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

Métodos para obtenção da árvore geradora mínima*

Model - Magazine Abakós - ICEI - PUC Minas

Fabio Franco de Azevedo[1]
Pedro Rafael Madeira Vasconcelos[2]
Thiago Teixeira Oliveira[3]

Resumo

Este documento tem como objetivo resolver o problema da obtenção da árvore geradora mínima em grafos direcionados ponderados por meio de dois métodos desenvolvidos por nós. Utilizando a linguagem Java, implementamos esses métodos e conduzimos uma análise de seu desempenho e comportamento. O programa permite ao usuário gerar um grafo aleatório especificando o número de vértices ou abrir um arquivo de texto contendo um grafo previamente salvo no diretório onde o código está localizado. O método 1(verifica a existência de dois caminhos internamente disjuntos (ou um ciclo) entre cada par de vértices no bloco. Essa verificação é crucial para identificar estruturas fundamentais no grafo que influenciam na formação da árvore geradora mínima), método 2 (Identifica articulações testando a conectividade após a remoção de cada vértice. Essa abordagem permite detectar pontos de articulação que, quando removidos, podem resultar em uma desconexão do grafo ou na formação de componentes conexos menores), além desses métodos, o método de Tarjan também foi utilizado para resolver o problema da árvore geradora mínima em grafos direcionados ponderados. Ele é um algoritmo clássico que encontra componentes fortemente conectados em um grafo direcionado e pode ser adaptado para encontrar a árvore geradora mínima.

Palavras-chave: Grafos Direcionados, Tarjan, Arborescência Geradora Mínima, Árvore Geradora Mínima, Pesquisa, Programa.

*Artigo apresentado à Revista Abakos

[1]Bacharel em Ciência da Computação, Brasil– fabio.franco@sga.pucminas.br
[2]Bacharel em Ciência da Computação, Brasil– pedro.madeira@sga.pucminas.br
[3]Bacharel em Ciência da Computação, Brasil– thiago.oliveira.1205087@sga.pucminas.br

Abstract

This document aims to address the problem of obtaining the Minimum Spanning Tree (MST) in weighted directed graphs through two methods we have developed. Using the Java programming language, we implemented these methods and conducted an analysis of their performance and behavior. The program allows the user to generate a random graph by specifying the number of vertices or to open a text file containing a previously saved graph in the directory where the code is located. Method 1 checks for the existence of two internally disjoint paths (or a cycle) between each pair of vertices in the block. This verification is crucial for identifying fundamental structures in the graph that influence the formation of the minimum spanning tree. Method 2 identifies articulation points by testing connectivity after the removal of each vertex. This approach allows for the detection of articulation points that, when removed, can result in a disconnection of the graph or the formation of smaller connected components. In addition to these methods, Tarjan's method was also used to solve the problem of the minimum spanning tree in weighted directed graphs. It is a classic algorithm that finds strongly connected components in a directed graph and can be adapted to find the minimum spanning tree. This document aims to solve the problem of obtaining the Minimum Spanning Arborescence (MST) in weighted directed graphs using three methods, evaluating the performance and behavior of each. Java was used to develop and implement the methods proposed by Edmonds (1967), Tarjan (1977), and Gabow et.al (1986) in the version by Böther et al (2023), all related to the mentioned topic. In total, eight classes were implemented, including Main, Edge, Graph, Edmonds, Tarjan, Gabow, FastestSpeedrun, and Girg, in addition to some Java classes like "ArrayList". The Main class contains all the functions aimed at solving the proposed problem. The program allows the user to generate a random graph by specifying the number of vertices or open a file containing a graph in txt format previously saved in the same directory where the code is located.

Keywords: Directed Graphs, Tarjan, Minimum Spanning Arborescence, Research, Program.

1 INTRODUÇÃO

A busca pela Árvore Geradora Mínima é um campo com uma variedade de estudos e soluções disponíveis. Sua definição envolve encontrar, em um grafo não direcionado com N vértices, o conjunto de (N-1) arestas de menor custo que conectam todos os N vértices do grafo, resultando na árvore geradora com a menor soma dos pesos das arestas. Embora os métodos mais conhecidos para resolver esse problema sejam o algoritmo de Kruskal e o de Prim, em muitos casos do mundo real, apenas um grafo direcionado pode resolver um problema específico, levantando a necessidade de obter a Árvore Geradora Mínima para um grafo direcionado.

A Árvore Geradora Mínima é definida como o menor conjunto de arestas, com o menor peso possível, que partindo de um vértice raiz consegue alcançar todos os outros vértices em um grafo direcionado. Este artigo foca em encontrar a Árvore Geradora Mínima dentro de um grafo direcionado, a partir de uma raiz previamente determinada.

O algoritmo para obter a Árvore Geradora Mínima foi descoberto de forma independente por Edmonds, Chu e Bock, com uma complexidade de O(nm). No entanto, essa complexidade pode se tornar cúbica dependendo da densidade do grafo. Isso incentivou o desenvolvimento de melhorias por Tarjan e Gabow, entre outros, que refinaram esse algoritmo. A versão de Tarjan alcançou uma complexidade de O(min(n^2, m log n)), enquanto a de Gabow et al. chegou a O(n log n + m).

Este estudo fornece descrições e implementações de cada um dos três algoritmos, assim como seus resultados. Os códigos implementados estão acessíveis em um repositório hospedado no >GitHub<.

2 DESENVOLVIMENTO

Para estabelecer uma formatação padrão dos grafos para os experimentos com cada um dos algoritmos de Árvore Geradora Mínima, foram desenvolvidas três classes principais: Main, Grafo e GFGRandomGraph. Essas classes constituem a base do projeto. Essas classes fornecem funcionalidades para manipular e gerar grafos, além de realizar análises como encontrar articulações e componentes biconexos.

2.1 Main

O Main contém a interface do programa, onde o usuário informa apenas o número de vértices para geração a da Árvore Geradora Mínima que será sempre aleatório.

- **main(String[] args):** No método principal, o programa solicita ao usuário que escolha o número de vértices do grafo que será gerado. Em seguida, o programa gera um grafo aleatório com o número de vértices especificado pelo usuário, verifica a existência de

dois caminhos internamente disjuntos entre todos os pares de vértices e encontra as articulações e os componentes biconexos no grafo, exibindo também o método proposto por Tarjan e o tempo de execução de todos os três métodos.

2.2 Grafo

A classe Grafo representa um grafo não direcionado e implementa métodos para adicionar arestas, verificar a existência de dois caminhos internamente disjuntos e encontrar articulações e componentes biconexos.

- **Atributos**

  Número de vértices: Número de vértices do grafo.

  ADJ: Lista de adjacência para representar o grafo.

  Tempo: Variável utilizada para controlar o tempo durante a busca em profundidade.

2.2.1 Metódos

- **Grafo(int v):** Construtor que inicializa o grafo com o número de vértices especificado.
- **addAresta(int v, int w):** Adiciona uma aresta entre os vértices v e w.
- **hasTwoDisjointPaths(int u, int v):** Verifica se existem dois caminhos internamente disjuntos entre os vértices u e v.
- **dfs(int u, boolean[] visited, int avoid):** Realiza uma busca em profundidade (DFS) a partir do vértice u, evitando o vértice avoid.
- **dfs(int u, boolean[] visited, int avoid, boolean[] visitedFirst):** Realiza uma DFS a partir do vértice u, evitando o vértice avoid e marcando os vértices visitados durante a primeira DFS.
- **findArticulacoes():** Encontra as articulações no grafo.
- **findArticulacoesDfs(int u, boolean[] visitado, int[] disc, int[] low, int[] parent, List<Integer> articulacoes):** Algoritmo de busca em profundidade modificado para encontrar articulações.
- **findBiconnectedComponents():** Encontra os componentes biconexos no grafo.
- **bccDfs(int u, int[] disc, int[] low, Stack<Integer> st, boolean[] stackMember, int time, List<List<Integer» bcc):** Algoritmo de busca em profundidade para encontrar componentes biconexos.

2.3 GFGRandomGraph

Essa classe é responsável por gerar um grafo aleatório com um número especificado de vértices.

- **Atributos**

  Vertices: Número de vértices do grafo.

  Edges: Número de arestas do grafo.

  AdjacencyList: Lista de adjacência para representar o grafo.

2.3.1 Metódos

- **GFGRandomGraph(int numVertices):** Construtor que gera um grafo aleatório com o número especificado de vértices.
- **addEdge(int v, int w):** Adiciona uma aresta entre os vértices v e w.
- **main(String[] args):** Método principal para testar a geração de um grafo aleatório.

3 IMPLEMENTAÇÃO

Para realização dos experimentos, utilizamos três grafos direcionados e ponderados, com 100, 1000, 10000 e 100000 vértices, respectivamente. No programa, os testes foram realizados exclusivamente utilizando o código. O usuário tem a opção de gerar um grafo aleatório especificando apenas o número total de vértices. Após a geração do grafo, o usuário pode escolher entre os seguintes métodos.

- **Método 1:** Verifica a existência de dois caminhos internamente disjuntos (ou um ciclo) entre cada par de vértices do bloco.
- **Método 2:** Identifica articulações testando a conectividade após a remoção de cada vértice.
- **Método 3:** Tarjan.

3.0.1 Tarjan

O Método de Tarjan, desenvolvido por Robert Tarjan em 1972, é um algoritmo para encontrar componentes fortemente conectados em um grafo direcionado adaptado para achar a

Árvore Geradora Mínima. Um componente fortemente conectado em um grafo é um conjunto máximo de vértices onde existe um caminho direcionado entre cada par de vértices.Ele usa uma abordagem baseada em Busca em Largura para explorar o grafo. Ele atribui a cada vértice dois valores: o tempo de descoberta (quando o vértice foi descoberto durante a busca em profundidade) e o valor "baixo"(o menor tempo de descoberta de qualquer vértice na subárvore da busca em profundidade a partir deste vértice). Usando esses valores, o algoritmo é capaz de identificar os componentes fortemente conectados no grafo.

4 EXPERIMENTOS

Após a implementação dos algoritmos, realizamos testes com cada um deles utilizando os mesmos grafos predefinidos, sendo estes os de 100, 1000, 10000 e 100000 vértices, além de gerações e caminhos aleatórios. Os códigos nos quais foram realizados os testes estão disponíveis no >GitHub<.

4.1 Arquivos Gerados Aleatóriamente

Para os grafos gerados, os seguintes resultados foram obtidos para cada um dos algoritmos:

- **Grafo de 100 vértices:**
  - **Método 1:** Tempo de excução: 0ms
  - **Método 2:** Tempo de excução: 1ms
  - **Tarjan:** Tempo de excução: 1ms
- **Grafo de 1000 vértices:**
  - **Método 1:** Tempo de excução: 18ms
  - **Método 2:** Tempo de excução: 18ms
  - **Tarjan** Tempo de excução: 16ms
- **Grafo de 10000 vértices:**
  - **Método 1:** Tempo de excução: 27ms
  - **Método 2:** Tempo de excução: 27ms
  - **Tarjan:** Tempo de excução: 24ms
- **Grafo de 100000 vértices:**
  - **Não foi possível realizar teste, pois o grafo não foi concluído.**

- **Tempo de execução de todos os três métodos**
  - **Grafo de 100 vértices:** 0s
  - **Grafo de 1000 vértices:** 177s
  - **Grafo de 10000 vértices:** 509s
  - **Grafo de 100000 vértices:** Não foi possível realizar teste, pois o grafo não foi concluído.

4.2 Arquivos Predefinidos

Para os grafos predefinidos, os seguintes resultados foram obtidos para cada um dos algoritmos:

- **Grafo de 100 vértices:**
  - **Método 1:** Tempo de excução: 0ms
  - **Método 2:** Tempo de excução: 1ms
  - **Tarjan:** Tempo de excução: 1ms
- **Grafo de 1000 vértices:**
  - **Método 1:** Tempo de excução: 8ms
  - **Método 2:** Tempo de excução: 8ms
  - **Tarjan** Tempo de excução: 6ms
- **Grafo de 10000 vértices:**
  - **Método 1:** Tempo de excução: 13ms
  - **Método 2:** Tempo de excução: 13ms
  - **Tarjan:** Tempo de excução: 7ms
- **Grafo de 100000 vértices:**
  - **Método 1:** Tempo de excução: 315ms
  - **Método 2:** Tempo de excução: 315ms
  - **Tarjan:** Tempo de excução: 277ms
- **Tempo de execução de todos os três métodos**
  - **Grafo de 100 vértices:** 0s
  - **Grafo de 1000 vértices:** 177s

- **Grafo de 10000 vértices:** 509s
- **Grafo de 100000 vértices:** 902s

Após desenvolver, implementar e analisar os resultados dos algoritmos para o problema, vários aspectos se destacam no comportamento de cada método para diferentes tipos de grafos.

Os experimentos revelaram que os algoritmos desempenham bem em grafos do mundo real, que apresentam características mais comuns em situações cotidianas. No entanto, em grafos densos gerados aleatoriamente, os algoritmos demonstraram um desempenho insatisfatório devido à sua alta complexidade.

Portanto, os algoritmos implementados apresentam resultados satisfatórios principalmente em cenários do mundo real, onde raramente alcançam o pior caso em termos de tempo de execução. Recomenda-se, em futuras pesquisas, investigar por que as instâncias do mundo real são mais simples e buscar melhorar o tempo de execução em grafos gerados aleatoriamente.

4.3 Conclusão

Neste trabalho foi possível analisar a diferença de desempenho entre diferentes algoritmos para encontrar a Árvore Geradora Mínima em grafos direcionados. Vemos que a escolha do algoritmo tem importância fundamental porque cada método possui suas próprias características e complexidades, influenciando diretamente no tempo de execução e na eficiência da solução encontrada.

Os experimentos realizados revelaram que, em situações do mundo real, onde os grafos tendem a ser menos densos, os algoritmos demonstraram um desempenho satisfatório. No entanto, em cenários com grafos densos, como os gerados aleatoriamente, a complexidade dos algoritmos pode resultar em tempos de execução prolongados e, consequentemente, em soluções menos eficientes.

Portanto, a escolha do algoritmo adequado é crucial para garantir um desempenho ótimo na busca pela Árvore Geradora Mínima, especialmente em ambientes onde a eficiência computacional é uma preocupação importante. Esta análise destaca a necessidade contínua de pesquisa e desenvolvimento de técnicas mais eficazes para lidar com diferentes tipos de grafos e melhorar o desempenho dos algoritmos existentes.

Referências

[1] MAXIMILIAN BÖTHER; KISSIG, O.; WEYAND, C. Efficiently Computing Directed Minimum Spanning Trees. Society for Industrial and Applied Mathematics eBooks, p. 86–95, 14 apr. 2024.

[2] TARJAN, R. E. Finding optimum branchings. Networks, v. 7, n. 1, p. 25–35, 1977.

[3] Graphs. Disponível em: <https://algs4.cs.princeton.edu/40graphs/>. Acesso em: 14 apr. 2024.

[4] TarjanSCC.java. Disponível em: <http://surl.li/mwiam>. Acesso em: 14 apr. 2024.

[5] GabowSCC.java. Disponível em: <https://acervolima.com/como-criar-um-grafico-aleatorio-usando-a-geracao-de-borda-aleatoria-em-java/>. Acesso em: 14 apr. 2024.

[8] ChatGPT. Disponível em: <https://chat.openai.com/share/9977f01d-f864-4609-b61f-1cf1e40922bd>. Acesso em: 14 apr. 2024.