

Árvores geradoras mínimas

1. Grafos com pesos nas arestas: $G = (V, E)$, $c: E \rightarrow \mathbb{R}$ (usualmente, $c(e) \geq 0$ para toda $e \in E$)
2. Neste capítulo: G será sempre conexo.
3. Queremos: *Árvore geradora* T de G com $c(E(T))$ mínimo (MST: 'minimum spanning tree')

Algoritmos

- ▷ Algoritmo de Prim
- ▷ Algoritmo de Kruskal
- ▷ Algoritmo de Boruvka

Definições para AGM-GENERIC(G, c)

- ▷ Suponha que A está contida em uma AGM de (G, c) . Uma aresta $e \in E(G) \setminus A$ é *boa para* A se $A \cup \{e\}$ também está contida em uma AGM de G .
- ▷ $(S, V \setminus S)$ *respeita* $A \subset E(G)$ se $A \cap E(S, V \setminus S) = \emptyset$.

Definições para AGM-GENERIC(G, c)

- ▷ Suponha que A está contida em uma AGM de (G, c) . Uma aresta $e \in E(G) \setminus A$ é *boa para* A se $A \cup \{e\}$ também está contida em uma AGM de G .
- ▷ $(S, V \setminus S)$ *respeita* $A \subset E(G)$ se $A \cap E(S, V \setminus S) = \emptyset$.

Propriedade 1. $G = (V, E)$ grafo conexo, $c: E \rightarrow \mathbb{R}$, $A \subset E$ contida em alguma AGM de (G, c) . Seja $(S, V \setminus S)$ um corte que respeita A . Se e tem peso mínimo dentre todas as arestas em $E(S, V \setminus S)$, então e é boa para A .

AGM-GENERIC(G, c)

1. $A \leftarrow \emptyset$
2. **enquanto** A não gera uma AG
3. **faça** encontre uma boa aresta $e = \{u, w\}$ para A
4. $A \leftarrow A \cup \{e\}$
5. **devolva** A

Algoritmo de Kruskal

Algoritmo de Kruskal

- ▷ Escolhemos para e em $\text{AGM-GENERIC}(G, c)$ a aresta de menor custo dentre aquelas arestas que conectam componentes distintos de $(V(G), A)$.

Algoritmo de Kruskal

- ▷ Escolhemos para e em $\text{AGM-GENERIC}(G, c)$ a aresta de menor custo dentre aquelas arestas que conectam componentes distintos de $(V(G), A)$. (Note que e é uma aresta boa.)

Algoritmo de Kruskal

- ▷ Escolhemos para e em $\text{AGM-GENERIC}(G, c)$ a aresta de menor custo dentre aquelas arestas que conectam componentes distintos de $(V(G), A)$. (Note que e é uma aresta boa.)

Correção decorre da Propriedade 1.

Algoritmo de Kruskal

- ▷ Escolhemos para e em $\text{AGM-GENERIC}(G, c)$ a aresta de menor custo dentre aquelas arestas que conectam componentes distintos de $(V(G), A)$. (Note que e é uma aresta boa.)

Correção decorre da Propriedade 1.

AA/Coelho: propriedade da escolha gulosa!

Algoritmo de Kruskal, implementação

Algoritmo de Kruskal, implementação

1. Ordene as arestas de G de forma que $c(e_1) \leq \dots \leq c(e_m)$

Algoritmo de Kruskal, implementação

1. Ordene as arestas de G de forma que $c(e_1) \leq \dots \leq c(e_m)$
2. **faça** $A \leftarrow \emptyset$

Algoritmo de Kruskal, implementação

1. Ordene as arestas de G de forma que $c(e_1) \leq \dots \leq c(e_m)$
2. **faça** $A \leftarrow \emptyset$
3. **para cada** $i = 1, \dots, m$, **faça**
4. **se** $(V(G), A \cup \{e_i\})$ é acíclico, **então** $A \leftarrow A \cup \{e_i\}$

Algoritmo de Kruskal, implementação

1. Ordene as arestas de G de forma que $c(e_1) \leq \dots \leq c(e_m)$
2. **faça** $A \leftarrow \emptyset$
3. **para cada** $i = 1, \dots, m$, **faça**
4. **se** $(V(G), A \cup \{e_i\})$ é acíclico, **então** $A \leftarrow A \cup \{e_i\}$
5. **devolva** $(V(G), A)$

Algoritmo de Kruskal, implementação (cont.)

Algoritmo de Kruskal, implementação (cont.)

1. Ordenação de $E(G)$: podemos usar ...

Algoritmo de Kruskal, implementação (cont.)

1. Ordenação de $E(G)$: podemos usar $\dots (O(m \log m) = O(m \log n))$
2. Como saber se $(V(G), A \cup \{e_i\})$ é acíclico?

Algoritmo de Kruskal, implementação (cont.)

1. Ordenação de $E(G)$: podemos usar ... ($O(m \log m) = O(m \log n)$)
2. Como saber se $(V(G), A \cup \{e_i\})$ é acíclico?
 - ▶ Algoritmos e estruturas de dados para a “evolução” de partições de conjuntos (“*Union-Find*”):

Algoritmo de Kruskal, implementação (cont.)

1. Ordenação de $E(G)$: podemos usar $\dots (O(m \log m) = O(m \log n))$
2. Como saber se $(V(G), A \cup \{e_i\})$ é acíclico?
 - ▷ Algoritmos e estruturas de dados para a “evolução” de partições de conjuntos (“*Union-Find*”): suponhamos que temos as funções $\text{UFfind}(u, v)$ e $\text{UFunion}(u, v)$.

Programa 20.5, Algoritmo de Kruskal

Programa 20.5, Algoritmo de Kruskal

```
void GRAPHmstE(Graph G, Edge mst[])
{
    int i, k; Edge a[maxE];
    int E = GRAPHedges(a, G);
    sort(a, 0, E-1);
    UFininit(G->V);
    for (i = 0, k = 0; i < E && k < G->V-1; i++)
        if (!UFfind(a[i].v, a[i].w))
        {
            UFunion(a[i].v, a[i].w);
            mst[k++] = a[i];
        }
}
```

Algoritmos e EDs para Union-Find

Algoritmos e EDs para Union-Find

Queremos $\text{UFfind}(u, v)$ e $\text{UFunion}(u, v)$.

Algoritmos e EDs para Union-Find

Queremos $\text{UFfind}(u, v)$ e $\text{UFunion}(u, v)$.

Problema: Suponha que recebemos arestas $e_i = (p_i, q_i)$ ($0 \leq i < M$) com $0 \leq p_i, q_i < N$, e um par (p, q) ($0 \leq p, q < N$).

Algoritmos e EDs para Union-Find

Queremos $\text{UFfind}(u, v)$ e $\text{UFind}(u, v)$.

Problema: Suponha que recebemos arestas $e_i = (p_i, q_i)$ ($0 \leq i < M$) com $0 \leq p_i, q_i < N$, e um par (p, q) ($0 \leq p, q < N$). Queremos saber se p e q estão no mesmo componente conexo do grafo sobre $0, \dots, N-1$ com arestas $e_1, \dots, e_M$.

Algoritmos e EDs para Union-Find

Queremos `UFfind(u,v)` e `UFunction(u,v)`.

Problema: Suponha que recebemos arestas $e_i = (p_i, q_i)$ ($0 \leq i < M$) com $0 \leq p_i, q_i < N$, e um par (p, q) ($0 \leq p, q < N$). Queremos saber se p e q estão no mesmo componente conexo do grafo sobre $0, \dots, N-1$ com arestas $e_1, \dots, e_M$. Fazemos

```
for(i = 0; i < M; i++) UFunction(p[i],q[i]);
```

e chamamos `UFfind(p,q)`.

Programa 1.1, Union-Find (Quick-find)

Programa 1.1, Union-Find (Quick-find)

```
/* Este programa gera uma floresta geradora */
#include <stdio.h>
#define N 10000
main()
{ int i, p, q, t, id[N];
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d\n", &p, &q) == 2)
  {
    if (id[p] == id[q]) continue;
    for (t = id[p], i = 0; i < N; i++)
      if (id[i] == t) id[i] = id[q];
    printf(" %d %d\n", p, q);
  }
}
```

Exemplo para union-find

Exemplo para union-find

p: 3 4 8 2 5 2 5 7 4 5 0 6
q: 4 9 0 3 6 9 9 3 8 6 2 1

Programa 1.2, Union-Find (Quick-union)

Programa 1.2, Union-Find (Quick-union)

```
/* Este programa gera uma floresta geradora */
#include <stdio.h>
#define N 10000
main()
{ int i, p, q, t, id[N];
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d\n", &p, &q) == 2) {
    for (i = p; i != id[i]; i = id[i]) ;
    for (j = q; j != id[j]; j = id[j]) ;
    if (i == j) continue;
    id[i] = j;
    printf(" %d %d\n", p, q);
  }
}
```

Programa 1.3, Union-find (Quick-union com pesos)

Programa 1.3, Union-find (Quick-union com pesos)

```
#define N 10000
main()
{ int i, j, p, q, id[N], sz[N];
  for (i = 0; i < N; i++) { id[i] = i; sz[i] = 1; }
  while (scanf("%d %d\n", &p, &q) == 2) {
    for (i = p; i != id[i]; i = id[i]) ;
    for (j = q; j != id[j]; j = id[j]) ;
    if (i == j) continue;
    if (sz[i] < sz[j])
      { id[i] = j; sz[j] += sz[i]; }
    else { id[j] = i; sz[i] += sz[j]; }
    printf(" %d %d\n", p, q);
  }
}
```

Quick-union com pesos

Quick-union com pesos

Propriedade 2 (Propriedade 1.3). *Seja F uma floresta computada pelo Programa 1.3 (quick-union com pesos). Se x pertence a uma árvore com i vértices, então a sua distância à raiz dessa árvore é no máximo $\log_2 i$.*

Demonstração. (Esboço) Provamos que esta propriedade se preserva ao unirmos duas árvores em F . Suponha que estas duas árvores têm $i \leq j$ vértices. Temos

$$1 + \log_2 i \leq \log_2(i + i) \leq \log(i + j),$$

e o resultado segue. □

Quick-union com pesos

Propriedade 2 (Propriedade 1.3). *Seja F uma floresta computada pelo Programa 1.3 (quick-union com pesos). Se x pertence a uma árvore com i vértices, então a sua distância à raiz dessa árvore é no máximo $\log_2 i$.*

Demonstração. (Esboço) Provamos que esta propriedade se preserva ao unirmos duas árvores em F . Suponha que estas duas árvores têm $i \leq j$ vértices. Temos

$$1 + \log_2 i \leq \log_2(i + i) \leq \log(i + j),$$

e o resultado segue. □

Conseqüência: complexidade do Programa 1.3 é $O(M \log N)$.

Quick-union com pesos

Propriedade 2 (Propriedade 1.3). *Seja F uma floresta computada pelo Programa 1.3 (quick-union com pesos). Se x pertence a uma árvore com i vértices, então a sua distância à raiz dessa árvore é no máximo $\log_2 i$.*

Demonstração. (Esboço) Provamos que esta propriedade se preserva ao unirmos duas árvores em F . Suponha que estas duas árvores têm $i \leq j$ vértices. Temos

$$1 + \log_2 i \leq \log_2(i + i) \leq \log(i + j),$$

e o resultado segue. □

Consequência: complexidade do Programa 1.3 é $O(M \log N)$. Há algoritmos para union-find *mais* eficientes! (Mas não veremos isto.)

Programa 20.5 (Kruskal), Complexidade

Programa 20.5 (Kruskal), Complexidade

Propriedade 3 (Propriedade 20.9). *Considere o Programa 20.5, e suponha que implementamos os algoritmos de union-find com quick-union com pesos e que as arestas de G são ordenadas por algum algoritmo de complexidade $O(m \log m)$. Então a complexidade do Programa 20.5 (Kruskal) será $O(m \log n)$.*

Programa 20.5 (Kruskal), Complexidade

Propriedade 3 (Propriedade 20.9). *Considere o Programa 20.5, e suponha que implementamos os algoritmos de union-find com quick-union com pesos e que as arestas de G são ordenadas por algum algoritmo de complexidade $O(m \log m)$. Então a complexidade do Programa 20.5 (Kruskal) será $O(m \log n)$.*

Observação: exemplo básico de algoritmo guloso (“propriedade da escolha gulosa”).

Programa 20.5 (Kruskal), Complexidade

Propriedade 3 (Propriedade 20.9). *Considere o Programa 20.5, e suponha que implementamos os algoritmos de union-find com quick-union com pesos e que as arestas de G são ordenadas por algum algoritmo de complexidade $O(m \log m)$. Então a complexidade do Programa 20.5 (Kruskal) será $O(m \log n)$.*

Observação: exemplo básico de algoritmo guloso (“propriedade da escolha gulosa”). “Matróides”

Programa 20.5 (Kruskal), Complexidade prática

Programa 20.5 (Kruskal), Complexidade prática

1. quicksort()

Programa 20.5 (Kruskal), Complexidade prática

1. quicksort()
2. Pesos inteiros: algoritmos de ordenação linear

Programa 20.5 (Kruskal), Complexidade prática

1. quicksort()
2. Pesos inteiros: algoritmos de ordenação linear
3. Quick-union com pesos e ‘compressão de caminhos’ (não definido): complexidade $O(m \log^* n)$, onde

$$\log^* n = \min\{i \geq 0 : \log_2^{(i)} n \leq 1\}.$$

($\log^* 2 = 1$, $\log^* 2^2 = 2$, $\log^* 2^{2^2} = 3$, $\log^* 2^{2^{2^2}} = \log^* 65536 = 4$, e $\log^* 2^{65536} = 5$, ...)

Programa 20.5 (Kruskal), Complexidade prática

1. quicksort()
2. Pesos inteiros: algoritmos de ordenação linear
3. Quick-union com pesos e ‘compressão de caminhos’ (não definido): complexidade $O(m \log^* n)$, onde

$$\log^* n = \min\{i \geq 0 : \log_2^{(i)} n \leq 1\}.$$

($\log^* 2 = 1$, $\log^* 2^2 = 2$, $\log^* 2^{2^2} = 3$, $\log^* 2^{2^{2^2}} = \log^* 65536 = 4$, e $\log^* 2^{65536} = 5$, ...)

4. Fila de prioridade (primeira prioridade: aresta mais barata)

AGMs no Stanford GraphBase

AGMs no Stanford GraphBase

▷ `miles_span`