AEDSIII - Algoritmos de Arvores Geradoras Minímas

Caio D.Alves[1,2]

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
CEP 35931-008 – João Monlevade – MG – Brasil

[2]Departamento de Computação e Sistemas (DECSI)
Universidade Federal de Ouro Preto (UFOP) – João Monlevade, MG – Brasil

{caio}@caio.damasceno@aluno.ufop.edu.br

1. Introdução

Os algoritmos de construção de Árvores Geradoras Mínimas (AGM) são essenciais na teoria dos grafos e possuem diversas aplicações práticas, como na otimização de redes de comunicação e no planejamento de circuitos. Este relatório aborda a implementação dos algoritmos de Prim e Kruskal, que foram primeiramente estudados e compreendidos por meio do material didático utilizado em sala de aula.

O exercício proposto envolveu a implementação desses algoritmos em Python, aplicando-os a dois grafos específicos que também foram utilizados durante as aulas. Os grafos são representados em formato JSON e diferem tanto em estrutura quanto em peso das arestas. O objetivo do trabalho é analisar a aplicabilidade dos algoritmos na construção de Árvores Geradoras Mínimas para ambos os grafos, destacando suas características e eficiência.

Neste relatório, além de descrever a implementação de cada algoritmo, faremos uma análise comparativa das suas características e complexidades, destacando as situações em que cada um deles se mostra mais eficiente. A ferramenta de modelos de linguagem, como o ChatGPT, foi utilizada para ajustar e otimizar os códigos desenvolvidos, proporcionando sugestões valiosas e corrigindo detalhes técnicos.

O código fonte completo para os algoritmos discutidos pode ser acessado no GitHub através do seguinte link: Acessar o código fonte no GitHub.

2. Representação dos Grafos em JSON

Os dois grafos utilizados para a aplicação dos algoritmos de Prim e Kruskal estão representados no formato JSON conforme abaixo:

```json
{
  "grafo1": {
    "nos": [0, 1, 2, 3, 4, 5],
    "arestas": [
      { "origem": 0, "destino": 1, "peso": 5 },
      { "origem": 0, "destino": 4, "peso": 15 },
      { "origem": 0, "destino": 5, "peso": 2 },
      { "origem": 1, "destino": 2, "peso": 12 },
      { "origem": 1, "destino": 3, "peso": 9 },
      { "origem": 1, "destino": 4, "peso": 22 },
      { "origem": 3, "destino": 5, "peso": 6 },
      { "origem": 1, "destino": 5, "peso": 4 },
      { "origem": 4, "destino": 2, "peso": 1 }
    ]
  },
  "grafo2": {
    "nos": ["A", "B", "C", "D", "E", "F"],
    "arestas": [
      { "origem": "A", "destino": "B", "peso": 6 },
      { "origem": "A", "destino": "F", "peso": 6 },
      { "origem": "A", "destino": "E", "peso": 9 },
      { "origem": "F", "destino": "C", "peso": 4 },
      { "origem": "F", "destino": "E", "peso": 8 },
      { "origem": "E", "destino": "B", "peso": 5 },
      { "origem": "E", "destino": "D", "peso": 4 },
      { "origem": "C", "destino": "E", "peso": 7 },
      { "origem": "C", "destino": "D", "peso": 8 }
    ]
  }
}
```

Listing 1. Representação dos grafos em formato JSON

Esses grafos servirão como base para a implementação e teste dos algoritmos de Prim e Kruskal, permitindo uma análise comparativa de seus desempenhos e eficiência na geração de Árvores Geradoras Mínimas.

3. Algoritmo de Prim

3.1. Descrição

O algoritmo de Prim é uma técnica eficiente para encontrar a Árvore Geradora Mínima (AGM) em um grafo não direcionado e ponderado. Ele opera de maneira gulosa, começando com um único nó e expandindo a árvore geradora adicionando repetidamente a aresta de menor peso que conecta um nó na árvore a um nó fora dela, até que todos os nós tenham sido incluídos na árvore geradora.

Inicialmente, este algoritmo foi aprendido e compreendido em sala de aula através do material didático [de Ouro Preto 2024]. A implementação em Python foi desenvolvida com o objetivo de explorar sua aplicabilidade na construção de árvores geradoras mínimas em grafos diversos.

3.2. Complexidade

A implementação clássica do algoritmo de Prim, utilizando uma lista não ordenada de arestas possíveis, possui uma complexidade de tempo de $O(V^2)$, onde V é o número de vértices no grafo. No entanto, ao utilizar uma fila de prioridade (como uma heap binária), a complexidade pode ser reduzida para $O(E \log V)$, onde E é o número de arestas, tornando o algoritmo mais eficiente para grafos esparsos.

3.3. Implementação em Python

```python
class AlgoritmoPrim:
    def __init__(self, grafo):
        self.grafo = grafo

    def encontrar_arvore_geradora_minima(self):
        """Encontra a  rvore  Geradora M nima  (AGM) usando o algoritmo de Prim."""
        nos_visitados = set()
        arestas_agm = []
        nos = list(self.grafo.nos)
        no_inicial = nos[0]
        nos_visitados.add(no_inicial)
        arestas_possiveis = [(no_inicial, destino, peso) for destino, peso in self.grafo
            .adjacencias[no_inicial]]

        while arestas_possiveis:
            arestas_possiveis.sort(key=lambda x: x[2])  # Ordena com base no peso
            menor_aresta = arestas_possiveis.pop(0)
            origem, destino, peso = menor_aresta

            if destino not in nos_visitados:
                nos_visitados.add(destino)
                arestas_agm.append((origem, destino, peso))

                for proxima_aresta in self.grafo.adjacencias[destino]:
                    if proxima_aresta[0] not in nos_visitados:
                        arestas_possiveis.append((destino, proxima_aresta[0],
                            proxima_aresta[1]))

        return arestas_agm
```

Listing 2. Implementação do Algoritmo de Prim em Python

Este código implementa o algoritmo de Prim para encontrar a Árvore Geradora Mínima (AGM) de um grafo não direcionado. A cada iteração, o algoritmo seleciona a aresta de menor peso que conecta um nó visitado a um nó não visitado, garantindo que a árvore geradora seja construída de maneira eficiente.

4. Algoritmo de Kruskal

4.1. Descrição

O algoritmo de Kruskal é uma técnica eficiente para encontrar a Árvore Geradora Mínima (AGM) de um grafo não direcionado e ponderado. Diferentemente do algoritmo de Prim, o algoritmo de Kruskal opera de forma a selecionar as arestas de menor peso do grafo, adicionando-as à AGM, desde que não formem um ciclo, até que todos os vértices estejam conectados.

Inicialmente, este algoritmo foi aprendido e compreendido em sala de aula através do material didático. A implementação em Python foi desenvolvida com o objetivo de explorar sua aplicabilidade na construção de árvores geradoras mínimas em grafos de diferentes configurações.

4.2. Complexidade

A complexidade do algoritmo de Kruskal é dominada pela ordenação das arestas e pelas operações do Union-Find, resultando em uma complexidade de tempo de $O(E \log E + E \log V)$, onde E é o número de arestas e V é o número de vértices. Esta complexidade é adequada para grafos esparsos, tornando o algoritmo eficiente para muitos problemas práticos.

4.3. Implementação da Estrutura Union-Find em Python

```python
class UnionFind:
    def __init__(self, n):
        """Inicializa a estrutura Union-Find para 'n' elementos."""
        self.pai = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        """Encontra o representante do conjunto de 'u'."""
        if u != self.pai[u]:
            self.pai[u] = self.find(self.pai[u])
        return self.pai[u]

    def union(self, u, v):
        """Une os conjuntos de 'u' e 'v'."""
        raiz_u = self.find(u)
        raiz_v = self.find(v)

        if raiz_u != raiz_v:
            # Uni o por rank
            if self.rank[raiz_u] > self.rank[raiz_v]:
                self.pai[raiz_v] = raiz_u
            elif self.rank[raiz_u] < self.rank[raiz_v]:
                self.pai[u] = raiz_v
            else:
                self.pai[raiz_v] = raiz_u
                self.rank[raiz_u] += 1
```

Listing 3. Implementação da Estrutura Union-Find em Python

4.4. Implementação do Algoritmo de Kruskal em Python

```python
from unionFind import UnionFind

class AlgoritmoKruskal:
    def __init__(self, grafo):
        self.grafo = grafo

    def encontrar_arvore_geradora_minima(self):
        """Encontra a  rvore  Geradora M nima  (AGM) usando o algoritmo de Kruskal."""
        agm = []
        # Ordena as arestas pelo peso
        arestas = sorted(self.grafo.arestas, key=lambda aresta: aresta['peso'])

        # Inicializa a estrutura Union-Find para controle dos ciclos
        uf = UnionFind(len(self.grafo.nos))
        indice_no = {no: i for i, no in enumerate(self.grafo.nos)}

        for aresta in arestas:
            origem = aresta['origem']
            destino = aresta['destino']
            peso = aresta['peso']

            # Obtenha o  ndice  dos  n  s  para o Union-Find
            u = indice_no[origem]
            v = indice_no[destino]

            # Verifica se a aresta cria um ciclo
            if uf.find(u) != uf.find(v):
                uf.union(u, v)
                agm.append((origem, destino, peso))

        return agm
```

Listing 4. Implementação do Algoritmo de Kruskal em Python

Este código implementa o algoritmo de Kruskal para encontrar a Árvore Geradora Mínima (AGM) de um grafo não direcionado. Utilizando uma estrutura Union-Find para

controlar a formação de ciclos, o algoritmo garante que a árvore geradora seja construída de maneira eficiente, minimizando o peso total.

5. Conclusão

Neste relatório, analisamos e implementamos dois algoritmos fundamentais para a construção de Árvores Geradoras Mínimas (AGM) em grafos: Prim e Kruskal. Cada um desses algoritmos apresenta características específicas que os tornam mais adequados para diferentes tipos de grafos e contextos de aplicação.

O algoritmo de Prim é eficiente para grafos densos, onde o número de arestas é alto em relação ao número de vértices. Sua abordagem de expansão gradual a partir de um nó inicial permite a construção da AGM de maneira eficiente, especialmente quando implementado com uma fila de prioridade.

Por outro lado, o algoritmo de Kruskal é mais adequado para grafos esparsos, onde o número de arestas é relativamente pequeno. Ele se destaca ao ordenar todas as arestas pelo peso e utiliza a estrutura de dados Union-Find para evitar a formação de ciclos, garantindo assim uma AGM mínima. Sua eficiência é especialmente notável em grafos com poucas arestas.

A implementação prática desses algoritmos em Python, utilizando grafos representados em formato JSON, nos proporcionou uma oportunidade de explorar suas diferenças de desempenho e aplicações em problemas reais. A experiência adquirida ao programar e testar esses algoritmos reforçou nossa compreensão de suas características, vantagens e limitações, demonstrando a importância de selecionar o algoritmo adequado conforme a estrutura e os requisitos do grafo em questão.

Referências

[de Ouro Preto 2024] de Ouro Preto, U. F. (2024). Arvores geradoras. Acesso em: 30 ago. 2024.