Relatório Projeto 2 - EDA

João Pedro Soares - Antônio Erick

July 2023

1 Problema 1

A resolução deste problema consiste primeiramente na leitura do arquivo inserido pelo usuário do tipo .txt, que consiste na utilização da variável *arq* do tipo *fstream* para leitura deste arquivo:

```cpp
int main(){
    fstream arq;
    arq.open("text.txt");
    int n = 0,m = 0;
    int u,v;
    //n = número de vertices.
    //m = número de arestas.
    //(u,v) = vizinhança.
    while(!arq.eof()){//Enquanto não chegar no fim do arquivo.
        arq >> n >> m;
        if (n == 0 && m == 0){
            continue;
        }
        //cout << "Numero de vertices = " << n << endl << "Numero de arestas = " << m << endl;
        vector<list<int>> grafo;
        grafo.resize(n);//Faço o vector ter tamanho n.
        for(int i = 0; i < m; i++){
            arq >> u >> v; //Leio u e v
            grafo[u].push_back(v); //Crio o par (u,v).
            //cout << "(" << u << "," << v << ") - ";
        }
        //Função que diz se posso pintar um grafo com apenas 2 cores ('Red' ou 'Blue')
        //de modo que nenhum grafo vizinho tenha a mesma cor ou seja, o grafo pode ser bipartido
        //ou visto como 2 conjuntos onde elementos do mesmo conjunto não façam vizinhança entre si.
        BFS(grafo,n);
        n = 0;
        m = 0;
    }
}
```

Figure 1:

Após isso, rodamos o arquivo dentro de um while que ficará em loop até seu término, aqui fazemos a construção do gráfico lendo as primeiras linhas que consistem no número de vértices e arestas do grafo em questão, se as variáveis n e m não se alterarem, implica dizer que a linha está vazia, logo avançamos para o próximo loop com o *continue*; ao ler o número de vértices e arestas criamos enfim o nosso grafo que será representado por um *vector* de *lista* de *inteiros*, sendo este do tamanho do número de vértices n.

Após isso, dentro de um for fazemos a construção propriamente dita do grafo, lendo par por par do arquivo e adicionando-os no vector $grafo$. Lido o primeiro grafo a função de busca em largura é chamada, denominada de BFS, passando como parâmetro uma referência de um *vector* de *lista* de *inteiros* e um *inteiro* representando o número de vértices; no nosso caso, vector $grafo$, e o inteiro n.

Terminada a verificação de coloração do grafo as variáveis n e m são reinicializadas e um novo ciclo começa.

Partindo então para a função de busca propriamente dita BFS que primeiramente cria um $vector\ cor$ de tamanho $grafo.size()$, e atribui para cada índice uma cor neutra, ou $undefined$ usada na nossa implementação. Conseguimos, então, trabalhar para simbolizar a coloração de um vértice e verificar se o $grafo$ pode ser ou não bipartido, ou dividido em duas cores alternadas como é descrito no problema.

```
//Busca em largura
void BFS(const vector<list<int>>& grafo,int vertices){
    //Criação e primeira coloração de cada vertice.
    vector<string> color;
    color.resize(grafo.size());
    for(int i = 0; i < grafo.size(); i++){
        color[i] = "undefined";
    }
    queue<int> fila; //Fila vazia.
```

Figure 2:

Assim, cria-se uma fila inicialmente vazia e o caminho no $grafo$ é iniciado sequencialmente por meio de um for, onde a primeira condição proposta é verificar se o índice, tratado no laço, do $vector$ de cores está simbolizada como $undefined$, se a condição for verdadeira, ela é pintada de vermelho e puxada para a fila, iniciando um novo laço de repetição enquanto a fila estiver cheia para tratar com a coloração dos elementos que estão na fila. Dentro desse laço é atribuído à variável u o valor presente no início da fila, retirando-o logo em sequência pelas funções respectivamente ditas: $fila.front()$ e $fila.pop()$

```
for(int index = 0; index < grafo.size(); index++){
    if(color[index] == "undefined"){
        fila.push(index); //Coloco na fila o primeiro vertice.
        color[index] = "Red";//Pinto de vermelho o primeiro vertice.
        while(!fila.empty()){
            int u = fila.front(); //Pego o primeiro da fila.
            fila.pop();//Retiro o primeiro da fila.
            for(int v : grafo[u]){
                //Tomada de decisão para pintura de um vertice ainda não visitado.
                if(color[v] == "undefined"){
                    if(color[u] == "Red"){
                        color[v] = "Blue";
                    }else if(color[u] == "Blue"){
                        color[v] = "Red";
                    }
                    fila.push(v);
                }else if(color[v] == color[u]){
                    cout << "Não!" << endl;
                    return;
                }
            }
        }
    }
}
cout << "Sim!" << endl;
```

Figure 3:

Com o valor de u em mãos, inicia-se um percurso pela lista do $grafo[u]$ para verificar seus vizinhos, atribuindo-lhes cores alternadas de u se a coloração do vizinho for $undefined$. Entretanto, se em algum momento a cor do vizinho já estiver "pintada" e possuir coloração igual a de u, a função printa Nao e finaliza-o. Se o laço não for encerrado durante todo o percurso, implica dizer que não existe vértices vizinhos que possuam as mesmas cores, logo a função retorna Sim e encerra.

Com relação a complexidade da função BFS temos:

- Primeiramente a declaração de *vector* e resize do mesmo, ambas as complexidades $O(1)$.
- Em seguida um for que ficará em loop de acordo com o tamanho do $grafo$, ou seja, o número de vértices. Dentro do for temos apenas uma atribuição ao *vector* color, $O(1)$. Portanto a complexidade será dependente apenas do número de vértices, logo será $O(V)$, sendo V o número de vértices no grafo.
- Em sequência temos mais uma declaração de variável, $O(1)$.
- E na sequência mais um for dependente do número de vértices, analisando o pior caso deste for será portanto $O(V)$, entretanto neste for temos algumas coisas a acrescentar dentro da condição if, $O(1)$:
  - Uma declaração e uma atribuição, ambas $O(1)$.
  - Um laço de repetição que ficará em loop enquanto a pilha estiver cheia, onde temos uma declaração e uma atribuição $O(1)$ e um novo laço for que andará pela lista representante dos vizinhos de um vértices, suas arestas, portanto, no pior caso, o número de iterações neste laço será o número de arestas do grafo, logo, $O(E)$, sendo E o número de arestas.

  Com o fim do for concluímos que seu grau de complexidade será $O(V+E)$, dado as complexidades internas do mesmo.

Colocando as complexidades significativas na mesa temos um for com complexidade $O(V)$ e outro $O(V+E)$, portanto na função geral o grau de complexidade será $O(V+E)$.

2 Problema 2

A leitura do arquivo é semelhante a do primeiro problema, com o uso novamente de uma variável do tipo $fstream$ para leitura do arquivo, a diferença entre os dois é o uso do *getline* através de um laço de repetição *while*, quebrando a linha em tokens e inserindo-os em um *vector* de *strings* geral. Após o fim da leitura do arquivo organizamos cada *string* do *vector* geral em partições, distribuindo os nomes, vizinhos e filmes em três diferentes *vector*'s, um para os nomes, outro para vizinhos, outro para os filmes.

```
input.open("input.txt");

if(!input.is_open()){
    cout << "Error ao abrir o arquivo!" << endl;
}
vector<string> nomes;
vector<string> filmes;
vector<string> vizinhos;
vector<string> geral;
string line;
while(getline(input,line)){
    stringstream ss {line};

    string token;

    while(getline(ss,token,';')){
        geral.push_back(token);
    }
}

for(int i = 0; i < geral.size();i++){
    if(i%3 == 0){
        nomes.push_back(geral[i]);
    }else if(i%3 == 1){
        filmes.push_back(geral[i]);
    }else if(i%3 == 2){
        vizinhos.push_back(geral[i]);
    }
}
```

Figure 4: Enter Caption

Tendo em mãos os três *vector*'s devidamente organizados, iniciamos uma *hash* do tipo *string*, *vector* de $pair < string, string >$, e inserimos nela os pares de ligação, o nome, ligado com o par (filme, vizinho), implementamos do nome para o vizinho, e do vizinho para o nome, dois grafos direcionados representados numa mesma *hash*, logo, temos a representação de um grafo não direcionado na *hash*.

```cpp
unordered_map <string,vector<pair<string,string>>> hash;

for(int i = 0; i < nomes.size();i++){
    string u = nomes[i];
    string edge = filmes[i];
    string v = vizinhos[i];

    hash[u].emplace_back(edge,v);
}
for(int i = 0; i < nomes.size(); i++){
    string u = nomes[i];
    string edge = filmes[i];
    string v = vizinhos[i];

    hash[v].emplace_back(edge,u);
}
```

Figure 5: Enter Caption

Após isso é feito um *clear* do *vector* nomes e atribuído novamente por meio de um for o primeiro elemento de cada índice da tabela *hash*, garantindo assim que todos os nomes que deveriam estar presentes no grafo estão alocados no *vector*.

Finalizado o for chamamos a função $baconNum$, que passa como parâmetro uma *hash* do tipo *string*, *vector* de $pair < string, string >$, além de um *vector* de *strings*; no nosso caso, a *hash* e o *vector* de nomes.

```cpp
  nomes.clear();
 for(auto&i : hash){
      nomes.push_back(i.first);
  }
 baconNum(hash,nomes);
```

Figure 6: Enter Caption

Falando sobre a função $baconNum$, tendo o *vector* de nomes em mãos, a primeira coisa a ser feita é ordenar ele e colocar os elementos em ordem alfabética como manda na descrição do problema, para isso usamos a função auxiliar *sort* da biblioteca *algorithm*, passando como parâmetro o início e o fim do *vector*, no caso, $nomes.begin()$ e $nomes.end()$. Com o *vector* ordenado inicializamos um for para o percorrer. Dentro deste for atribuímos a uma variável *inteira* bacon o valor retornado ao chamarmos uma função de busca BFS de estrutura semelhante a do primeiro problema, passando como parâmetro a *hash* e uma *string* referente a posição do *vector*, atribuído a variável i. Falando da função BFS como já dito ela é semelhante a do primeira problema, porém retorna um inteiro, para sabermos quanto um vértice teve que andar para chegar no "Kevin Bacon", para isso iniciamos uma *string* com "Kevin Bacon" e fazemos uma condição para sabermos se a *string* dada como parâmetro já é o próprio "Kevin Bacon", se for, retorna -2, se não, o programa segue e cria-se uma fila de $pair < string, int >$ e adicionamos a ele a *string* passada como parâmetro e o inteiro 0, além de uma declaração de uma *hash* do tipo *set* para "pintar" um nó já visitado chamado cinza. Iniciamos um laço para percorrer a fila com o caminho até "Kevin Bacon" verificando o vizinho da *string* dada; se for "Kevin Bacon" retorno o número do caminho traçado, se não o laço continua com o próximo nome que é o vizinho da *string* passada, fazendo isso até encontrar "Kevin Bacon", se o for se encerrar percorrendo todos os nomes, então retorna -1, implicando que o nome não possui ligação com "Kevin Bacon".

```
int BFS(const unordered_map <string,vector<pair<string,string>>> hash,string n){
    string kb = "Kevin Bacon";

    if(n == kb){ //É o KB
        return -2;
    }
    queue<pair<string,int>> fil;
    fil.push({n,0});

    unordered_set<string> cinzas;
    cinzas.insert(n);

    while(!fil.empty()){
        string nome = fil.front().first;
        int bn = fil.front().second;
        fil.pop();

        if(nome == kb){
            return bn;
        }
        for(auto& i : hash.at(nome)){
            string vizinho = i.second;
            if(cinzas.find(vizinho) == cinzas.end()){

                fil.push({vizinho,bn+1});
                cinzas.insert(vizinho);
            }
        }
    }
    return -1; //sinaliza que nao chegou ao KB
```

Figure 7: Enter Caption

Após o retorno da função, compara-se e printa-se a mensagem de acordo com o inteiro resultante como visto a seguir:

Sobre a complexidade, a função BFS possui a mesma complexidade da já apresentada no primeiro problema, sendo portanto $O(V+E)$, sendo V o número de vértices e E o número de arestas do grafo. Entretanto na função $baconNum$ primeiramente o chamado da função *sort* que usa o QuickSort em sua imple-

```cpp
void baconNum(const unordered_map <string,vector<pair<string,string>>> hash,vector<string> nomes){
    sort(nomes.begin(),nomes.end());

    for(string& i : nomes){
        int bacon = BFS(hash,i);
        if(bacon == -1){
            //Não chegou ao Kevin Bacon
        }else if(bacon == -2){
            cout  << "O número de Bacon de " << i << " é " << 0 << " pelo fime " << endl;
        }else{
            cout << "O número de Bacon de " << i << " é " << bacon << " pelo fime " << hash.at(i)[0].first << endl;
        }
    }
}
```

Figure 8: Enter Caption

mentação, tendo por sua vez, complexidade $O(N \log_N)$, como já provada pelo professor em aula, ademais, como nosso *vector* de nomes possui o número de vértices no grafo, a complexidade pode ser substituída por $O(V \log_V)$. Além disso possui um for que ficará em loop percorrendo todos os vértices, portanto, complexidade de $O(V)$, dentro deste for a função BFS é chamada em todas as iterações, portanto a complexidade deste for será $O(V * (V + E))$. Portanto a complexidade total da função $baconNum$ será $O(V \log_V + V * (V + E))$.

3 Problema 3

A leitura do arquivo é mais rápida do que a dos problemas anteriores pelo fato de cada arquivo ter apenas um grafo, dessa vez cria-se dois vetores, um *vector* para armazenar o grafo lido no arquivo, sendo este, **NÃO DIRECIONADO** e um outro *vector* que será trabalhado, este sendo **DIRECIONADO**, tudo isso por meio de um for semelhante aos outros problemas. Vale destacar que nesse problema declaramos duas variáveis globais, sendo elas *int* "global" inicializada com 0 e um *vector* do tipo $pair < int, int >$ "pontes". Após a leitura do arquivo a função de busca em profundidade DFS que recebe como parâmetro um *vector* de lista de inteiros é chamada, passando nosso grafo **NÃO DIRECIONADO** como parâmetro.

```cpp
int main(){
    fstream arq;
    arq.open("grafo2.txt");
    int n, m;
    arq >> n >> m;
    vector <list<int>> grafo; //Grafo nao direcionado
    vector <list<int>> grafoDirec; //Grafo direcionado
    grafo.resize(n);
    grafoDirec.resize(n);

    if(arq.is_open()){
        for(int i = 0; i < m; i++){
            int u, v;
            arq >> u >> v;
            grafo[u].push_back(v); // Criando (u,v)
            grafo[v].push_back(u); //Criando (v,u) por ser não direcionado

            grafoDirec[u].push_back(v);// Criando (u,v) no grafo direcionado
        }
    }

    DFS(grafo);// Essa função irá descobrir as pontes.
```

Figure 9: Enter Caption

- A função DFS como já dita, faz uma busca em profundidade pelo grafo. Primeiramente dando um *clear* no *vector* pontes. A DFS declara um *vector* de *strings*, além 4 outros do tipo *int* de nomes: color(representando a cor do vértice), pi(representando o pai do vértice), upperTime(representando o tempo de encerramento do vértice), lowerTime(representando o tempo de descoberta do vértice) e o minTime (representando a menor distância entre o vértice e seus vizinhos); atribuindo logo após, 0 a variável "global".

Logo após um laço é inicializado, e para cada índice branco no *vector* de cores, é chamada a função $DFSVisit$ que passa, todos as variáveis declaradas anteriormente dentro da função de maneira referenciada, além de um *int* i e o grafo em si.

```cpp
void DFS(vector <list<int>> grafo){
    pontes.clear();
    vector<string> color(grafo.size(),"Branco"); //Inicio o vetor de cores todos brancos
    vector<int> pi(grafo.size(),-1); //-1 representa que não há pai para o vertice
    vector<int> upperTime(grafo.size(),0); //Tempo de encerramento
    vector<int> LowerTime(grafo.size(),0);//Tempo de descoberta
    vector<int> minTime(grafo.size(),0);//Menor tempo entre o vertice e seus vizinhos

    global = 0;

    for(int i = 0; i < grafo.size();i++){
        if(color[i] == "Branco"){
            DFSVisit(grafo,i,color,upperTime,lowerTime,pi,minTime);
        }
    }
}
```

Figure 10: Enter Caption

- Dentro dessa $DFSVisit$ a primeira coisa a se fazer é incrementar a variável "global", atribuir seu valor ao *vector* lower e ao *vector* minTime, atribuir a cor "Cinza" ao *vector* de cores, simbolizando que o vértice foi descoberto e por último declarar um *int* que conta o número de filhos que o vértice tem.
  Logo após é iniciado um laço de repetição que tem por função andar pela lista de um vértice através de um inteiro k, percorrendo todos os seus vizinhos, onde, para cada iteração fazemos:
    · Se vizinho em questão possuir cor "Branca":
    · Atribuímos o valor de i ao *vector* pi de índice k,
    · Incrementamos a variável filhos.
    · Chamamos recursivamente $DFSVisit$, passando as variáveis criadas como parâmetro, além do grafo e do índice k, para verificar novos caminhos no vértice.
    · Atualizamos o minTime do índice i.
    · Fazemos a verificação da menor distância entre dois vértices e adicionamos o $pair < i, k >$ no *vector* "pontes".
    · Se k for diferente do índice i no *vector* pi, apenas atualizamos o minTime de índice i.

  Após o fim do for, colorimos o vértice trabalhado de "Preto", simbolizando que o vértice já foi finalizado, incrementamos a variável "global" e atribuímos seu valor ao índice referente ao vértice do *vector* upper.

```
void DFSVisit(vector<list<int>> grafo, int i,vector<string>& color, vector<in
    global++;
    lower[i] = global; // Tempo de descoberta
    color[i] = "Cinza"; //Vertice já descoberto
    int filhos = 0; //Conta quantos filhos o vertice tem
    minTime[i] = global; //Menor tempo de descoberta

    //Para cada elemento na adj. pesquise.
    for(auto& k : grafo[i]){
        if(color[k] == "Branco"){
            pi[k] = i;
            filhos++;
            DFSVisit(grafo,k,color,upper,lower,pi,minTime);
            minTime[i] = min(minTime[i],minTime[k]);
            if(minTime[k] > minTime[i]){
                pontes.push_back(make_pair(i,k));
            }
        }else if(k != pi[i]){
            minTime[i] = min(minTime[i],minTime[k]);
        }

    }
    color[i] = "Preto";
    global++;
    upper[i] = global;

}
```

Figure 11: Enter Caption

Finalizada a busca em Profundidade DFS adicionamos todas os pares do *vector* "pontes" no *vector* de grafo **DIRECIONADO**. E em seguida, por meio de mais um laço verificamos os vértices que não tem saída, alternando as arestas sequenciadas no seu caminho e inserindo no *vector* grafo **DIRECIONADO** e removendo a aresta que não é ponte, aquela que deixa o vértice sem saída.

```cpp
//Adicionando a aresta ponte de volta.
for(int i = 0; i < pontes.size();i++){
    int u = pontes[i].first;
    int v = pontes[i].second;

    grafoDirec[v].push_back(u);
}

//Corrigindo vertices que deveriam ter saída mas não tem.
for(int i = 0; i < grafoDirec.size(); i ++){
    if(grafoDirec[i].size() == 0){ // Lista vazia
        int min = grafo[i].front();
        for(int& k : grafo[i]){
            if(k < min){
                min = k; //achando o menor;
            }
        }
        grafoDirec[i].push_back(min); //Adicionando a aresta
        grafoDirec[min].remove(i); //Removendo a aresta que não é ponte
    }
}
```

Figure 12: Enter Caption

Finalizada a implementação, a sequência do código é uma formatação de saída, onde, utilizando a função *sort*, ordenamos a lista de adjacência de cada índice no *vector*. Em seguida ocorre a impressão do grafo no formato desejado.

```cpp
//Deixando o grafo em ordem para a impressão
for(int i = 0; i < grafoDirec.size(); i++){
    grafoDirec[i].sort();
}

//Imprimindo o grafo
for(int i = 0; i < grafoDirec.size(); i++){
    for(int& k : grafoDirec[i]){
        cout << "(" << i << "," << k << ")" << endl;
    }
}
cout << "#" << endl;
```

Figure 13: Enter Caption

Falando sobre as complexidades significativas, temos que a função DFS apresenta um for que ficará em loop caminhando pelo grafo, logo sua complexidade será baseada no número de vértices no grafo, portanto, $O(V)$, dentro deste for ocorre V iterações ,no pior caso, de chamadas da função $DFSVisit$, que por sua vez percorre uma lista de adjacências, determinada pelas arestas de um vértice, utilizamos cores para ter a certeza de que visitaremos cada aresta e cada vértice apenas uma vez, garantindo que a complexidade da função $DFSVisit$ seja dependente do número de arestas no grafo, sendo portanto $O(E)$, logo, a complexidade da função DFS será $O(V + E)$, sendo V o número de vértices, e E o número de arestas.

4 Bibliografia

4.1 Questão 1

- As aulas ministradas pelo professor durante o semestre.
- Slides do professor, com relação a ideia de implementação da busca em largura e complexidade da função.

4.2 Questão 2

- As aulas ministradas pelo professor durante o semestre.
- Slides do professor, com relação a ideia de implementação da busca em largura e complexidade da função .
- Do link:
  Para auxílio da Tabela Hash no C++

4.3 Questão 3

- As aulas ministradas pelo professor durante o semestre.
- Slides do professor sobre grafos, com relação a ideia de implementação da busca em profundidade e complexidade da função.
- Do link: Para auxílio na resolução do Problema