EDA - Projeto Final

Ezequiel dos Santos Melo, Gustavo Fernandes de Barros

8 de julho de 2023

1 [Problema 1] Colorindo os vértices do grafo com duas cores

Meus primeiros pensamentos sobre a resolução foram de certa forma intuitivos, visto que, tratando-se somente de duas cores, talvez fosse possível obter a resposta testando, ou seja, colorindo cada vértice do grafo para verificar se o mesmo pode ser colorido com duas cores ou não.

Após pesquisar sobre o problema da $\varkappa$-coloração de vértices, descobri uma relação direta entre a veracidade desta propriedade para duas cores e grafos bipartidos, onde há um teorema no qual um grafo pode ser colorido com duas cores **se e somente se** ele for bipartido. Desta forma, é possível aproveitar-se deste teorema para resolver a questão.

Uma forma algorítmica de resolver essa questão é usando busca em largura para descobrir se um grafo é bipartido é de fato verificando se é possível colorir o mesmo com duas cores de forma que vértices adjacentes tenham cores distintas, **percorrendo todos os vértices do grafo, colorindo-os e verificando a cor de seus vizinhos.** Caso haja em algum momento vizinhos de cores semelhantes, logo conclui-se que o grafo não é bipartido, portanto não é possível que o mesmo seja colorido com duas cores.

```
bool Bicoloration(const Graph& graph) {
    int number_of_vertices = graph.get_number_of_vertices();

    vector<char> colors(number_of_vertices, ' ');
    vector<bool> visited_vertices(number_of_vertices, false);

    for(int i = 0; i < number_of_vertices; i++) {
        if(!visited_vertices[i]) {
            queue<int> queue_of_vertices;
            queue_of_vertices.push(i);
            colors[i] = 'R';

            while(!queue_of_vertices.empty()) {
                int current_vertice = queue_of_vertices.front();
                queue_of_vertices.pop();
                visited_vertices[current_vertice] = true;

                for(const auto& edge : graph.neighbors(current_vertice)) {
                    int neighbor = edge.get_destiny();

                    if(colors[neighbor] == ' ') {
                        if(colors[current_vertice] == 'R') colors[neighbor] = 'B';

                        else if(colors[current_vertice] == 'B') colors[neighbor] = 'R';

                        queue_of_vertices.push(neighbor);
                    }
```

```
                        else if(colors[neighbor] == colors[current_vertice]) return false;
                    }
                }
            }
        }

    return true;
}
```

Analisando a complexidade, temos um loop que percorre todos os vértices do grafo, sendo O(V), onde V é o número de vértices. A inicialização dos vetores auxiliares são constantes, portanto não influenciam na complexidade, enquanto o loop mais interno a função pode percorrer todas as arestas do grafo, além de realizar algumas verificações acerca dos vizinhos do vértice atual, podendo atingir uma complexidade O(A), onde A é o número de arestas. Logo, pode-se dizer que a complexidade desse algoritmo é **O(V + A).**

2 [Problema 2] Seis graus de Kevin Bacon

Acredito que os primeiros empecilhos em relação a essa questão foram em relação a leitura do arquivo em si, de que forma seria possível extrair todos os atores e transformá-los em vértices, da mesma forma com os filmes que teriam de virar arestas, e principalmente pelo fato de que muitos atores tinham seus nomes repetidos no arquivo, o que atrapalhava na criação das arestas.

Passamos algum tempo pensando em usar templates nas classes de grafo e aresta, incluvise também chegamos a criar uma classe vértice para armazenar o nome dos atores, porém não prosseguimos com a abordagem por encontrar alguns obstáculos de lógica no caminho, no entanto enquanto fazíamos essa questão, pensamos novamente nessa possibilidade, provavelmente funcionaria e seria mais rentável do que a abordagem utilizada, porém rersolvemos utilizar lista de adjacências junto de estruturas de dados adicionais para auxílio.

Basicamente implementamos um algoritmo para ler o arquivo e ir criando as arestas enquanto isso, ao mesmo tempo que mantinhamos vetores para armazenar atores e filmes separadamente, já que basicamente cada linha do arquivo é uma aresta no grafo. Para evitar as repetições de atores, o algoritmo verificava para todo ator no arquivo se o mesmo já estava no vetor antes de adicioná-lo, além de que sempre eram mantidos os últimos filme e ator em variáveis para construção da aresta.

```
void create_graph(const string& file_name, Graph& graph, vector<string>& actors, vector<
    ifstream file(file_name);
    string line;

    while(getline(file, line)) {
        istringstream new_line(line);
        string element;

        int counter, current_actor, current_movie;

        counter = current_actor = current_movie = 0;

        while(getline(new_line, element, ';')) {
            if(counter == 0) {
                if(find(actors.begin(), actors.end(), element) == actors.end()) {
                    actors.push_back(element);
                }

                current_actor = distance(actors.begin(), find(actors.begin(), actors.end
```

```
        }

        if(counter == 2) {
            if(find(actors.begin(), actors.end(), element) == actors.end()) actors.p

            Edge edge(current_actor, distance(actors.begin(), find(actors.begin(), a

            graph.insert(edge);
        }

        else if(counter == 1) {
            movies.push_back(element);

            current_movie = distance(movies.begin(), find(movies.begin(), movies.end
        }

        counter++;
    }
}

file.close();
}
}
```

Para esse algoritmo, o que pesa mais além da iteração sobre os elementos do arquivo é justamente a questão da verificação de atores repetidos, já que busca e inserção em vetores pode assumar complexidade O(n) e, tratando-se do fato de que essa operação é frequentemente realizada duas vezes por iteração, podemos considerar a complexidade de $\mathbf{O(n^2)}$.

Após a implementação do grafo, a questão pode ser resolvida com uma busca por largura, partindo do vértice desejado e contando até chegar ao destino, no caso o vértice do ator Kevin Bacon.

Utilizamos uma fila para armazenar o índice dos atores que têm de ser visitados, assim como também utilizamos um vetor para marcar os vértices já visitados, assim inicia-se um loop para visitar todos os vértices até encontrar Kevin Bacon. A cada iteração compara-se o ator de interesse com o ator atual da fila, se forem diferentes, a variável correspondente ao número de Bacon é incrementada e a busca continua.

```
int bacon_number(const Graph& graph, const vector<string>& actors, const vector<string>&
    int actor_index = distance(actors.begin(), find(actors.begin(), actors.end(), actor)
    int number;

    if(actor_index != actors.size()) {
        number = 0;

        queue<int> actors_queue;
        vector<bool> visited(graph.get_number_of_vertices(), false);

        actors_queue.push(actor_index);
        visited[actor_index] = true;

        while(!actors_queue.empty()) {
            int current_actor = actors_queue.front();
            actors_queue.pop();

            if(current_actor != actor_index) number++;
```

```
            for(const Edge& edge : graph.neighbors(current_actor)) {
                int neighbor = edge.get_destiny();

                if(!visited[neighbor]) {
                    visited[neighbor] = true;

                    actors_queue.push(neighbor);
                }
            }
        }
    }

    return number;
}
```

Nesse caso a complexidade se dá pelo número de atores e número de arestas presentes no grafo, onde é necessário fazer uma busca linear no vetor de atores para encontrar o ator de interesse e percorrer os vizinhos de cada ator, que podemos aproximar pelo número de arestas do grafo, resultando em complexidade **O(n + m)**, onde n é o número de vértices e m é o número de arestas.

3 [Problema 3] Vias de mão dupla