Projeto 2 - Grafos

Luiz Guilherme Moreira Leite - 537949

July 6, 2023

1 Questão 1 - Problema das Cores

Essa função resolve o primeiro problema percorrendo o grafo em largura, atribuindo e comparando cores. Ela começa em um vértice e verifica seus vizinhos. Se um vizinho não estiver colorido, a função tenta colori-lo com uma cor diferente da do vértice atual. Se um vizinho já estiver colorido, a função compara as cores e verifica se são iguais. Se forem iguais, a função altera o valor de uma flag, invalidando o grafo. Caso contrário, ela continua a busca no grafo. O motivo de ter usado um algoritmo baseado em busca é simples, nesse caso foi a maneira que eu pensei de percorrer todo o grafo.

A complexidade do algoritmo é $\mathcal{O}(n*(v*x))$. A qual n é o tamanho do vetor de grafos, v é número de vertices do grafo e x é o numero de vizinhos que cada vertice do grafo contem. Dessa forma no pior caso onde todos os valores são iguais seria $\mathcal{O}(n*n*n)$

2 Questao 2 - Número de Kevin Bacon

As funções buildList, calculaNumeroBacon e print são essenciais para resolver o problema. Cada uma delas desempenha uma parte específica do processo. A função buildList é responsável por criar a lista de adjacência. A função calculaNumeroBacon realiza o cálculo dos números de Bacon. E a função print tem a finalidade de imprimir na tela a saída de forma ordenada.

2.1 buildList

Basicamente, essa função tem como parâmetro uma string com o caminho até o arquivo. Esse arquivo é lido linha por linha e cada linha é dividida por meio da função split. Essa função split retorna uma lista de adjacência. Se essa lista tiver tamanho 3, são criados os dois atores (os dois vértices do grafo) e o filme (a aresta que liga os dois vértices). Em seguida, é verificado se esses atores já estão presentes na lista de adjacência. Se não estiverem, é criado um novo ator e adicionado à lista. Além disso, o filme é adicionado à lista de filmes do ator 1 e do ator 2. Após isso, é fechado o arquivo e retornado a lista.

Sua Complexidade é dada por $\mathcal{O}(n)$ a qual n é número de linhas do arquivo

2.2 calculaNumeroBacon

Essa função recebe a lista de adjacência e um ator como parâmetro, e calcula o número de Bacon para cada ator da lista com base no ator fornecido. Ela define o ator inicial com o número de Bacon 0 e o filme como vazio. Em seguida, realiza uma busca em largura para calcular o número de Bacon. Durante a busca em largura, o número de Bacon atual é calculado incrementando o número de Bacon do ator atual em 1. Para cada filme em que o ator atual aparece, é feito um loop através de todos os pares na lista de adjacência. É verificado se o ator coestrela é diferente do ator atual e se o filme em questão também está presente na lista de filmes do ator coestrela. Isso indica uma conexão entre os dois atores através do filme. Se o ator coestrela ainda não foi visitado (ou seja, seu número de Bacon é -1), ele é marcado com o número de Bacon atualizado, o filme é registrado como a conexão entre os atores e o ator coestrela é adicionado à fila de atores para processamento posterior.

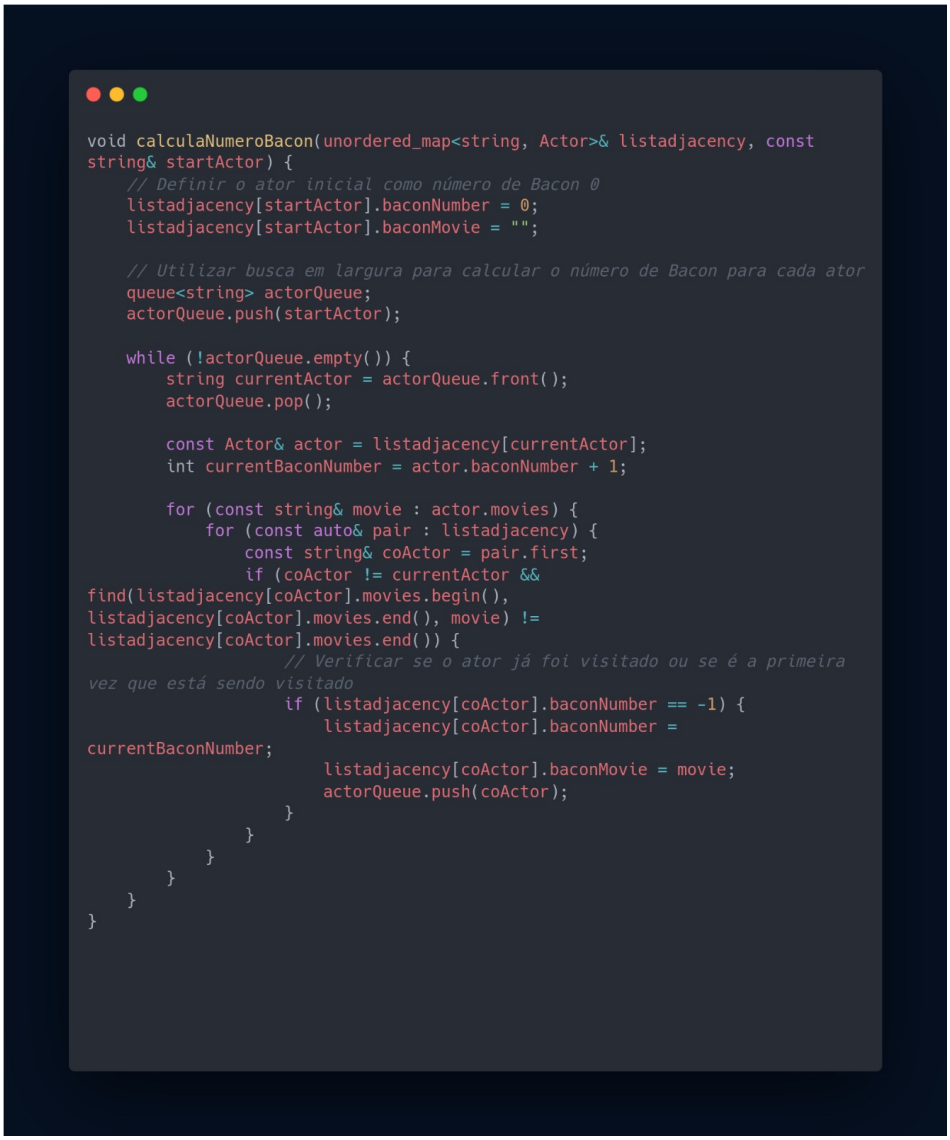
Sua Complexidade é dada por $\mathcal{O}(n*(x*z))$ a qual n é número de atores na lista de adjacência, x é número de filmes e z é o numero de pares

```cpp
string verify(vector<vector<list<int>>> grafos) {
    //criando a string de resultado
    string result = "";
    //percorrendo o vetor de grafos
    for(int a = 0; a < grafos.size(); a++){
        //vetor para armazenar as cores dos vertices
       vector<int> Colors(grafos[a].size());
        //flag para indicar se o grafo é valido ou invalido
        int flag = 0;
        //fila para a busca em largura(BFS)
        queue<int> fila;
        //atribuindo a cor 1 ao vertice inicial do grafo
        Colors[0] = 1;
        //inserindo o vertice inicial na fila
        fila.push(0);
        //referencia ao grafo atual
        auto grafo = grafos[a];
        //Fazendo a busca e comparando as cores
        while(!fila.empty()){
            //obtendo o proximo vertice da fila
            int u = fila.front();
            //desempilhando o vertice da fila
            fila.pop();
            //percorrendo os vertices adjacentes a u
            for(auto v : grafo[u]){
                //verificando se o vertice v ainda nao foi visitado(cor igual a 0)
                if(Colors[v] == 0){
                    //verificando se a cor do vertice u é igual a 1
                    if(Colors[u] == 1){
                        //atribuindo a cor 2 ao vertice v
                        Colors[v] = 2;
                    }else{
                        //atribuindo a cor 1 ao vertice v
                        Colors[v] = 1;
                    }
                    //inserindo o vertice v na fila para processamento posterior
                    fila.push(v);
                }
                //verificando se há conflito de cores entre u e v
                else{
                    if(Colors[u] == Colors[v]){
                        flag = 1;
                    }
                }
            }
        }
        //verificando o valor da flag após o processamento do grafo
        if(flag == 0){
            result += "SIM\n";
        }
        else{
            result += "NAO\n";
        }
    }
    return result;
}
```

Figure 1: Função da Priemira Questão

```cpp
// Função para ler o arquivo de entrada e construir a lista de adjacência
(simulando um grafo)
unordered_map<string, Actor> buildList(const string& filename) {
    unordered_map<string, Actor> listAdjacency;
    ifstream inputFile(filename);
    string line;

    while (getline(inputFile, line)) {
        vector<string> parts = split(line, ';');

        if (parts.size() == 3) {
            string ator1 = parts[0];
            string film = parts[1];
            string ator2 = parts[2];

            // Adicionar o ator 1 à lista de adjacência
            if (listAdjacency.find(ator1) == listAdjacency.end()) {
                Actor novoAtor;
                novoAtor.name = ator1;
                // inicialmente, definido como -1 para indicar não visitado
                novoAtor.baconNumber = -1;
                listAdjacency[ator1] = novoAtor;
            }

            // Adicionar o ator 2 à lista de adjacência
            if (listAdjacency.find(ator2) == listAdjacency.end()) {
                Actor novoAtor;
                novoAtor.name = ator2;
                novoAtor.baconNumber = -1;
                listAdjacency[ator2] = novoAtor;
            }

            // Adicionar o filme à lista de filmes do ator 1
            listAdjacency[ator1].movies.push_back(film);

            // Adicionar o filme à lista de filmes do ator 2
            listAdjacency[ator2].movies.push_back(film);
        }
    }

    inputFile.close();
    return listAdjacency;
}
```

Figure 2: buildList



```
void calculaNumeroBacon(unordered_map<string, Actor>& listadjacency, const
string& startActor) {
    // Definir o ator inicial como número de Bacon 0
    listadjacency[startActor].baconNumber = 0;
    listadjacency[startActor].baconMovie = "";

    // Utilizar busca em largura para calcular o número de Bacon para cada ator
    queue<string> actorQueue;
    actorQueue.push(startActor);

    while (!actorQueue.empty()) {
        string currentActor = actorQueue.front();
        actorQueue.pop();

        const Actor& actor = listadjacency[currentActor];
        int currentBaconNumber = actor.baconNumber + 1;

        for (const string& movie : actor.movies) {
            for (const auto& pair : listadjacency) {
                const string& coActor = pair.first;
                if (coActor != currentActor &&
find(listadjacency[coActor].movies.begin(),
listadjacency[coActor].movies.end(), movie) !=
listadjacency[coActor].movies.end()) {
                    // Verificar se o ator já foi visitado ou se é a primeira
vez que está sendo visitado
                    if (listadjacency[coActor].baconNumber == -1) {
                        listadjacency[coActor].baconNumber =
currentBaconNumber;
                        listadjacency[coActor].baconMovie = movie;
                        actorQueue.push(coActor);
                    }
                }
            }
        }
    }
}
```

Figure 3: calculaNumero

```
void print(const unordered_map<string, Actor>& adjacencyList) {
    set<string> actors; // Usando um conjunto para manter os nomes dos atores
em ordem alfabética

    // Adicionar todos os atores à lista
    for (const auto& pair : adjacencyList) {
        actors.insert(pair.first);
    }

    // Imprimir o número de Bacon para cada ator
    for (const string& actor : actors) {
        const Actor& currentActor = adjacencyList.at(actor);
        cout << "O numero de Bacon de " << currentActor.name << " é " <<
currentActor.baconNumber;
        cout << " pelo filme " << currentActor.baconMovie << endl;
    }
}
```

Figure 4: print

2.3 print

Essa função recebe como parametro uma lista de adjacência contendo os atores.A função print é responsável por imprimir a saída do cálculo do número de Bacon para cada ator da lista de adjacência, utiliza-se do set para imprimir o nóme dos atores em forma ordenada

Sua complexidade é dada por $\mathcal{O}(n+m)$ a qual n é o número de pares na lista e m o número de atores no set