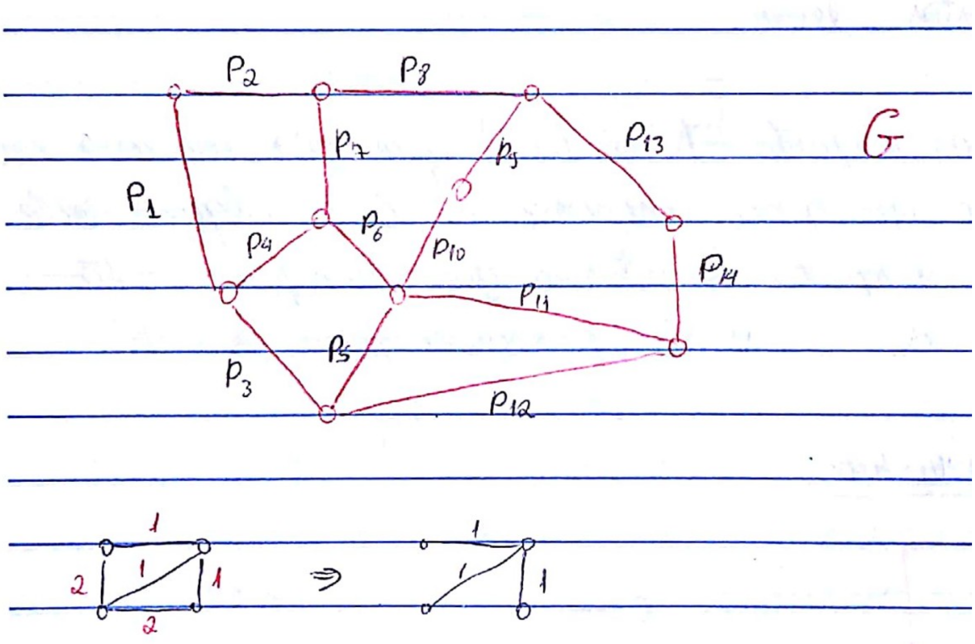
>> Grafos

> Arvore Geradora Mínima



Algoritmos clássicos: Prim e Kruskal
- Invariante mantida por ambos os algoritmos: O conjunto A c E(G) é subconjunto de arestas de alguma arvore geradora mínima de G.
- A cada iteração é escolhida uma aresta de E(G)\A considerada segura para manter a invariante ao entrar em A.

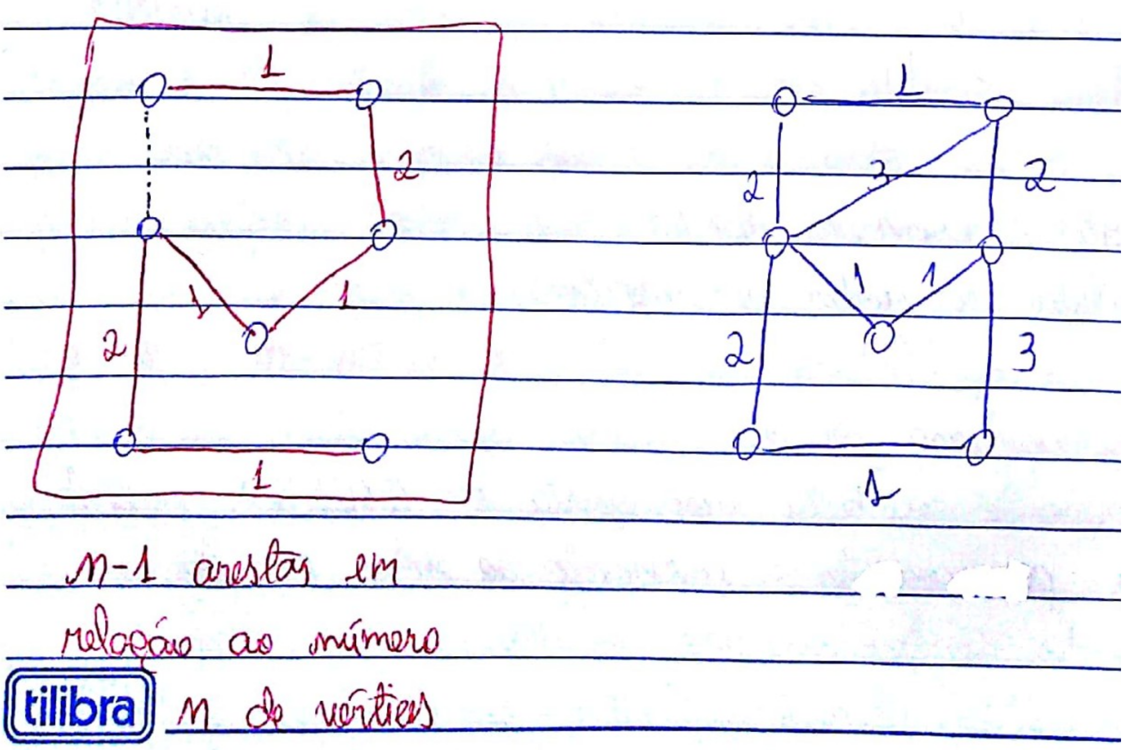
Definição: dado um grafo G = (V, E) e um subconjunto S q V não vazio, denominamos o conjunto de arestas com uma extremidade em S e a outra em V\S de corte de arestas denotado por [S, V – S]

Definição: Dado um conjunto de arestas A e um corte [S, V – S], ambos do mesmo grafo, dizemos que o corte respeita A se A interseção [S, V – S] = Vazio.

Definição: Dado um corte [S, V – S] de um grafo ponderado em arestas, as arestas de peso mínimo no corte são chamadas de arestas leves.

Teorema: Dado um conjunto A c E(G) que seja incluído em alguma arvore geradora mínima de G e algum corte [S, V – S] que respeita A, temos que qualquer aresta leve (u, v) de [S, V – S] é segura para A.

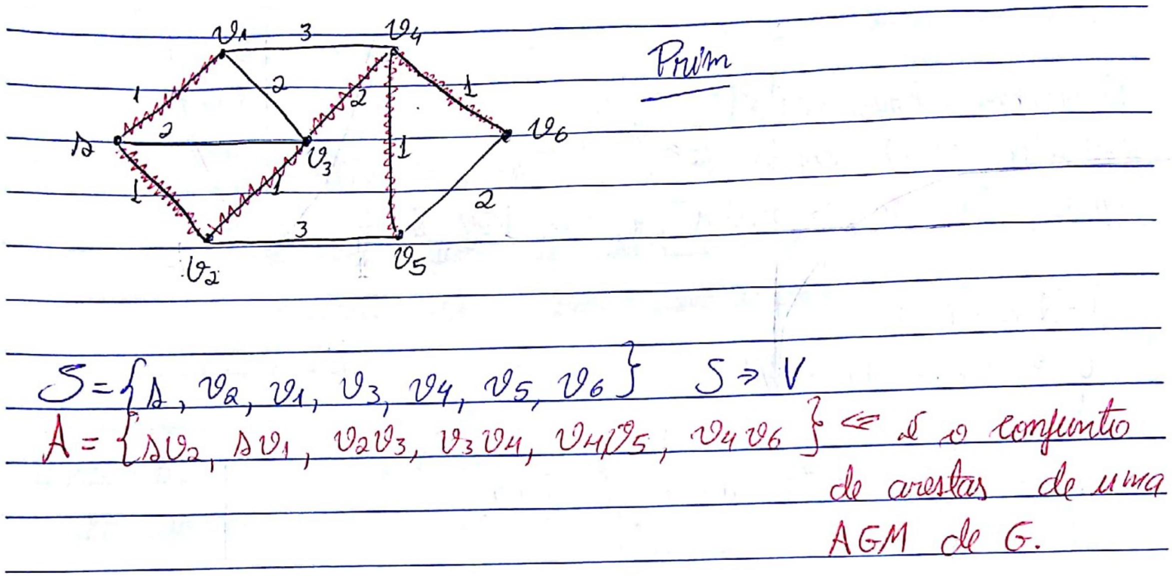
Algoritmo de Kruskal:



Algoritmo: Kruskal(G)
Entrada: Grafo conexo G(V, E) ponderado em arestas
Saída: Conjunto de arestas em uma AGM de G

```
Ordenar(E)
U ← Union-Find(V)
A ← Vazio
para cada aresta uv e E faca:
|     se |A| = |V| - 1 entao:
|     |     pare
|     se Find(U, u) != Find(U, v) entao:
|     |     A ← A uniao {uv}
|     |     Union(U, u, v)
retorna A
```

Algoritmo de Prim



* Manter uma fila de prioridades com todas as arestas do corte atul [S, V – S]
* Manter um conjunto com todos os vértices já inclusos em S
* Manter uma lista das atestas relacionadas
\# Sabendo as extremidades da próxima aresta escolhida, sabemos como atualizar S e H

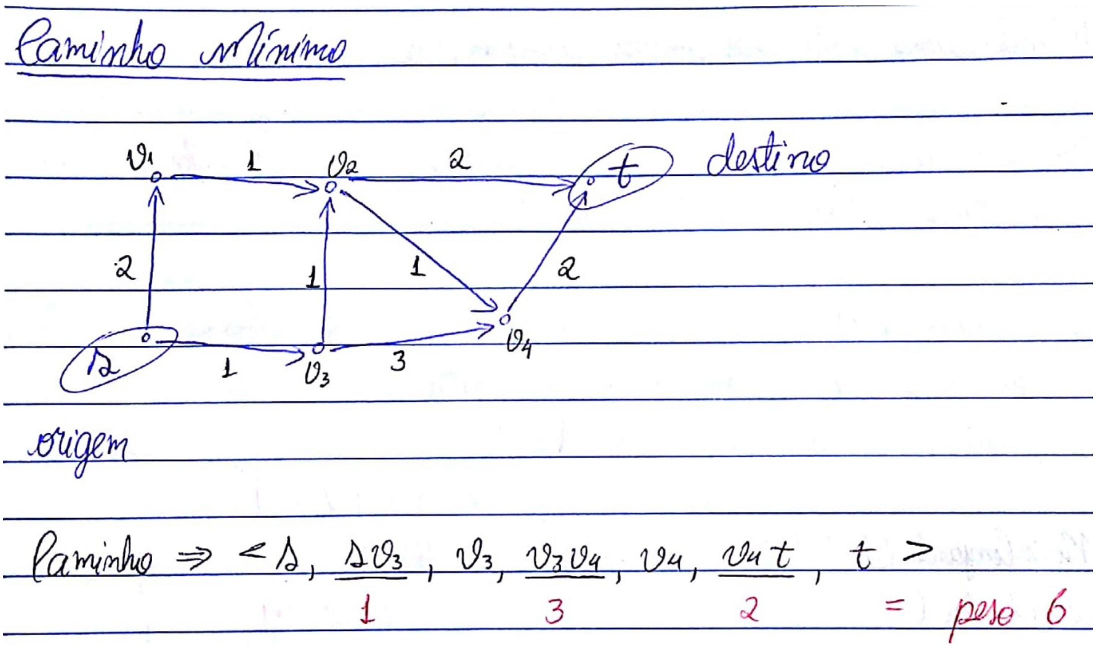
Algoritmo: Prim(G)
Entrada: Grafo(V,E) ponderado em arestas
Saída: Conjuto de arestas de uma AGM em G

```
S ← NovoConjunto()
A ← NovaLista()
H ← NovaFilaPrioridades()
S ← {v0}

para cada aresta v0w faca:
|     H ← H uniao {v0w}
enquanto |S| != n faca:
|     se H != Vazio entao:
|     |     retorne Vazio
|     senao:
|     |     uv ← ExtrairMinimo(H)
|     |     A ← A uniao {uv}
|     |     considere w dentre {u,v} que não pertence s S
|     |     S ← S uniao {w}
|     |     para cada aresta wy faca:
|     |     |     se y !e S entao:
|     |     |     |     H ← H uniao {wy}
retorne A
```

Reconhecimento de ausência de solução:
Kruskal: Quando acabarem as arestas sem termos escolhido n-1 para a solução
Prim: Quando S != V e [S, V – S] = Vazio

> Caminho Mínimo



Dentre todos os caminhos que começam em s e terminam em t, num grafo, determinar um com peso mínimo, levando em conta a distancia e o peso de cada aresta.

* Caminho mínimo com origem e destino únicos
* Caminho mínimo com origem fixa (s)
    -> Queremos a resposta de s para todos os demais.

*Caminho mínimo com destino fixo(t)
    -> queremos a resposta a partir de cada vertice ate t

*Caminho mínimo entre todos os pares.

Operação de relaxação de arestas
*Para cada vértice vamos manter ma estimativa do custo da solução ótima
d[v] >= &(s,v)
        |
        '------------> peso do menor caminho de s a v.

* Relaxação de aresta significa tentar atualizar a estimativa de um vértice observando uma aresta.
*Vamos atualizar d[v] apenas com custos de caminhos existentes no grafo.

Algoritmo: Relaxar(uv, d, w)
Entrada: Aresta uv, estimativas d, pesos w

```
salto ← d[u] + w[uv]
se d[v] > salto entao:
| d[v] ← salto
| retorne V
retorne F
```

Caminho Mínimo

&(s,v) representa o valor de um CM de s p/ v.

1. Desigualdade Triangular:

&(s,u) + Puv >= &(s,v)

2. Limite Superior

Vamos manter d[v] >= &(s,v), porém, uma vez conseguindo igualdade, d[v] não será mais alterado.

3. Ausência de Caminho

Se não existir caminho sv diferente, d[v]=+infinito durante todo o algoritmo.

4. Convergência

Se s-->u->v é um caminho mínimo para u e v quaisquer e conseguirmos d[u]=&(s,u) em algum momento anterior a ultima relaxação de uv, então d[v]=&(s,v) dessa relaxação em diante.

5. Relaxação de caminho

Se s->v0->v1->v2->...->vk-1->vk é um caminho mínimo de s a vk e, ao longo das relaxações, conseguirmos relaxar as arestas desse caminho em ordem, após a ultima relaxação certamente d[vk]=&(s,vk).

(Independentemente dessas relaxações ocorrerem intercaladas por de outras arestas).

Algoritmo: Belman-Ford(G,s,P)

Entrada: Grafo G = (V,E), vértice s pertencente a V(G), função de pesos P:E(G)→R.
Saída: Árvore de caminhos mínimos em G com raiz s segundo os pesos p.
Requisito: Não podem haver ciclos negativos em G.

```
d ← NovoVetor(|V|)
pi ← NovoVetor(|V|)
para cada v pertencente a V(G) faca:
| d[v] = +infinito;
| pi[v] = lambda;
d[s] ← 0;
pi[s] ← s;
para i ←1,2,...,|V|-1 faca:
| para cada uv pertencente a E(G) faca:
| | se Relaxar(uv,d,p) entao:
| | | pi[v] ←u
retorne pi
```

Dijkstra

Vamos manter como invariante:

O vértice v ainda não escolhido que tenha valor de d[v] mínimo necessariamente tem d[v]=&(s,v)

Suponha uma sequencia de escolhas e atualizações segundo o processo de DIJKSTRA.

Suponha, por absurdo, que u é o primeiro dessa sequencia que não tinha d[u]=&(s,u). --(I)

Certamente u /= s.

Caso d[u] = +infinito (absurdo)

Caso d[u] /= +infinito

Existe caminho de s p/ u com pelo menos mais um outro vértice.

Seja P um caminho mínimo qualquer em G entre s e u.

s-->x->y-->u

d[y] = &(s,y) < &(s,u) < d[u]   por (I), temos outro absurdo.

Em suma, não tem como não existir por que não tem como chegar em d[u] se os anteriores já foram escolhidos.

Concluímos que não existe um primeiro vértice que deu errado, ou seja, é impossível de dar errado.

Algoritmo: Djikstra(G,s,P)
Entrada: Grafo G = (V,E), vértice s pertencente a V(G), função de pesos P:E(G)→R.
Saída: Árvore de caminhos mínimos em G com raiz s segundo os pesos p.
Requisito: Não podem haver arestas negativas em G

```
d← NovoVetor(|V|)
pi ← NovoVetor(|V|)
Q ← NovaFilaPrioridades()
para cada v pertencente a V(G) faca:
| d[v] = +infinito;
| pi[v] = lambda;
d[s] ← 0;
pi[s] ← s;
enquanto Q != Vazio faca:
|    u ← ExtrairMinimo(Q)
|    para cada vertice v adjacente a u faca:
|    |    alt ← dist[u] + dist_entre(uv)
|    |    se alt  < dist[v] entao:
|    |    |    dist[v] ← alt
|    |    |    pi[v] ← u
retorne pi
```