_data = {'sizes': [], 'runtimes': []}
        start_time = time.time()
        create_graph(name, size, graph_type, number_of_conflicts)
        end_time = time.time()
        runtime = end_time - start_time
        runtime_data['sizes'].append(size)
        runtime_data['runtimes'].append(runtime)
        print(f"Graph size: {size}, Runtime: {runtime:.6f} seconds")


    def measure_runtime(self, method):
        start_time = time.time()
        method()
        end_time = time.time()
        return end_time - start_time

    def generate_histogram(self, label = ''):
        conflicts_per_vertex_slo = [len(v.conflicts) for v in
self.vertices.values()]
        #conflicts_per_vertex_sodl = [v.degree for v in
self.vertices.values()]

        plt.hist(conflicts_per_vertex_slo,
bins=range(max(conflicts_per_vertex_slo) + 2), align='left', alpha=0.5,
label= label)
        #plt.hist(conflicts_per_vertex_sodl,
bins=range(max(conflicts_per_vertex_sodl) + 2), align='left',
alpha=0.5, label='SODL')

        plt.xlabel('Number of Conflicts')
        plt.ylabel('Number of Vertices')
        plt.title('Distribution of Conflicts per Vertex')
        plt.legend()
        plt.show()
```

```
def create_graph(name, size, graph_type, number_of_conflicts=None):
    graph = Graph(name, size)
    for i in range(size):
        graph.add_vertex(i)

    if graph_type == 'uniform':
        while number_of_conflicts > 0:
            v1, v2 = random.randint(0, size-1), random.randint(0, size-
1)
            if v2 not in graph.vertices[v1].conflicts and v1 != v2:
                graph.add_conflict(v1, v2)
                number_of_conflicts -= 1
    elif graph_type == 'skewed':
        probabilities = np.linspace(0, 1, size)
        # Normalize to form a probability distribution
        probabilities /= probabilities.sum()
        while number_of_conflicts > 0:
            v1, v2 = np.random.choice(size, 2, p=probabilities)
            if v2 not in graph.vertices[v1].conflicts and v1 != v2:
                graph.add_conflict(v1, v2)
                number_of_conflicts -= 1
    elif graph_type == 'normal':
        while number_of_conflicts > 0:
            v1, v2 = np.random.normal(size/2, size/6, 2).astype(int)
            v1, v2 = np.clip(v1, 0, size-1), np.clip(v2, 0, size-1)
            if v2 not in graph.vertices[v1].conflicts and v1 != v2:
                graph.add_conflict(v1, v2)
                number_of_conflicts -= 1

    return graph
# Example usage
uniform_graph = create_graph('Uniform Distribution Graph', 10,
'uniform', 15)
uniform_graph.output_stats_and_file('uniform_vertex_color_output.csv')
uniform_graph.analyze_graph('Uniform Distribution Graph', size=10,
number_of_conflicts=15)
uniform_graph.analyze_graph_run_time('Uniform Distribution Graph', 10,
'uniform', 15)

skewed_graph = create_graph('Skewed Distribution Graph', 10, 'skewed',
15)
skewed_graph.output_stats_and_file('skewed_vertex_color_output.csv')
skewed_graph.analyze_graph(name='Skewed Distribution Graph', size=10,
number_of_conflicts=15)
skewed_graph.analyze_graph_run_time('Skewed Distribution Graph', 10,
'uniform', 15)


normal_graph = create_graph('Normal Distribution Graph', 10, 'normal',
15)
normal_graph.output_stats_and_file('normal_vertex_color_output.csv')
normal_graph.analyze_graph(name='Normal Distribution Graph', size=10,
number_of_conflicts=15)
normal_graph.analyze_graph_run_time('Skewed Distribution Graph', 10,
'uniform', 15)
```

Raw Data and Analysis Scripts:

The code for all the analysis was the same but we called it with the different algorithms. In order to not take up more pages we will just show this one for the SLO algorithm.

```
import random
import time
import numpy as np
import matplotlib.pyplot as plt

def measure_time(graph_function, *args, **kwargs):
    start_time = time.time()
    result = graph_function(*args, **kwargs)
    end_time = time.time()
    return result, end_time - start_time

def analyze_graph(graph):
    _, slo_time = measure_time(graph.smallest_last_ordering)
    color_map, coloring_time = measure_time(graph.greedy_coloring_with_slo)
    unique_colors = len(set(color_map.values()))
    E = sum(len(v.conflicts) for v in graph.vertices.values()) // 2
    V = len(graph.vertices)
    density = 2 * E / (V * (V - 1)) if V > 1 else 0
    average_degree = 2 * E / V

    print(f"Graph Analysis for '{graph.name}':")
    print(f"  Time for Smallest Last Ordering: {slo_time:.4f} seconds")
    print(f"  Time for Coloring: {coloring_time:.4f} seconds")
    print(f"  Total Number of Colors Used: {unique_colors}")
    print(f"  Number of Vertices (V): {V}")
    print(f"  Number of Edges (E): {E}")
    print(f"  Graph Density: {density:.4f}")
    print(f"  Average Degree: {average_degree:.2f}")

    return unique_colors, slo_time, coloring_time, density, average_degree

sizes = np.linspace(100, 10000, 10).astype(int)
conflicts = np.linspace(500, 50000, 10).astype(int)

slo_times = {'Uniform': [], 'Skewed': [], 'Normal': []}
coloring_times = {'Uniform': [], 'Skewed': [], 'Normal': []}
total_colors_used = {'Uniform': [], 'Skewed': [], 'Normal': []}

for size, conflict in zip(sizes, conflicts):
    uniform_graph = create_graph('Uniform Distribution Graph',
size,'uniform', conflict)
    total_colors, slo_time, coloring_time, _, _ =
analyze_graph(uniform_graph)
    slo_times['Uniform'].append(slo_time)
    coloring_times['Uniform'].append(coloring_time)
    total_colors_used['Uniform'].append(total_colors)
```

```
    skewed_graph = create_graph('Skewed Distribution Graph', size, 'skewed',
conflict)
    total_colors, slo_time, coloring_time, _, _ = analyze_graph(skewed_graph)
    slo_times['Skewed'].append(slo_time)
    coloring_times['Skewed'].append(coloring_time)
    total_colors_used['Skewed'].append(total_colors)

    normal_graph = create_graph('Normal Distribution Graph', size, 'normal',
conflict)
    total_colors, slo_time, coloring_time, _, _ = analyze_graph(normal_graph)
    slo_times['Normal'].append(slo_time)
    coloring_times['Normal'].append(coloring_time)
    total_colors_used['Normal'].append(total_colors)

# Plotting results
fig, ax = plt.subplots(3, 1, figsize=(10, 10))
ax[0].plot(sizes, slo_times['Uniform'], label='Uniform SLO Time')
ax[0].plot(sizes, slo_times['Skewed'], label='Skewed SLO Time')
ax[0].plot(sizes, slo_times['Normal'], label='Normal SLO Time')
ax[0].set_xlabel('Graph Size')
ax[0].set_ylabel('Smallest Last Ordering Time (seconds)')
ax[0].set_title('SLO Time vs Graph Size')
ax[0].legend()

ax[1].plot(sizes, coloring_times['Uniform'], label='Uniform Coloring Time')
ax[1].plot(sizes, coloring_times['Skewed'], label='Skewed Coloring Time')
ax[1].plot(sizes, coloring_times['Normal'], label='Normal Coloring Time')
ax[1].set_xlabel('Graph Size')
ax[1].set_ylabel('Coloring Time (seconds)')
ax[1].set_title('Coloring Time vs Graph Size')
ax[1].legend()

ax[2].plot(sizes, total_colors_used['Uniform'], label='Uniform Colors Used')
ax[2].plot(sizes, total_colors_used['Skewed'], label='Skewed Colors Used')
ax[2].plot(sizes, total_colors_used['Normal'], label='Normal Colors Used')
ax[2].set_xlabel('Graph Size')
ax[2].set_ylabel('Total Colors Used')
ax[2].set_title('Colors Used vs Graph Size')
ax[2].legend()

plt.tight_layout()
plt.show()
```