Otimização de Infraestrutura de Redes utilizando Árvores Geradoras Mínimas e Algoritmo de Kruskal

Eric de Gusmão Pino
Ciência da Computação
UNIMA
Maceió, Brasil
ericdegusmaopino@gmail.com

***Abstract*—A otimização de custos em conectividade é um dos desafios centrais em engenharia de redes e logística. Este artigo aborda o problema computacional da Árvore Geradora Mínima (Minimum Spanning Tree - MST) aplicado à implantação de infraestrutura de fibra óptica. O objetivo é conectar um conjunto de localidades minimizando o custo total. Foi implementada uma solução baseada no Algoritmo de Kruskal utilizando a linguagem Go (Golang), aproveitando recursos modernos como genéricos e bibliotecas de teste robustas. Os experimentos, realizados em um processador Intel i9, demonstraram alta eficiência, processando grafos com 5.000 arestas em aproximadamente 0.25 milissegundos, além de validar a integridade da solução através de testes de unidade e Fuzzing.**

***Index Terms*—Teoria dos Grafos, Árvore Geradora Mínima, Kruskal, Go, Fuzz Testing, Otimização.**

I. Introdução

Grafos são estruturas matemáticas fundamentais para modelar relações entre objetos. Um dos problemas clássicos nessa área é o da Árvore Geradora Mínima (AGM), que consiste em encontrar um subconjunto de arestas que conecte todos os vértices de um grafo ponderado, sem formar ciclos e com o menor peso total possível.

Este problema possui vasta aplicação prática. Em redes de computadores, é a base do *Spanning Tree Protocol* (STP) para evitar loops de broadcast. Em logística, auxilia no planejamento de malhas viárias. Em design de circuitos VLSI, otimiza as conexões entre componentes reduzindo o calor e atraso de sinal.

Este trabalho foca na aplicação da AGM para o planejamento de infraestrutura de telecomunicações. O problema proposto é: dado um conjunto de cidades e o custo para passar fibra óptica entre elas, qual é a configuração que conecta todas as cidades com o menor custo financeiro? A solução foi implementada em Go, escolhida por sua eficiência de compilação e execução.

II. Fundamentação Teórica

A. *Definição Formal do Problema*

Seja um grafo não direcionado, conexo e ponderado $G = (V, E)$, onde V representa o conjunto de vértices (cidades) e E o conjunto de arestas (cabos). Cada aresta $(u, v) \in E$ possui um peso $w(u, v) \geq 0$. O objetivo é encontrar um subgrafo acíclico $T \subseteq E$ que conecte todos os vértices tal que a soma dos pesos seja minimizada:

$$w(T) = \sum_{(u,v)\in T} w(u, v) \tag{1}$$

Se o grafo tiver V vértices, a AGM terá exatamente $|V| - 1$ arestas.

B. *Áreas de Aplicação*

- **Telecomunicações:** Cabeamento de redes WAN/LAN.
- **Clusterização:** O algoritmo de *Single-linkage clustering* é essencialmente uma variante da AGM.
- **Aproximação para PCV:** A AGM é usada como heurística para encontrar limites inferiores no Problema do Caixeiro Viajante.

C. *Algoritmo Escolhido: Kruskal*

Existem dois algoritmos gulosos principais para este problema: Prim e Kruskal. O Algoritmo de Prim cresce a árvore a partir de um nó, sendo mais eficiente para grafos densos ($E \approx V^2$).

O **Algoritmo de Kruskal**, escolhido para este trabalho, seleciona as arestas de menor peso globalmente, desde que não formem ciclos. É ideal para grafos esparsos e sua complexidade é dominada pela ordenação das arestas: $O(E \log E)$. A detecção de ciclos é feita eficientemente usando a estrutura de dados *Union-Find* (Conjuntos Disjuntos).

III. Implementação da Solução

A solução foi desenvolvida em **Go 1.23**, utilizando o pacote ‘slices‘ para ordenação otimizada.

A. *Estruturas de Dados*

O grafo é representado por uma lista de arestas, facilitando a iteração necessária para o Kruskal.

```go
type Edge struct {
   Source int
   Dest  int
   Weight int
}

type Graph struct {
   V    int  // Numero de Vertices
   Edges []Edge // Lista de Arestas
}
```

B. Lógica do Algoritmo

A função ‘KruskalMST‘ executa os seguintes passos: 1. Ordena as arestas por peso (crescente). 2. Inicializa a estrutura Union-Find. 3. Itera sobre as arestas; se os vértices de origem e destino estão em conjuntos diferentes (‘findIterative‘), une-os (‘Union‘) e adiciona a aresta ao resultado. 4. Interrompe ao atingir $V-1$ arestas.

```
// Exerto da funcao principal
slices.SortFunc(graph.Edges, func(a, b Edge)
    int {
   return cmp.Compare(a.Weight, b.Weight)
})

for _, edge := range graph.Edges {
   if edgesCount >= treeSize { break }

   rootX := findIterative(parent, edge.Source)
   rootY := findIterative(parent, edge.Dest)

   if rootX != rootY {
      result = append(result, edge)
      totalWeight += edge.Weight
      // Logica de Union by Rank...
   }
}
```

IV. Execução dos Testes e Resultados

A validação foi realizada em três frentes: testes unitários de cenários conhecidos, testes de robustez (Fuzzing) e análise de desempenho (Benchmark). O ambiente de teste utilizou um processador Intel Core i9-12900HX.

A. Entradas Simuladas (Cenários)

Foram criados cenários específicos para garantir a corretude:

- **Triângulo Básico:** Grafo cíclico simples para verificar a eliminação da aresta mais custosa.
- **Grafo Desconexo:** Para verificar se o algoritmo lida corretamente com florestas.
- **Multigrafo:** Várias arestas entre os mesmos nós; o algoritmo deve escolher apenas a de menor peso.

B. Teste de Robustez (Fuzzing)

Utilizou-se o recurso nativo de Fuzzing do Go (‘testing.F‘) para gerar milhares de grafos aleatórios. O teste verificou três invariantes matemáticos para qualquer entrada arbitrária: 1. O número de arestas na solução nunca excede $V-1$. 2. O peso total retornado iguala a soma das arestas da lista. 3. **Não existem ciclos** na solução final.

O Fuzzing não detectou falhas, garantindo robustez para entradas imprevisíveis.

C. Análise de Desempenho

Para simular um cenário realista de grande escala (ex: rede metropolitana), executou-se o ‘BenchmarkKruskalLarge‘ com 1.000 vértices e 5.000 arestas.

O tempo de execução de 0.25 milissegundos é extremamente baixo, confirmando a eficiência da implementação $O(E \log E)$ e do compilador Go. Isso viabiliza o uso do algoritmo em sistemas de tempo real, como recálculo de rotas em protocolos de rede.

TABLE I
Resultados do Benchmark

Parâmetro	Valor
Vértices	1.000
Arestas	5.000
Operações (Média)	3.933 execuções
Tempo por Operação	**254.413 ns** ($\approx$ 0.25 ms)
Alocação de Memória	40 KB/op

V. Conclusão

Este trabalho apresentou uma implementação robusta do algoritmo de Kruskal para resolução de problemas de Árvore Geradora Mínima. A utilização da linguagem Go, aliada a estruturas de dados eficientes como Union-Find com compressão de caminho, resultou em um software de alto desempenho.

A aplicação de técnicas modernas de verificação, como Fuzzing, garantiu que a solução não apenas funciona para casos ideais, mas é resiliente a entradas complexas. Os resultados obtidos (0.25ms para 1000 nós) superam largamente os requisitos para a maioria das aplicações industriais de logística e redes.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," 3rd ed. MIT Press, 2009.

[2] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," Proc. Am. Math. Soc., vol. 7, 1956.

[3] A. A. A. Donovan and B. W. Kernighan, "The Go Programming Language," Addison-Wesley, 2015.

[4] IEEE, "IEEE Editorial Style Manual," IEEE Periodicals, Transactions/Journals Dept., 2016.