AEDSIII - Fluxo em Redes

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 fluxo de redes desempenham um papel crucial na teoria dos grafos e têm aplicações práticas em diversas áreas, como otimização de redes de transporte e sistemas de comunicação. Este relatório aborda a implementação dos algoritmos de Sucessivos Caminhos Mínimos e Ford-Fulkerson, que foram estudados e analisados com base no material didático utilizado em sala de aula.

O exercício proposto envolveu a implementação desses algoritmos em Python, aplicando-os a dois grafos específicos representados em formato JSON. Estes grafos diferem em estrutura e características das arestas. O objetivo do trabalho é analisar a aplicabilidade dos algoritmos na resolução de problemas de fluxo máximo em redes, destacando suas características e eficiência.

Neste relatório, além de descrever a implementação de cada algoritmo, faremos uma análise comparativa das suas características e complexidades, evidenciando 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 fluxo estão representados no formato JSON conforme abaixo:

```json
{
  "graph_1": {
    "nodes": [
      { "id": "s" },
      { "id": "1" },
      { "id": "2" },
      { "id": "3" },
      { "id": "4" },
      { "id": "t" }
    ],
    "edges": [
      { "source": "s", "target": "1", "weight": 10 },
      { "source": "1", "target": "2", "weight": 7 },
      { "source": "1", "target": "3", "weight": 3 },
      { "source": "2", "target": "4", "weight": 3 },
      { "source": "3", "target": "4", "weight": 3 },
      { "source": "4", "target": "t", "weight": 7 }
    ]
  },
  "graph_2": {
    "nodes": [
      { "id": "s" },
      { "id": "1" },
      { "id": "2" },
      { "id": "3" },
      { "id": "4" },
      { "id": "t" }
    ],
    "edges": [
      { "source": "s", "target": "1", "capacity": 1, "cost": 2 },
      { "source": "1", "target": "2", "capacity": 2, "cost": 1 },
      { "source": "1", "target": "3", "capacity": 3, "cost": 1 },
      { "source": "1", "target": "4", "capacity": 5, "cost": 2 },
      { "source": "2", "target": "4", "capacity": 2, "cost": 1 },
      { "source": "3", "target": "4", "capacity": 3, "cost": 1 },
      { "source": "4", "target": "t", "capacity": 1, "cost": 2 }
    ]
  }
}
```

Listing 1. Representação dos grafos em formato JSON

Esses grafos servirão como base para a implementação e teste dos algoritmos de Sucessivos Caminhos Mínimos e Ford-Fulkerson, permitindo uma análise comparativa de seus desempenhos e eficiência na solução de problemas de fluxo máximo em redes.

3. Algoritmo de Ford-Fulkerson

3.1. Descrição

O algoritmo de Ford-Fulkerson é um método utilizado para encontrar o fluxo máximo em uma rede de fluxo. Ele baseia-se na técnica de encontrar caminhos de aumento em um grafo residual e aumentar o fluxo total através desses caminhos. O algoritmo começa com um fluxo inicial de 0 e, iterativamente, busca caminhos do nó fonte ao nó sumidouro onde ainda há capacidade residual. Cada caminho encontrado é utilizado para aumentar o fluxo até que não seja mais possível encontrar tais caminhos.

Este algoritmo foi inicialmente estudado e compreendido através do material didático utilizado em sala de aula [de Ouro Preto 2024]. A implementação em Python foi desenvolvida com o intuito de explorar sua aplicabilidade na maximização do fluxo em redes com capacidades não negativas.

3.2. Complexidade

A complexidade do algoritmo de Ford-Fulkerson depende do método utilizado para encontrar os caminhos de aumento. Se o algoritmo de busca em largura (BFS) for empregado, a complexidade do algoritmo é $O(E \cdot f)$, onde E é o número de arestas e f é o valor máximo do fluxo. No entanto, o desempenho real pode variar com base na escolha da técnica para encontrar os caminhos de aumento e na estrutura específica do grafo. A utilização do BFS garante uma eficiência adequada para muitos casos práticos, mas em grafos densos ou com grandes capacidades, o algoritmo pode enfrentar limitações.

3.3. Implementação em Python

```python
from collections import defaultdict
class FordFulkerson:
    def __init__(self, graph):
        self.graph = graph
        self.residual_graph = defaultdict(dict)

    def bfs(self, source, sink, parent):
        visited = set()
        queue = [source]
        visited.add(source)

        while queue:
            u = queue.pop(0)
            for v in self.residual_graph[u]:
                if v not in visited and self.residual_graph[u][v] > 0:
                    queue.append(v)
                    visited.add(v)
                    parent[v] = u
                    if v == sink:
                        return True
        return False

    def ford_fulkerson(self, source, sink):
        max_flow = 0
        parent = {}

        for edge in self.graph['edges']:
            self.residual_graph[edge['source']][edge['target']] = edge['weight']

        while self.bfs(source, sink, parent):
            path_flow = float('Inf')
            s = sink

            while s != source:
                path_flow = min(path_flow, self.residual_graph[parent[s]][s])
                s = parent[s]

            max_flow += path_flow

            v = sink
            while v != source:
                u = parent[v]
                self.residual_graph[u][v] -= path_flow
                self.residual_graph[v][u] = self.residual_graph.get(v, {}).get(u, 0) +
                    path_flow
                v = parent[v]

        return max_flow
```

Listing 2. Implementação do Algoritmo FORD-FULKERSON

Esta implementação do algoritmo de Ford-Fulkerson é projetada para encontrar o fluxo máximo em uma rede de fluxo, utilizando o método de busca em largura para identificar os caminhos de aumento e ajustar o fluxo residual de maneira eficiente.

4. Algoritmo de Sucessivos Caminhos Mínimos

4.1. Descrição

O algoritmo de Sucessivos Caminhos Mínimos é uma técnica eficaz para resolver o problema de fluxo máximo em redes de fluxo com custos. O algoritmo funciona ao encontrar iterativamente o caminho de custo mínimo no grafo residual e aumentar o fluxo ao longo desse caminho até que não haja mais caminhos de aumento com custo reduzido. Essa abordagem permite a maximização do fluxo enquanto minimiza o custo total do fluxo enviado.

Este algoritmo foi inicialmente estudado e compreendido através do material didático fornecido em sala de aula. A implementação em Python foi desenvolvida com o objetivo de explorar sua aplicabilidade na otimização de redes de fluxo, permitindo a análise de problemas práticos de fluxo máximo e custo mínimo.

4.2. Complexidade

A complexidade do algoritmo de Sucessivos Caminhos Mínimos é influenciada pelo algoritmo de Bellman-Ford utilizado para encontrar os caminhos de custo mínimo e pelas atualizações das capacidades residuais. Em geral, a complexidade é $O(V \cdot E^2)$, onde V é o número de vértices e E é o número de arestas. Embora a complexidade possa parecer alta, o algoritmo é eficiente para muitos casos práticos, especialmente quando aplicado a grafos com custos e capacidades bem definidos. .

4.3. Implementação do Algoritmo de Sucessivos Caminhos Mínimos

```python
from collections import defaultdict

class SuccessiveShortestPaths:
    def __init__(self, graph):
        self.graph = graph
        self.residual_graph = defaultdict(lambda: defaultdict(int))
        self.cost_graph = defaultdict(lambda: defaultdict(int))

    def bellman_ford(self, source):
        distances = defaultdict(lambda: float('Inf'))
        parent = {}
        distances[source] = 0

        for _ in range(len(self.graph['nodes']) - 1):
            for edge in self.graph['edges']:
                u, v = edge['source'], edge['target']
                capacity = self.residual_graph[u][v]
                cost = self.cost_graph[u][v]
                if capacity > 0 and distances[u] + cost < distances[v]:
                    distances[v] = distances[u] + cost
                    parent[v] = u

        return distances, parent

    def successive_shortest_paths(self, source, sink):
        max_flow = 0
        min_cost = 0

        for edge in self.graph['edges']:
            self.residual_graph[edge['source']][edge['target']] = edge['capacity']
            self.cost_graph[edge['source']][edge['target']] = edge['cost']

        while True:
            distances, parent = self.bellman_ford(source)

```

```
            if distances[sink] == float('Inf'):
                break

            path_flow = float('Inf')
            v = sink

            while v != source:
                u = parent[v]
                path_flow = min(path_flow, self.residual_graph[u][v])
                v = parent[v]

            max_flow += path_flow

            v = sink
            while v != source:
                u = parent[v]
                self.residual_graph[u][v] -= path_flow
                self.residual_graph[v][u] += path_flow
                min_cost += path_flow * self.cost_graph[u][v]
                v = parent[v]

        return max_flow, min_cost
```

Listing 3. Implementação do Algoritmo de Sucessivos Caminhos Mınimos Em Python

Esta implementação do algoritmo de Sucessivos Caminhos Mínimos permite a maximização do fluxo em redes enquanto minimiza o custo total, utilizando uma abordagem iterativa que encontra e ajusta os caminhos de custo mínimo no grafo residual.

5. Conclusão

Neste relatório, analisamos e implementamos dois algoritmos fundamentais para a resolução de problemas de fluxo máximo em redes: Ford-Fulkerson e Sucessivos Caminhos Mínimos. Cada um desses algoritmos possui características distintas que os tornam adequados para diferentes contextos e tipos de grafos.

O algoritmo de Ford-Fulkerson é amplamente utilizado para encontrar o fluxo máximo em redes de fluxo. Sua abordagem de busca por caminhos de aumento e ajuste de fluxo residual é eficaz em muitas situações, especialmente quando combinada com a busca em largura (BFS). Sua simplicidade e flexibilidade o tornam uma escolha robusta para resolver problemas de fluxo máximo em grafos com capacidades não negativas.

Por outro lado, o algoritmo de Sucessivos Caminhos Mínimos se destaca na otimização de redes com custos, buscando minimizar o custo total do fluxo enquanto maximiza a quantidade de fluxo. Utilizando o algoritmo de Bellman-Ford para encontrar caminhos de custo mínimo e ajustando o fluxo de forma iterativa, este método é eficiente para grafos onde o custo associado ao fluxo é uma consideração importante.

A implementação prática desses algoritmos em Python, com grafos representados em formato JSON, permitiu explorar suas aplicabilidades em problemas reais de fluxo máximo e custo mínimo. A experiência adquirida ao programar e testar esses algoritmos reforçou nossa compreensão de suas características, vantagens e limitações, demonstrando a importância de escolher o algoritmo adequado com base nas necessidades específicas do problema e na estrutura do grafo.

Referências

[de Ouro Preto 2024] de Ouro Preto, U. F. (2024). Fluxo em redes. Acesso em 13 de setembro de 2024.