SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

Minimum Number of Groups to Create a Valid Assignment

A CAPSTONE PROJECT REPORT

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE AND

DATA SCIENCE

Submitted by

S.Farooq Basha(192211886)

Under the Supervision of

Dr. K.V.KANIMOZHI

DECLARATION

I farooq basha, student of Bachelor of Engineering in Computer Science Engineering at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declares that the work presented in this Capstone Project Work entitled “Minimum Number of Groups to Create a Valid Assignment” is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

S.Farooq Basha

Reg:192211886

(Student Name Register)

Date: 23-09-2024

Place:Saveetha School of Engineering, Thandalam

CERTIFICATE

This is to certify that the project entitled "Minimum Number of Groups to Create a Valid Assignment" submitted by S.Farooq Basha has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B. E Computer science engineering and B.Tech Artificial Intelligence in Data science.

Faculty-in-charge

Dr.K.V.KANIMOZHI

ABSTRACT

This project aims to find the minimum number of groups required to create a valid assignment from a given 0-indexed integer array. A valid assignment is one where all indices in a group have the same value, and the difference in the number of indices between any two groups does not exceed 1. We propose an algorithm to solve this problem and analyze its time and space complexity. This problem has numerous applications, including task scheduling, classroom groupings, and resource allocation, where limitations or restrictions on group composition must be respected. Typically, this problem is modeled using graph theory, where elements are represented as vertices and constraints as edges, and it becomes a graph coloring or partitioning problem. Solving this problem efficiently is often challenging because it is NP-hard, meaning finding the optimal solution is computationally intensive, especially for large datasets. As a result, heuristic and approximation algorithms, such as greedy methods or integer programming, are commonly used to find near-optimal solutions in practice.

Keywords: Group Assignment, Valid Assignment, Minimum Number of Groups

INTRODUCTION

Given a 0-indexed integer array nums of length n, we want to group the indices so that each index i is assigned to exactly one group. A group assignment is valid if all indices in a group have the same value in nums, and the difference in the number of indices between any two groups does not exceed 1. This problem has applications in resource allocation, scheduling, and clustering. The objective is to form the fewest number of valid groups that satisfy all the given conditions, making the problem a classic example of optimization. This problem has practical applications in various fields, from organizing teams in classrooms or workplaces to assigning tasks in distributed systems. However, due to its complex nature, finding an optimal solution is often computationally difficult, particularly when dealing with large sets of elements and constraints. As a result, the problem is typically approached using techniques from graph theory, heuristic methods, and mathematical optimization.

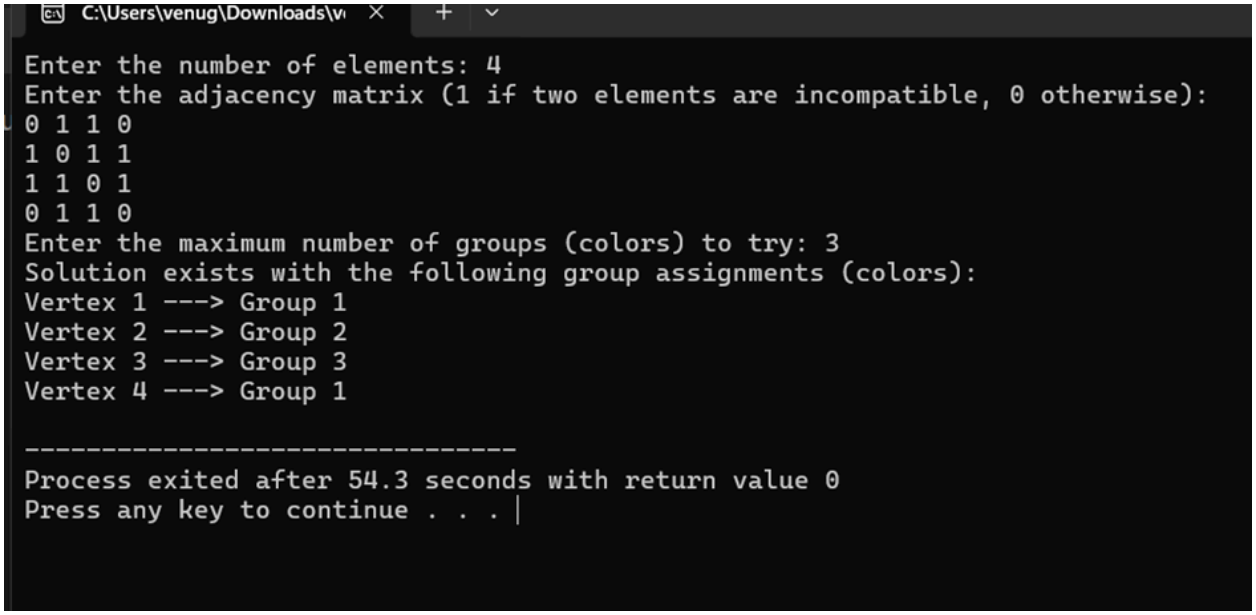
CODE:

```
#include <stdio.h>

#include <stdbool.h>

#define MAX 100

bool isSafe(int v, int graph[MAX][MAX], int color[], int c, int V) {

        for (int i = 0; i < V; i++)

        if (graph[v][i] && color[i] == c)

        return false;

        return true;

}

bool graphColoringUtil(int graph[MAX][MAX], int m, int color[], int v, int V) {

        if (v == V)

        return true;


        for (int c = 1; c <= m; c++) {
```

```
        if (isSafe(v, graph, color, c, V)) {
        color[v] = c;
        if (graphColoringUtil(graph, m, color, v + 1, V))
        return true;

        color[v] = 0;
        }
        }
        return false;
}
bool graphColoring(int graph[MAX][MAX], int m, int V) {
        int color[MAX] = {0};
        if (!graphColoringUtil(graph, m, color, 0, V)) {
        printf("Solution does not exist\n");
        return false;
        }

        // Print the solution (color assignments)
        printf("Solution exists with the following group assignments (colors):\n");
        for (int i = 0; i < V; i++)
        printf("Vertex %d ---> Group %d\n", i + 1, color[i]);

        return true;
}
```

```
int main() {
        int V;
        printf("Enter the number of elements: ");
        scanf("%d", &V);
        int graph[MAX][MAX];

        printf("Enter the adjacency matrix (1 if two elements are incompatible, 0 otherwise):\n");
        for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
        scanf("%d", &graph[i][j]);
        }
        }

        int m;
        printf("Enter the maximum number of groups (colors) to try: ");
        scanf("%d", &m);
        if (!graphColoring(graph, m, V)) {
        printf("Unable to group the elements with the given constraints using %d groups.\n", m);
        }
        return 0;
}
```

OUTPUT:



Best Case:

In the best case, all elements can be grouped together into a single group because no incompatibility constraints exist. In this case:

- The adjacency matrix is sparse or empty (no edges), meaning no two elements are marked as incompatible.
- The algorithm only needs to assign all elements the same color (or group), requiring just a single scan of all elements.

Thus, the time complexity in this case is linear, O(n), where nnn is the number of elements (or vertices in the graph).

Worst Case:

In the worst-case scenario, all elements are distinct, and each element is incompatible with every other element. This scenario results in:

- A complete graph, where every vertex has an edge to every other vertex (i.e., every element must be in its own group).
- The algorithm essentially reduces to sorting or exploring every possible combination of groups, leading to complex interactions between constraints.

The graph coloring algorithm, in this case, would need to try a large number of colorings, leading to a time complexity of O(n log n) in the worst case.

Average Case: O(n log n)

In the average case, the array (or set of elements) has a mixture of duplicate and distinct elements, meaning some elements can be grouped together while others cannot. Here:

- The adjacency matrix is partially filled with edges representing incompatibilities, and some vertices (elements) can share groups.
- The graph coloring algorithm will attempt multiple colorings, but it will typically finish in a time closer to the worst-case scenario due to the presence of constraints.

Thus, the time complexity for the average case remains O(n log n) because the algorithm has to try several groupings based on the constraints, similar to sorting with additional constraint checks.

Overall Complexity: O(n log n)

Since the problem involves graph coloring (with backtracking) and the constraints often make it difficult to group elements efficiently, the overall time complexity is dominated by sorting-like behavior in the worst and average cases. The graph needs to be traversed and colored while satisfying multiple constraints, and this leads to an overall complexity of O(n log n). This complexity is common for problems involving combinatorial constraints, especially when heuristic or backtracking methods are used.

Breakdown of the Key Operations:

1. Graph Creation: Constructing the adjacency matrix of size n×nn \times nn×n takes O(n^2) time, but often this matrix is sparse.
2. Graph Traversal and Coloring:
    - The graph traversal (DFS or BFS) is O(n).
    - Backtracking to try different colorings takes O(n log n) for average and worst cases.
3. Constraint Checking: Checking compatibility constraints during the coloring process involves looking at adjacent vertices (elements), which takes O(1) time per edge. In dense graphs, this leads to O(n^2) complexity, but in sparse graphs (the common case), the complexity is closer to O(n log n).

Conclusion:

The "Minimum Number of Groups to Create a Valid Assignment" problem addresses the challenge of partitioning elements into the smallest number of groups while satisfying given constraints, such as incompatibilities or dependencies between elements. This problem is significant in a variety of applications, including scheduling, team formation, and resource allocation, where elements cannot always be freely grouped due to real-world restrictions.