CS301 Project Presentation

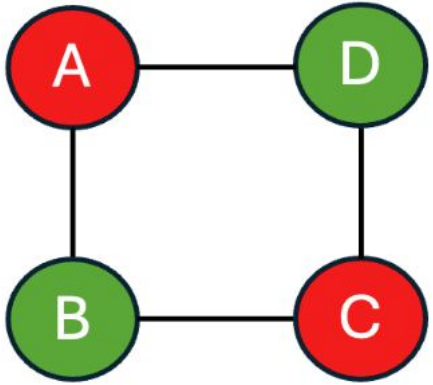
Group Members:

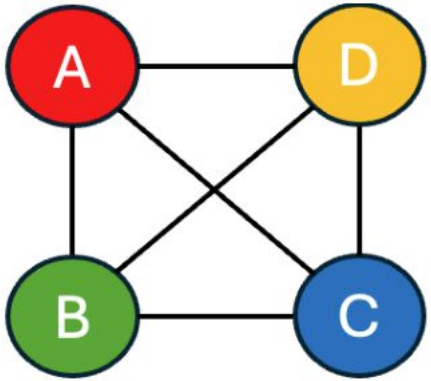
Ayça Elif Aktaş - 27802

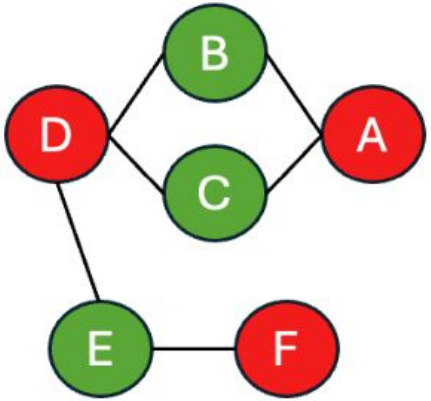
Mehmet Talha Güven - 27829

1. Problem Description

- Graph coloring is the process of assigning colors to vertices so that no two adjacent vertices have the same color.
- The Graph Coloring Problem consists of determining the minimum number of different colors in order to carry out a coloring of graph itself.
- Some example graphs:







Real World Applications

- **Map Coloring:** Assigning colors to regions on a map such that adjacent regions do not share the same color is known as the classic map coloring issue. Applications of this topic include resource distribution in geographical areas, political border separation, and mapping.
- **Scheduling:** Graph coloring effectively models and solves scheduling challenges by allocating tasks or events to resources or time slots, such as assigning exam time slots to avoid conflicts for students.
- **Register Allocation in Compilers:** In order to reduce load and store operations, graph coloring is used in compiler optimization to help allocate processor registers for variables.
- **Wireless Frequency Assignment:** By assigning distinct frequencies or channels to each device, graph coloring can be used to minimize interference in wireless communication networks.

2. Algorithm Description (Brute Force)

- The brute force technique for the graph coloring problem goes over all potential graph colorings in order to find a valid coloring with the minimum number of colors.
- This approach is inefficient since it considers every possible color combination for the vertices, resulting in exponential time complexity.
- Despite its inefficiency, if there is any possible solution, the brute force method ensuring to find it, by trying all coloring combinations in given enough time.

Step 1: Initialize an empty list to store the final coloring with minimum number of color. Also Initialize a variable to store the minimum color.

Step 2: From one color up to n (number of vertex) colors generate all permutations of colorings for the vertices.

Step 2.1: For each permutation in all permutations:

Step 2.2: Assign colors to vertices based on the current permutation and form the current graph.

Step 2.3: Check if the coloring is proper (no adjacent vertices have the same color).

Step 2.3.1: For each edge in the graph:

Step 2.3.2: If the colors of the edge's endpoints are the same:

Step 2.3.3: Mark the coloring as improper.

Step 2.4: If the coloring is proper:

Step 2.4.1: If the number of colors used is less than the minimum:

Step 2.4.2: Update the minimum number of colors and the corresponding coloring.

Step 3: Return the coloring with the minimum number of colors.

2. Algorithm Description (Heuristic)

- Utilizing an iterative method, the Greedy Coloring Algorithm colors each vertex separately, selecting the lowest color that doesn't clash with the colors of nearby vertices.
- At each step, it chooses a vertex and evaluates its neighbors' colors. It then assigns the smallest available color that has not been used by any nearby vertex. This process repeats until all vertices are colored.
- The Greedy Coloring Algorithm uses at most $\Delta + 1$ colors, where Δ is the graph's maximum degree.
- The Greedy Coloring Algorithm is efficient, running in polynomial time, and offers a fair estimate to find the minimum coloring of the graph.

3. Algorithm Analysis (Brute Force)

- **Time Complexity:** Let n be the number of vertices in the graph. Let m be the number of edges. For k colors, there are k^n possible ways to assign these colors to n vertices. The algorithm checks color assignments starting from one color up to n colors. The complexity for each k involves generating k^n colorings and checking each coloring against m edges to ensure it is proper. The algorithm will have to check every edge and compare the vertices that are connected by that edge. That would give us (m+n) steps to do.Remember that we need to repeat this step for every possible vertex-color mapping. Given that k^n grows exponentially with increasing k, the dominant term in the sum is when k= n. Thus, the worst-case time complexity can be simplified to: **O(n^n × (m +n))**.

- **Space Complexity:**The space complexity of the brute force graph coloring algorithm primarily depends on the storage requirements for: the graph itself, the storage of colors for each vertex, the space needed to maintain a list of valid colorings found during execution. In our case the algorithm stores the graph using an adjacency list which takes O(V+E) space, where V is the number of vertices and E is the number of edge.In our case the algorithm stores only the best coloring found so far and the current number of colors that are used. In such a case, the space complexity remains O(V). The primary space usage is from the graph storage and the array of current vertex colors. Thus, the space complexity is **O(V+E)** when using an adjacency list, with an additional O(V) for the current vertex colors, not significantly altering the overall complexity.

3. Algorithm Analysis (Heuristic)

- **Time Complexity:** The Graph Coloring Problem heuristic algorithm worst-case scenario occurs if the graph is fully connected. The worst-case time complexity is $\Theta(n \cdot \Delta)$, where n is the number of vertices and Δ is the maximum degree. This arises because each vertex may need to check all its neighbors (up to n-1) for color assignment, requiring $O(\Delta)$ time per vertex, and this process iterates over all n vertices.

- **Space Complexity:** The colors that are assigned to each vertex needs to be stored by the algorithm. Since each vertex is assigned a color, the space needed for storing the colors is proportional to the number of vertices in the graph, which leads to a space complexity of O(n), where n is the number of vertices.

4. Sample Generation

- The random instance generator algorithm produces graphs represented as adjacency lists, using sample size, number of vertices, and edge probability as inputs.
- **Validation Function:** is_valid_graph(graph) checks for no self-loops and ensures the graph is undirected by verifying that if vertex A is connected to B, then B is also connected to A.
- **Graph Generation:** generate_valid_graph(num_vertices, edge_probability) initializes a graph and adds edges based on edge probability, ensuring an undirected structure.
- **Sample Generation:** generate_graph_samples(num_samples, num_vertices, edge_probability) generates the required number of valid graph samples by repeatedly creating and validating graphs until the sample list is complete.

```python
import random

def is_valid_graph(graph):

    for vertex, edges in graph.items():

        if vertex in edges:
            return False
        for adjacent in edges:
            if vertex not in graph[adjacent]:
                return False
    return True

def generate_valid_graph(num_vertices, edge_probability):

    graph = {i: set() for i in range(1, num_vertices + 1)}
    for i in range(1, num_vertices + 1):
        for j in range(i + 1, num_vertices + 1):
            if random.random() < edge_probability:
                graph[i].add(j)
                graph[j].add(i)
    return graph


def generate_graph_samples(num_samples, num_vertices, edge_probability):

    samples = []
    while len(samples) < num_samples:
        graph = generate_valid_graph(num_vertices, edge_probability)
        if is_valid_graph(graph):
            samples.append(graph)
    return samples
```

Figure 5: Implementation of random sample generation

5. Algorithm Implementation

- **is_proper_coloring(graph, coloring):** Iterates over each vertex and its neighbors to check for conflicts, returning False if any adjacent vertices share the same color, and True if the coloring is valid.

- **brute_force_graph_coloring(graph):** Lists the vertices and initializes variables to track the minimum number of colors and the optimal coloring. Loops through possible color numbers, generating all color assignments and checking their validity. Updates the minimum color count and optimal coloring if a valid configuration with fewer colors is found. Returns the optimal coloring with the fewest colors ensuring no two adjacent vertices share the same color.

```python
from itertools import product

def is_proper_coloring(graph, coloring):
    for vertex, edges in graph.items():
        for neighbor in edges:
            if coloring[vertex] == coloring[neighbor]:
                return False
    return True

def brute_force_graph_coloring(graph):
    vertices = list(graph.keys())
    min_colors = float('inf')
    optimal_coloring = None

    for num_colors in range(1, len(vertices) + 1):
        for coloring_permutation in product(range(num_colors), repeat=len(vertices)):
            current_coloring = {vertex: color for vertex, color in zip(vertices, coloring_permutation)}
            if is_proper_coloring(graph, current_coloring):
                if num_colors < min_colors:
                    min_colors = num_colors
                    optimal_coloring = current_coloring
        if optimal_coloring and min_colors == len(set(optimal_coloring.values())):
            break

    return optimal_coloring
```

Figure 6: Implementation of brute-force graph coloring algorithm

5. Heuristic Algorithm

Step 1: Initialize an empty dictionary 'coloring' to store the colors assigned to vertices.

Step 2: Initialize a list 'available_colors' to store the available colors.

Step 3: Initialize a variable 'max_color' to store the maximum color used.

Step 4: Sort the vertices of the graph in descending order of degree.

Step 5: For each vertex v in sorted_vertices:

- **Step 5.1:** neighbors = GetNeighbors(v) # Get the neighbors of vertex v.
- **Step 5.2:** used_colors = {coloring[n] for n in neighbors if coloring[n] is not None} # Get the colors used by neighbors.
- **Step 5.3:** available_colors = set(range(max_color + 2)) - used_colors # Get the available colors for vertex v.
- **Step 5.4:** coloring[v] = min(available_colors) # Assign the smallest available color to vertex v.
- **Step 5.5:** max_color = max(max_color, coloring[v]) # Update the maximum color used.

Step 6: Return coloring, max_color # Return the coloring and the maximum color used.

6. Experimental Analysis of The Performance

- The performance study carried out in section 3.2 is an upper bound on the algorithm's time and space complexity; it does not specify the performance in real-life applications.
- k randomly generated distinct graphs for each input size, k = 200 to assess the effectiveness of a heuristic approach for graph coloring. Each graph with a given edge probability that range in size from 50 to 200 vertices
- We found that equation for the fitted line is y = 1.755682 * x + -14.530963. The slope of the line is 1.755682, indicating that the average running time grows at a rate proportional to the number of vertices (x). In this case, the y-intercept of the line, which is -14.530963, is not practically significant because it reflects the baseline average running time when there are no vertices. Consider T(n)=n^a + (lower order terms) as the theoretical time complexity function, where n represents the number of vertices and a is the slope in the log-log plot. Slope of the line can give insights into the algorithm's practical time complexity. **Since a is 1.755682**, the algorithm's time complexity is **O(n^1.755682). As a result, in practice, algorithm performs better than the worst case running time of O(n · Δ) that is stated in section 3.2**

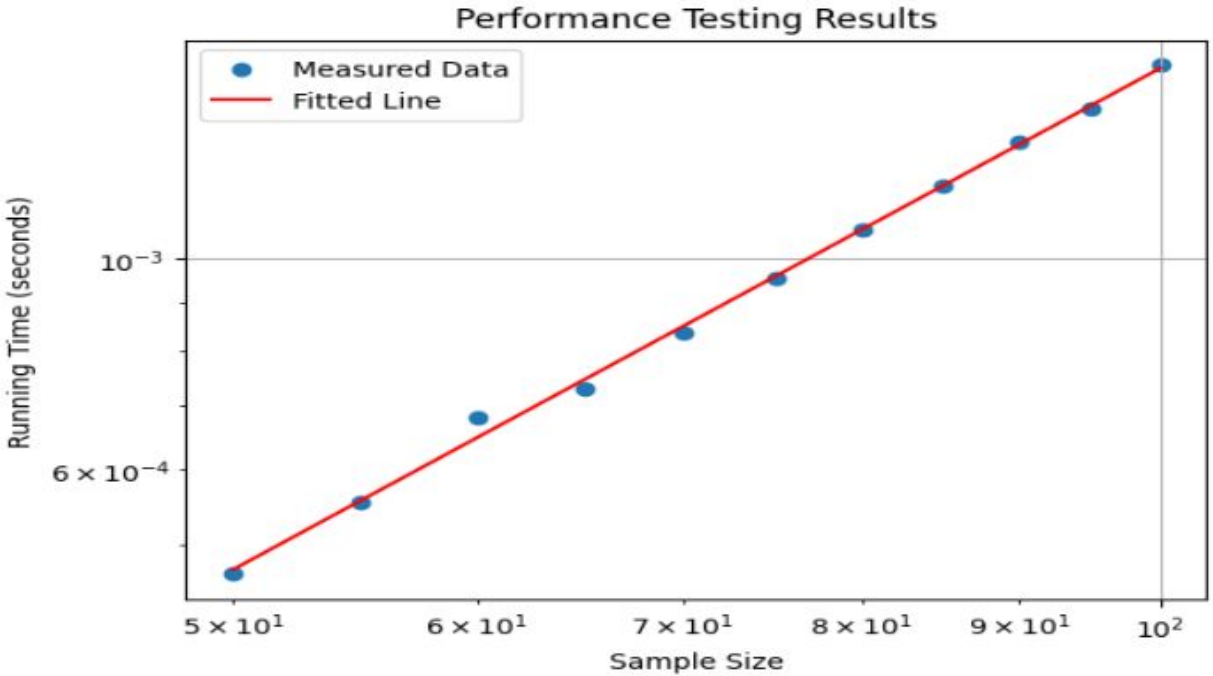


Figure 8: Performance Testing Result

7. Experimental Analysis of the Quality

- The experiments we did (Table 1) displays the results of a comparison of heuristic and brute force algorithms. As we suspected, the heuristic approach does not always produce the correct solution. Correct means whether it gives the minimum number of colors required to color the graph (the graph's chromatic number). The quality of the approaches also gets measure with the number of colors they use to color the graphs. Minimum number of colors means better quality. Table 1 shows the outcomes of providing random graphs to both algorithms, with input size, chromatic number, and ratio of results.

- The graph at Figure 10 shows the relationship between input size and the heuristic solution **quality** versus brute force solution quality. The red line (line of best fit) summarizes the statistical trend in the data. It is useful to analyze the overall behavior of the heuristic solution's quality in comparison to the brute force solution as the input size increases, smoothing out particular deviations. The line of best fit indicates a general upward trend in the ratio as input size grows. This suggests that when the problem grows larger, the heuristic approach tends to produce a lower quality (higher number of colors needed to color the graph) than the brute force. The graph shows that the heuristic does not maintains its performance relative to the brute force solution, but actually get worse as the problem's complexity increases

INPUT SIZE	NUMBER OF COLORS NEEDED FOR THE BRUTE FORCE	NUMBER OF COLORS NEEDED FOR THE HEURISTIC	RATIO OF HEURISTIC/BRUTE FORCE
1	1	1	1.0
2	1	2	2.0
3	2	3	1.5
4	2	3	1.5
5	2	3	1.5
6	2	4	2.0
7	3	3	1.0
8	3	3	1.0
9	3	4	1.33
10	3	3	1.0
11	4	4	1.0
12	4	4	1.0
13	4	4	1.0
14	4	4	1.0
15	3	5	1.66

Table 1

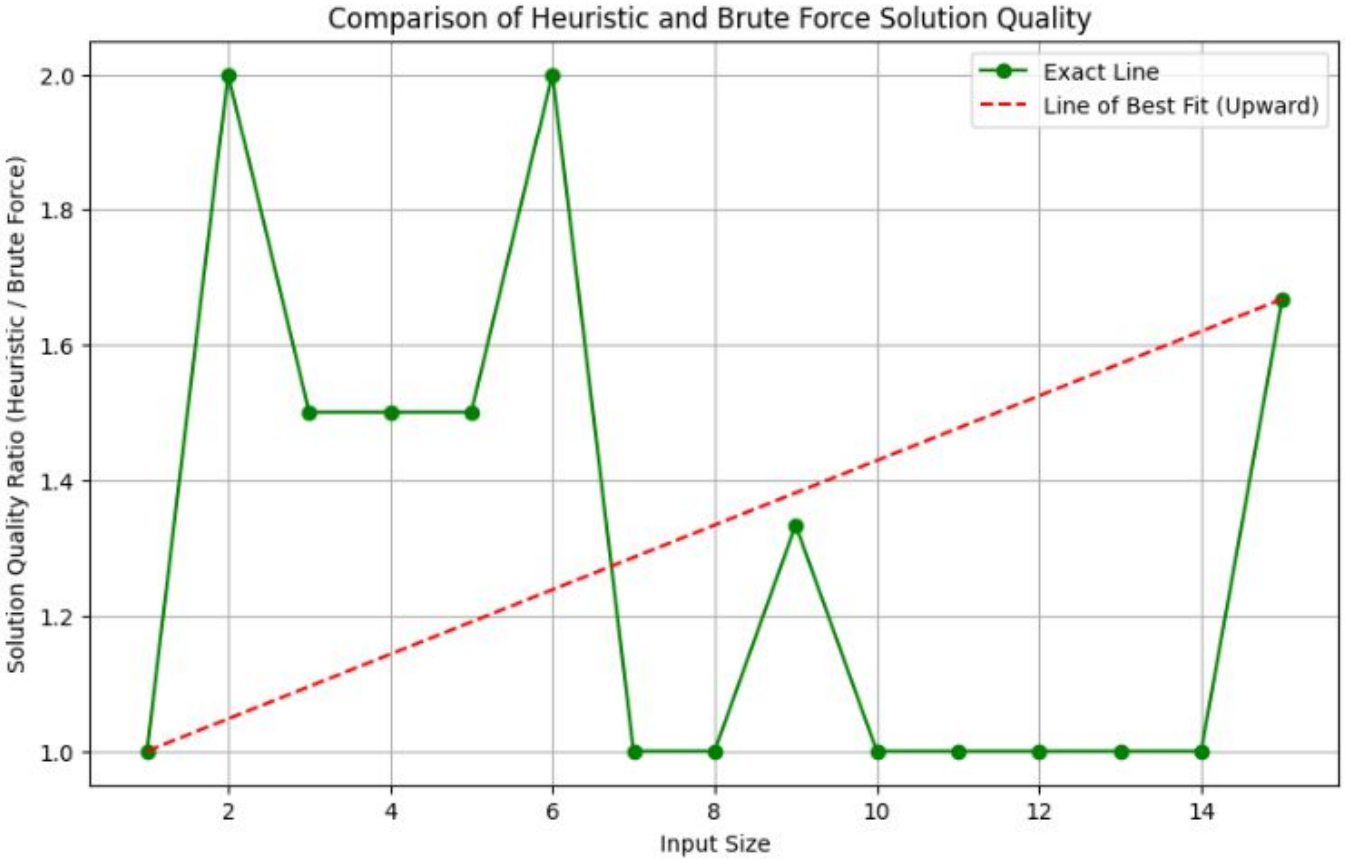


Figure 10: Results of the experiments for the size

8. Experimental Analysis of the Correctness of the Implementation (Functional Testing)

- Black box testing is a software testing method that evaluates the functionality of an application without peering into its internal structures or workings. We apply black box testing to the heuristic graph coloring algorithm we proposed.

- All test cases are successfully passed by the algorithm, underscoring its robustness and effectiveness. The tests span a range of graph types, from simple to complex, ensuring comprehensive coverage of potential real-world scenarios

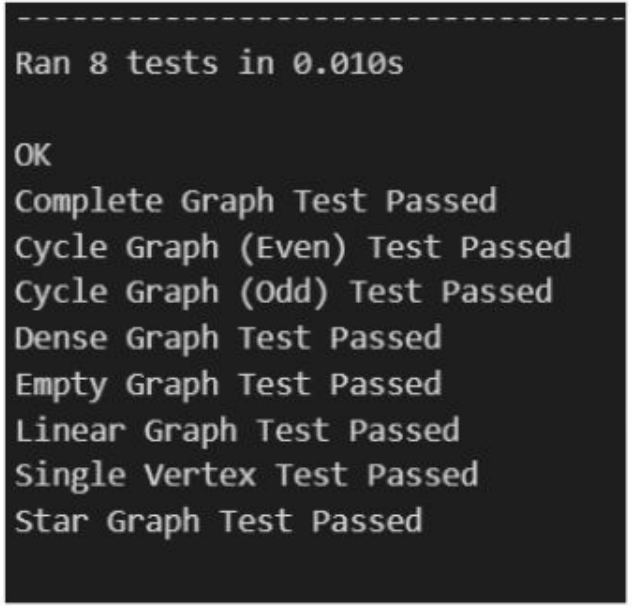


Figure 14 :Block Box Test Results

- White box testing, or clear box testing, involves examining the internal structure and logic of an algorithm. In this section, we delve into the white box testing of our heuristic graph coloring algorithm, focusing on statement coverage, decision coverage, and path coverage.

- The results of the tests highlighted both strengths and areas needing improvement. While the heuristic graph coloring algorithm demonstrated strong performance in certain scenarios, such as selecting the maximum degree node and coloring complete graphs the algorithm needs significant improvement in handling isolated nodes, non connected graphs, and sparse graphs to optimize color usage and efficiency

```
.FF.F
======================================================================
FAIL: test_coloring_no_edges (__main__.WhiteBoxTestGraphColoring.test_coloring_no_edges)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\aycaaelifaktas\AppData\Local\Temp\ipykernel_27712\1118664743.py", line 13, in test_coloring_no_edges
    self.assertEqual(len(set(colors_used.values())), 1, "Should color isolated nodes with the same color")
AssertionError: 2 != 1 : Should color isolated nodes with the same color

======================================================================
FAIL: test_non_connected_graph (__main__.WhiteBoxTestGraphColoring.test_non_connected_graph)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\aycaaelifaktas\AppData\Local\Temp\ipykernel_27712\1118664743.py", line 27, in test_non_connected_graph
    self.assertEqual(len(set(colors_used.values())), 2, "Non-connected graph parts can reuse colors")
AssertionError: 3 != 2 : Non-connected graph parts can reuse colors

======================================================================
FAIL: test_sparse_graph_color_reuse (__main__.WhiteBoxTestGraphColoring.test_sparse_graph_color_reuse)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\aycaaelifaktas\AppData\Local\Temp\ipykernel_27712\1118664743.py", line 33, in test_sparse_graph_color_reuse
    self.assertTrue(len(set(colors_used.values())) <= 2, "Sparse graphs should reuse colors efficiently")
AssertionError: False is not true : Sparse graphs should reuse colors efficiently

----------------------------------------------------------------------
Ran 5 tests in 0.005s

FAILED (failures=3)
Coloring Complete Graph Test Passed
Select Node Max Degree Test Passed
```

Figure 16: : White Box Test Results

9.Discussion

- The heuristic algorithm has a time complexity of $O(n \cdot \Delta)$ for n vertices and Δ represents the highest degree, significantly less than the brute force algorithm in theory. As a result, the algorithm's applicability to real-world problems improves. However, there is also a difficulty with the heuristic algorithm. As we have shown, the method loses correctness as the input increases.

- All of the tests that are applied black-box testings, passed successfully by the algorithm but some of the white box tests failed. As for the performance measures, experimental results (see to Chapter 6) showed that the average running of the algorithm's time complexity is O(n^1.755682). As a result, in practice, algorithm performs better than the worst-case running time of $O(n \cdot \Delta)$ that is stated in theory.

- In practice, the algorithm's running time is less than its worst-case scenario. This was to be expected, given that the algorithm's worst-case running time occurs so rarely. The heuristic technique clearly outperforms the brute force algorithm in terms of time performance. However, when input sizes increase, we lose some accuracy while saving time.We demonstrated this loss in Chapter 7 by comparing the precise accurate findings.