UNIVERSIDADE FEDERAL UBERLÂNDIA

CAMPUS MONTE CARMELO

CURSO BACHARELADO EM SISTEMAS DE INFORMAÇÃO

RHUAN FLORES CUNHA FERNANDES

CLÉSIO RODRIGUES DA SILVA JÚNIOR

3° TRABALHO DE ESTRUTURA DE DADOS II

Monte Carmelo-MG

06/2023

SUMÁRIO

1 DEFINIÇÃO DE ALGORITMO DE COLORAÇÃO DE GRAFOS

O algoritmo de …..

2 RESOLUÇÃO DO PROBLEMA

O problema nos informava que um professor tem uma classe com 25 alunos dispostos em 5 filas com 5 alunos em cada, formando um quadrado de dimensões 5x5. Muito preocupado com a integridade acadêmica, esse professor decidiu adotar um método que leva em consideração a proximidade física dos estudantes para eliminar todas as possibilidades de cola durante a aplicação da prova final.

Ele considera que os alunos podem ser influenciados pelos colegas que estão próximos pela horizontal, vertical e diagonal, e por isso esses alunos adjacentes não podem ter o mesmo modelo (cor) de prova.

Contudo, é bastante trabalhoso elaborar 25 modelos diferentes de prova, por isso ele solicitou a nossa ajuda para criar um algoritmo em linguagem C que o permita descobrir o número mínimo de modelos de provas que ele deve elaborar, a quantidade de cada modelo para não haver desperdício de papel e também um mapa com a melhor distribuição dessas provas pela sala.

Para resolver esse problema, inicialmente utilizamos **structs** para representar a estrutura do grafo e a lista de adjacência dos vértices (não foi viável criar uma matriz de adjacência pois ficaria muito grande, tamanho 25x25). Após isso, criamos algumas funções para manipularem o grafo: adicionar vértices, verificar adjacentes e colorir o grafo.

A função **addVert** serve para adicionar os vértices à respectiva adjacência, recebendo como parâmetros um ponteiro para a lista de adjacência e o valor do vértice adicionado e o armazenamos em um novo nó do tipo Noptr. Após isso verificamos se a lista de adjacência está vazia: se sim, adicionamos esse nó no início da lista, e caso contrário percorremos até a última posição e o adicionamos no final. Sempre vamos atualizando essa lista com base na verificação das condições estabelecidas.

```
22  // Função para adicionar um vértice à lista de adjacência de outro vértice
23  void addVert(AdjList* listaAdjacente, int vert) {
24      Noptr* newNode = (Noptr*)malloc(sizeof(Noptr));
25      newNode->vert = vert;
26      newNode->prox = NULL;
27
28      if (listaAdjacente->inicio == NULL) {
29          listaAdjacente->inicio = newNode;
30      } else {
31          Noptr* atual = listaAdjacente->inicio;
32          while (atual->prox != NULL) {
33              atual = atual->prox;
34          }
35          atual->prox = newNode;
36      }
37  }
```

A função **verificaAdjacente** irá verificar se dois vértices do grafo são adjacentes com base na lista de adjacência. Recebe por parâmetros o grafo, o primeiro e segundo vértices que precisa verificar. A partir daí, a função percorre a lista de adjacência do primeiro vértice informado e verifica se algum deles é igual ao segundo vértice fornecido e retorna 1 se os vértices forem vizinhos ou 0 caso contrário.

```
40  // Função para verificar se dois vértices são adjacentes
41  int verificaAdjacente(Vertice grafo[], int u, int v) {
42      Noptr* atual = grafo[u].listaAdjacente->inicio;
43      while (atual != NULL) {
44          if (atual->vert == v) {
45              return 1;
46          }
47          atual = atual->prox;
48      }
49      return 0;
50  }
```

Por fim temos a função **colorirGrafo** que atribui as cores aos vértices de modo a impedir que vértices próximos tenham a mesma cor de prova, inviabilizando a “cola” dos alunos. Iniciamos usando a corAtual = 1 (que posteriormente iremos definir com base num vetor de cores) e verificamos sequencialmente se os vértices já estão coloridos. Se o vértice não está colorido precisamos verificar se os seus vizinhos já possuem essa cor que está em uso: se algum adjacente já tiver essa cor, pulamos para o próximo vértice e repetimos as etapas; se os adjacentes não tiverem a mesma cor, atribuímos essa cor à ele e vamos para o próximo. Só iremos incrementar a variável corAtual após percorrer todas as 25 posições e verificarmos que ainda existem vértices sem cor.

```
52  // Função para colorir o grafo
53  void colorirGrafo(Vertice grafo[]) {
54      int corAtual = 1;
55      int numVertices = SIZE * SIZE;
56
57      while (1) {
58          int todosColoridos = 1;
59
60          for (int i = 0; i < numVertices; i++) {
61              if (grafo[i].cor == 0) {
62                  int adjacenteColorido = 0;
63
64                  for (int j = 0; j < numVertices; j++) {
65                      if (verificaAdjacente(grafo, i, j) && grafo[j].cor == corAtual) {
66                          adjacenteColorido = 1;
67                          break;
68                      }
69                  }
70                  if (!adjacenteColorido) {
71                      grafo[i].cor = corAtual;
72                  } else {
73                      todosColoridos = 0;
74                  }
75              }
76          }
77          if (todosColoridos) {
78              break;
79          }
80          corAtual++;
81      }
82  }
```

Na função **main()** iniciamos um vetor chamado grafo de tamanho 5x5 para representar todos os vértices (alunos) e definimos um vetor de cores que será utilizado na impressão dos resultados.

Iniciamos o grafo e construímos a lista de adjacência através de uma estrutura de repetição com a finalidade de verificar se o vértice possui um tipo de vizinho (superior, inferior, esquerdo, direito, superior esquerdo, superior direito, inferior esquerdo e inferior direito): se possuir, adicionamos na lista de adjacência daquele vértice e verificamos o próximo.

Após essas etapas, chamamos a função para colorir o grafo de acordo com as restrições mencionadas. Também utilizamos contadores para verificar quantas cópias de cada modelo/cor de prova devem ser utilizados de modo que não haja desperdício. Para finalizar, imprimimos a quantidade de cores diferentes usadas, a quantidade de cópias de cada prova e o mapa com a distribuição das provas na sala de aula.

3 CÓDIGO:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5  // Tamanho da grade (5x5)

// Estrutura para representar um vértice do grafo
typedef struct Noptr {     int vert;
    struct Noptr* prox;
} Noptr;

typedef struct {
    Noptr* inicio;
} AdjList;

typedef struct {
    int cor;
    AdjList* listaAdjacente;
} Vertice;
```

```
// Função para adicionar um vértice à lista de adjacência de um vértice
void addVert(AdjList* listaAdjacente, int vert) {
    Noptr* newNode = (Noptr*)malloc(sizeof(Noptr));
    newNode->vert = vert;
    newNode->prox = NULL;

    if (listaAdjacente->inicio == NULL) {
        listaAdjacente->inicio = newNode;
    } else {
        Noptr* atual = listaAdjacente->inicio;
        while (atual->prox != NULL) {
            atual = atual->prox;
        }
        atual->prox = newNode;
    }
}



// Função para verificar se dois vértices são adjacentes
int verificaAdjacente(Vertice grafo[], int u, int v) {
    Noptr* atual = grafo[u].listaAdjacente->inicio;
    while (atual != NULL) {
        if (atual->vert == v) {
            return 1;
        }
        atual = atual->prox;
    }
    return 0;
}
```

```
// Função para colorir o grafo
void colorirGrafo(Vertice grafo[]) {
    int corAtual = 1;
    int numVertices = SIZE * SIZE;

    while (1) {
        int todosColoridos = 1;

        for (int i = 0; i < numVertices; i++) {
            if (grafo[i].cor == 0) {
                int adjacenteColorido = 0;

                for (int j = 0; j < numVertices; j++) {
                    if (verificaAdjacente(grafo, i, j) && grafo[j].cor == corAtual) {
                        adjacenteColorido = 1;
                        break;
                    }
                }
                if (!adjacenteColorido) {
                    grafo[i].cor = corAtual;
                } else {
                    todosColoridos = 0;
                }
            }
        }
        if (todosColoridos) {
            break;
        }
        corAtual++;
    }
}
```

```
int main() {
    Vertice grafo[SIZE * SIZE];  // Grafo representado como um array de vértices
    char cores[][15] = {"", "Rosa", "Verde claro", "Verde escuro", "Azul"};


    // Inicialização do grafo e construção da lista de adjacência para cada vértice
    for (int i = 0; i < SIZE * SIZE; i++) {
        grafo[i].cor = 0;  // Cor indefinida
        grafo[i].listaAdjacente = (AdjList*)malloc(sizeof(AdjList));
        grafo[i].listaAdjacente->inicio = NULL;


        // Ajuste da qtd de arestas para vértices nas bordas
        int lin = i / SIZE;
        int col = i % SIZE;


        // Adiciona os vizinhos à lista de adjacência
        if (lin > 0) {
            addVert(grafo[i].listaAdjacente, i - SIZE);  // Vizinho superior
        }
        if (lin < SIZE - 1) {
            addVert(grafo[i].listaAdjacente, i + SIZE);  // Vizinho inferior
        }
        if (col > 0) {
            addVert(grafo[i].listaAdjacente, i - 1);  // Vizinho à esquerda
        }
        if (col < SIZE - 1) {
            addVert(grafo[i].listaAdjacente, i + 1);  // Vizinho à direita
        }
        if (lin > 0 && col > 0) {
            addVert(grafo[i].listaAdjacente, i - SIZE - 1);  // Vizinho superior esquerdo
        }
```

```
    if (lin > 0 && col < SIZE - 1) {
        addVert(grafo[i].listaAdjacente, i - SIZE + 1);  // Vizinho superior direito
    }
    if (lin < SIZE - 1 && col > 0) {
        addVert(grafo[i].listaAdjacente, i + SIZE - 1);  // Vizinho inferior esquerdo
    }

    if (lin < SIZE - 1 && col < SIZE - 1) {
        addVert(grafo[i].listaAdjacente, i + SIZE + 1);  // Vizinho inferior direito
    }
}



// Chamada da função de coloração do grafo
colorirGrafo(grafo);



// Contagem do número de provas diferentes e distribuição
int numProvas = 0;
int numCadaProva[SIZE * SIZE] = {0};
int provaPorAluno[SIZE * SIZE];

for (int i = 0; i < SIZE * SIZE; i++) {
    if (grafo[i].cor > numProvas) {
        numProvas = grafo[i].cor;
    }
    numCadaProva[grafo[i].cor]++;
    provaPorAluno[i] = grafo[i].cor;
}
```

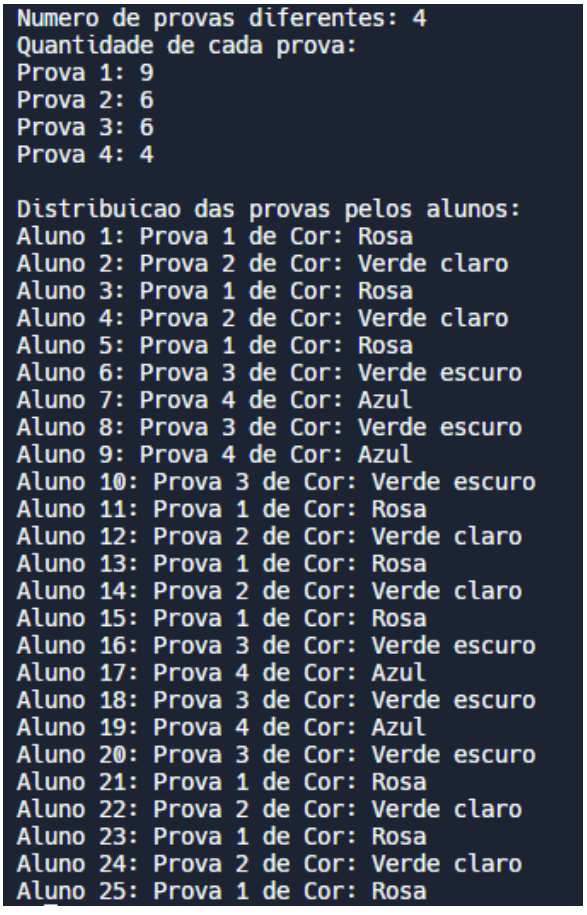
```
    // Impressão dos resultados
    printf("Numero de provas diferentes: %d\n", numProvas);
    printf("Quantidade de cada prova:\n");
    for (int i = 1; i <= numProvas; i++) {
        printf("Prova %d: %d\n", i, numCadaProva[i]);
    }

    printf("\nDistribuicao das provas pelos alunos:\n");
     for (int i = 0; i < SIZE * SIZE; i++) {
        printf("Aluno %d: Prova %d de Cor: %s\n", i+1, provaPorAluno[i],
cores[provaPorAluno[i]]);
    }

    return 0;
}
```

4 SAÍDAS:

Após a execução do algoritmo, essa foi a saída:



5 RESOLUÇÃO MANUSCRITA:

Após a saída do algoritmo, decidimos resolver de forma manuscrita para conferir os resultados: