Universidade Federal de Juiz de Fora

Trabalho Parte 3 Grafos - 2024.3

Documentação

Integrantes:
Felipe Lazzarini Cunha
Luiz Alberto Werneck Silva

17 de março de 2025

Sumário

Capítulo 1

Observações Iniciais

1.1 Do código

Repositório da parte 3 presente em: https://github.com/Muowl/trabalho-grafos-DCC059
Para atender de forma eficiente à demanda da AGMG com nossas bases de dados extraídas do conjunto de dados de Stanford - https://snap.stanford.edu/data/, refatoramos o código no que tange à leitura e implementação das funções anteriormente implementadas nas partes 1 e 2 do trabalho.
Separamos em branches no GitHub para mantermos toda a base desenvolvida ao longo do período, sendo a parte 3 na main e as partes 1 e 2 nas branches ajustes-parte-1 e parte-2 respectivamente.

1.2 Do ambiente de testes

Durante as execuções e testes, por questões de limitação de hardware dos integrantes algumas vezes tivemos que limitar o tamanho dos grafo e eventuais funções para que pudessemos estuda-las. Como melhor explicado no capítulo do código, pelos motivos citados acima, o arquivo grafo.txt foi gerado com auxílio de IA baseado no arquivo web-Google.txt presente nas entradas (mantendo a restrição pedida de ao menos 5000 nós).

1.3 Contribuíções

Durante o desenvolvimento, de forma similar as partes 1 e 2, Felipe ficou responsável pela parte de grafo_lista e suas funções auxiliares como

lista_encadeada.h.
Luiz ficou responsável pela implementação de grafo_matriz e funções relacionadas.

Capítulo 2

Código

2.1 Funções Auxiliares

Durante o desenvolvimento criamos estruturas auxiliares para facilitação do desenvolvimento. No caso do nosso trabalho foram: leitura.cpp, vetor.h e lista_encadeada.h. Além disso criamos uma estrutura para sanitizar os grafos obtidos de Stanford, chamado de sanitizador.cpp Segue nas seções abaixo uma explicação sobre cada estrutura.

2.1.1 leitura.cpp

O arquivo `leitura.cpp` é responsável por fazer a leitura e armazenamento das arestas de um grafo. Ele:

- Ignora as quatro primeiras linhas do arquivo de entrada (que normalmente contêm metadados).
- Lê as arestas (pares de nós) e armazena no `Vetor<std::pair<int, int»`.
- Identifica os nós presentes (armazenando-os em `nos_presentes`) e gera pesos sintéticos para cada aresta (gravados em `arestas_com_peso`).

No final, o arquivo fornece métodos para exibir os nós presentes e verificar rapidamente se um nó está ou não no grafo.
Isso foi necessário para melhorar a eficiência do uso de memória, pelo fato de que antes dessa adição, o grafo que tinha um nó com um id 875230 para ser adicionado à matriz era necessário a mesma ser de ordem 875230, mesmo que não tivesse nós entre esse nó e outro, por exemplo, com id 100000, sendo muito ineficiente no uso de memória.

2.1.2 vetor.h

A classe `Vetor` foi criada para manipular dinamicamente coleções de dados sem depender de bibliotecas padrão como `std::vector`. Principais características:

- Utiliza `malloc`, `realloc` e `free` para gerenciar a memória.
- Possui métodos como `push_back` e `resize`, que aumentam automaticamente a capacidade conforme necessário.
- As construções dos elementos são feitas via *placement new*, permitindo gerenciar explicitamente o ciclo de vida de cada objeto armazenado.

2.1.3 lista_encadeada.h

A estrutura `ListaEncadeada` fornece uma lista encadeada simples. Destaques:

- Possui ponteiros `head` e `tail` para facilitar operações de inserção no final (`push_back`).
- Conta com métodos para limpar (`clear`) e verificar se está vazia (`empty`).
- Oferece `remove_if`, que recebe um predicado para remover elementos que correspondam a uma condição.

2.1.4 sanitizador.cpp

O arquivo `sanitizador.cpp` cuida de "limpar" (ou filtrar) dados de um grafo de entrada:

- Copia as quatro primeiras linhas do arquivo de origem para manter informações iniciais.
- Converte cada aresta (par `(origem, destino)`) em uma representação ordenada (menor nó primeiro).
- Usa `qsort` para ordenar as arestas e escreve somente arestas únicas no arquivo de saída.

Isso é útil para remover duplicidades e manter um conjunto de arestas consistente.

2.2 Funções Principais

2.2.1 GrafoMatriz

A classe `GrafoMatriz` implementa uma representação de grafo utilizando matriz de adjacência. Principais características:

- A estrutura armazena os vértices e arestas em uma matriz bidimensional, onde cada célula `matriz[i][j]` indica a presença ou o peso da aresta entre os vértices `i` e `j`.
- Oferece operações eficientes para verificação de adjacência entre dois vértices (complexidade O(1)).
- Possui métodos para inserção e remoção de arestas com complexidade O(1).
- Implementa algoritmos de busca como BFS (Busca em Largura) e DFS (Busca em Profundidade).
- Suporta algoritmos de caminho mínimo como Dijkstra e Floyd-Warshall, aproveitando a estrutura de matriz para cálculos diretos.

Estrutura Interna

A implementação utiliza:

- Uma matriz de pesos (valores reais ou inteiros) para representar as arestas.
- Um mapeamento entre os IDs originais dos nós e índices internos para otimizar o uso de memória.
- Estruturas auxiliares para armazenar metadados como grau de cada vértice.

Vantagens e Desvantagens

Vantagens:

- Acesso direto às informações de adjacência entre quaisquer vértices em tempo constante.
- Implementação simples para grafos densos.

- Facilidade na implementação de algoritmos que requerem verificação frequente de adjacência.

Desvantagens:

- Uso de memória elevado (O(n^2) para n vértices), tornando-se ineficiente para grafos esparsos.
- Operações de adição ou remoção de vértices podem ser custosas, exigindo realocação da matriz.
- Travessia de adjacências de um vértice específico requer varredura de uma linha inteira da matriz (O(n)).

Algoritmos Implementados

No escopo deste trabalho, o `GrafoMatriz` implementa:

- Cálculo do fecho transitivo direto e indireto.
- Verificação de conexidade e componentes conectados.
- Algoritmos de caminho mínimo para grafos ponderados.
- Detecção de ciclos e circuitos.

Gerenciamento de Grafos Grandes

Para lidar eficientemente com grafos com mais de 5000 nós, o `GrafoMatriz` implementa importantes otimizações:

- **Representação Compacta**: Em vez de alocar uma matriz com tamanho baseado no maior ID de nó (que pode ser muito grande), a implementação cria uma matriz compacta baseada apenas nos nós realmente presentes no grafo.
- **Mapeamento de IDs**: Utiliza um sistema de mapeamento entre os IDs originais dos nós e índices internos compactos. Por exemplo, se o grafo contém apenas os nós com IDs 5, 1000 e 50000, eles são mapeados internamente para os índices 0, 1 e 2, economizando memória significativamente.
- **Redimensionamento Dinâmico**: A matriz é redimensionada conforme necessário através do método `redimensionar_matriz`.

- **Busca Eficiente**: Para manter a eficiência na conversão entre IDs originais e índices compactos, utiliza-se busca binária, garantindo operações de $O(\log n)$.

Essas otimizações permitem que o `GrafoMatriz` trabalhe com grafos grandes sem exigir quantidades proibitivas de memória. Por exemplo, um grafo com nós de IDs muito dispersos (como 0, 10000 e 1000000) que normalmente exigiria uma matriz de 1000001×1000001, é representado usando apenas uma matriz 3×3 mais as estruturas de mapeamento.

Para grafos com 5000 nós efetivos, independentemente do intervalo de IDs originais, a matriz ocupará memória apenas proporcional a esses 5000 nós, resultando em uma redução substancial no uso de memória.

2.2.2 GrafoLista

A classe `GrafoLista` implementa a representação do grafo por meio de listas de adjacência, onde cada vértice mantém uma lista das arestas que partem dele. Esta implementação foi otimizada com um mecanismo de cache matricial:

- `matriz_cache`: Uma matriz que armazena os pesos das arestas para consulta em tempo constante O(1)
- `existencia_cache`: Uma matriz booleana que indica rapidamente se uma aresta existe
- Métodos auxiliares `inicializarCache()` e `limparCache()` gerenciam o ciclo de vida deste cache

Esta abordagem híbrida combina as vantagens de espaço da representação por lista de adjacência (eficiente para grafos esparsos) com o desempenho de acesso O(1) típico de representações matriciais. Quando é necessário verificar se uma aresta existe (`existeAresta()`) ou consultar seu peso (`getPesoAresta()`), o sistema utiliza o cache em vez de percorrer a lista encadeada, reduzindo significativamente o tempo de execução para grafos grandes, especialmente durante algoritmos que fazem múltiplas consultas sobre a existência de arestas, como é comum nos métodos de detecção de comunidades.

Estrutura Interna

A implementação do `GrafoLista` utiliza uma combinação sofisticada de estruturas de dados:

- **Array de listas encadeadas**: Um array de tamanho n (número de vértices), onde cada posição contém uma lista encadeada das arestas que partem daquele vértice.
- **Matriz de cache de pesos**: Uma matriz $n \times n$ que armazena o peso de cada aresta para acesso em tempo constante.
- **Matriz de existência**: Uma matriz booleana $n \times n$ que indica rapidamente se existe uma aresta entre dois vértices.
- **Métodos de gerenciamento de memória**: Funções como `inicializarCache()` e `limparCache()` que controlam o ciclo de vida das estruturas auxiliares.

Para cada aresta (v_1, v_2) com peso p, a implementação mantém não apenas um objeto `Aresta` na lista de adjacência de v_1, mas também atualiza as entradas correspondentes nas matrizes de cache: `matriz_cache[v1][v2] = p` e `existencia_cache[v1][v2] = true`.

Vantagens e Desvantagens

Vantagens:

- **Eficiência em grafos esparsos**: A estrutura base de listas encadeadas ocupa espaço proporcional ao número de arestas (O(|E|)), não ao quadrado do número de vértices.
- **Consultas em tempo constante**: O uso do cache matricial permite verificar a existência e o peso de arestas em O(1), superando a limitação clássica das listas de adjacência.
- **Compromisso adaptativo**: Para grafos com densidade média, o sistema balanceia automaticamente entre eficiência de espaço e tempo.
- **Modificações eficientes**: A estrutura permite inserções e remoções de arestas sem reorganização de estruturas grandes.

Desvantagens:

- **Consumo de memória do cache**: As matrizes de cache ocupam $O(n^2)$ de memória, o que pode ser significativo para grafos muito grandes.
- **Sobrecarga de manutenção**: É necessário manter as listas encadeadas e o cache sincronizados, aumentando o custo das operações de modificação.

- **Complexidade de implementação**: A estrutura híbrida é mais complexa de manter do que abordagens puramente baseadas em listas ou matrizes.
- **Inicialização custosa**: A criação do cache exige $O(n^2)$ operações, o que pode ser um gargalo na inicialização de grafos muito grandes.

Algoritmos Implementados

O `GrafoLista` implementa todos os algoritmos básicos requeridos para o trabalho com grafos em geral e detecção de comunidades em particular:

- **Operações fundamentais**: Adição, remoção e verificação de arestas, todas otimizadas pelo sistema de cache.
- **Carregamento de arquivos**: Método `carregarDoArquivo` que utiliza a classe `leitura` para importar dados de arquivos externos.
- **Suporte à detecção de comunidades**: Fornece as interfaces necessárias para os algoritmos de detecção (Guloso, Randomizado e Relativo), com acesso eficiente às relações de adjacência.
- **Travessias e consultas**: Permite acesso eficiente aos vizinhos de cada vértice, essencial para algoritmos de expansão de comunidades.
- **Impressão e debug**: O método `imprimirGrafo()` fornece representações visíveis da estrutura do grafo.

A implementação das operações básicas (como `adicionarAresta`, `removerAresta`, `existeAresta`, e `getPesoAresta`) é particularmente otimizada graças ao sistema de cache, permitindo que algoritmos complexos como os de detecção de comunidades operem com eficiência em grafos de grande porte.

Conclusão sobre Lista de Adjacências

A implementação do `GrafoLista` demonstra uma abordagem híbrida efetiva que combina a eficiência espacial das listas de adjacência com a eficiência temporal das matrizes de adjacência. O sistema de cache implementado (através de `matriz_cache` e `existencia_cache`) resolve uma limitação tradicional das representações por lista—o tempo linear para verificar a existência de arestas específicas—sem incorrer no custo total de memória de uma representação puramente matricial.

Esta estratégia é particularmente adequada para grafos esparsos de grande porte (como o `web-Google.txt`), onde uma representação puramente matricial seria proibitiva em termos de memória, mas as consultas frequentes de aresta demandadas pelos algoritmos de detecção de comunidade tornariam uma implementação de lista pura ineficiente. A natureza adaptativa da estrutura permitiu processamento eficiente mesmo com grafos contendo milhares de vértices, apoiando diretamente os requisitos do estudo sobre AGMG em redes sociais complexas.

Capítulo 3

Algoritmos de Detecção de Comunidades

A detecção de comunidades foi a abordagem escolhida para resolver o problema da Árvore Geradora Mínima Generalizada (AGMG) proposto no trabalho. Este problema consiste em encontrar uma árvore geradora com custo mínimo que inclua exatamente um vértice de cada grupo/comunidade do grafo. Através dos algoritmos de detecção de comunidades, conseguimos particionar o grafo em grupos coesos, permitindo posteriormente selecionar representantes de cada comunidade para construir a árvore geradora mínima generalizada.

Para implementar esta solução, desenvolvemos três algoritmos distintos de detecção de comunidades: Guloso, Randomizado e Relativo. Cada um desses algoritmos visa identificar grupos de nós fortemente conectados entre si e fracamente conectados com o restante do grafo, mas utilizando estratégias diferentes.

3.1 Algoritmo Guloso

3.1.1 Visão Geral

O algoritmo Guloso implementado busca formar comunidades através de um processo incremental baseado na densidade das conexões. Opera com dois parâmetros principais:

- `limiarDensidade`: Controla a proporção mínima de conexões que um vértice deve ter com a comunidade atual para ser incluído.
- `limiteNos`: Limita o número máximo de nós por comunidade, evitando comunidades excessivamente grandes.

3.1.2 Funcionamento e Implementação

O algoritmo segue uma abordagem direta:

- Inicia com um vértice não atribuído a nenhuma comunidade.
- Expande a comunidade gradativamente, incluindo vértices adjacentes que atendam ao critério de densidade de conexões.
- Continua expandindo até que nenhum vértice adjacente possa ser adicionado à comunidade ou o limite de nós seja atingido.
- Cria uma nova comunidade com outro vértice não atribuído e repete o processo.

Para grafos muito grandes (acima de 10.000 vértices), a implementação adota uma abordagem ultrassegura: seleciona um pequeno conjunto de vértices iniciais e cria comunidades muito limitadas em tamanho para evitar consumo excessivo de recursos.

O método `expandirComunidade` é central na implementação e utiliza um conjunto de vértices já verificados para evitar repetições. Um vértice é adicionado à comunidade apenas se possuir conexões suficientes (determinadas pelo método `deveAdicionarVertice`) com membros já presentes.

3.1.3 Otimizações e Limitações

Para grafos grandes, várias otimizações foram implementadas:

- Limitação direta do número de vértices por comunidade.
- Uso de estruturas de dados eficientes para rastrear vértices já verificados.
- Amostragem de nós quando o grafo possui milhares de vértices, focando nas regiões mais densas.
- Limites rígidos de tempo para expansão de comunidade, interrompendo o processo quando necessário.

A principal limitação do algoritmo é sua tendência a formar comunidades muito grandes em grafos com alta densidade ou muitas arestas, sendo esse o principal motivo da imposição do limite de nós por comunidade.

3.2 Algoritmo Randomizado

3.2.1 Visão Geral

O algoritmo Randomizado utiliza princípios probabilísticos para formar comunidades, introduzindo aleatoriedade controlada no processo. Seus principais parâmetros são:

- `probabilidadeConexao`: Probabilidade base para considerar um vértice conectado a uma comunidade.
- `numComunidadesAlvo`: Número inicial de comunidades a serem formadas.
- `semente`: Valor para inicializar o gerador de números aleatórios, garantindo reprodutibilidade.

3.2.2 Funcionamento e Implementação

O algoritmo opera em duas fases principais:

1. **Inicialização**: Cria um número predefinido de comunidades com vértices escolhidos aleatoriamente como sementes.
2. **Expansão**: Para cada vértice não atribuído, calcula uma probabilidade de pertencer a cada comunidade existente, baseando-se na estrutura de conexões e em um componente aleatório.

A implementação utiliza uma referência ao `leitor` para acessar os nós presentes no grafo, o que permite o sorteio de vértices aleatórios de maneira eficiente, mesmo em grafos esparsos com IDs não sequenciais.

Para grafos muito grandes, o algoritmo utiliza uma abordagem simplificada que cria um número reduzido de comunidades demonstrativas com poucos nós aleatórios, impondo limites rígidos no tamanho e quantidade de comunidades.

3.2.3 Funções Especializadas

A implementação inclui funções especializadas:

- `sortearNoAleatorio`: Seleciona um nó aleatório da lista de nós presentes, utilizando o `leitor` quando disponível.
- `gerarNumeroAleatorio`: Produz valores aleatórios entre 0 e 1 para cálculos probabilísticos.

- `calcularProbabilidade`: Avalia a probabilidade de um vértice se juntar a uma comunidade específica, baseando-se em suas conexões.

3.2.4 Características Probabilísticas

A natureza probabilística do algoritmo permite:

- Escapar de mínimos locais que algoritmos determinísticos como o Guloso podem encontrar.
- Adaptar-se a diferentes estruturas de grafo sem ajustes excessivos de parâmetros.
- Encontrar comunidades com formas não tradicionais que poderiam ser ignoradas por abordagens gulosas.

Contudo, essa aleatoriedade também introduz variabilidade nos resultados: execuções sucessivas com a mesma semente podem produzir partições diferentes, o que pode ser tanto uma vantagem (explorando diferentes soluções) quanto uma desvantagem (menor reproducibilidade).

3.3 Algoritmo Relativo

3.3.1 Visão Geral

O algoritmo Relativo é um meta-algoritmo que refina resultados iniciais obtidos por outro método de detecção de comunidades. Seus parâmetros incluem:

- `tipoAlgoritmoInicial`: Define qual algoritmo usar para inicialização (GULOSO ou RANDOMIZADO).
- `limiarMelhoria`: Limiar mínimo de melhoria na modularidade para aceitar a movimentação de um vértice.
- `maxIteracoes`: Número máximo de iterações do processo de refinamento.
- `leitor`: Referência ao leitor de dados (necessário para algoritmos randomizados).

3.3.2 Processo de Dois Estágios

O algoritmo opera em dois estágios distintos:

1. **Inicialização**: Utiliza outro algoritmo (Guloso ou Randomizado) para criar comunidades iniciais.
2. **Refinamento**: Move iterativamente vértices entre comunidades para maximizar a modularidade global.

Durante a inicialização, o algoritmo cria e executa uma instância do algoritmo escolhido, copiando suas comunidades resultantes. O refinamento então avalia, para cada vértice, o ganho de modularidade ao movê-lo para outras comunidades, executando apenas movimentações que melhorem a modularidade acima do limiar definido.

3.3.3 Adaptações para Grafos Grandes

Para grafos muito grandes (acima de 10.000 vértices), o algoritmo implementa adaptações significativas:

- Omite completamente a fase de refinamento, utilizando apenas o resultado do algoritmo inicial.
- Limita o número de vértices avaliados durante o refinamento (no máximo 1.000).
- Interrompe o processo após poucas iterações mesmo que ainda haja melhorias.
- Utiliza implementações seguras dos algoritmos inicializadores, já adaptadas para grafos grandes.

Essas adaptações são essenciais para viabilizar a execução em grafos como o `web-Google.txt`, com centenas de milhares de vértices.

3.3.4 Cálculo de Ganho de Modularidade

O núcleo do algoritmo está no método `calcularGanhoModularidade`, que avalia o impacto na modularidade global ao mover um vértice da sua comunidade atual para outra. Este cálculo considera:

- A soma dos pesos das conexões internas da comunidade antes e depois da movimentação.

- A soma total dos pesos das arestas que conectam a comunidade ao resto do grafo.
- O grau ponderado do vértice sendo movido.

A implementação utiliza fórmulas otimizadas para calcular incrementalmente o ganho sem recalcular a modularidade total a cada movimentação, o que seria computacionalmente proibitivo.

3.4 Comparação entre os Algoritmos

3.4.1 Desempenho e Escalabilidade

Os três algoritmos apresentam diferentes características de desempenho:

- **Algoritmo Guloso**: Mais rápido e com menor complexidade computacional, adequado para análises iniciais ou grafos de tamanho moderado.
- **Algoritmo Randomizado**: Moderadamente eficiente com boa escalabilidade, especialmente quando seus parâmetros probabilísticos são ajustados para limitar a expansão.
- **Algoritmo Relativo**: Mais computacionalmente intensivo devido à fase de refinamento, mas com resultados superiores em termos de modularidade das comunidades detectadas.

Para grafos extremamente grandes, todos os algoritmos implementam versões "ultrasseguras"que priorizam a viabilidade computacional sobre a precisão dos resultados.

3.4.2 Qualidade das Comunidades

Em termos de qualidade das partições geradas:

- **Algoritmo Guloso**: Produz comunidades compactas e bem definidas, mas pode não capturar estruturas complexas ou comunidades de formatos irregulares.
- **Algoritmo Randomizado**: Mais flexível na detecção de comunidades com estruturas diversas, mas com maior variabilidade nos resultados.
- **Algoritmo Relativo**: Gera as partições de maior qualidade em termos de modularidade, especialmente em grafos menores onde o refinamento completo é viável.

3.4.3 Casos de Uso Ideais

Cada algoritmo tem aplicações preferenciais:

- **Algoritmo Guloso**: Ideal para análises rápidas, grafos médios ou como componente de algoritmos híbridos.
- **Algoritmo Randomizado**: Útil para exploração inicial de estruturas comunitárias ou quando se deseja diversidade nas soluções.
- **Algoritmo Relativo**: Preferível para análises detalhadas onde a qualidade da detecção é prioritária sobre a velocidade, ou quando se tem uma boa inicialização disponível.

Na prática, a escolha do algoritmo depende do tamanho do grafo, dos recursos computacionais disponíveis e dos objetivos específicos da análise de comunidades. Para grafos extraídos do Stanford Network Analysis Project, como o `web-Google.txt`, as versões adaptadas para grafos grandes são essenciais para viabilizar qualquer análise de comunidades.