Identificação de Pontes em Grafos e Caminho Euleriano

Ana Fernanda Souza Cancado ‖ anafcancado8@gmail.com

Arthur de Sá Braz de Matos ‖ arthursbmatos9@gmail.com

Gabriel Praes Bernardes Nunes ‖ gabriel.praes@gmail.com

Guilherme Otávio de Oliveira ‖ gooliveira@sga.pucminas.br

Júlia Pinheiro Roque ‖ juliapinheiroroque960@gmail.com

PUC Minas

March 9, 2025

Resumo

Este relatório descreve a implementação e análise de dois algoritmos para identificação de pontes em grafos: o método Naive [1] baseado em testes de conectividade e o algoritmo de Tarjan [3]. Além disso, implementamos o Algoritmo de Fleury [2] para encontrar um caminho euleriano em um grafo euleriano. Os tempos computacionais das abordagens foram comparados para grafos de diferentes tamanhos.

1 Introdução

Neste trabalho, é abordado o problema da identificação de pontes e a aplicação do Algoritmo de Fleury para encontrar caminhos e ciclos eulerianos.

Uma ponte é uma aresta que, se removida, aumenta o número de componentes conexos do grafo. Identificar pontes é importante em várias aplicações práticas, um exemplo, é no contexto de redes de computadores, em que ao indetificar essa aresta que faria parte da rede ficar desconectada, é possível buscar maneiras de minimizar o impacto de falhas.

O Algoritmo de Fleury é utilizado para encontrar caminhos ou ciclos eulerianos. Um caminho euleriano percorre todas as arestas do grafo uma única vez sem repetir nenhuma aresta. Para ele existir, é necessário que o grafo seja conexo e tenha todos os vértices com grau par ou exatamente 2 de grau ímpar. Já um ciclo euleriano é um caminho euleriano que começa e termina no mesmo vértice, sem repetir aresta. Para isso, o grafo precisa ser conexo e todos os vértices precisam ter grau par.

A presença de uma ponte pode afetar o encontro de um caminho euleriano, já que se ela for removida, o grafo se tornará desconexo, o que impossibilita a existência de um caminho euleriano. Por isso, o algoritmo de Fleury evita a remoção dessas arestas.

2 Metodologia

2.1 Identificação de Pontes

Foram implementados dois métodos para identificação de pontes:

- **Método Naive**: Remove-se cada aresta e testa-se a conectividade do grafo.
  - `isConnected()` - percorre os vértices e realiza uma busca em profundidade (DFS), marcando os vértices visitados. Se algum vértice com vizinhos não for visitado, a função retorna `false`. Caso contrário, o grafo é considerado conexo e retorna `true`.
  - `isBridgeNaive(int u, int v)` - remove temporariamente a aresta (u, v) e verifica a conectividade.

- `findBridgesNaive()` - encontra todas as pontes no grafo. Para isso, percorre todas as arestas e utiliza `isBridgeNaive()` para identificar as pontes. As pontes encontradas são armazenadas em um vetor e retornadas ao final.

- **Algoritmo de Tarjan**: Baseado em DFS e *low-link values*.
  - `bridgeUtil()` - Realiza a busca em profundidade (DFS) e é responsável por calcular os valores `disc` e `low` para cada vértice.
    - `disc[u]`: Representa o tempo de descoberta do vértice u.
    - `low[u]`: Representa o menor tempo de descoberta alcançável a partir do vértice u.

    O algoritmo percorre os vértices adjacentes v de u:
    - Se v não foi visitado, o algoritmo realiza uma chamada recursiva para v e, após isso, atualiza `low[u]` com o valor mínimo entre `low[u]` e `low[v]`.
    - Se `low[v]` for maior que `disc[u]`, então a aresta (u, v) é uma ponte, pois não há outro caminho para v que não passe pela aresta (u, v).
    - Se v já foi visitado e não é o pai de u, então o algoritmo atualiza `low[u]` com o menor valor entre `low[u]` e `disc[v]`.
  - `findBridgesTarjan()` - Inicializa as estruturas necessárias e executa `bridgeUtil()`.

2.2 Análise de Complexidade

A abordagem Naive para encontrar pontes em um grafo percorre todas as arestas e, para cada uma, verifica se sua remoção desconecta o grafo. Isso é feito removendo a aresta, verificando a conectividade com uma busca em profundidade (DFS) e, em seguida, restaurando a aresta. A complexidade dessa solução é $O(V \times E(V + E))$, pois para cada uma das E arestas, executamos uma verificação de conectividade que custa $O(V + E)$.

O algoritmo de Tarjan para encontrar *pontes* em um grafo não direcionado utiliza uma busca em profundidade (DFS) modificada para calcular os menores tempos de descoberta acessíveis para cada vértice. A complexidade do algoritmo é $O(V + E)$, pois cada vértice e aresta é visitado no máximo duas vezes. Isso torna o método eficiente, mesmo para grafos grandes e esparsos.

2.3 Caminho Euleriano

A presença de um Caminho Euleriano pode influenciar significativamente a identificação de pontes em um grafo. Em grafos com Ciclo Euleriano, onde não existem pontes, os algoritmos de identificação de pontes retornam um conjunto vazio.

O algoritmo de Fleury é um método para encontrar um Caminho Euleriano em um grafo, garantindo que cada aresta seja percorrida apenas uma vez. A implementação do método `Graph::fleury` realiza os seguintes passos:

1. Verifica se existe um Caminho Euleriano: Percorre os vértices e conta quantos possuem grau ímpar. Se existirem mais de dois, o grafo não possui Caminho Euleriano, e é retornado um caminho vazio.

2. Cria uma Cópia do Grafo: Para não modificar o grafo original, ele é copiado.

3. Determina o Vértice de Início: Se existir vértices de grau ímpar, o caminho inicia em um deles.

4. Percorre as Arestas Evitando Pontes: Se existirem várias opções, ele evita escolher uma ponte para que o grafo não se desconecte antes da conclusão do percurso (pode ser feito utilizando o método Naive ou o de Tarjan).

5. Remove a Aresta e Continua o Caminho: A aresta escolhida é removida e avança para o próximo vértice até que todas as arestas tenham sido percorridas e o caminho tenha sido finalizado.

3 Experimentos e Resultados

Testamos os algoritmos em grafos aleatórios de tamanhos variados. A tabela abaixo apresenta os tempos médios obtidos.

Tamanho do Grafo	Método Naive (ms)	Tarjan (ms)
100	22	1
1000	17367	101
10000	1527808	1434
100000	indeterminado	indeterminado

Table 1: Tempos médios de execução.

Com base nos resultados obtidos em função do número de vértices do grafo, observa-se que, mesmo para grafos menores (100 vértices), o algoritmo de Tarjan já apresenta uma vantagem significativa em termos de tempo de execução. Como ilustrado na Tabela 1, para um grafo com 100 vértices, o método Naíve leva 22 ms para concluir a execução, enquanto o algoritmo de Tarjan executa em apenas 1 ms. Essa diferença se torna ainda mais expressiva quando aumentamos consideravelmente a quantidade de vértices do grafo testado. Podemos verificar esse comportamento analisando a razão entre os tempos de execução medidos. Para 1.000 vértices, a relação entre os tempos dos dois algoritmos é aproximadamente:

$$\frac{17367}{101} \approx 172$$

Esse fator de crescimento confirma a expectativa teórica baseada na ordem de complexidade dos algoritmos. Enquanto o algoritmo de Tarjan possui complexidade O(V+E), garantindo um crescimento próximo ao linear, o método Naíve apresenta complexidade O(V×(V+E)), o que leva a um crescimento quadrático no pior caso.

À medida que o número de vértices aumenta, esse comportamento se torna ainda mais evidente. Para 10.000, o aumento é de 8697,19

Para 100.000 vértices, o método Naíve torna-se impraticável, com tempo de execução indeterminados devido à escalabilidade inadequada. O algoritmo de Tarjan, por outro lado, mantém um tempo de execução viável para 10.000 vértices (1,4 segundos), mas também se torna inviável para 100.000 vértices. Isso sugere que, para problemas em larga escala, mesmo algoritmos otimizados podem necessitar de abordagens ainda mais eficientes, como paralelização ou heurísticas especializadas.

Esses resultados demonstram a superioridade do algoritmo de Tarjan para a identificação de pontes em grafos, especialmente em cenários com um grande número de vértices e arestas, onde a eficiência do método se torna um fator crítico para a viabilidade da análise.

4 Conclusão

Os experimentos demonstram a superioridade do algoritmo de Tarjan em relação ao Naive para a identificação de pontes e do algoritmo de Fleury. Através dos testes em grafos de diferentes tamanhos, o método Naive torna-se impraticável para grafos com um grande número de vértices. O algoritmo de Tarjan manteve tempos de execução viáveis até grafos com 10.000 vértices, embora também tenha se tornado inviável para grafos ainda maiores.

A Tabela 1 evidencia essa diferença de desempenho, mostrando que, para 100 vértices, o algoritmo de Tarjan já é significativamente melhor que o Naive, sendo aproximadamente 22 vezes mais rápido. Essa vantagem cresce exponencialmente à medida que o número de vértices aumenta, devido a diferença de complexidades dos métodos: enquanto Tarjan é de $O(V+E)$, o Naive tem complexidade $O(V \times (V+E))$.

O algoritmo de Tarjan manteve um desempenho aceitável, executando em 1,4 segundos para 10.000 vértices, mas também se tornou inviável para 100.000 vértices. Dessa forma, concluímos que o algoritmo de Tarjan é a melhor escolha para a execução do algoritmo de Fleury em grafos de médio porte, garantindo um tempo de execução significativamente menor em comparação ao método Naive.

Responsabilidades da Equipe

- Ana Fernanda: Implementação do método Naive.
- Arthur: Implementação do algoritmo de Tarjan.
- Gabriel: Implementação do Algoritmo de Fleury.
- Guilherme, Júlia e Ana Fernanda: Escrita do relatório.

- Guilherme e Júlia: Análise de complexidade e análise dos resultados.

References

[1] A strong-connectivity algorithm and its applications in data flow analysis. *Computers Mathematics with Applications*, 7(1):67–72, 1981.

[2] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[3] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.