Notas de Aula - AED2 – Grafos: árvore geradora mínima
Prof. Jefferson T. Oliva

Suponha que queremos resolver o seguinte problema: dado um conjunto de computadores, onde cada par de computadores pode ser ligado usando uma quantidade de fibra ótica, encontrar uma rede interconectando-os que use a menor quantidade de fibra ótica possível.

Este problema pode ser modelado através de um grafo não orientado ponderado onde os vértices representam os computadores, as arestas representam as conexões que podem ser construídas e o peso/custo de uma aresta representa a quantidade de fibra ótica necessária.

Abstração do problema:
Nessa modelagem, o problema que queremos resolver é encontrar um subgrafo gerador (que contém todos os vértices do grafo original), conexo (para garantir a interligação de todas as cidades) e cuja soma dos custos de suas arestas seja a menor possível. Obviamente, o problema só tem solução se o grafo for conexo. Daqui pra frente vamos supor que o grafo de entrada é conexo. Além disso, o subgrafo gerador procurado é sempre uma árvore (supondo que os pesos são positivos).

Formalização do problema no slide 5.

Exemplo no 6.

Veremos dois algoritmos gulosos clássicos para resolver o problema:
- algoritmo de Prim
- algoritmo de Kruskal

Algoritmo de Prim

Nesse algoritmo, **A** é uma árvore com raiz **r**, escolhido arbitrariamente. Em cada iteração, o algoritmo considera o corte **gamma(C)**, onde **C** é o conjunto de vértices que são extremos de **A**.

O que é corte? Nesse caso, são arestas extremas de A que ligam aos vértices restantes do grafo.

Ele encontra uma *aresta leve* **(u, v)** (menor peso) neste corte e acrescenta-a ao conjunto A e começa outra iteração até que **A** seja uma árvore geradora.

Um detalhe de implementação importante é como encontrar eficientemente uma aresta leve no corte.

A ordem de exploração do vértices é de acordo com o peso da aresta.

Ver slides entre 10 e 18.

O algoritmo mantém durante sua execução as seguintes informações **(slide 19)**:
- Todos os vértices que não estão na árvore estão em uma fila de prioridade (de mínimo) **Q**.

- Cada vértice **v** de **Q** tem uma chave **key[v]** que indica o menor peso de qualquer aresta ligando v a algum vértice da árvore. Se não existir nenhuma aresta, então **key[v] = infinito**.
- A variável **π[u]** indica o **pai** de **u** na árvore. Então: A = **{(u, π[u]) : u ∈ V − {r} − Q}**.

Explicação do algoritmo no slide 20:
- Entre as linhas 1 e 3, os elementos do vetor **key** são iniciados com infinito e os de **π**, com NULL.
- Como a raiz r é o vértice de onde deve ser começada a geração da árvore, em key[r] é atribuído zero (linha 4) e o seu pai permanecerá NULL ao final da execução do algoritmo.
- Na linha 5, todos os vértices são enfileirados
- Em seguida, o loop do laço na linha 6 é executado enquanto houver vértice na fila
→ A = **{(u, π[u]) : u ∈ V − {r} − Q}**
→ Os vértices que estão na árvore são os que não estão na fila (**V - Q**)
→ Para todos os vértices que estão na fila e que possuem pai, as suas chaves (**key[v]**) são menores que infinito, ou seja, **key[v]** é o peso de uma aresta leve (v, **π[v]**)
- Assim, na linha 7, um vértice da fila incidente em uma aresta leve que cruza o corte (**V – Q, Q**) é identificado (Com exceção da primeira raiz, em que **u = r** por causa da linha 4). Em outras palavras, é identificado o vértice de menor custo que liga a árvore A à fila. Na linha 7, um elemento é desenfileirado
- Na linha 8, são explorados os vértices adjacentes a **u** de forma ordenada por peso
- Para isso, são testadas duas condições na linha 9: se um adjacente **v** ainda está na fila (ou seja, não foi adicionado na árvore) e se o peso entre as arestas **u** e **v** (**w(u, v)**) é menor do que está armazenado em **key[v]**. Se é a primeira visita ao vértice, lógico que **key[u]** irá mudar na primeira comparação, pois no mesmo está atribuído infinito.
- Caso as condições da linha 9 sejam atendidas, o pai de **v** será **u** (**π[u] = v**) e o key[v] será o peso da aresta que liga u e v (key[v] = w(u, v)), como ocorre nas linhas 10 e 11 respectivamente.
- Lembre-se, enquanto a fila não estiver vazia, os comandos entre as linhas 7 e 11 serão executados

Ver slides entre 21 e 63.

Complexidade (para Q implementada como heap):
- Iniciar key e pi: O(|V|)
- Inicialização de Q: |V| log |V|
- O comando EXTRACT-MIN: O(log |V|), por mais que a remoção tenha custo O(1), a reorganização da heap é na ordem de log |V|. Como esse algoritmo é executado |V| vezes, então a complexidade da linha 7 é na ordem de O(|V| log |V|)
- Linhas entre 8: O(|E|) arestas são exploradas, no total
- Linha 9: percorrer heap é na ordem de log |V|
- Logo, o tempo de execução nas linhas de 8 a 11 é na ordem de O(|E| log |V|)
- Custo total: O(|V| log |V| + |E| log |V|)
- Supondo que o número de arestas seja sempre maior que o de vértices, o custo total, de forma simplificada, é na ordem de O(|E| log |V|)

Algoritmo de Kruskal

No algoritmo de Kruskal, o subgrafo **F = (V , A)** é uma floresta. Inicialmente, A é vazio.

Em cada iteração, o algoritmo escolhe uma aresta **(u, v)** de menor peso que liga vértices de componentes (árvores) distintos **C** e **C′** de **F = (V , A)**

Note que **(u, v)** é uma aresta leve do corte **δ(C)**.

Ele acrescenta **(u, v)** ao conjunto **A** e começa outra iteração até que **A** seja uma árvore geradora.

Um detalhe de implementação importante é como encontrar a aresta de menor peso ligando componentes distintos.

Ver exemplo nos slides entre 66 e 80

Implementação genérica (slide 81)

```
AGM-KRUSKAL(G, w)
1  A ← ∅
2  Ordene as arestas em ordem não-decrescente de peso
3  para cada (u, v) ∈ E  nessa ordem  faça
4      se  u e v estão em componentes distintos de (V, A)
5          então  A ← A ∪ {(u, v)}
6  devolva A
```

Se o par de vértices já estiver na árvore, a respectiva aresta que ambos não deve ser adicionado, pois formaria um ciclo

Problema: Como verificar eficientemente se u e v estão no mesmo componente (árvore) da floresta GA = (V , A)?

Inicialmente GA = (V , ∅), ou seja, GA corresponde à floresta onde cada componente é um vértice isolado.

Ao longo do algoritmo, esses componentes são modificados pela inclusão de arestas em A.

Uma estrutura de dados para representar **GA = (V , A)** deve ser capaz de executar eficientemente as seguintes operações:
- Dado um vértice **u**, determinar o componente de **GA** que contém **u** e
- Dados dois vértices **u** e **v** em componentes distintos **C** e **C′**, fazer a união desses em um novo componente.

ED para conjuntos disjuntos: Uma estrutura de dados para conjuntos disjuntos mantém uma coleção **{S1, S2, ..., Sk}** de conjuntos disjuntos dinâmicos (isto é, eles mudam ao longo do tempo).
- Cada conjunto é identificado por um representante que é um elemento do conjunto.
- Quem é o representante é irrelevante, mas se o conjunto não for modificado, então o representante não pode mudar.

ED para conjuntos disjuntos
- Uma estrutura de dados para conjuntos disjuntos deve ser capaz de executar as seguintes operações:
→ **Make-Set(x)**: cria um novo conjunto **{x}**.
→ **Union(x, y)**: une os conjuntos (disjuntos) que contém **x** e **y**, digamos **Sx** e **Sy**, em um novo conjunto **Sx** ∪ **Sy**. Os conjuntos **Sx** e **Sy** são descartados da coleção.
→ **Find-Set(x)** devolve um apontador para o representante do (único) conjunto que contém **x**.
Componentes conexos (slide 85):

```
CONNECTED-COMPONENTS(G)
1  para cada vértice v ∈ V[G] faça
2      MAKE-SET(v)
3  para cada aresta (u, v) ∈ E[G] faça
4      se FIND-SET(u) ≠ FIND-SET(v)
5          então UNION(u, v)

SAME-COMPONENT(u, v)
1  se FIND-SET(u) = FIND-SET(v)
2      então devolva SIM
3      senão devolva NÃO
```

Interpretação do algoritmo connected-components
- linhas 1 e 2 constrói um componente (árvore) para cada vértice
- linhas 3, 4 e 5, para cada aresta, se o representante de cada árvore for diferente (se os dois vértices estão em árvores diferentes), então unimos os vértices u e v

Interpretação do algoritmo connected-components: verificar se os vértices (árvores) u e v são os mesmos (ou possuem o mesmo representante).

Versão completa do algoritmo de Kruskal (slide 86):

```
AGM-KRUSKAL(G, w)
1  A ← ∅
2  para cada v ∈ V[G] faça
3      MAKE-SET(v)
4  Ordene as arestas em ordem não-decrescente de peso
5  para cada (u, v) ∈ E nessa ordem faça
6      se FIND-SET(u) ≠ FIND-SET(v)
7          então A ← A ∪ {(u, v)}
8                UNION(u, v)
9  devolva A
```

Interpretação do código
- linha 1: árvore vazia
- linhas 2 e 3: cria o conjunto de árvores (floresta)
- linha 4: ordenar as arestas em ordem crescente
- linha 5: loop para verificar cada aresta
- linha 6: verificar se o representante dos vértices u e v é o mesmo. Em outras palavras, é verificado se os dois vértices estão na mesma árvore

- linhas 7 e 8: se passar no teste da linha 6, a aresta é anexada na árvore e as árvores que contêm u e v são unidas.
Cada conjunto pode ser representado por uma lista encadeada. O representante é o primeiro elemento da lista (slide 87).

Cada nó tem um campo que aponta para o representante.

Guarda-se um apontador para o fim da lista.

Ver slide 88.

Complexidade (pseudo-código apresentado no slide 86)
- MAKE-SET: O(|V|)
- Ordenação das arestas: O(|E| log |E|)
- Linhas 5-8: por volta de O(|E|)
- Custo total: O(|E| log |E|)

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Marin, L. O. Grafos: Árvore Geradora Mínima. AE23CP - Algoritmos e Estrutura de Dados II. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.