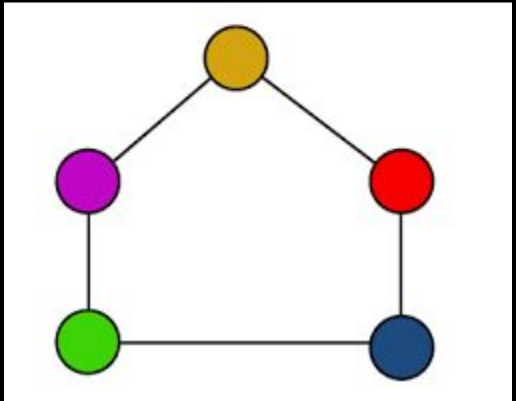
# Augmented Solution

# Quick Recap

## Min Graph Coloring Problem

What is the minimum number of colors needed to color a graph G, such that no two adjacent vertices share the same color?



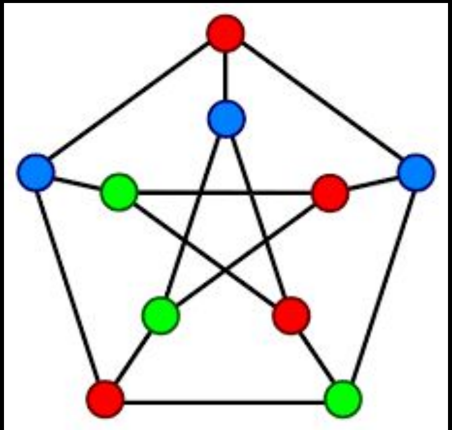


Figure 1, Figure 2

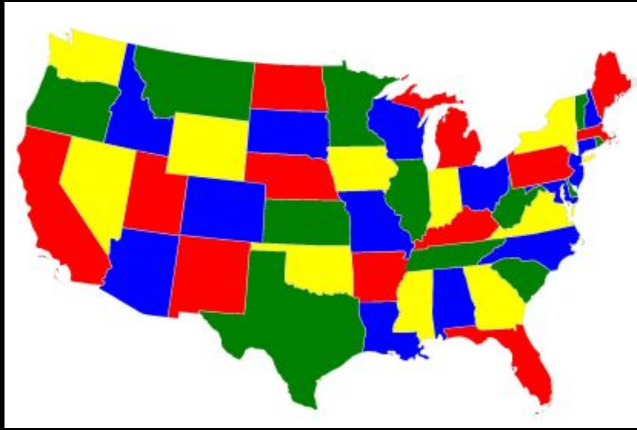
# Why is this important?

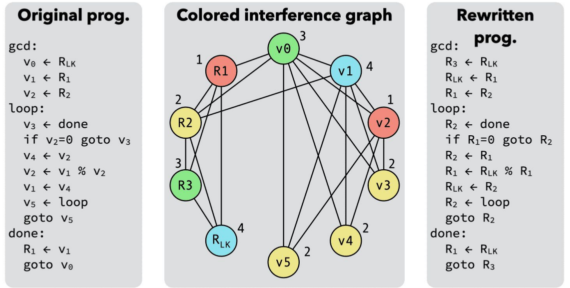
## Min Graph Coloring is NP-complete

- There is no known polynomial-time algorithm that solves the problem optimally for all instances

- Optimal solutions are expensive (exponential time), so approximation methods are more practical and commonly used

- Real world applications include frequency assignment in wireless networks, register allocation in compilers, scheduling problems, and map coloring






This second coloring is also correct, but produces worse code!

Figure 1, Figure 2

# Exact Solution

## Backtracking algorithm

- Try to color the graph with just one color, incrementally increases the number of colors as necessary
- If a conflict arises (a  color cannot be safely added), the algorithm backtracks, trying a different color
- Repeat recursively until a valid coloring is found
- Time complexity: $O(m^V)$

```
class Graph:
    function __init__(vertices):
        initialize vertices
        create empty adjacency list for each vertex

    function add_edge(u, v):
        add v to adjacency list of u
        add u to adjacency list of v

    function is_safe(vertex, color_assignment, color):
        for each neighbor in adjacency list of vertex:
            if neighbor's color is the same as color:
                return False
        return True

    function graph_color_util(color_assignment, colors, index):
        if index equals number of vertices:
            return True

        current_vertex = vertices[index]
        for each color in colors:
            if is_safe(current_vertex, color_assignment, color):
                assign color to current_vertex
                if graph_color_util(color_assignment, colors, index + 1):
                    return True
                remove color assignment from current_vertex
        return False

    function find_min_coloring():
        for num_colors from 1 to number of vertices:
            initialize empty color_assignment
            colors = list of colors from 0 to num_colors - 1
            if graph_color_util(color_assignment, colors, 0):
                return num_colors, color_assignment
        return None

function main():
    read number of edges
    read each edge and store in edges list
    extract and sort all unique vertices
    create Graph instance with vertices
    add all edges to the graph
    start timer
    num_colors, color_assignment = graph.find_min_coloring()
    print num_colors
    print elapsed time
```

# Approximate Solution

## Greedy algorithm

- Creates a python dictionary *color* where each vertex is assigned a value of -1
- Select the first vertex and assign it the first color
- Iterate through each vertex
- For uncolored vertices, initialize a list of available colors
- Mark colors used by adjacent vertices as unavailable
- Assign the first available color to the current vertex
- Time complexity: $O(V^2 + E)$

```python
def greedy_coloring(graph):
    # Initialize color dictionary with -1 for all vertices
    color = {vertex: -1 for vertex in graph}

    # Assign first color to the first vertex
    first_vertex = next(iter(graph))
    color[first_vertex] = 0

    # Assign colors to remaining vertices
    for vertex in graph:
        if color[vertex] != -1:
            continue  # Skip already colored vertex

        # Initialize all colors as available
        available_colors = [True] * len(graph)

        # Mark colors of adjacent vertices as unavailable
        for neighbor in graph[vertex]:
            if color[neighbor] != -1:
                available_colors[color[neighbor]] = False

        # Find the first available color
        for c in range(len(graph)):
            if available_colors[c]:
                color[vertex] = c
                break

    # Return color assignment
    return color
```

## Exact

Exponential

Very slow

Produces correct results

## Approximate

Polynomial

Very fast

May not produce an optimal coloring

non_optimal_v15_e200_2.txt

```
Running test case: non_optimal_v15_e200_2.txt
Exact Colors: 7, Time: 232.208240 seconds
Approx Colors: 9, Time: 0.000225 seconds
```

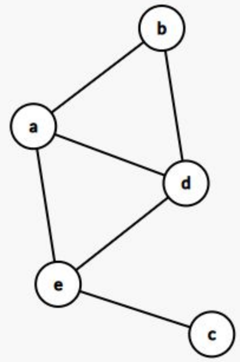
# Can we improve the approximation?

Performance heavily depends on the order in which vertices are processed

E is processed last, forcing a new color

non_optimal_v5_e6.txt

```
6
b a
b d
d a
c e
e a
e d
# Running test case: non_optimal_v5_e6.txt
# Exact Colors: 3, Time: 0.000032 seconds
# Approx Colors: 4, Time: 0.000025 seconds
```



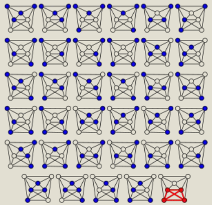
# Augmenting the approximate solution

## DSatur heuristic graph coloring algorithm

- Daniel Brélaz invented the DSatur algorithm in 1979
- Similar to greedy, but chooses the next vertex to color based on the number of colors already used by its neighbors (with highest saturation)
- Defined as the degree of saturation of a given vertex (hence the name)
- Dynamically orders vertices
- Not always optimal, but produces correct results for bipartite, cycle, and wheel graphs
- Time complexity: O(V + E) with additional overhead to update saturation

```python
def dsatur_coloring(graph):
    # Initialize color assignment for each vertex to -1 (uncolored)
    color = {vertex: -1 for vertex in graph}

    # Initialize saturation and degree for each vertex
    saturation = {vertex: 0 for vertex in graph}
    degree = {vertex: len(neighbors) for vertex, neighbors in graph.items()}

    # Select the vertex with the highest degree to start
    current_vertex = vertex_with_highest_degree(graph, degree)
    color[current_vertex] = 0  # Assign the first color

    # Remove the colored vertex from the set of uncolored vertices
    uncolored_vertices = set(graph.keys())
    uncolored_vertices.remove(current_vertex)

    # Update saturation for neighbors of the first colored vertex
    update_saturation(saturation, graph, current_vertex)

    while uncolored_vertices:
        # Select the uncolored vertex with the highest saturation
        current_vertex = select_highest_saturation_vertex(uncolored_vertices, saturation, degree)

        # Determine the smallest available color not used by neighbors
        available_color = find_smallest_available_color(graph, color, current_vertex)
        color[current_vertex] = available_color  # Assign the color

        # Remove the vertex from the set of uncolored vertices
        uncolored_vertices.remove(current_vertex)

        # Update saturation for neighbors of the newly colored vertex
        update_saturation(saturation, graph, current_vertex)

    return color

# Helper function to select the vertex with the highest degree
def vertex_with_highest_degree(graph, degree):
    return max(graph.keys(), key=lambda v: degree[v])

# Helper function to update saturation values
def update_saturation(saturation, graph, colored_vertex):
    for neighbor in graph[colored_vertex]:
        if color[neighbor] == -1:
            saturation[neighbor] = calculate_saturation(graph, color, neighbor)

# Helper function to calculate saturation for a vertex
def calculate_saturation(graph, color, vertex):
    return len(set(color[neighbor] for neighbor in graph[vertex] if color[neighbor] != -1))

# Helper function to select the next vertex to color
def select_highest_saturation_vertex(uncolored, saturation, degree):
    return max(
        uncolored,
        key=lambda v: (saturation[v], degree[v])
    )

# Helper function to find the smallest available color for a vertex
def find_smallest_available_color(graph, color, vertex):
    used_colors = {color[neighbor] for neighbor in graph[vertex] if color[neighbor] != -1}
    smallest_color = 0
    while smallest_color in used_colors:
        smallest_color += 1
    return smallest_color
```

Chromatic Number (x): The smallest number of colors needed to achieve a valid coloring.



*A clique in a graph is a set of vertices all of which are pairwise adjacent. The chromatic number of a graph is at least the size of the largest clique in the graph.*

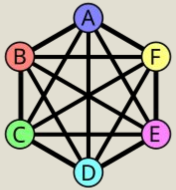
# Lower Bound

- A guarantee that the chromatic number is at least this value
- Can be defined as the size of the largest found clique in the graph
- Max clique is NP-complete, no known polynomial solution
- Instead, we can use a heuristic approach
- Check for triangles (LB=3) or presence of edges (LB=2)

Lower Bound ≤ Chromatic Number ≤ Upper Bound

```python
def compute_bounds(graph):
    # Check if the graph is empty
    if not graph:
        return (1, 1)

    # Compute Upper Bound
    # Delta (Δ) is the maximum degree of any vertex in the graph
    delta = max(len(neighbors) for neighbors in graph.values()) if graph else 0
    upper_bound = delta + 1  # Based on Brooks' Theorem

    # Compute Lower Bound
    has_edge = False
    for vertex in graph:
        if graph[vertex]:  # If the vertex has at least one neighbor
            has_edge = True
            break

    if not has_edge:
        lower_bound = 1  # No edges means only one color is needed
    else:
        found_triangle = False
        for vertex in graph:
            neighbors = graph[vertex]
            if len(neighbors) < 2:
                continue  # Need at least two neighbors to form a triangle
            # Check all pairs of neighbors to find a triangle
            for i in range(len(neighbors)):
                for j in range(i + 1, len(neighbors)):
                    neighbor1 = neighbors[i]
                    neighbor2 = neighbors[j]
                    if neighbor2 in graph.get(neighbor1, []):
                        found_triangle = True
                        break
                if found_triangle:
                    break
            if found_triangle:
                break

        if found_triangle:
            lower_bound = 3  # A triangle requires at least three colors
        else:
            lower_bound = 2  # At least two colors are needed if there are edges but no triangles

    return (lower_bound, upper_bound)
```

Chromatic Number (x): The smallest number of colors needed to achieve a valid coloring.



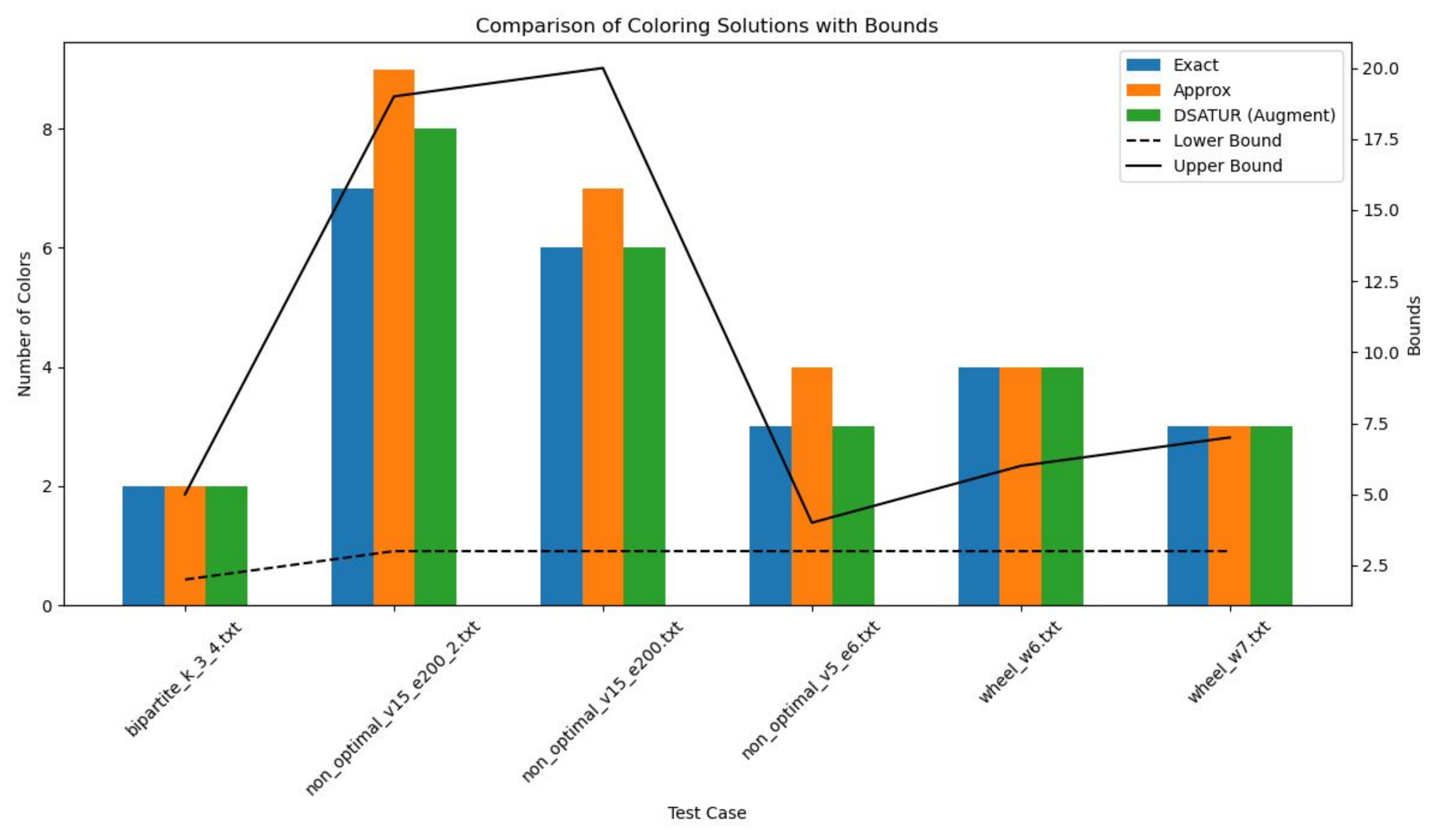

*Complete graphs need one more color than their maximum degree. They and the odd cycles are the only exceptions to Brooks' theorem.*

# Upper Bound

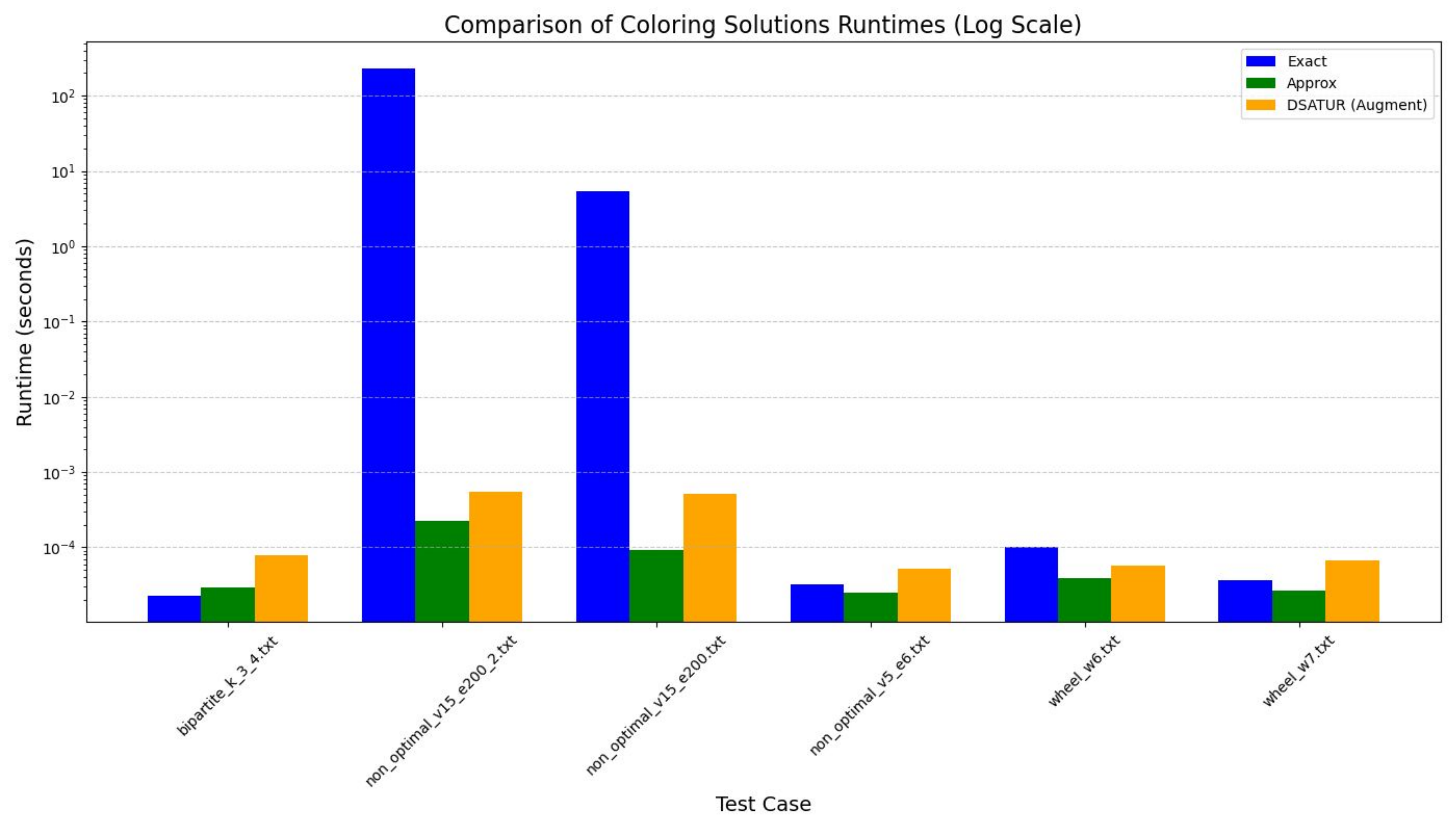
- Brooks' theorem states a relationship between the maximum degree of a graph and its chromatic number
- A connected graph in which every vertex has at most Δ neighbors, the vertices can be colored with only Δ colors
- Exception of two cases, complete graphs and cycle graphs of odd length, which require Δ + 1 colors
- Simply adding 1 guarantees the upper bound holds
- Calculated as Δ + 1 using Brooks' theorem in polynomial time

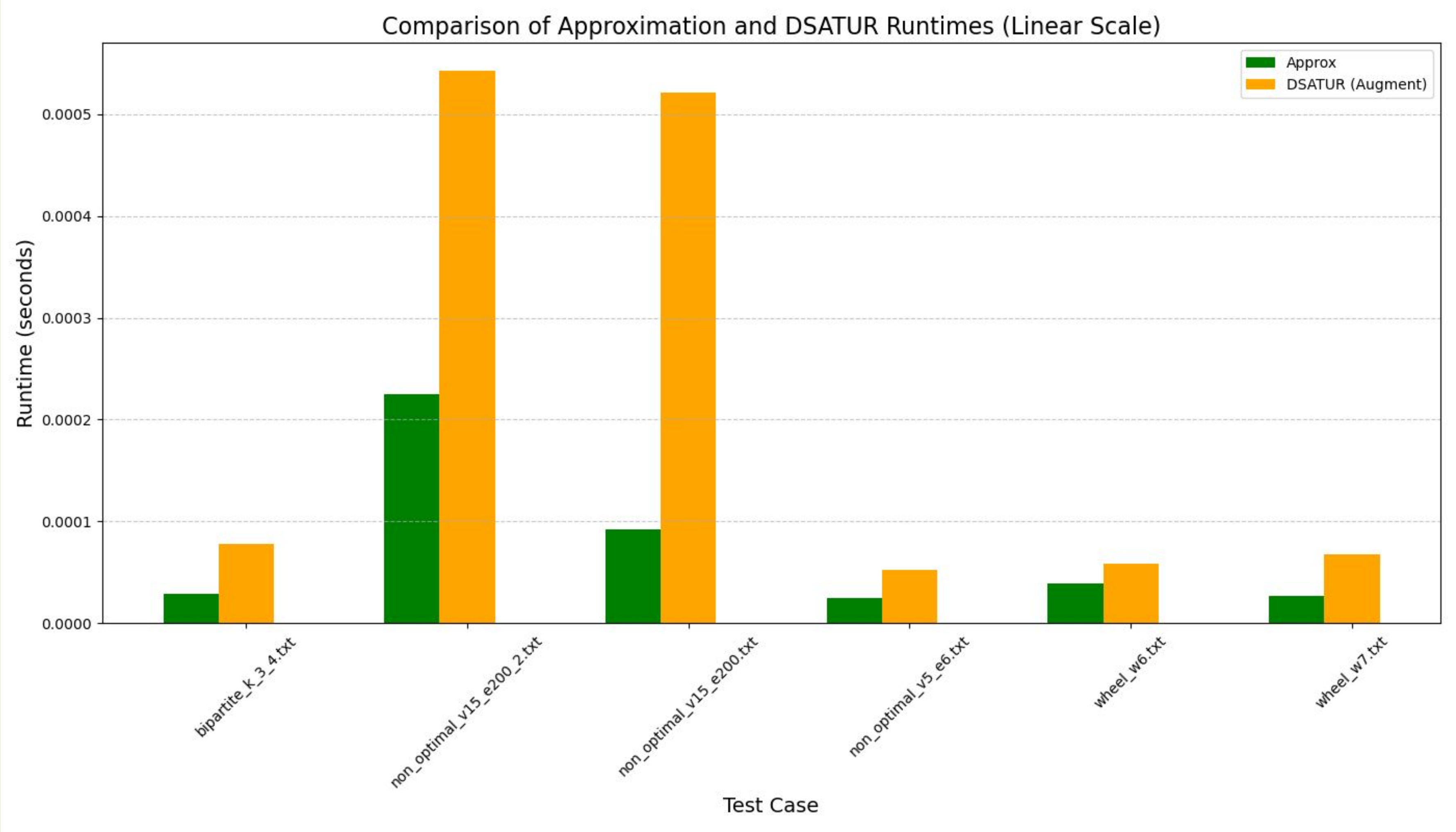
Lower Bound ≤ Chromatic Number ≤ Upper Bound

```python
def compute_bounds(graph):
    # Check if the graph is empty
    if not graph:
        return (1, 1)

    # Compute Upper Bound
    # Delta (Δ) is the maximum degree of any vertex in the graph
    delta = max(len(neighbors) for neighbors in graph.values()) if graph else 0
    upper_bound = delta + 1  # Based on Brooks' Theorem

    # Compute Lower Bound
    has_edge = False
    for vertex in graph:
        if graph[vertex]:  # If the vertex has at least one neighbor
            has_edge = True
            break

    if not has_edge:
        lower_bound = 1  # No edges means only one color is needed
    else:
        found_triangle = False
        for vertex in graph:
            neighbors = graph[vertex]
            if len(neighbors) < 2:
                continue  # Need at least two neighbors to form a triangle
            # Check all pairs of neighbors to find a triangle
            for i in range(len(neighbors)):
                for j in range(i + 1, len(neighbors)):
                    neighbor1 = neighbors[i]
                    neighbor2 = neighbors[j]
                    if neighbor2 in graph.get(neighbor1, []):
                        found_triangle = True
                        break
                if found_triangle:
                    break
            if found_triangle:
                break

        if found_triangle:
            lower_bound = 3  # A triangle requires at least three colors
        else:
            lower_bound = 2  # At least two colors are needed if there are edges but no triangles

    return (lower_bound, upper_bound)
```



Comparison of Coloring Solutions with Bounds
Exact
Approx
DSATUR (Augment)
Lower Bound
Upper Bound
Number of Colors
Bounds
Test Case
0
2
4
6
8
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
bipartite_k_3_4.txt
non_optimal_v15_e200_2.txt
non_optimal_v15_e200.txt
non_optimal_v5_e6.txt
wheel_w6.txt
wheel_w7.txt


Comparison of Coloring Solutions Runtimes (Log Scale)
Exact
Approx
DSATUR (Augment)
Runtime (seconds)
$10^{2}$
$10^{1}$
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
bipartite_k_3_4.txt
non_optimal_v15_e200_2.txt
non_optimal_v15_e200.txt
non_optimal_v5_e6.txt
wheel_w6.txt
wheel_w7.txt
Test Case


Comparison of Approximation and DSATUR Runtimes (Linear Scale)
Approx
DSATUR (Augment)
Runtime (seconds)
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
bipartite_k_3_4.txt
non_optimal_v15_e200_2.txt
non_optimal_v15_e200.txt
non_optimal_v5_e6.txt
wheel_w6.txt
wheel_w7.txt
Test Case

# Conclusion

- DSATUR algorithm improves upon the greedy approximation, providing better colors with slight overhead (still polynomial)

- Heuristic bounds help contextualize the quality of our approximations, especially on large graphs where exact solutions are impractical

run_compare_test_cases.sh

```
Running test case: bipartite_k_3_4.txt
Exact Colors: 2, Time: 0.000023 seconds
Approx Colors: 2, Time: 0.000029 seconds
Augment Colors: 2, Time: 0.000078 seconds
Lower Bound: 2
Upper Bound: 5
----------------------------
Running test case: non_optimal_v15_e200_2.txt
Exact Colors: 7, Time: 232.208240 seconds
Approx Colors: 9, Time: 0.000225 seconds
Augment Colors: 8, Time: 0.000543 seconds
Lower Bound: 3
Upper Bound: 19
----------------------------
Running test case: non_optimal_v15_e200.txt
Exact Colors: 6, Time: 5.332889 seconds
Approx Colors: 7, Time: 0.000092 seconds
Augment Colors: 6, Time: 0.000521 seconds
Lower Bound: 3
Upper Bound: 20
----------------------------
Running test case: non_optimal_v5_e6.txt
Exact Colors: 3, Time: 0.000032 seconds
Approx Colors: 4, Time: 0.000025 seconds
Augment Colors: 3, Time: 0.000052 seconds
Lower Bound: 3
Upper Bound: 4
----------------------------
Running test case: wheel_w6.txt
Exact Colors: 4, Time: 0.000103 seconds
Approx Colors: 4, Time: 0.000039 seconds
Augment Colors: 4, Time: 0.000058 seconds
Lower Bound: 3
Upper Bound: 6
----------------------------
Running test case: wheel_w7.txt
Exact Colors: 3, Time: 0.000037 seconds
Approx Colors: 3, Time: 0.000027 seconds
Augment Colors: 3, Time: 0.000067 seconds
Lower Bound: 3
Upper Bound: 7
----------------------------
```