Projeto e Análise em Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

21 de maio de 2019

Arvore Geradora Mínima

Contextualição

- Considere o seguinte problema: Uma universidade possui vários campi no estado da Bahia, e deseja interligá-los usando o menor custo de fibra.
- Este problema pode ser modelado através de um grafo não orientado, onde os vértices representam cada campus e as arestas as ligações entre os campi.
- Cada aresta deve ser ponderada pelo custo da ligação, ou seja, o custo da fibra ótica, neste exemplo.

Árvore Geradora Mínima

- Para resolver o problema, precisamos encontrar um SUBGRAFO GERADOR, *conexo* (para garantir a interligação entre todos os campi) e cuja soma dos custos de suas arestas seja o menor possível.
- O grafo original precisa ser conexo, senão não teríamos um subgrafo conexo. Vamos considerar que nosso problema sempre apresenta um grafo conexo.
- O subgrafo gerador procurado é uma árvore, pois se houvesse um ciclo significa que existem dois vértices com dois caminhos diferentes ligando-os, neste caso, eliminar um caminho tornaria um custo menor.

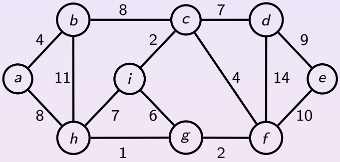
O Problema da Árvore Geradora Mínima

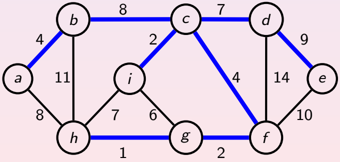
- Entrada: Grafo conexo $G = (V, E)$ com pesos $w(u, v)$ para cada aresta $\{u, v\}$.
- Saída: Subgrafo gerador conexo T de G, cujo peso total

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

seja o menor possível.

Exemplo


a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
8
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
8
10
1
2

Estratégia gulosa

- De uma forma genérica, a abordagem no problema envolve um algoritmo guloso. A escolha da aresta que constrói a árvore é construída de forma incremental.
- O conjunto de arestas A escolhido mantém o seguinte invariante:
  - No início de cada iteração, A está contido em uma Árvore Geradora Mínima.
  - Em cada iteração, determina-se uma aresta (u, v) tal que $A' = A \cup \{(u, v)\}$ também satisfaz o invariante.
- A aresta determinada é chamada de *aresta segura*
- Dois algoritmos utilizam esta abordagem:
  1. Algoritmo de Prim
  2. Algoritmo de Kruskal

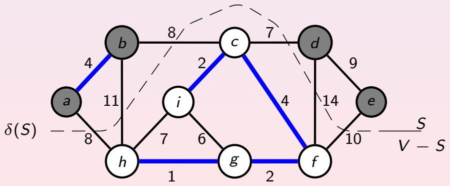
Algoritmo Guloso

Algoritmo $\mathrm{AGM}(G, w)$
  $A \leftarrow 0$
  enquanto A não é uma árvore geradora **faça**
    Encontre aresta (u, v) segura para A
    $A \leftarrow A \cup \{(u, v)\}$
  retorne A

- O *algoritmo* está correto!
- Como o *laço* somente para se A for uma árvore geradora, então em cada iteração A está contido em uma AGM que chamaremos de T. Logo, existe uma *aresta segura* (u, v) em $T - A$.
- Para definir o algoritmo é necessário especificar como encontrar uma "aresta segura".

Encontrando uma aresta segura

- Vamos considerar o grafo $G(V, E)$ e tomamos $S \subset V$.
- Considere $\delta(S)$ o corte de S, ou seja, o conjunto de arestas que têm um extremo em S e outro em $V - S$.
- Seja $A \subset E$ contido em T, uma árvore geradora mínima.
- $\delta(S)$ respeita A se $A \cap \delta(S) = \emptyset$. Ou seja, o corte de S não contém nenhuma aresta de A.

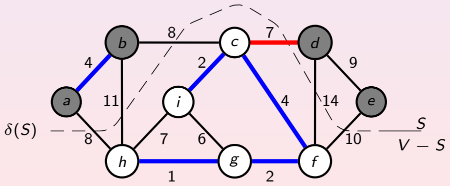


Achando uma aresta segura

Uma aresta do corte é *leve*, se tiver o menor peso entre as arestas.

Teorema

Seja G um grafo com pesos nas arestas dado por w. Seja A um subconjunto de arestas contido em uma Árvore Geradora Mínima. Seja $\delta(S)$ um corte que respeita A e (u, v) uma aresta leve deste corte. Então (u, v) é uma aresta segura.



Achando uma aresta segura

Demonstração.

Vamos supor T uma árvore geradora mínima e $A \subset E(T)$, com $(u, v) \notin E(T)$. vamos tomar P o único caminho de u a v em T. escolhemos

$$(x, y) = \{e : e \in E(P) \cap \delta(S) | w(x, y) = \max(w(e))\}$$

uma aresta do corte $\delta(S)$ em P. Vamos fazer $A' = E(T) - (x, y) \cup \{(u, v)\}$. Ficamos com: $w(A') = w(E(T)) - w(x, y) + w(u, v)$, como (u, v) é leve no corte, então $w(u, v) \leqslant w(x, y)$. Logo: $w(A') \leqslant w(E(T))$. $T' = G(V, A')$ é também uma árvore geradora mínima e contém a aresta (u, v). Como $(x, y) \notin A$, $A \cup \{(u, v)\} \subset E(T')$. (u, v) é aresta segura. $\square$

Algoritmos Prim e Kruskal

- Ambos algoritmos utilizam esta estratégia gulosa. Eles se baseiam no seguinte corolário ao teorema anterior:

Corolário

Seja G um grafo com pesos nas arestas dado por w. Seja A um subconjunto de arestas contido em uma Árvore Geradora Mínima. Seja C um componente (árvore) de $G_A = (V, A)$. Se (u, v) é uma aresta leve de $\delta(V(C))$, então (u, v) é segura para A.

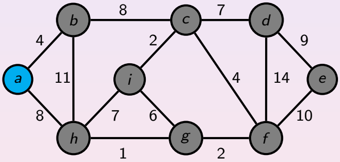
Demonstração.

Imediato a partir do teorema. $\square$

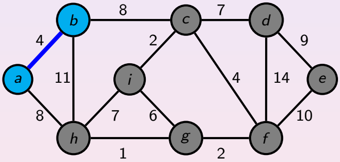
PRIM: Aplicando a escolha gulosa no corte

- A é o conjunto de arestas de uma árvore com raiz r (escolhido arbitrariamente no início).
- Inicialmente $A = \emptyset$
- A cada iteração, o algoritmo define o corte $\delta(V(C))$, onde C é o conjunto de vértices definido pela árvore de raiz r.
- Ele encontra uma aresta leve (u, v) neste corte e acrescenta-a ao conjunto A, agregando o extremo de (u, v) que não pertença à árvore com raiz em r como nova folha da árvore.
- O algoritmo repete a iteração até que $T = T(C, A)$ seja uma árvore geradora do grafo original, neste caso, $V \equiv C$.
- A dificuldade no algoritmo é encontrar eficientemente uma aresta leve no corte.

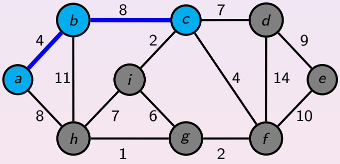
Aplicando o algoritmo



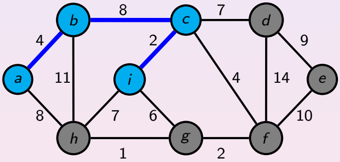
Aplicando o algoritmo



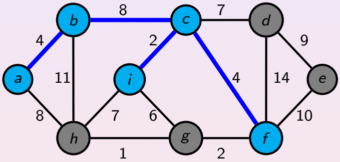
Aplicando o algoritmo



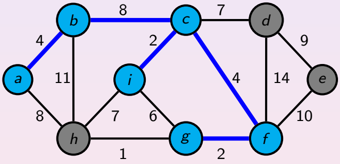
Aplicando o algoritmo



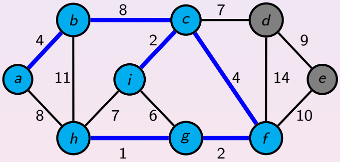
Aplicando o algoritmo



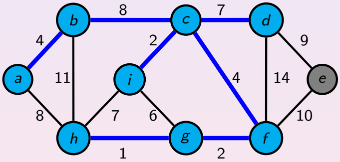
Aplicando o algoritmo



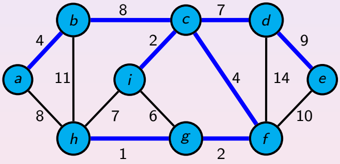
Aplicando o algoritmo



Aplicando o algoritmo



Aplicando o algoritmo



Características do Algoritmo

Informações definidas no algoritmo:

- Fila Q com todos os vértices
- Cada vértice v em Q tem uma chave $key[v]$ que indica o menor peso de qualquer aresta que o liga a algum vértice da árvore. Se não existir, $key[v] = \infty$
- A variável $\pi[u]$ indica o pai de u na árvore.
- A árvore $T = T(C, A)$ é definida pelo conjunto de arestas $A = \{(u, \pi[u]) : u \in V - \{r\} - Q\}$

O algoritmo

Algoritmo $\textsc{Prim}(G, w, r)$
    para $u \in V(G)$ **faça**
        $key[u] \leftarrow \infty$
        $\pi[u] \leftarrow nulo$
    $key[r] \leftarrow 0$
    $Q \leftarrow V(G)$
    enquanto $Q \neq \emptyset$ **faça**
        $u \leftarrow ExtrairMinimo(Q)$
        para $v \in Adj[u]$ **faça**
            se $v \in Q$ **e** $w(u, v) < key[v]$ **então**
                $\pi[v] \leftarrow u$
                $key[v] \leftarrow w(u, v)$

Corretude do Algoritmo

Invariantes:

- No início de cada iteração do laço 7–12:
  - $A = \{(u, \pi[u]) : u \in V - \{r\} - Q\}$.
  - O conjunto de vértices da árvore é exatamente $V[G] - Q$.
  - Para cada $v \in Q$, se $\pi[v] \neq nulo$, então $key[v]$ é o peso de uma aresta $(v, \pi[v])$ de menor peso ligando v a um vértice $\pi[v]$ na árvore.
- Esse invariantes garantem que o algoritmo sempre escolhe uma aresta segura para acrescentar a A e portanto, o algoritmo está correto.

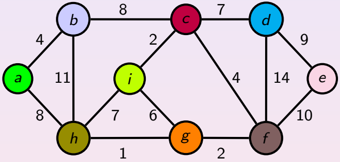
Complexidade do Algoritmo

- A complexidade do algoritmo depende de como a fila Q é implementada.
- Considere Q como um min-heap.
- As linhas 2–6 podem ser executadas em tempo $O(V)$.
- O laço da linha 7 é executado $|V|$ vezes e cada operação *ExtrairMinimo* consome tempo $O(\log V)$, resultando em um tempo total $O(V \log V)$ para todas as chamadas de *ExtrairMinimo*.
- O laço das linhas 9–12 é executado $O(E)$ vezes no total.
- Ao atualizar a chave de um vértice na linha 11 é feita uma atualização na fila que consome tempo $O(\log V)$.
- O tempo total é $O(V \log V + E \log V) = O(E \log V)$.

Kruskal: Aplicando a escolha gulosa no corte

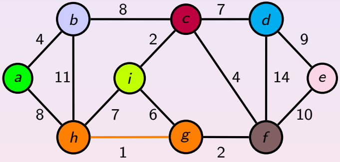
- T representa cada árvore da floresta com todos vértices, e inicialmente sem arestas. $A = \emptyset$
- para o vértice u, $T[u] = u$.
- S é um "saco" com $E(G)$ que não estão em A. Inicialmente S contém todo $E(G)$.
- A cada iteração, o algoritmo escolhe a aresta $<u, v>$ de menor peso em "S". Se u e v estão em árvores distintas ($T[u] \neq T[v]$), então $<u, v>$ pertence a um corte $\delta(V(C))$, que respeita A, logo pode ser inserida em A, C é o conjunto de vértices da árvore de u.
- $<u, v>$ sendo leve (é a mais leve do saco), todos vértices que eram $T[v]$ passam a ser $T[u]$.
- O algoritmo repete a iteração a floresta contenha apenas uma árvore.

Aplicando o algoritmo



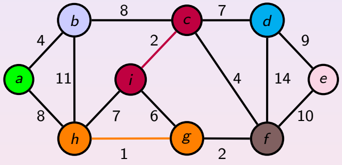
$A = \{\ \} \rightarrow |A| = 0, |V| = 9$
saco = $\{<g, h>, <c, i>, <f, g>, <a, b>, <c, f>, <g, i>, <c, d>, <h, i>, <a, h>, <b, c>, <d, e>, <e, f>, <b, h>, <d, f>\}$

Aplicando o algoritmo



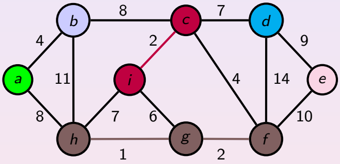
$A = \{<g,h>\} \rightarrow |A| = 1, |V| = 9$
saco = $\{<c,i>, <f,g>, <a,b>, <c,f>, <g,i>, <c,d>, <h,i>, <a,h>, <b,c>, <d,e>, <e,f>, <b,h>, <d,f>\}$

Aplicando o algoritmo



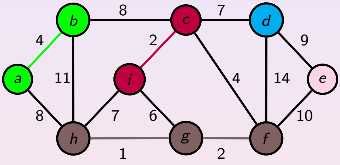
$A = \{<g, h>,\ <c, i>\} \rightarrow |A| = 2, |V| = 9$
saco = $\{<f, g>,\ <a, b>,\ <c, f>,\ <g, i>,\ <c, d>,\ <h, i>,\ <a, h>,\ <b, c>,\ <d, e>,\ <e, f>,\ <b, h>,\ <d, f>\}$

Aplicando o algoritmo



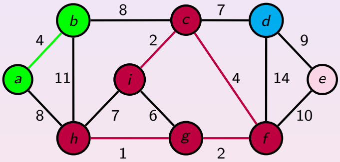
$A = \{\texttt{<}g, h\texttt{>},\ \texttt{<}c, i\texttt{>},\ \texttt{<}f, g\texttt{>}\} \to |A| = 3, |V| = 9$
saco = $\{\texttt{<}a, b\texttt{>},\ \texttt{<}c, f\texttt{>},\ \texttt{<}g, i\texttt{>},\ \texttt{<}c, d\texttt{>},\ \texttt{<}h, i\texttt{>},\ \texttt{<}a, h\texttt{>},\ \texttt{<}b, c\texttt{>},\ \texttt{<}d, e\texttt{>},\ \texttt{<}e, f\texttt{>},$
$\texttt{<}b, h\texttt{>},\ \texttt{<}d, f\texttt{>}\}$

Aplicando o algoritmo



$A = \{<a, b>,\ <g, h>,\ <c, i>,\ <f, g>\} \rightarrow |A| = 4, |V| = 9$
saco = $\{<c, f>,\ <g, i>,\ <c, d>,\ <h, i>,\ <a, h>,\ <b, c>,\ <d, e>,\ <e, f>,\ <b, h>,\ <d, f>\}$

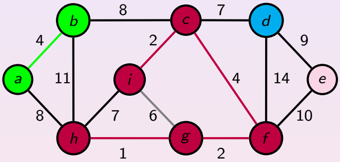
Aplicando o algoritmo



$A = \{<c, f>,\ <a, b>,\ <g, h>,\ <c, i>,\ <f, g>\} \rightarrow |A| = 5, |V| = 9$

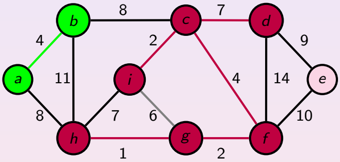
saco = $\{<g, i>,\ <c, d>,\ <h, i>,\ <a, h>,\ <b, c>,\ <d, e>,\ <e, f>,\ <b, h>,\ <d, f>\}$

Aplicando o algoritmo



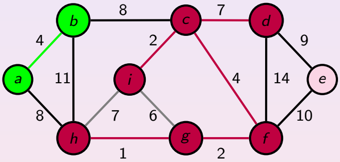
$A = \{<c, f>,\ <a, b>,\ <g, h>,\ <c, i>,\ <f, g>\} \rightarrow |A| = 5, |V| = 9$
saco = $\{<c, d>,\ <h, i>,\ <a, h>,\ <b, c>,\ <d, e>,\ <e, f>,\ <b, h>,\ <d, f>\}$

Aplicando o algoritmo



$A = \{<c,d>,\ <c,f>,\ <a,b>,\ <g,h>,\ <c,i>,\ <f,g>\} \rightarrow |A| = 6, |V| = 9$
saco = $\{<h,i>,\ <a,h>,\ <b,c>,\ <d,e>,\ <e,f>,\ <b,h>,\ <d,f>\}$

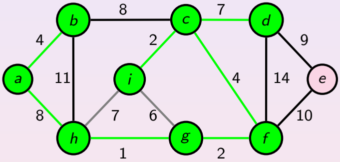
Aplicando o algoritmo



$A = \{<c,d>,\ <c,f>,\ <a,b>,\ <g,h>,\ <c,i>,\ <f,g>\} \rightarrow |A| = 6, |V| = 9$

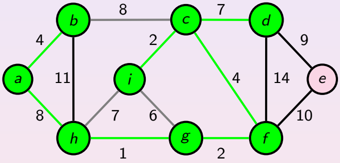
$\texttt{saco} = \{<a,h>,\ <b,c>,\ <d,e>,\ <e,f>,\ <b,h>,\ <d,f>\}$

Aplicando o algoritmo



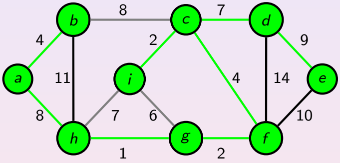
$A = \{<a, h>,\ <c, d>,\ <c, f>,\ <a, b>,\ <g, h>,\ <c, i>,\ <f, g>\} \rightarrow |A| = 7, |V| = 9$
`saco` = $\{<b, c>,\ <d, e>,\ <e, f>,\ <b, h>,\ <d, f>\}$

Aplicando o algoritmo



$A = \{<a,h>,\ <c,d>,\ <c,f>,\ <a,b>,\ <g,h>,\ <c,i>,\ <f,g>\} \rightarrow |A| = 7, |V| = 9$
saco = $\{<d,e>,\ <e,f>,\ <b,h>,\ <d,f>\}$

Aplicando o algoritmo



$A = \{<d,e>,\ <a,h>,\ <c,d>,\ <c,f>,\ <a,b>,\ <g,h>,\ <c,i>,$
$<f,g>\} \rightarrow |A| = 8, |V| = 9$
`saco =` $\{<e,f>,\ <b,h>,\ <d,f>\}$
FIM DO ALGORITMO!!!

Características do Algoritmo

Informações definidas no algoritmo:

- Fila de prioridade S com todas as arestas
- Cada vértice u em $V(G)$ $T[u] \leftarrow u$ que indica que existem $|V|$ árvores na floresta definida por T
- Para cada aresta <u, v> da fila S, se $T[u]! = T[v]$, para todo $x \in V(G)$, se $T[x] = v$, então $T[x] \leftarrow u$. A floresta diminui em 1 a quantidade de árvores.
- O algoritmo termina quando a quantidade de árvores é 1.

O algoritmo

Algoritmo $\textsc{Kruskal}(G, w, r)$
    para $u \in V(G)$ **faça**
        $T[u] \leftarrow u$
    $S \leftarrow E(G)$ S é fila de prioridade
    $A \leftarrow 0$
    enquanto $A < (|V| - 1)$ **faça**
        $e \leftarrow ExtrairMinimo(S)$
        se $T[e.u] \neq T[e.v]$ **então**
            para $x \in V(G)$ **faça**
                se $T[x] = T[e.v]$ **então**
                    $T[x] \leftarrow T[e.u]$
            $A \leftarrow A + 1$

Corretude do Algoritmo

Invariantes:

- No início de cada iteração do laço 8–14:
  - $A < |V| - 1$ representa o número de arestas na Floresta que possui as árvores indicadas por T, e $|V| - A$ o número de árvores na Floresta (todos vértices estão na floresta).
  - S indica no início da fila a aresta mais leve que ainda não pertence à Floresta.
- Esse invariantes garantem que o algoritmo sempre escolhe uma aresta leve para acrescentar a A, e esta aresta somente será acrescida a A, se ela ligar árvores distintas, logo não cria ciclos e é portanto segura.
- O algoritmo está correto.

Complexidade do Algoritmo

- A complexidade do algoritmo depende de como a fila Q é implementada.
- Considere Q como um min-heap.
- O laço da linha 2 pode ser executado em tempo $O(V)$.
- A linha 4 pode ser executada em tempo $O(E)$ se for usado o `make_heap`;
- O laço da linha 6 é executado no máximo E vezes.
- A linha 7 é em tempo $\Theta(1)$.
- O laço da linha 8 é em tempo $O(V)$
- Portanto o laço da linha 6 é executado em tempo $O(EV)$.
- O tempo total é $O(V + E + EV) = O(EV)$.

Exemplo: URI 1152 - Estradas Escuras

Nestes dias se pensa muito em economia, mesmo em Byteland. Para reduzir custos operacionais, o governo de Byteland decidiu otimizar a iluminação das estradas. Até agora, todas as rotas eram iluminadas durante toda noite, o que custava 1 Dólar Byteland por metro a cada dia. Para economizar, eles decidiram não iluminar mais todas as estradas e desligar a iluminação de algumas delas. Para ter certeza que os habitantes de Byteland continuem a se sentirem seguros, eles querem otimizar o sistema de tal forma que após desligar a iluminação de algumas estradas à noite, sempre existirá algum caminho iluminado de qualquer junção de Byteland para qualquer outra junção.
Qual é a quantidade máxima de dinheiro que o governo de Byteland pode economizar, sem fazer os seus habitantes sentirem-se inseguros?

Entrada:

A entrada contém vários casos de teste. Cada caso de teste inicia com dois números **m** ($1 \leqslant$**m**$\leqslant 200000$) e **n** (**m**$-1 \leqslant$**n**$\leqslant 200000$), que são o número de junções de Byteland e o número de estradas em Byteland, respectivamente. Seguem **n** conjuntos de três valores inteiros, **x**, **y** e **z**, especificando qual será a estrada bidirecional entre **x** e **y** com **z** metros ($0 \leqslant$**x**, **y**$<$**m** e **x**$\neq$**y**).
A entrada termina com **m**$=$**n**$=0$. O grafo especificado em cada caso de teste é conectado. O tamanho total de todas as estradas em cada caso de teste é menor do que 231.

Saída:

Para cada caso de teste imprima uma linha contendo a máxima quantidade diária de dólares de Byteland que o governo pode economizar.

Declarando os tipos

- T é um vetor de inteiros indicando a árvore que cada vértice pertence (os vértices são os índices do vetor).
- E é um vetor de arestas cada aresta é indicada por uma estrutura par: `<w,<u,v>>`, onde w é o peso da aresta e u e v os vértices da aresta.
- s é uma fila de prioridade de arestas ordenadas pela função *compare*
- total contém o custo de todas arestas do grafo e fim apenas da agm. O resultado desejado é total-fim.

Solução:

```
#include <iostream>
#include <utility>
#include <queue>
#include <vector>

using namespace std;

class compare {
public:
 bool operator() (const pair<int, pair<int,int> >& a,
                  const pair<int, pair<int,int> >& b) const {
   bool res;
   res = a.first > b.first;
   if(a.first == b.first) {
    res = a.second.first > b.second.first;
    if(a.second.first == b.second.first)
     res = a.second.second > b.second.second;
   }
   return res;
 }
};
```

$T[u] = u$ e $S \leftarrow E(G)$

```
int main() {
int n, m, eu, ev, ew, total, fim;
 cin >> n >> m;
 while(n) {
  vector<int> T;
  vector<pair<int, pair<int, int> > > E;
  for(int i = 0; i < n; i++)
   T.push_back(i);
  total = 0;
  for(int i = 0; i < m; i++) {
   cin >> eu >> ev >> ew;
   if (ev < eu) E.push_back(make_pair(ew,make_pair(ev,eu)));
   else E.push_back(make_pair(ew,make_pair(eu,ev)));
   total+=ew;
  }
  priority_queue<pair<int, pair<int, int> >,
                 vector<pair<int, pair<int, int> > >, compare>
                 s(E.begin(),E.end());
```

ExtrairMinimo e agregar árvores (colocando ‘e’ em A)

```
   int a = 0;
   fim = 0;
   while(a < (n-1)) {
    auto e = s.top(); s.pop();
    if(T[e.second.first] != T[e.second.second]) {
     int oldT = T[e.second.second];
     for(int x=0; x<n; x++)
      if(T[x] == oldT) {
       T[x] = T[e.second.first];
      }
     fim+=e.first;
     a++;
    }
   }
   cout << (total-fim) << endl;
   cin >> n >> m;
  }
  return 0;
}
```