Florestas geradoras maximais de custo mínimo em grafos dinâmicos

Chung Jin Shian **Orientadora: Cristina Gomes Fernandes**

Departamento de Ciência da Computação, Instituto de Matemática, Estatística e Ciência da Computação,
Universidade de São Paulo

Resumo

Grafos dinâmicos permitem modelar problemas em que o grafo sofre alterações ao longo do tempo. Um dos problemas fundamentais nesse contexto é a manutenção de uma árvore geradora de custo mínimo de um grafo dinâmico. Estudamos vários algoritmos propostos por Holm, de Lichtenberg e Thorup [1] para variantes desse problema. O foco foi no algoritmo para manter uma floresta maximal de custo mínimo (MSF) no contexto decremental, em que se dá suporte eficiente à remoção de arestas. Esse algoritmo foi implementado e testado em grafos dinâmicos com dezenas de milhares de vértices.

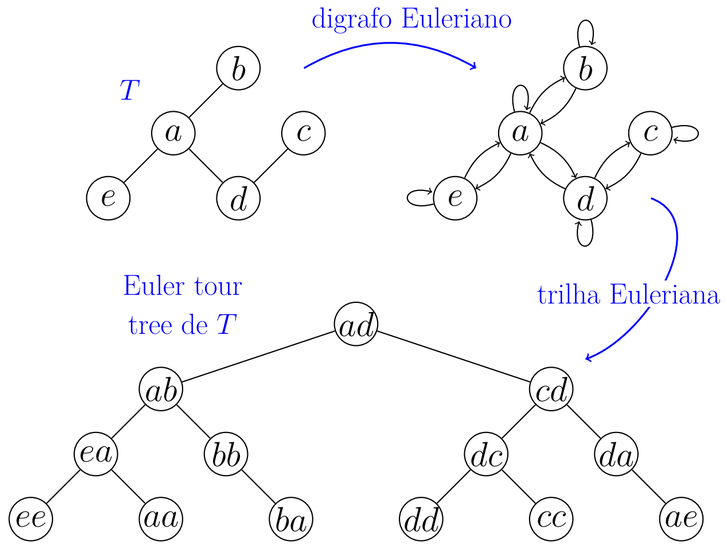
Conexidade em grafos dinâmicos

O problema da conexidade em grafos dinâmicos visa uma implementação eficiente da biblioteca abaixo:

- **`grafoDinâmico(`**n**`)`**: contrói e devolve um grafo dinâmico com n vértices e sem arestas;
- **`conectadosGD(`**G**`, `**u**`, `**v**`)`**: devolve verdadeiro se os vértices u e v estão na mesma componente de G e falso caso contrário;
- **`adicioneGD(`**G**`, `**u**`, `**v**`)`**: adiciona a aresta uv no grafo G;
- **`removaGD(`**G**`, `**u**`, `**v**`)`**: remove a aresta uv do grafo G.

Ideia: Fatiar o grafo G em níveis. Cada aresta de G possui um nível entre 1 e $\lceil \lg n \rceil$, onde n é o número de vértices de G. Uma aresta, ao ser inserida em G, começa com o nível $\lceil \lg n \rceil$ e, durante o algoritmo, seu nível vai sendo decrementado. Seja G_i o subgrafo de G com as arestas de G de nível menor ou igual a i. Para cada nível i, o algoritmo mantém uma floresta maximal F_i de G_i. Além disso, ele mantém também o subgrafo R_i de G com as arestas de nível i que não estão em F_i, chamadas de arestas reserva.

Cada grafo R_i é mantido por suas listas de adjacências. Já cada floresta F_i é mantida em uma estrutura de dados específica para florestas dinâmicas, baseada em Euler tour trees. Cada componente de F_i é armazenada como uma Euler tour tree. Em nossa implementação, Euler tour trees são implementadas como splay trees.



O algoritmo mantém as seguintes invariantes:

- F_i é floresta maximal de G_i para $1 \leq i \leq \lceil \lg n \rceil$;
- $F_i \subseteq F_{i+1}$ para $1 \leq i \leq \lceil \lg n \rceil - 1$;
- Cada componente de F_i tem no máximo 2^i vértices.

Nos pseudocódigos abaixo, esboçamos a remoção de uma aresta do grafo. As Euler tour trees carregam informação extra para que a implementação das linhas 6, 7, 10 e 11 do método `substituaAresta` seja eficiente.

```
removaGD(G, u, v)
Entrada: Recebe dois vértices u e v adjacentes do grafo G.
Efeito: Remove a aresta uv do grafo G.
1  i ← G.nível[u, v]
2  nível[u, v] ← NIL                                  ▷ marcamos uv como removida
3  L ← G.nívelMax
4  se uv ∈ G.F_L então                                ▷ uv é aresta da floresta
5      para j ← i até L faça
6          removaFD(G.F_j, u, v)                      ▷ remove uv da floresta F_j
7      substituaAresta(G, i, u, v)
8  senão                                              ▷ uv é aresta reserva
9      removaLA(G.R_i, u, v)                          ▷ remove uv do grafo R_i
```

```
substituaAresta(G, i, u, v)
Entrada: Recebe dois vértices u e v de componentes distintas do grafo G, e um nível i.
Efeito: Adiciona à floresta, se existir, uma aresta de G substituta para xy, de nível ≥ i.
1  L ← G.nívelMax
2  para j ← i até L faça
3      T_u ← splay(G.F_j.nó[u, u])                    ▷ torna o nó uu raiz da componente de u em F_j
4      T_v ← splay(G.F_j.nó[v, v])                    ▷ torna o nó vv raiz da componente de v em F_j
5      se T_u.tam > T_v.tam então T_u ↔ T_v
6      enquanto T_u.arestasDeNível > 0 faça           ▷ rebaixa as arestas de T_u de nível j
7          nóXY ← procureArestaDeNível(T_u)
8          T_u ← splay(nóXY)
9          rebaixeNívelDaAresta(G, nóXY, j)
10     enquanto T_u.arestasReservasDeNível > 0 faça   ▷ há arestas reserva de nível j
11         nóXX ← procureNóIncideArestaReservaDeNível(T_u)   incidentes a T_u?
12         T_u ← splay(nóXX)
13         (x, x) ← nóXX.vértices
14         para y ∈ G.R_j[x] faça                     ▷ se é substituta, inclui e termina, senão rebaixa
15             se testeSubstituta(G, x, y, j) então retorne
```

```
testeSubstituta(G, x, y, j)
Entrada: Recebe dois vértices adjancentes x e y do grafo G e um nível j.
Saída: Devolve verdadeiro se a aresta xy é substituta e falso caso contrário.
1  removaLA(G.R_j, x, y)
2  se conectadosGD(G, x, y) então                     ▷ a aresta xy não é substituta
3      G.nível[x, y] ← j − 1
4      adicioneLA(G.R_{j−1}, x, y)                    ▷ xy passa a ser reserva de nível j − 1
5      retorne falso
6  senão                                              ▷ a aresta xy é substituta
7      L ← G.nívelMax
8      para k ← j até L faça                          ▷ adiciona xy às florestas F_j, ..., F_L
9          adicioneFD(G.F_k, x, y)
10     retorne verdadeiro
```

Na nossa implementação, que utiliza splay trees, alguns dos métodos têm consumo amortizado por operação.

- **`grafoDinâmico(`**n**`)`**: $O(n \lg n)$;
- **`conectadosGD(`**G**`, `**u**`, `**v**`)`**: amortizado $O(\lg n)$;
- **`adicioneGD(`**G**`, `**u**`, `**v**`)`**: amortizado $O(\lg n)$;
- **`removaGD(`**G**`, `**u**`, `**v**`)`**: amortizado $O(\lg^2 n)$.

MSF decremental

O problema da manutenção de uma MSF decremental do grafo visa uma implementação eficiente dos métodos da biblioteca abaixo:

- **`MSFDecremental(`**n**`, `**E**`)`**: constrói e devolve o grafo ponderado G com n vértices e as arestas ponderadas dadas no conjunto E;
- **`consultePesoMSF(`**G**`)`**: devolve o peso de uma MSF do grafo ponderado G;
- **`removaMSF(`**G**`, `**u**`, `**v**`)`**: remove a aresta uv do grafo ponderado G.

Ideia: No loop das linhas 10 a 15 do método `substituaAresta`, busca-se por uma aresta substituta numa ordem arbitrária. Na implementação da MSF decremental, a ideia é olhar as arestas candidatas à substituta em ordem crescente de peso. Para isso, as listas de adjacências de R_i são substituídas por min-heaps, onde a chave para um vizinho v é o peso da aresta uv. Assim, nas linhas 11 e 14 do `substituaArestaMSF`, basta procurar por uma aresta em R_i que incida em T_u, que tenha o menor peso e que conecte T_u e T_v, para podermos manter o peso mínimo de uma MSF do grafo ao longo das remoções de arestas.

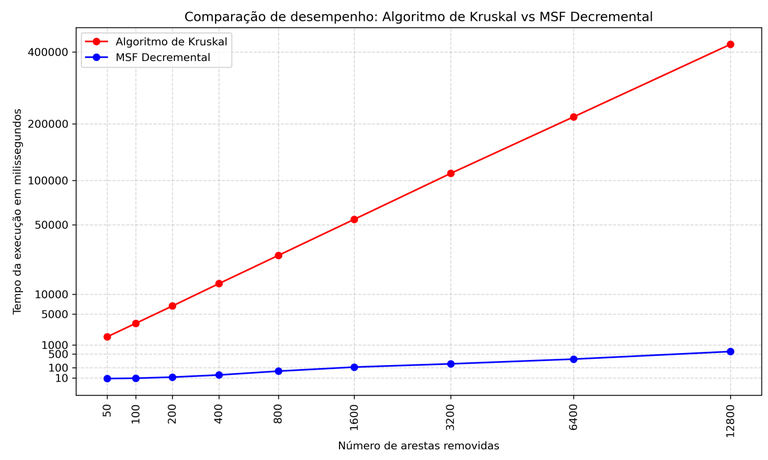
Na figura abaixo, veja que, nas linhas 11 e 14, passamos a procurar uma aresta de peso mínimo em R_i incidente a T_u. As linhas 1 a 10 são idênticas às do método `substituaAresta`.

```
substituaArestaMSF(G, i, u, v)
Entrada: Recebe dois vértices u e v de componentes distintas do grafo G, e um nível i.
Efeito: Adiciona à floresta, se existir, uma aresta de G substituta para xy, de nível ≥ i.
1  L ← G.nívelMax
2  para j ← i até L faça
3      T_u ← splay(G.F_j.nó[u, u])                    ▷ torna o nó uu raiz da componente de u em F_j
4      T_v ← splay(G.F_j.nó[v, v])                    ▷ torna o nó vv raiz da componente de v em F_j
5      se T_u.tam > T_v.tam então T_u ↔ T_v
6      enquanto T_u.arestasDeNível > 0 faça
7          nóXY ← procureArestaDeNível(T_u)
8          T_u ← splay(nóXY)
9          rebaixeNívelDaAresta(G, nóXY, j)
10     enquanto T_u.arestasReservasDeNível > 0 faça
11         nóXX := procureNóIncideArestaDePesoMínimo(R_j, T_u)
12         T_u ← splay(nóXX)
13         (x, x) ← nóXX.vértices
14         (y, w) := consulteMinLAMSF(R_j, x)         ▷ vizinho de x de aresta de peso
15         se testeSubstitutaMSF(G, x, y, j) então retorne   mínimo em R_j
```

Abaixo está o consumo de tempo da implementação:

- **`MSFDecremental(`**n**`, `**E**`)`**: $O(|E| \lg n)$;
- **`consultePesoMSF(`**G**`)`**: $O(1)$;
- **`removaMSF(`**G**`, `**u**`, `**v**`)`**: amortizado $O(\lg^2 n)$.

Testamos a remoção das arestas de um grafo G com 20.000 vértices e mais de 140.000 arestas, realizando uma comparação da performance do algoritmo de Kruskal e da MSF decremental que implementamos.



Arestas removidas	peso da MST	Algoritmo de Kruskal (ms)	MSF Decremental (ms)
50	33264874	1688	8
100	33268603	3379	9
200	33286808	6768	13
400	33331654	13695	24
800	33428133	27408	56
1600	33607175	54915	111
3200	34064910	110084	174
6400	34820142	215978	302
12800	36600884	426767	613

Comentários finais

No texto do TCC, além da descrição detalhada dos algoritmos apresentados aqui bem como de suas análises de consumo de tempo, também descrevemos um algoritmo para o problema da MSF totalmente dinâmico, que dá suporte eficiente não só à remoção de arestas, como também à inserção de arestas. Esse algoritmo faz uso do algoritmo da MSF decremental apresentado acima.

Informações e contato

Para mais informações, acesse a página do trabalho: `https://linux.ime.usp.br/~cjinshian/`

Endereço para contato: `cjinshian77@usp.br`

Referências

[1] Holm, J., de Lichtenberg, K., Thorup, M., "**Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity**," *Journal of the ACM*, 48(4): 723–760, 2001.