****

**Green University of Bangladesh**

**Department of Computer Science and Engineering(CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Spring, Year:2024), B.Sc. in CSE (Day)**

**LAB REPORT NO #4**

**Course Title: Artificial Intelligence Lab**

**Course Code: CSE 316 Section: 213D3**

**Lab Experiment Name: Implement Graph Coloring Algorithm.**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Shoaib Ahmed | 212902019 |

**Lab Date : 04-05-24**

**Submission Date : 11-05-24**

**Course Teacher’s Name : Md. Zahidul Hasan**

**[For Teachers use only: Don’t Write Anything inside this box]**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature:.....................**  **Comments:.............................................. Date:..............................** |

**1. TITLE OF THE LAB EXPERIMENT**

Implement Graph Coloring Algorithm.

**2. OBJECTIVES/AIM**

* To understand the concept of Graph Coloring and its applications in computer science.
* To implement the Graph Coloring algorithm using Python.
* To learn how to read graph data from a file and process it for use in an

algorithm.

* To explore the use of recursion in solving complex problems such as Graph

Coloring.

* To analyze the effectiveness of the Graph Coloring algorithm in terms of its

ability to find a solution with a given number of colors.

**3. PROCEDURE / ANALYSIS / DESIGN**

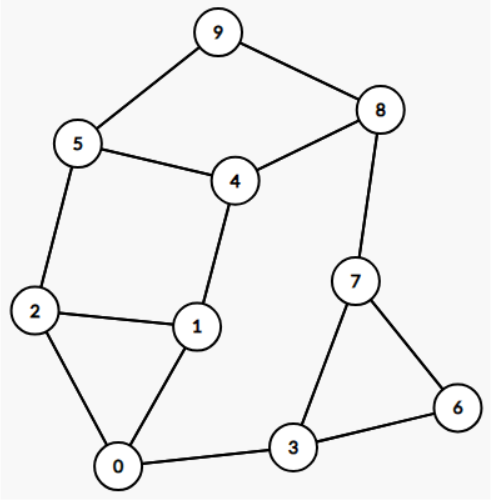


Fig-4.1: Demo Graph for coloring

If 2 colors are given to color the graph, It won’t be possible. Because there are 2

triangle nodes. 0, 1, 2 and 3, 6, 7. So we at least need 3 colors

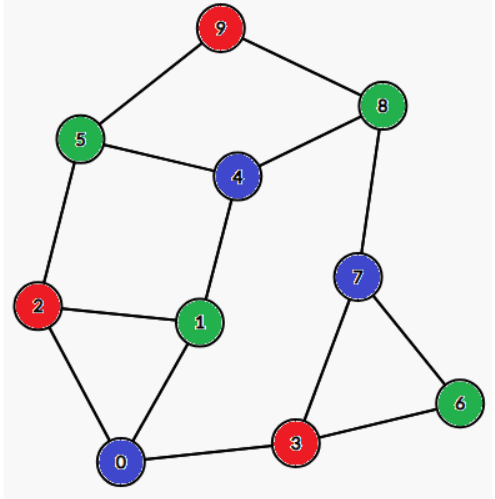


Fig-4.2: Colored Graph

**4. IMPLEMENTATION**

Code:

|  |
| --- |
| import os  def isSafe(graph, node, color, graphColored):  for i in graph[node]:  if color == graphColored[i]:  return False  return True  def solve(graph, numOfColors, currentNode, graphColored):  if currentNode == len(graph):  return True    for color in range(1, numOfColors + 1):  if isSafe(graph, currentNode, color, graphColored):  graphColored[currentNode] = color  if solve(graph, numOfColors, currentNode + 1, graphColored):  return True  graphColored[currentNode] = 0    return False  def main():  filepath = "24-05-18/lab-report-4/colorGraph.txt"  graph = {}  numOfColors = 0    if not os.path.isfile(filepath):  print(f"File path {filepath} does not exist.")  else:  try:  with open(filepath, "r") as file:  lines = file.readlines()  n = len(lines)  for i in range(n - 1):  graph[i] = list(map(int, lines[i].split()))  numOfColors = int(lines[-1])  except PermissionError:  print(f"Permission denied for file {filepath}.")    graphColored = [0] \* n    if solve(graph, numOfColors, 0, graphColored):  print("Solution exists")  for i in range(n):  print(f"Node {i} is colored with color {graphColored[i]}")  else:  print("Solution does not exist")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

**5. TEST RESULT / OUTPUT**

Suppose 3 colors are given to paint the graph.

|  |
| --- |
| 1 2 3  0 2 4  0 1 5  0 6 7  1 5 8  2 4 9  3 7  3 6 8  4 7 9  5 8  3 #This is the number of Color. |

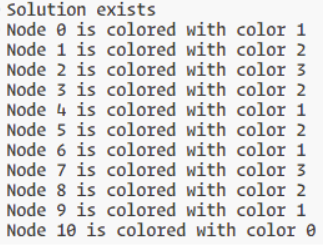


Fig-4.3: Output of graph coloring

**6. ANALYSIS AND DISCUSSION**

Here I implement a graph coloring algorithm using backtracking to determine if it's possible to color a given graph with a specified number of colors such that no two adjacent nodes share the same color. It begins by attempting to read graph data from a file (colorGraph.txt), handling potential file errors gracefully. The solve function recursively attempts to assign colors to each node, ensuring adjacency constraints are met using the isSafe function. If successful, it prints that a solution exists and displays the color assignments for each node; otherwise, it indicates that no solution exists. The algorithm's efficiency depends on the graph's structure and the number of colors, with worst-case performance potentially exponential in terms of the number of nodes and colors due to the nature of backtracking. Improvements could include optimizations like heuristics for color assignment or pruning techniques to reduce unnecessary exploration of color choices, especially for larger or denser graphs.