

Júlio César Gonzaga Ferreira Silva [Pontificia Universidade Católica de Minas Gerais | jcgfsilva@sga.pucminas.br]
Luiz Fernando Antunes da Silva Frassi [Pontificia Universidade Católica de Minas Gerais | lfasfrassi@sga.pucminas.br]
Suzane Lemos de Lima [Pontificia Universidade Católica de Minas Gerais | slima@sga.pucminas.br]
Rafael Castro Guimarães Cristino [Pontificia Universidade Católica de Minas Gerais | rafael.cristino@sga.pucminas.br]
Pedro Augusto Gomes Ferreira de Albuquerque [Pontificia Universidade Católica de Minas Gerais | pagfalbuquerque@sga.pucminas.br]

Department of Computer Science, Pontifical Catholic University of Minas Gerais (PUC Minas), Av. Dom José Gaspar, 500, Coração Eucarístico, Belo Horizonte, MG, 30535-901, Brazil.

Abstract: Grafos desempenham papel crucial na resolução de problemas computacionais em diversas áreas, tais como análise de redes, otimização de rotas e modelagem de sistemas complexos. Este trabalho implementa e compara duas metodologias distintas para a identificação de ciclos em grafos não direcionados: uma baseada em permutações de vértices e outra baseada em busca em profundidade (caminhamento em grafos). Ambas as abordagens foram implementadas utilizando listas de adjacência e avaliadas quanto ao desempenho computacional e precisão dos resultados. Os experimentos realizados evidenciam trade-offs significativos entre essas abordagens, destacando variações de desempenho relacionadas ao tamanho e à estrutura do grafo analisado.

Keywords: Algoritmos em Grafos, Detecção de Ciclos, Grafos Não Direcionados, Caminhamento em Grafos, Complexidade Computacional, CPP

1 Introdução

Algoritmos baseados em grafos desempenham um papel fundamental na resolução de problemas em diversas áreas, como análise de redes sociais, otimização de rotas e modelagem de sistemas biológicos. Um problema clássico nesse domínio é a enumeração de ciclos em grafos não-direcionados, que consiste em identificar todas as sequências fechadas de vértices sem repetições intermediárias. Essa tarefa possui aplicações práticas, como detecção de redundâncias em redes ou análise de conectividade, mas apresenta desafios computacionais à medida que o tamanho do grafo aumenta. Neste trabalho, implementamos e comparamos duas abordagens distintas para resolver esse problema: uma baseada na permutação de vértices e outra utilizando caminhamento no grafo, ambas desenvolvidas em C/C++. Nosso objetivo é avaliar o desempenho dessas soluções em termos de eficiência, considerando diferentes tamanhos de grafos. Este relatório descreve as implementações, os experimentos realizados e os resultados obtidos, além de uma análise comparativa entre os métodos

2 Metodologia do caminhamento

Nesta seção, descrevemos a representação do grafo adotada, os detalhes da implementação do algoritmo baseado em caminhamento.

2.1 Representação do Grafo

O grafo foi representado utilizando uma lista de adjacência implementada na classe `graphVet`. Cada vértice é um objeto da classe `vertex`, que armazena um identificador (`uVertex`), um indicador de visitação (`visitado`), um ponteiro para os vizinhos (`prox`) e um vetor de caminhos (`caminho`) que registra as sequências exploradas. A escolha por listas de adjacência foi motivada por sua eficiência em grafos esparsos, comuns

em aplicações reais, e pela facilidade de acesso aos vizinhos de um vértice, essencial para o algoritmo de caminhamento.

2.2 Implementação do Algoritmo de Caminhamento

O algoritmo implementado para enumerar ciclos é baseado em uma busca em profundidade **DFS** modificada, conforme a função `dfs`. Para cada vértice inicial v do grafo $G=(V,E)$, o algoritmo explora recursivamente os vizinhos por meio da função `deep`, rastreando caminhos e identificando ciclos. O processo funciona da seguinte forma:

- **Exploração Recursiva:** A função `deep` visita os vizinhos de v (obtidos por `ShowProx`) e constrói caminhos no formato $u-v$, evitando revisitar vértices já marcados como visitados ou caminhos duplicados **verificados por ExisteCaminho**. Quando um vizinho é explorado, ele é adicionado ao caminho atual e a busca continua recursivamente.
- **Detecção de Ciclos:** Um ciclo é identificado quando um caminho tem mais de dois vértices $k > 2$ e retorna ao vértice inicial, conforme filtrado em `contarStrings2`. A função `removePermutacoesIguais` elimina ciclos equivalentes (ex.: "0-1-2-0" e "1-2-0-1").
- **Saída:** Os ciclos são armazenados em `ciclo` e exibidos com o número total calculado por `vet.size()`.
- O grafo de teste padrão `SetDefault` constrói um gráfico equivalente ao exemplo do enunciado.

3 Metodologia da permutação

Nesta seção, descrevemos a representação do grafo adotada, os detalhes da implementação do algoritmo baseado em permutação.

3.1 Representação do Grafo

O grafo foi representado utilizando uma matriz de adjacência implementada como um vetor bidimensional de booleanos (**vector<vector<bool>> adj**). Essa escolha foi motivada pela simplicidade de verificar a existência de arestas entre vértices, uma operação frequente no algoritmo de permutação. Cada posição **adj[u][v]** indica se há uma aresta entre os vértices u e v . Para um grafo com V vértices, a matriz possui tamanho $V \times V$, com valores **true** para arestas existentes e **false** caso contrário. Apesar de a matriz de adjacência ocupar $O(v^2)$ em memória, ela foi adequada para os experimentos com grafos pequenos, como o da Figura 1 do enunciado, e permitiu uma implementação direta do algoritmo.

Descrição da Implementação: Algoritmo de Permutação de Vértices O algoritmo proposto para encontrar ciclos baseia-se na geração de subconjuntos de vértices e na verificação de ciclos válidos por meio de permutações. A implementação foi realizada em C++ e segue os seguintes passos:

3.2 Implementação do Algoritmo de Permutação

- **Geração de Subconjuntos:** A função `gerarSubconjuntos` utiliza uma abordagem binária para criar todos os subconjuntos possíveis dos V vértices do grafo. Para n vértices, são gerados $2^n - 1$ subconjuntos (**excluindo o vazio**), mas apenas aqueles com 3 ou mais vértices são considerados, pois ciclos com menos de 3 vértices não existem em grafos simples não-direcionados.
- **Verificação de Ciclos:** Para cada subconjunto, a função `next_permutation` da biblioteca padrão de C++ é utilizada para gerar todas as permutações dos vértices no subconjunto. Uma permutação é considerada um ciclo se: Existe uma aresta entre cada par de vértices consecutivos (**adj[sub[i]][sub[i+1]]=true**). Existe uma aresta entre o último e o primeiro vértice (**adj[sub[sub.size()-1]][sub[0]]=true**).
- **Eliminação de Redundâncias:** Como diferentes permutações podem representar o mesmo ciclo (ex.: $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ e $1 \rightarrow 2 \rightarrow 0 \rightarrow 1$), os vértices de cada ciclo válido são armazenados em um **set<int>** (conjunto ordenado), e os ciclos únicos são mantidos em um **set<set<int>**. Isso garante que cada ciclo seja contado apenas uma vez, independentemente da ordem ou do ponto de início.
- **Saída:** Os ciclos únicos são exibidos no formato de sequência circular, repetindo o primeiro vértice ao final para indicar o fechamento do ciclo (ex.: **0 1 2 0**).

4 Conclusão

Conclusão e reflexão dos dados obtidos

4.1 Grafo direcionado

Com relação aos algoritmos feitos, é perfeitamente possível adaptá-lo para funcionar em um grafo direcionado. No caso

da nossa implementação específica, desenvolvida em C++, não seriam necessárias alterações significativas, já que o grafo não direcionado é interpretado como um caso especial de grafo direcionado. Isso ocorre porque um grafo não direcionado pode ser modelado como um grafo direcionado com arestas bidirecionais (ou seja, para cada aresta u,v , existem (u,v) e (v,u) em E). Nossa implementação já considera a existência de arestas entre vértices adjacentes durante o caminhamento, utilizando uma abordagem que, implicitamente, suporta essa equivalência. Assim, o algoritmo de caminhamento, construído para explorar caminhos e identificar ciclos em grafos não direcionados, mantém sua funcionalidade em grafos direcionados.

4.2 Desempenho

Caminhamento

1. Grafo completo com 4 vértices: O tempo de execução foi de **6070 microssegundos**.
2. Grafo padrão (do exemplo): O tempo de execução foi de **37158 microssegundos**.
3. Grafo completo com 8 vértices: O tempo de execução foi de **49884671 microssegundos**.

Permutação

1. Grafo completo com 4 vértices: O tempo de execução foi de **618 microssegundos**.
2. Grafo padrão (do exemplo): O tempo de execução foi de **15488 microssegundos**.
3. Grafo completo com 8 vértices: O tempo de execução foi de **539602 microssegundos**.

Devido à quantidade de operadores e métodos de ajuste utilizados no caminhamento, ele se torna extremamente lento, dando à permutação uma grande vantagem em termos de velocidade.

Responsabilidades do Grupo

Contribuições dos Autores

Luiz Fernando e **Júlio César** foram responsáveis pelo algoritmo de caminhamento.

Pedro Augusto foi responsável pelo algoritmo de permutação.

Rafael Castro e **Suzane Lemos** descreveram os detalhes da implementação, os experimentos e os resultados obtidos.