UNIVERSIDADE FEDERAL DO AGRESTE DE PERNAMBUCO

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ISMAEL DIOGENYS DOS SANTOS CORREIA

ANÁLISE DE GRAFOS DIRECIONADOS EM REDES SOCIAIS

RECOMENDAÇÕES POR MEIO DO ALGORITMO DE KOSARAJU

E COMPONENTES FORTEMENTE CONEXAS

GARANHUNS—PE

2024

Sumário

1. Introdução 3
   1.1 Contexto e Objetivo do Projeto 3
   1.2 Abordagem Escolhida 3
2. Conceitos Teóricos 4
   2.1 Noções Básicas sobre grafos 4
   2.2 Algoritmos Principais Utilizados 4
3. Descrição do Projeto 6
   3.1 Problema a ser Resolvido 6
   3.2 Requisitos do Sistema 6
   3.3 Ferramentas e Tecnologias Utilizadas 6
4. Implementação 7
   4.1 Estrutura do Grafo 7
   4.2 Algoritmos Implementados 8
   4.3 Exemplos de Execução 11
5. Teste e Resultados 13
6. Conclusão 16
   6.1 Resumo dos Resultados 16
   6.2 Melhorias Futuras 16
7. Fontes Consultadas 17

1. Introdução

1.1 Contexto e Objetivo do Projeto

Nos dias atuais, as redes sociais desempenham um papel fundamental na interação entre indivíduos, permitindo a troca de informações, experiências e recomendações. Com o crescimento exponencial dessas plataformas, torna-se essencial desenvolver métodos que otimizem a experiência do usuário, promovendo conexões significativas. Neste contexto, a análise de grafos se apresenta como uma ferramenta poderosa, permitindo representar usuários e suas interações como vértices e arestas, respectivamente.

O objetivo deste projeto é implementar um sistema de recomendações de amizade em uma rede social fictícia, utilizando o algoritmo de Kosaraju para identificar componentes fortemente conexas. A identificação dessas componentes permite agrupar usuários que possuem conexões significativas, facilitando a sugestão de novas amizades com base em relações existentes. Ao explorar as propriedades dos grafos e aplicar o algoritmo de Kosaraju, este trabalho busca aprimorar a relevância das recomendações, oferecendo uma abordagem eficiente e fundamentada na estrutura da rede social.

1.2 Abordagem Escolhida

A abordagem adotada neste projeto baseia-se na modelagem da rede social como um grafo, onde os usuários são representados por vértices e suas interações, como amizades e seguidores, são representadas por arestas. Essa representação em grafos possibilita uma análise mais profunda das relações entre os usuários, permitindo a identificação de padrões e a recomendação de novos vínculos.

Para identificar as conexões significativas dentro da rede, utilizamos o algoritmo de Kosaraju, que é eficiente para determinar componentes fortemente conexas em um grafo direcionado. Este algoritmo é particularmente adequado para o nosso objetivo, pois permite identificar grupos de usuários que estão fortemente interligados, ou seja, aqueles que possuem um número elevado de interações entre si. Assim, a identificação dessas componentes não apenas revela a estrutura da rede, mas também fornece uma base sólida para as recomendações de amizade.

Após a identificação das componentes fortemente conexas, a lógica de recomendação foi implementada para sugerir novas amizades. Os usuários são recomendados a se conectar com aqueles que pertencem à mesma componente, facilitando a formação de novas interações dentro de grupos já estabelecidos. Essa abordagem não apenas melhora a relevância das recomendações, mas também promove uma maior coesão social dentro da rede, maximizando as chances de interações significativas entre os usuários.

2. Conceitos Teóricos

2.1 Noções Básicas sobre Grafos

Neste projeto, utilizamos um grafo direcionado, onde cada aresta indica uma relação de "seguimento" entre os usuários. Nesse contexto, o vértice de saída representa um usuário que segue o vértice de entrada. Assim, cada aresta simboliza a ação de um usuário seguir outro, refletindo as interações na rede social de forma clara e estruturada.

Além disso, utilizamos o conceito de componentes fortemente conexas, que são subgrafos onde existe um caminho direcionado entre todos os pares de vértices. Isso significa que, dentro de uma componente fortemente conexa, cada usuário pode alcançar qualquer outro usuário por meio das arestas do grafo. Essa característica é fundamental para o sistema de recomendações, pois permite identificar grupos de usuários com interações mais intensas, possibilitando a sugestão de novos seguidores com base nas conexões existentes.

Formalmente uma componente fortemente conexa (SCC) de um grafo orientado G = (V,E) é um conjunto máximo de vértices $C \subseteq V$, tal que, para todo par de vértice u e $v \in C$, existe um caminho de u para v e um caminho de v para u, isto é $u \rightarrow v$ e $v \rightarrow u$.

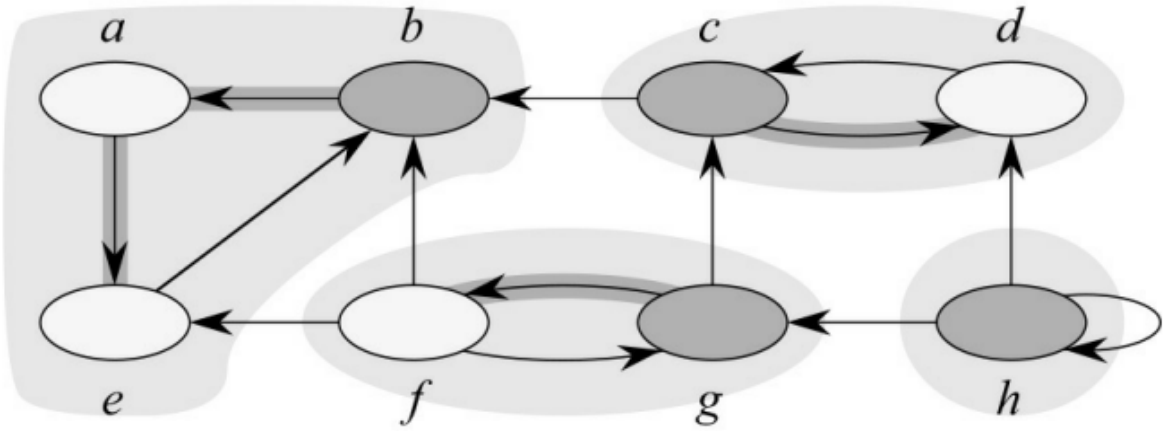

Figura 1. Cada região sombreada representa uma componente fortemente conexa dentro de um grafo direcionado.

Fonte: CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: Teoria e Prática*. 3. ed. Rio de Janeiro: Elsevier, 2013, seção 22.5.

2.2 Algoritmos Principais Utilizados

A Busca em Profundidade (*Depth-First Search - DFS*) é um algoritmo fundamental para explorar grafos. Iniciando em um vértice, ele explora o máximo possível ao longo de cada ramo antes de retroceder. No nosso projeto, a *DFS* é utilizada para identificar componentes fortemente conexas em um grafo direcionado. Durante a execução, os vértices são marcados como visitados e armazenados em uma pilha. Além disso, um temporizador é registrado para marcar a ordem de visitação dos vértices, informação que é crucial para o algoritmo de Kosaraju, que utiliza essa ordem para determinar a estrutura do grafo.

O algoritmo de Kosaraju é uma abordagem eficiente para encontrar componentes fortemente conexas em um grafo direcionado. Ele opera em duas passagens principais. Na primeira passagem, o algoritmo realiza uma Busca em Profundidade (*DFS*) no grafo original, marcando os vértices como visitados e armazenando-os em uma pilha na ordem em que são finalizados. Essa ordem é crucial, pois representa a sequência em que os vértices completam sua exploração.

Em seguida, na segunda passagem, o grafo é transposto, invertendo todas as arestas. A *DFS* é então executada novamente no grafo transposto, utilizando a ordem dos vértices armazenada na pilha. Cada vez que uma nova *DFS* é iniciada, todos os vértices alcançados pertencem a uma componente fortemente conexa. O algoritmo de Kosaraju é eficiente, operando em tempo linear em relação ao número de vértices e arestas do grafo, ou seja, O(V + E). No entanto, essa eficiência linear é garantida somente quando o grafo é representado por listas de adjacências. Essa característica torna o algoritmo uma escolha popular para problemas relacionados a componentes fortemente conexas em grafos direcionados.8

```
STRONGLY-CONNECTED-COMPONENTS(G).
1   chamar DFS(G) para calcular tempos de término u.f para cada vértice u
2   calcular G^T
3   chamar DFS(G^T) mas, no laço principal de DFS, considerar os vértices em ordem
        decrescente de u.f (como calculado na linha 1)
4   dar saída aos vértices de cada árvore na floresta em profundidade formada na linha 3
        como uma componente fortemente conexa separada
```

Figura 2. Pseudocódigo do Algoritmo de Kosaraju.

Fonte: CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: Teoria e Prática*. 3. ed. Rio de Janeiro: Elsevier, 2013, seção 22.5.

3. Descrição do Projeto

3.1 Problema a ser Resolvido

Em um ambiente digital cada vez mais conectado, as redes sociais enfrentam o desafio de oferecer experiências personalizadas que incentivem a interação entre os usuários. Um aspecto fundamental dessa personalização é a recomendação de amizades, que visa conectar usuários com interesses e comportamentos semelhantes. No entanto, muitas abordagens tradicionais de recomendação não consideram a estrutura subjacente das conexões entre os usuários, resultando em sugestões que podem ser irrelevantes ou ineficazes.

Nesse contexto, a utilização de componentes fortemente conexas se mostra uma solução lógica e eficaz. As componentes fortemente conexas representam grupos de usuários que estão interligados através de relações mútuas, formando comunidades onde os interesses e características são mais alinhados. Recomendar usuários que pertencem à mesma componente fortemente conexa aumenta a probabilidade de que essas novas conexões sejam bem-sucedidas, pois essas comunidades tendem a compartilhar interesses comuns.

Além disso, a identificação dessas comunidades através do algoritmo de Kosaraju não apenas melhora a relevância das recomendações, mas também enriquece a dinâmica social da rede. Conectar usuários que já têm amigos em comum fortalece o senso de pertencimento, resultando em uma rede social mais ativa e interativa. Assim, a abordagem baseada em componentes fortemente conexas não só resolve o problema de recomendações inadequadas, mas também promove o engajamento dos usuários e a formação de comunidades vibrantes.

3.2 Requisitos do Sistema

Para o correto funcionamento do sistema, é necessário fornecer um arquivo de entrada que contenha a definição do grafo representando a rede social. O arquivo deve especificar inicialmente o número de vértices (usuários) e arestas (conexões de seguimento), seguido pelas informações detalhadas de cada conexão. Cada vértice representa um usuário, enquanto cada aresta indica a ação de seguir entre dois usuários. Assim, para cada aresta, o arquivo deve informar qual usuário (vértice de origem) está seguindo outro usuário (vértice de destino).

3.3 Ferramentas e Tecnologias Utilizadas

O projeto foi totalmente desenvolvido em linguagem Java, utilizando a Eclipse IDE. Além disso, foram utilizadas as bibliotecas padrão da linguagem, como `java.util` para manipulação de estruturas de dados como listas e conjuntos, e `java.io` para operações de leitura e escrita de arquivos. A escolha da linguagem Java se deu por sua robustez, portabilidade e pelo vasto suporte oferecido para manipulação de grafos e algoritmos relacionados. A utilização da *Eclipse IDE* facilitou o desenvolvimento e forneceu ferramentas de depuração.

4. Implementação

4.1 Estrutura do Grafo

Para garantir que o algoritmo de Kosaraju opere com complexidade linear (O(V + E)), é fundamental que o grafo seja implementado utilizando listas de adjacências. Abaixo, apresentamos um diagrama de classes UML que ilustra as principais classes do projeto.

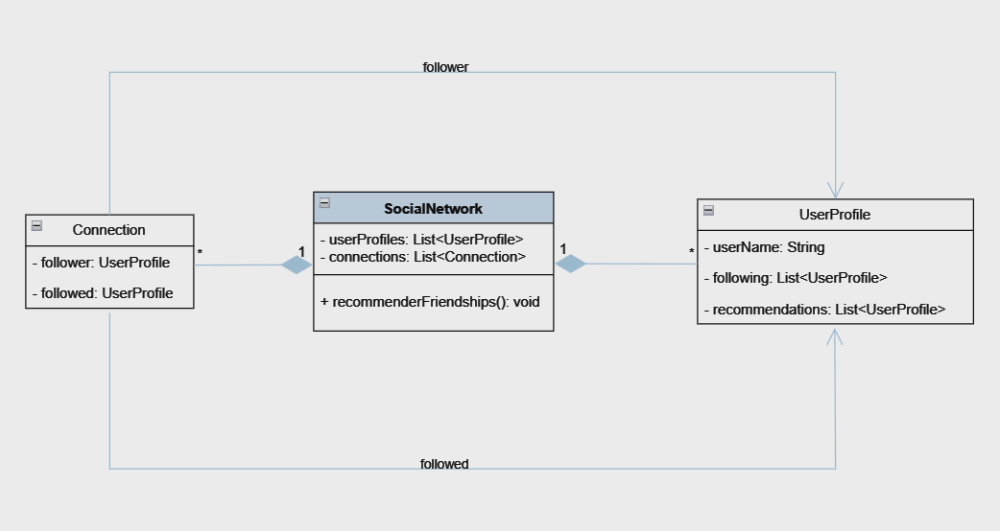

Figura 3. Diagrama de Classes UML do projeto.
Fonte: Elaborado pelo site draw.io.

Na Figura 3, apresentamos uma visão geral da implementação do grafo. A classe responsável por representar o grafo é denominada *SocialNetwork*, composta por uma lista de objetos *UserProfile* (que correspondem aos vértices do grafo) e uma lista de *Connection* (que representam as arestas do grafo). Cada *UserProfile* possui, entre seus atributos, um identificador do tipo *String*, uma lista chamada *Following*, que armazena os perfis de outros usuários seguidos (ou seja, uma lista de objetos *UserProfile*), e uma lista de recomendações de amizade chamada *recommendations*. Essa lista armazena sugestões de perfis que podem ser seguidos. Já a classe *Connection* é composta por dois atributos do tipo *UserProfile*: o *follower*, representando o usuário que segue, e o *followed*, representando o usuário seguido, estabelecendo assim a relação direcional entre os vértices.

4.2 Algoritmos Implementados

O algoritmo de busca em profundidade (*DFS*) é fundamental para a execução do algoritmo de Kosaraju. Abaixo, apresentamos a imagem do código desenvolvido para a implementação da *DFS*, que serve como um componente essencial na identificação das componentes fortemente conexas do grafo. Essa implementação permite a exploração eficaz de todos os vértices e arestas, estabelecendo uma base sólida para o funcionamento do algoritmo de Kosaraju.

```java
public void DFS() {
    this.initializer();
    this.setTimer(0);

    for(UserProfile user: this.userProfiles) {
        if(user.getColor().equals(Color.WHITE)) {
            DFS_Visit(user);
        }
    }
}

private void DFS_Visit(UserProfile user) {
    user.setDiscoveryTime(++this.timer);
    user.setColor(Color.GRAY);

    for(UserProfile v: user.getFollowing()) {
        if(v.getColor().equals(Color.WHITE)) {
            v.setParent(user);
            DFS_Visit(v);
        }
    }

    user.setColor(Color.BLACK);
    user.setFinishTime(++this.timer);
}

private void DFS_Visit(UserProfile user, List<UserProfile> currentComponent) {
    user.setDiscoveryTime(++this.timer);
    user.setColor(Color.GRAY);

    currentComponent.add(user);

    for(UserProfile v: user.getFollowing()) {
        if(v.getColor().equals(Color.WHITE)) {
            v.setParent(user);
            DFS_Visit(v, currentComponent);
        }
    }

    user.setColor(Color.BLACK);
    user.setFinishTime(++this.timer);
}
```

Figura 4. Implementação da Busca em Profundidade.
Fonte: Captura de tela da *Eclipse IDE*.

É importante ressaltar a sobrecarga do método *DFS_Visit*. Essa sobrecarga se deve à necessidade de inserir as componentes fortemente conexas em uma lista apropriada para essa finalidade. Com essa abordagem, conseguimos gerenciar a coleta de usuários pertencentes a cada componente, permitindo uma organização mais eficiente das informações e facilitando o processamento subsequente das recomendações de amizade.

Para uma melhor clareza e facilidade de manutenção, o algoritmo de Kosaraju, implementado pelo método *stronglyConnectedComponents*, é estruturado de maneira a invocar métodos auxiliares e privados. Essa abordagem modulariza a lógica do algoritmo, permitindo um entendimento mais profundo de cada etapa do processo e facilitando futuras modificações ou correções. A seguir, apresentamos a estrutura do código que foi desenvolvido para a implementação do algoritmo de Kosaraju, destacando a importância dos métodos auxiliares no fluxo do algoritmo.

```
82    public List<List<UserProfile>> stronglyConnectedComponents(){
83        this.DFS();
84
85        SocialNetwork transposed = this.transpose();
86
87        List<UserProfile> orderedProfiles = getOrderedProfiles();
88
89        initializeTransposedGraph(transposed);
90
91        List<List<UserProfile>> stronglyConnectedComponents = new ArrayList<>();
92
93        findStronglyConnectedComponents(transposed, orderedProfiles, stronglyConnectedComponents);
94
95        return stronglyConnectedComponents;
96    }
```

Figura 5. Implementação do algoritmo de Kosaraju pelo método stronglyConnectedComponents.
Fonte: Captura de tela da *Eclipse IDE*.

Na linha 83, o algoritmo invoca o método de busca em profundidade (DFS) no grafo original, preparando-o para a identificação de componentes fortemente conexas. Em seguida, na linha 85, o grafo é transposto, ou seja, a direção de todas as arestas é invertida. Na linha 87, os perfis de usuários são ordenados de forma decrescente com base no tempo de finalização da busca em profundidade (*DFS_Visit*) para cada usuário, e esses perfis ordenados são armazenados na lista *orderedProfiles*. Na linha 89, o grafo transposto é inicializado, preparando-o para uma nova execução da busca em profundidade. Logo após, na linha 91, uma estrutura do tipo *ArrayList* é criada para armazenar as componentes fortemente conexas do grafo. Na linha 93, o método *findStronglyConnectedComponents* é chamado, recebendo como parâmetros o grafo transposto, a lista de perfis ordenados e a lista onde as componentes fortemente conexas serão armazenadas. Finalmente, na linha 95, o método retorna a lista de listas contendo as componentes fortemente conexas identificadas no grafo.

Por fim, apresentamos o método responsável por gerar novas recomendações de amizade para cada perfil de usuário, utilizando as componentes fortemente conectadas identificadas pelo algoritmo de Kosaraju. Para proporcionar maior clareza e facilitar a manutenção, o método invoca um método auxiliar que, por sua vez, chama outro método auxiliar, conforme detalhado abaixo.

```
18    public void recommenderFriendships() {
19        List<List<UserProfile>> stronglyConnectedComponents = this.stronglyConnectedComponents();
20        this.printStronglyConnectedComponents(stronglyConnectedComponents);
21
22        for(List<UserProfile> component: stronglyConnectedComponents) {
23            processComponentRecommendations(component);
24        }
25    }
```

Figura 6. Método principal para a geração de recomendações.
Fonte: Captura de tela da *Eclipse IDE*.

Na linha 19, de *recommenderFrienships*, é criada uma lista de listas para armazenar as componentes fortemente conectadas, conforme fornecido pelo método *stronglyConnectedComponents*. Na linha 20, essas componentes são exibidas na tela. Em seguida, nas linhas 22 a 24, o processamento das recomendações de amizade em cada componente é realizado por meio do método auxiliar *processComponentRecommendations*. O detalhamento deste método será apresentado a seguir.

```
32    private void processComponentRecommendations(List<UserProfile> component) {
33        for(UserProfile user: component) {
34            user.getRecommendations().clear();
35            generateRecommendationsForUser(user, component);
36        }
37    }
```

Figura 7. Implementação do método responsável por iterar cada usuário da componente passada como parâmetro por *recommenderFriendships*.
Fonte: Captura de tela da *Eclipse IDE*.

O método *processComponentRecommendations* apresenta uma iteração que percorre todos os usuários de uma componente específica. Na linha 34, a lista de recomendações do usuário é esvaziada, garantindo que novas recomendações sejam geradas de forma adequada. Em seguida, na linha 35, o método auxiliar *generateRecommendationsForUser* é invocado, passando tanto o usuário atual quanto a componente fortemente conectada à qual esse usuário pertence. A descrição deste método pode ser encontrada abaixo.

```
45    private void generateRecommendationsForUser(UserProfile user, List<UserProfile> component) {
46        for(UserProfile otherUser: component) {
47            if(!otherUser.getFollowing().contains(user) && !user.equals(otherUser)) {
48                user.addRecomendation(otherUser);
49            }
50        }
51
52        printRecommendationsForUser(user);
53    }
```

Figura 8. Método responsável por adicionar à lista de recomendações.
Fonte: Captura de tela da *Eclipse IDE*.

O método *generateRecommendationsForUser* é responsável por adicionar as recomendações apropriadas a um usuário específico. Na linha 46, inicia-se uma iteração que percorre cada perfil na componente. Na linha 47, é verificado se o usuário atual da iteração não está na lista de seguidores do usuário principal e se ambos não são iguais. Se essa verificação for atendida, na linha 48, o usuário em iteração é adicionado à lista de recomendações do usuário principal. Por fim, na linha 52, a lista de recomendações do usuário principal é exibida na tela utilizando o método *printRecommendationsForUser*.

4.3 Exemplos de Execução

Para a realização dos testes do projeto, utilizou-se um arquivo .txt como entrada, o qual é lido pelo programa. Após a leitura, os objetos essenciais para a execução do programa são criados de acordo com as informações fornecidas no arquivo. A seguir, apresentamos um exemplo de arquivo de teste utilizado.

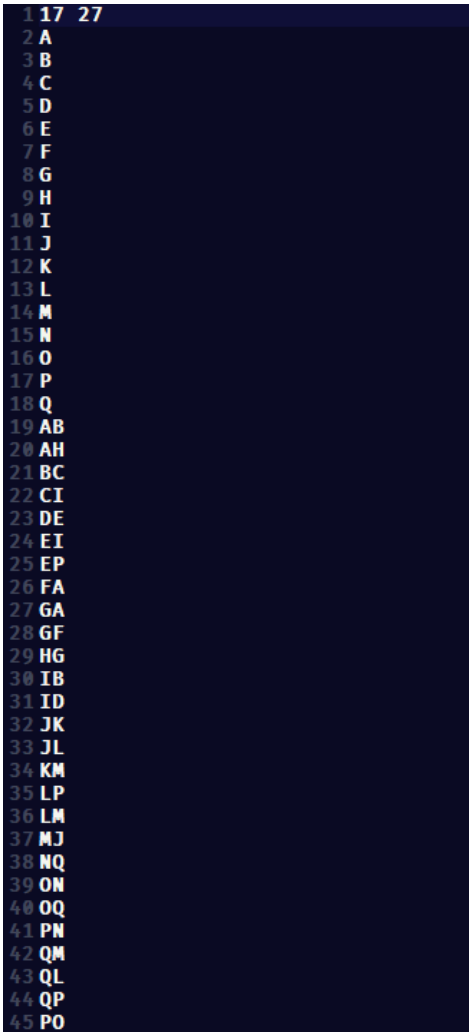

Figura 9. Arquivo de entrada para o programa.
Fonte: Imagem tirada diretamente da *Eclipse IDE*.

Na linha inicial do arquivo de entrada, são informadas a quantidade de vértices e arestas do grafo, correspondendo, respectivamente, à quantidade de perfis de usuários e conexões entre eles na rede social fictícia. Em seguida, são listados os nomes dos perfis (usando letras do alfabeto para simplificar o entendimento). Posteriormente, é detalhado quais perfis cada usuário segue — por exemplo: o perfil 'A' segue o perfil 'B'.

Foi desenvolvida uma classe Main responsável pela execução de um teste do sistema. O código dessa classe é apresentado a seguir.

```java
public class Main {

    public static void main(String[] args) throws FileNotFoundException {
        GraphGenerator graphGenerator = new GraphGenerator();
        SocialNetwork socialNetwork = graphGenerator.generate("C:\\Users\\Diogenes\\eclipse-workspace\\GraphProject/input.txt");

        socialNetwork.printGraph();
        System.out.println();

        socialNetwork.recommenderFriendships();
    }

}
```

Figura 10. Classe usada para testar a execução do programa.
Fonte: Captura de tela da *Eclipse IDE*.

Na linha 7, é instanciado um objeto da classe *GraphGenerator*, responsável pela leitura do arquivo de entrada. Em seguida, na linha 8, um objeto da classe *SocialNetwork* é criado ao chamar o método *generate* do objeto *graphGenerator*, passando como parâmetro uma string que contém o caminho para o arquivo de entrada (*input.txt*). Nesse ponto, o grafo que simula a rede social já está criado. Na linha 10, o método *printGraph* é chamado para exibir o grafo na tela, mostrando os perfis de usuários e as respectivas conexões (ou seja, quem cada usuário segue). Por fim, na linha 13, o método *recommenderFriendships* é invocado para gerar e exibir as recomendações de amizade para cada perfil, com base nas componentes fortemente conectadas a que pertencem.

Após todos esses passos, o programa está pronto para ser executado e testado. Com a leitura do arquivo de entrada concluída e o grafo da rede social devidamente estruturado, as funcionalidades de exibição e recomendação podem ser verificadas em detalhes.

5. Testes e Resultados

Ao executarmos a classe Main da Figura 10, o programa deve ler o arquivo de entrada e imprimir corretamente os perfis, juntamente com os perfis que cada um segue. Além disso, o programa deve identificar e exibir na tela as componentes fortemente conexas da rede social simulada. Logo após, ele gerará e apresentará as devidas recomendações de amizade para cada perfil com base nas componentes fortemente conectadas.

Abaixo, encontra-se uma representação visual do grafo correspondente ao arquivo de entrada da Figura 9.

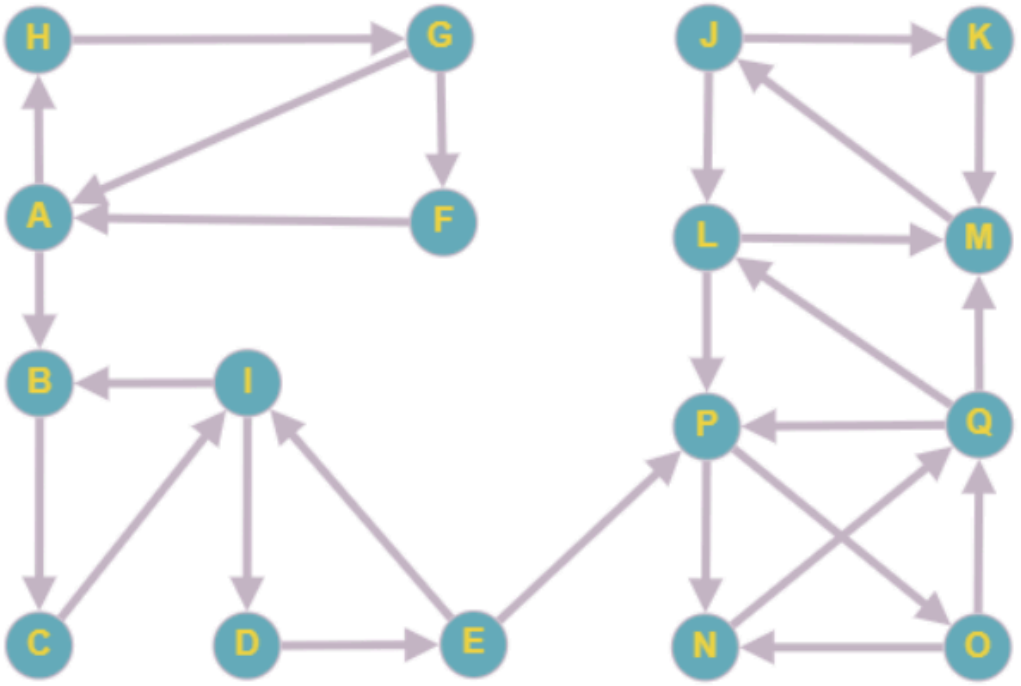

Figura 11. Exemplo visual do grafo de entrada da figura 9.
Fonte: Captura de tela tirada após criação no site graphonline.ru.

Vamos analisar este grafo. Cada seta que sai de um determinado vértice (que, no programa, representa os perfis de usuário) indica que o perfil de entrada é seguido pelo perfil de saída.

Seja S(α) o conjunto de perfis, pertencentes à mesma componente fortemente conexa de α, que o perfil α segue. Por exemplo, para o perfil Q, temos: S(Q) = {L, M, P}.

Pela definição de componente fortemente conexa apresentada na Seção 2, podemos observar que o conjunto de componente fortemente conectadas (SCCs) deste grado de rede social G é dado por SCC(G) = {{A, F, G, H}, {B, C, D, E, I}, {J, K, L, M, N, O, P, Q}}, que são as componentes que o algoritmo deve reconhecer e imprimir na tela para visualização.

Vamos analisar isoladamente a componente א = {A, F, G, H} e indicar o conjuntos R(α), que é o conjunto de recomendações para cada perfil naquela componente.

Para o perfil A, temos que S(A) = {H}; para H, S(H) = {G}; para G, S(G) = {A, F}; e, por fim, para o perfil F, S(F) = {A}.

Os conjuntos R{α} e S{α} devem formar a componente א, ou seja R{α} ∪ S{α} = א. No entanto, esses conjuntos devem ser disjuntos: R{α} ∩ S{α} = ∅, o que significa que as recomendações para um determinado perfil α não podem incluir perfis que ele já segue.

Assim, o conjunto R(α) de recomendações para cada perfil dentro de א é: R(A) = {G, F}, R(F) = {H, G}, R(G) = {H} e R(H) = {A, F}. Essas informações devem ser apresentadas na tela para validar a corretude do código.

A seguir, apresentaremos o resultado obtido após a execução do programa. Como a saída é muito extensa, mostraremos por partes.

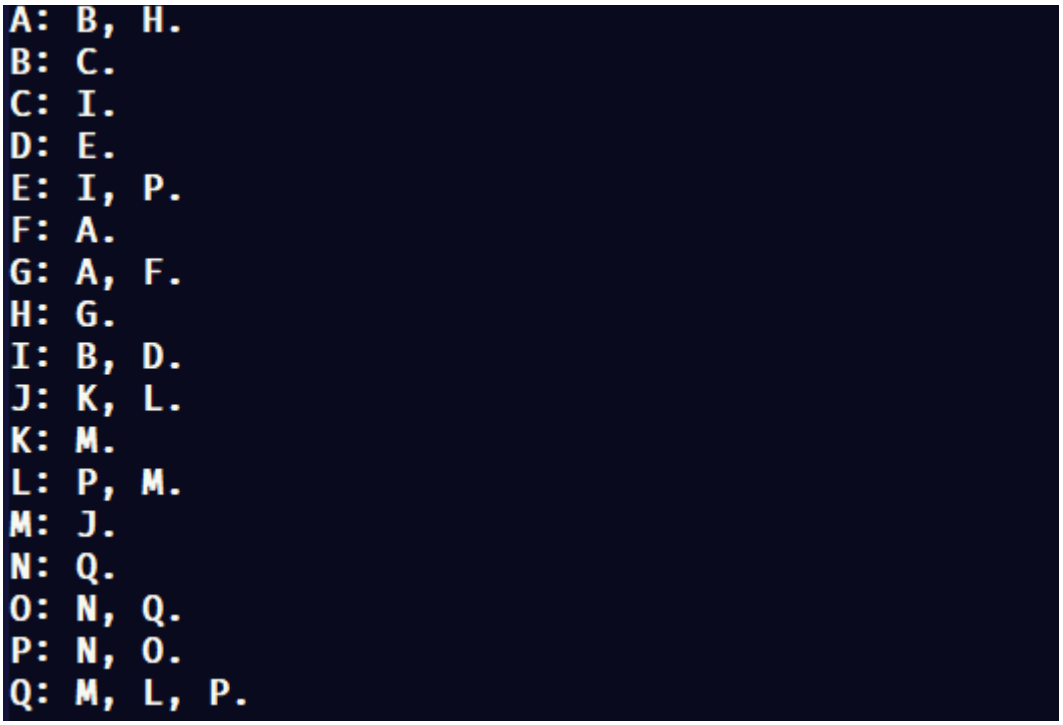

Figura 12. Primeira parte da saída após execução do programa.
Fonte: Captura de tela do terminal da Eclipse IDE.

Esta primeira parte do resultado corresponde à linha 10 da figura 10, onde o método *printGraph* do objeto *socialNetwork* é invocado. A saída mostra de forma clara a listagem de todos os perfis e os perfis que cada um segue. Os resultados estão alinhados com as expectativas e refletem com precisão a representação visual do grafo apresentada na figura 11.

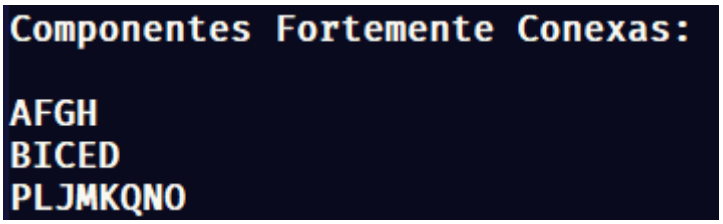

Figura 13. Segunda parte da saída após execução do programa.
Fonte: Captura de tela do terminal da Eclipse IDE.

Nesta segunda parte, são exibidas as componentes fortemente conexas do grafo da rede social, que constituem o cerne do objetivo do programa. Mais uma vez, observamos que a saída do programa é consistente com a representação visual da figura 11, reconhecendo e apresentando de forma clara as componentes definidas como SCC na seção 2.

Por fim, chegamos à parte mais importante da execução do programa: a geração de recomendações para cada perfil de usuário, com base na componente fortemente conexa a que ele pertence. Para fins de verificação, analisaremos apenas a componente que exploramos anteriormente: {A, F, G, H}. Nessa etapa, o programa deverá apresentar as recomendações corretas para cada perfil dessa componente, conforme as regras previamente estabelecidas.

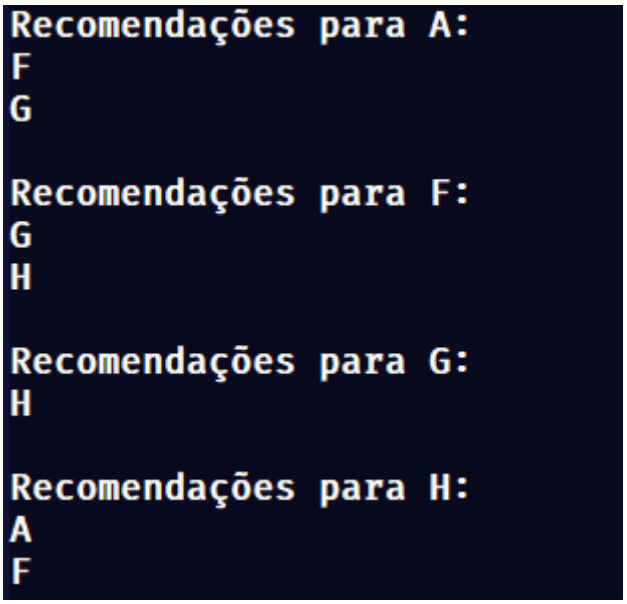

Figura 14. Parte final da saída após execução do programa.
Fonte: Captura de tela do terminal da Eclipse IDE.

Ao compararmos as informações analisadas na figura 11 com a saída do programa, observamos uma clara coerência entre os dados apresentados. Isso nos permite validar a correção do código, confirmando que ele opera conforme o esperado.

6. Conclusão

6.1 Resumo dos Resultados

O programa desenvolvido para simular uma rede social baseada em um grafo direcionado demonstrou resultados satisfatórios ao processar e apresentar as informações de forma coerente. Ao executar a classe Main, o programa conseguiu ler corretamente o arquivo de entrada, exibindo a listagem de todos os perfis de usuário e suas respectivas conexões. Essa etapa inicial, visível na primeira parte da saída, correspondeu fielmente à representação gráfica do grafo.

Em seguida, o programa identificou e apresentou as componentes fortemente conexas (SCCs) da rede social, confirmando a implementação correta desse conceito fundamental. A análise detalhada da componente {A, F, G, H} revelou a capacidade do sistema de gerar recomendações de amizade, respeitando as regras estabelecidas de que as recomendações não devem incluir perfis que um usuário já segue.

Por fim, ao comparar as saídas do programa com as informações extraídas da representação visual do grafo, constatou-se uma consistência entre os dados. Isso valida a eficácia e a correção do código, garantindo que o sistema opera de acordo com as expectativas estabelecidas. Em suma, os resultados obtidos demonstram que o programa cumpre seu objetivo de identificar interações na rede social e gerar recomendações personalizadas de maneira eficiente.

Melhorias Futuras

6.2 Melhorias Futuras

Uma melhoria futura seria a expansão do modelo da rede, que se basearia na inclusão de diversas dimensões sociais e comportamentais para criar um ambiente de simulação mais robusto e realista. Essa expansão poderia envolver a introdução de diferentes tipos de relacionamentos e conexões temporárias, que refletiriam as dinâmicas sociais do mundo real. Ao permitir que os usuários se conectem de várias maneiras, o modelo se tornaria mais representativo das interações humanas e, consequentemente, das recomendações de amizade seriam mais precisas e relevantes.

Além disso, seria benéfico integrar atributos aos perfis dos usuários, como interesses pessoais, localização geográfica e histórico de interações. Esses dados poderiam ser utilizados para aprimorar o sistema de recomendações, sugerindo conexões com base em interesses comuns e localização. Por exemplo, um usuário que se interessa por fotografia poderia receber recomendações de outros usuários que compartilham esse hobby, facilitando a formação de comunidades e interações significativas dentro da rede.

Por fim, a análise de comunidades e grupos dentro da rede social poderia oferecer insights mais profundos sobre a dinâmica social. A detecção de comunidades poderia revelar grupos de usuários que compartilham interesses ou comportamentos semelhantes, permitindo recomendações ainda mais direcionadas. Essa análise poderia ser enriquecida com dados de interações externas, como eventos ou tendências sociais, ampliando o escopo do sistema e possibilitando uma simulação mais realista e interativa.

Com essas melhorias, o modelo de rede não apenas se tornaria mais abrangente, mas também mais eficaz em simular e entender as complexidades das interações sociais,

resultando em uma experiência mais rica e significativa para os usuários.

7. Fontes Consultadas

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: teoria e prática.* Tradução de Gustavo A. A. Ferreira e André C. C. Santos. São Paulo: Editora Campus, 2010.