A B A K Ó S
PUC Minas Instituto de Ciências Exatas e Informática

Trabalho Prático I*

Model - Magazine Abakós - ICEI - PUC Minas

Fernanda Mendes Gomes[1]
Hugo Portes Araujo Cattoni[2]
Pedro Olyntho Carvalho Alves[3]

Resumo

Este artigo apresenta a documentação do Primeiro Trabalho Prático da disciplina Teoria dos Grafos e Computabilidade, referente à identificação de componentes biconexos em grafos nao direcionados. Os componentes biconexos, ou blocos do grafo, são os subgrafos maximais do Grafo G biconexos em vértices ou isomorfos a K_2. Neste trabalho, foram desenvolvidos três métodos para identificação de blocos. O primeiro, realiza o reconhecimento a partir da identificação de dois caminhos internamente disjuntos entre todos os pares de vértice do bloco. Já o segundo, identificando as articulações do grafo e testando a conectividade após a remoção de cada vértice. Por fim, o terceiro, é uma implementação do algoritmo proposto por Robert Tarjan em 1972. Para comparar as três execuções, foram realizados experimentos em grafos não direcionados gerados aleatoriamente com 100, 1.000, 10.000 e 100.000 vértices respectivamente.

Palavras-chave: Grafos. Grafos Não Direcionados. Tarjan. Componentes Biconexos. Teoria dos Grafos e Computabilidade. Articulações. Caminhos Disjuntos. Blocos. LaTeX.

*Artigo apresentado como documentação ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais para disciplina Teoria dos Grafos e Computabilidade.

[1], E-mail:fernandamendesgomes@gmail.com
Instituto de Ciências Exatas e de Informática da PUC Minas, Brasil.
[2], E-mail:hcattoni@sga.pucminas.br
Instituto de Ciências Exatas e de Informática da PUC Minas, Brasil.
[3], E-mail:pedro.olyntho@sga.pucminas.br
Instituto de Ciências Exatas e de Informática da PUC Minas, Brasil.

1 INTRODUÇÃO

Componentes biconexos em grafos são subgrafos conexos de um grafo não direcionado que não podem ser divididos em dois ou mais componentes conexos a partir da remoção de uma aresta ou um vértice. Um grafo G é caracterizado como biconexo se e somente se existir dois caminhos sem vértices internos em comum entre cada par de vértices do grafo. O problema de identificar esse tipo de componente apresenta uma série de aplicações como: identificar áreas críticas em rede de computadores e auxiliar o planejamento de rotas de transporte. No presente trabalho, serão exploradas três maneiras de encontrar os blocos de um grafo: verificando caminhos disjuntos entre vértices, identificando as articulações e testando a conectividade após a remoção de vértices e, finalmente, aplicando o método desenvolvido por TARJAN.

2 IMPLEMENTAÇÃO

Esta seção apresenta a implementação do trabalho prático: descrição da classe Grafo utilizada nos projetos e dos três algoritmos desenvolvidos para encontrar os componentes biconexos de um grafo não direcionado. Vale ressaltar que, durante o desenvolvimento, o ambiente Visual Studio Code foi empregado em conjunto com a linguagem Java. Ademais, foram criados três projetos que implementam cada um dos métodos desenvolvidos para a resolução do problema. A classe Grafo e as operações implementadas nos três projetos foram escritas de forma a serem executadas em uma única Thread de execução.

2.1 Grafo: Representação e Considerações Gerais

As soluções utilizando Tarjan e Articulações têm em comum a classe Grafo, a qual é a principal do projeto. Na representação do grafo, optou-se pela utilização da estrutura de lista de adjacência, tendo em vista a facilidade de busca de vizinhos e de adição e remoção de arestas. A lista foi implementada como um objeto do tipo *Map<Integer, List<Integer»*, onde a chave do mapa corresponde ao número do vértice, enquanto o valor é uma lista de inteiros que contém os vértices adjacentes ao vértice correspondente à chave.

Já a solução que encontra os blocos por meio de caminhos disjuntos, não utiliza a classe Grafo. A lista de adjâcencia do grafo, estrutura utilizada no método, é uma propriedade da classe DisjointPaths. Os caminhos disjuntos do grafo são armazenados em um objeto do tipo *List<List<Integer»*, onde o primeiro item de cada lista interna contém o vértice de origem da busca, e o último o vértice de destino. Os outros itens da lista interna são os vértices presentes no caminho.

2.2 Algoritmos

Esta subseção introduz os algoritmos desenvolvidos para a resolução do problema de encontrar blocos em grafos não direcionados. Na primeira solução, foram utilizados caminhos disjuntos para encontrar os componentes biconexos do grafo. Já a segunda, parte da identificação das articulações. Por fim, a última abordagem é uma implementação do algoritmo de Tarjan para solucionar o mesmmo problema.

2.2.1 Caminhos Disjuntos

Na solução, a análise dos caminhos disjuntos para cada par de vértices do grafo foi utilizada para encontrar os blocos do grafo. Para qualquer par de vértices pertencentes ao bloco, existe pelo menos dois caminhos distintos que os conectam.

Inicialmente, os caminhos disjuntos para todo par de vértice são identificados e são armazenados em uma estrutura do tipo *List<List<Integer»*. Posteriormente, ocorre uma iteração na lista em que estão armazenadas as listas com os caminhos, verificando se existem mais de um caminho entre o vértice de destino e origem. Caso existam, os vértices contidos no caminho são adicionados a um bloco.

As pontes do grafo são identificadas quando existe somente um caminho do vértice origem para o destino, e esse caminho contém apenas uma aresta.

2.2.2 Identificação de Articulações

Nesta solução, as articulações, vértices cuja remoção desconecta o grafo ou aumenta o número de componentes conectados do grafo, foram utilizadas para encontrar os componentes biconexos.

Inicialmente, as articulações do grafo são identificadas por meio da remoção de cada vértice e posterior checagem do número de componentes após a remoção. Se após a remoção do vértice o grafo tem seu número de componentes aumentado, o vértice em questão é uma articulação.

Em seguida, é criada um clone do grafo sem seus pontos de articulação. A partir dessa cópia, são identificados os componentes conexos resultantes. Em seguida, todas as arestas das articulações são testadas para identificar a qual componente cada uma delas pertence, o que permite encontrar os blocos do grafo

Como este método não foi capaz de lidar com o caso das pontes, em que os vértices que as compõem formam um componente, elas foram identificadas usando o algoritmo proposto por Tarjan.

2.2.3 Método de Tarjan (1972)

O algoritmo de Tarjan desenvolvido utilizou além da classe Grafo, a classe Aresta, que representa as arestas de um grafo. Nessa classe, são armazenados como inteiros a origem e o destino de cada objeto do tipo Aresta. Além disso, a implementação de Tarjan foi feita utilizando dois métodos implementados na classe Grafo: *encontrarComponentesBiconexos* e *encontrarComponentesBiconexosAux*. O primeiro é iniciado criando um contador num e três estruturas de dados: um mapa que armazena o número de cada vértice, outro mapa que armazena o menor número que pode ser alcançado por um vértice e uma pilha para armazenar as arestas do grafo. Em seguida, o método principal percorre cada vértice do grafo, se algum vértice ainda não foi visitado, o método auxiliar é chamado.

No método auxiliar, um número e um valor inicial de lowpt, ou menor número alcançado pelo vértice, são atribuídos e o contador é incrementado. Em seguida, a lista de vizinhos do vértice é percorrida, se algum vizinho não tiver seu número atribuído, é feita uma chamada recursiva do método *encontrarComponentesBiconexosAux* , onde o vizinho é o vértice atual e o vértice original é pai desse vizinho. A aresta entre o vértice original e o vizinho é empilhada.

Quando a chamada recursiva é concluída, o valor de lowpt é atualizado para o vértice original e os valores de lowpt são comparados entre o vértice vizinho e o original. Se o valor original for maior ou igual ao do vizinho, o vértice original é definido como uma articulação e é criado um novo componente biconexo do grafo. Posteriormente, as arestas são desempilhadas e os vértices de origem e destino de cada aresta são adicionados ao componente até que a aresta entre os vértices original e vizinho seja encontrada. Nesse momento, o componente é adicionado à lista de componentes biconexos.

Finalmente, o método auxiliar é chamado novamente para cada vizinho que ainda não tenha um número definido. Ao final de todas as chamadas recursivas, a lista de componentes biconexos é retornada, ou seja, todos os componentes foram encontrados.

3 ANÁLISE DOS RESULTADOS

Nesta seção são descritos os testes realizados utilizando cada uma das abordagens e seus resultados. Além disso, o desempenho das soluções propostas será discutido e comparado, levando em consideração o tempo para obter os componentes biconexos do grafo em cada uma delas.

3.1 Testes Realizados

Para testar os algoritmos, foram gerados grafos não direcionados aleatórios com 100, 1.000, 4.000 e 10.000 vértices. A quantidade de arestas nos grafos resultantes corresponde a

2, 5, 10 e 20 vezes o número de vértices de tal grafo. Dada a necessidade de calcular o tempo médio de execução de cada algoritmo, todos os testes foram realizados três vezes para cada dimensão pré determinada para o grafo.

Os testes foram realizados em um processador AMD Ryzen 5 5600x. No caso de algum dos testes não ser concluído em até uma hora, a execução é abortada e o resultado do teste é definido como *timeout (t.out)*.

3.2 Análise Comparativa dos Resultados

Ao analisar as tabelas 1 e 2, nota-se que em grafos com poucos vértices qualquer uma das soluções pode ser implementada, eis que todos os tempos são relativamente baixos. Entretanto, ao processar grafos com mais de 1.000 vértices, as diferenças são mais explícitas, sendo os tempos da solução que utiliza caminhos disjuntos significativamente maior do que o das outras duas. Por outro lado, os tempos obtidos pelo algoritmo de Tarjan, seja de 1.000 ou de 4.00 vértices, são bem menores, dada a baixa a complexidade do algoritmo.

Para 4.000 vértices, a solução de caminhos disjuntos ultrapassou o limite de execução de uma hora, configurando timeout. Já o método que utiliza as articulações e o de Tarjan foram capazes de processar até 4.000 vértices com número de arestas igual a 20 vezes o número de vértices. A primeira, a partir de 10.000 vértices, alcançou timeout; enquanto a segunda gerou stack overflow. Entende-se que a solução de Tarjan teve essa limitação pois utiliza uma abordagem recursiva.

Finalmente, cumpre ressaltar que o desempenho das soluções piora à medida que o número de vértices e arestas aumenta. Para grafos com 100 vértices, todos os algoritmos apresentam tempos razoáveis, enquanto que para grafos maiores (a partir de 1.000 vértices), o tempo de execução torna-se significativo e, para grafos com 10.000 ou mais vértices, os algoritmos não conseguem terminar devido a problemas de timeout ou estouro de pilha.

Tabela 1 – Tempo em Milissegundos dos Testes Realizados

Vértices	Arestas	Caminhos Disjuntos			Articulações			Tarjan		
100	**2**	198	198	199	28	29	27	32	30	37
100	**5**	165	167	172	29	30	29	33	31	32
100	**10**	185	198	176	36	36	34	36	33	34
100	**20**	232	226	217	51	50	51	39	35	38
1.000	**2**	483282	559836	513282	2917	2956	2817	145	128	127
1.000	**5**	532456	327823	476456	5605	5751	5791	114	113	113
1.000	**10**	249666	249393	247345	10404	10958	11000	122	122	121
1.000	**20**	228305	211380	238674	20914	20602	20994	154	137	138
4.000	**2**	T.OUT	T.OUT	T.OUT	186584	184803	179301	429	427	428
4.000	**5**	T.OUT	T.OUT	T.OUT	634829	586768	535805	372	349	374
4.000	**10**	T.OUT	T.OUT	T.OUT	1217473	1214484	1114484	428	420	436
4.000	**20**	T.OUT	T.OUT	T.OUT	2303798	2347655	2248550	442	546	577
10.000	**2**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
10.000	**5**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
10.000	**10**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
10.000	**20**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
100.000	**2**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
100.000	**5**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
100.000	**10**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.
100.000	**20**	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	T.OUT	S.O.	S.O.	S.O.

Tabela 2 – Tempo Médio em Milissegundos dos Testes Realizados

Vértices	Arestas	Caminhos Disjuntos	Articulações	Tarjan
100	**2**	198,3333333	28	33
100	**5**	167	29	32
100	**10**	185	36	34
100	**20**	226	51	38
1.000	**2**	513282	2917	128
1.000	**5**	476456	5751	113
1.000	**10**	249393	10958	122
1.000	**20**	228305	20914	138
4.000	**2**	TIMEOUT	184803	428
4.000	**5**	TIMEOUT	586768	372
4.000	**10**	TIMEOUT	1214484	428
4.000	**20**	TIMEOUT	2303798	546
10.000	**2**	TIMEOUT	TIMEOUT	STACK OVERFLOW
10.000	**5**	TIMEOUT	TIMEOUT	STACK OVERFLOW
10.000	**10**	TIMEOUT	TIMEOUT	STACK OVERFLOW
10.000	**20**	TIMEOUT	TIMEOUT	STACK OVERFLOW
100.000	**2**	TIMEOUT	TIMEOUT	STACK OVERFLOW
100.000	**5**	TIMEOUT	TIMEOUT	STACK OVERFLOW
100.000	**10**	TIMEOUT	TIMEOUT	STACK OVERFLOW
100.000	**20**	TIMEOUT	TIMEOUT	STACK OVERFLOW

4 CONCLUSÃO

Considerando a relevância da identificação de componentes biconexos em grafos, o presente estudo examinou três abordagens potenciais: a utilização de caminhos disjuntos, o emprego de articulações e, por fim, a implementação do algoritmo proposto por Robert Tarjan em 1972, todas utilizando uma única Thread de execução.

Com base nos testes realizados em grafos de diferentes tamanhos e densidades, concluiu-se que a solução proposta por Robert Tarjan é a mais eficiente para encontrar componentes biconexos em grafos, especialmente em grafos com grande quantidade de vértices e arestas. Já a solução de caminhos disjuntos foi a menos eficiente, apresentando tempos consideravelmente maiores e sendo capaz de processar somente 1.000 vértices em até uma hora. Entretanto, destaca-se que tanto a solução que utiliza articulações quanto a de Robert Tarjan apresentaram limitações em grafos com número de vértices superior a 4.000. Enquanto a primeira alcançou timeout por conta da sua complexidade, a segunda, sendo uma abordagem recursiva, gerou stack overflow.

Referências

TARJAN, Robert. Depth-first search and linear graph algorithms. **SIAM Journal on Computing**, v. 1, n. 2, p. 146–160, 1972. Disponível em: <https://doi.org/10.1137/0201010>.