Técnicas para Programação Competitiva

Algoritmos Envolvendo Grafos

Prof. Andrei Braga

UNIVERSIDADE FEDERAL DA FRONTEIRA SUL

Conteúdo

- Árvores geradoras de peso mínimo
- Árvores geradoras de peso mínimo - Algoritmo de Prim
- Árvores geradoras de peso mínimo - Algoritmo de Kruskal
- Caminhos de peso mínimo
- Caminhos de peso mínimo - Algoritmo de Dijkstra
- Referências

Conteúdo

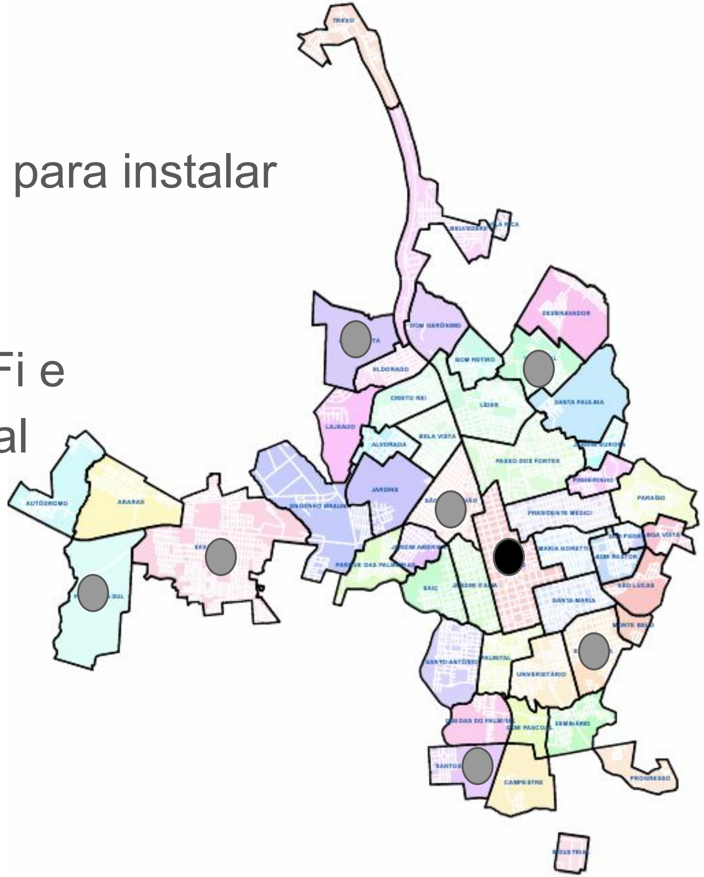
- **Árvores geradoras de peso mínimo**
- Árvores geradoras de peso mínimo - Algoritmo de Prim
- Árvores geradoras de peso mínimo - Algoritmo de Kruskal
- Caminhos de peso mínimo
- Caminhos de peso mínimo - Algoritmo de Dijkstra
- Referências

Problema

- Suponha que a nossa empresa foi contratada para instalar pontos de acesso Wi-Fi em Chapecó

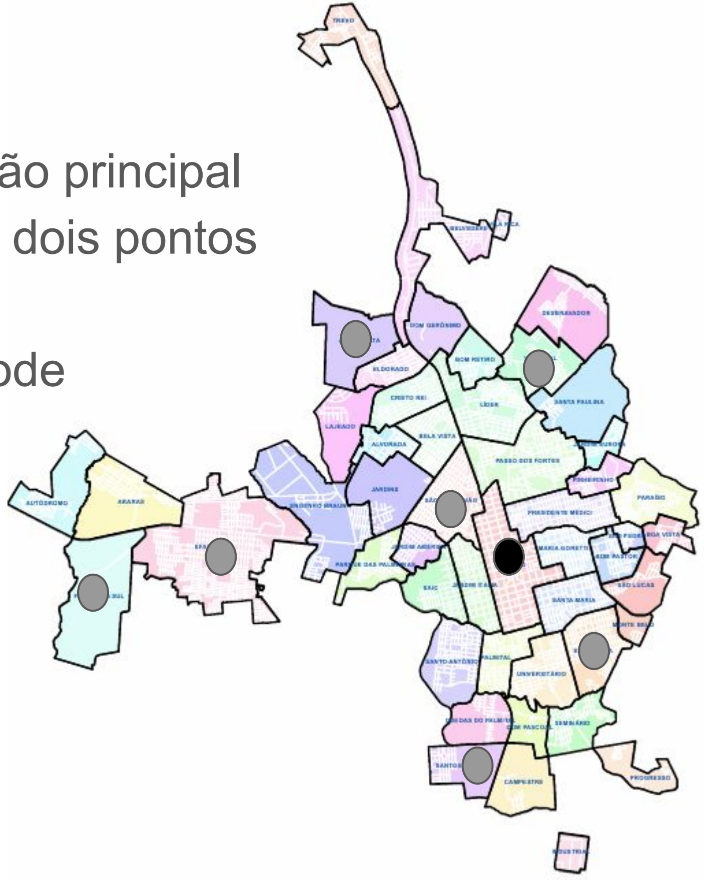
Problema

- Suponha que a nossa empresa foi contratada para instalar pontos de acesso Wi-Fi em Chapecó
- Foram selecionados bairros da cidade onde deverão ser instalados pontos de acesso Wi-Fi e a nossa empresa possui uma estação principal de onde será fornecida a comunicação com a Internet aos pontos de acesso Wi-Fi



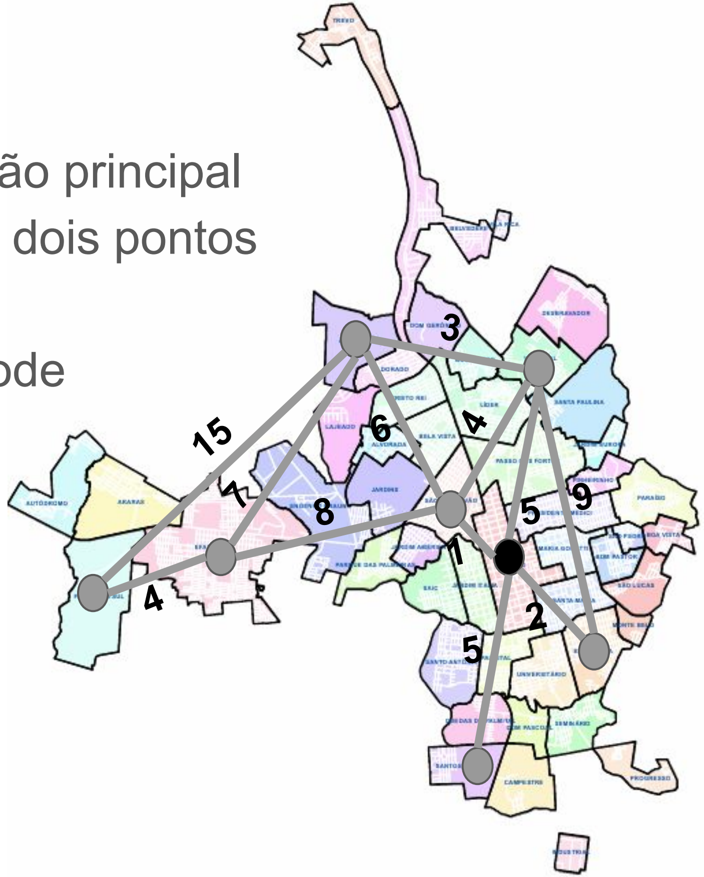
Problema

- Poderemos usar cabos para conectar a estação principal a um ponto de acesso Wi-Fi ou para conectar dois pontos de acesso Wi-Fi
- Cada conexão por cabo tem um custo, que pode depender da distância entre os pontos que serão conectados e de outros fatores
- Podemos não considerar conexões entre alguns pares de pontos, por estas conexões serem inviáveis ou por algum outro motivo



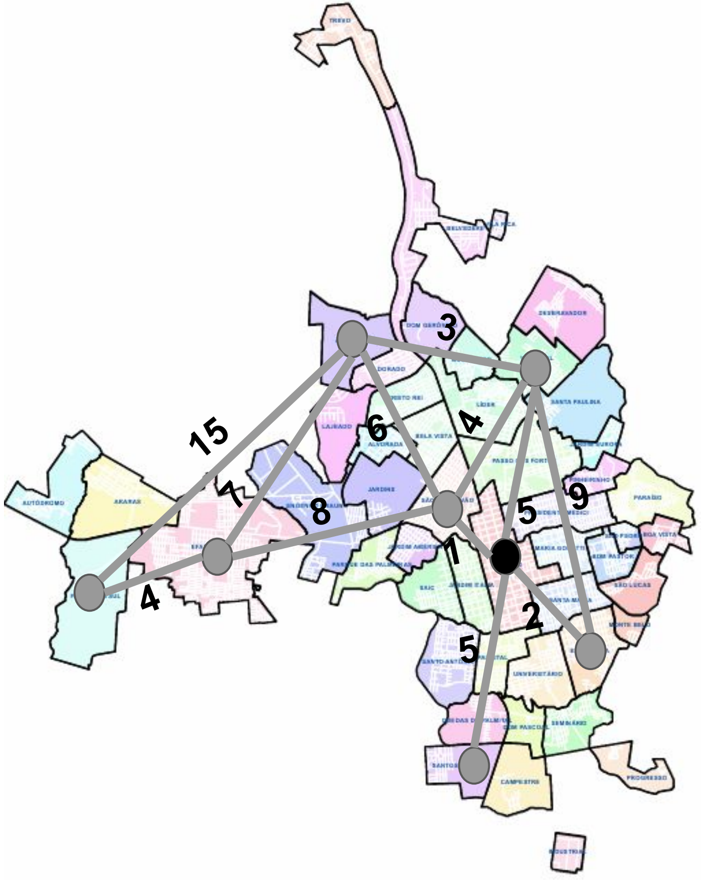
Problema

- Poderemos usar cabos para conectar a estação principal a um ponto de acesso Wi-Fi ou para conectar dois pontos de acesso Wi-Fi
- Cada conexão por cabo tem um custo, que pode depender da distância entre os pontos que serão conectados e de outros fatores
- Podemos não considerar conexões entre alguns pares de pontos, por estas conexões serem inviáveis ou por algum outro motivo



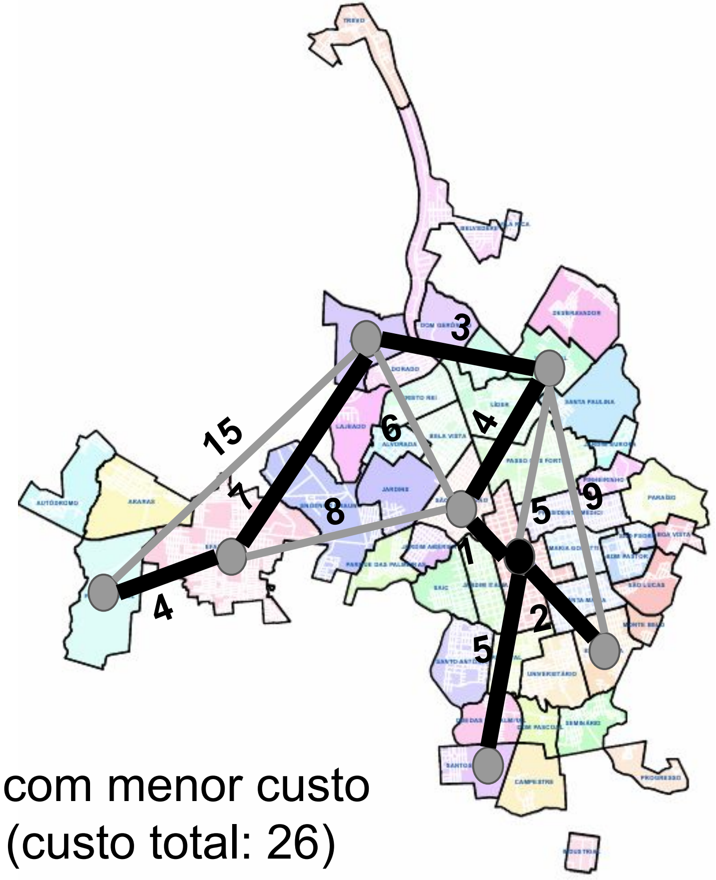
Problema

- **Problema:** Como conectar todos os pontos de acesso Wi-Fi e a estação principal tendo o menor custo possível?



Problema

- **Problema:** Como conectar todos os pontos de acesso Wi-Fi e a estação principal tendo o menor custo possível?



Solução com menor custo possível (custo total: 26)

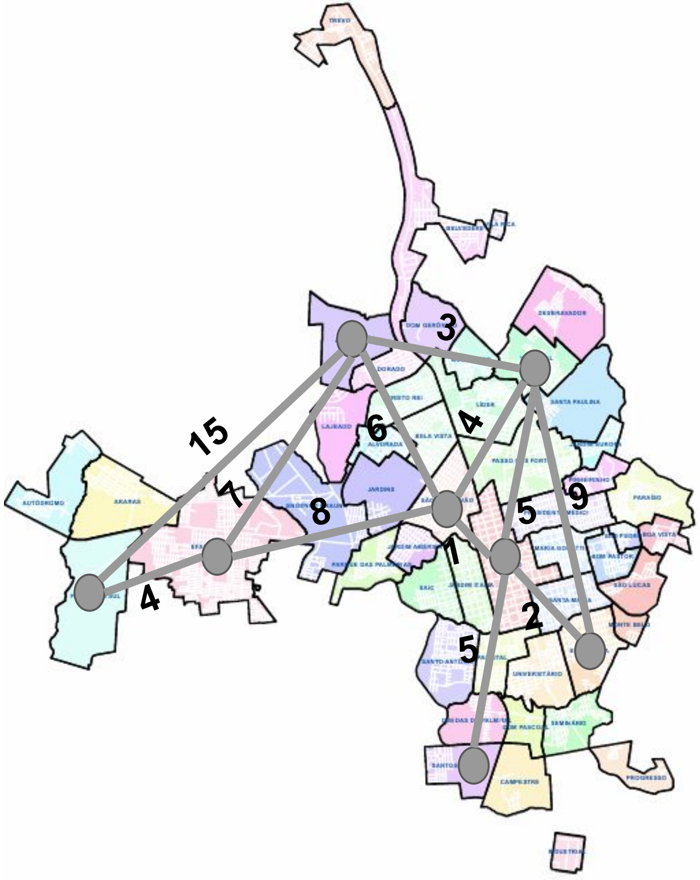
Problema - Modelagem com grafos

- Podemos modelar este problema usando um grafo *G* tal que
  - os vértices de *G* representam os pontos a serem conectados (pontos de acesso Wi-Fi e a estação principal) e
  - as arestas de *G* representam as conexões por cabo que podemos realizar,
  - com cada aresta de *G* tendo um **peso**, que representa o **custo** da conexão correspondente

- O nosso **objetivo** é, então, encontrar um subconjunto de arestas de *G* que
  - conecte todos os vértices de *G*,
  - não contenha ciclos e
  - tenha peso total mínimo

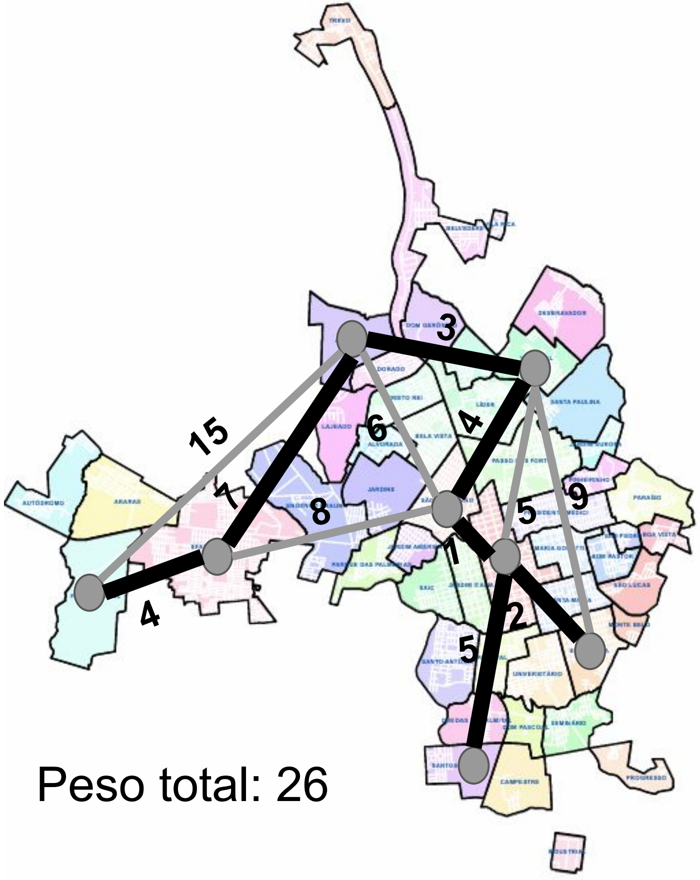
Problema - Modelagem com grafos

- O nosso objetivo é, então, encontrar um subconjunto de arestas de *G* que
  - conecte todos os vértices de *G*,
  - não contenha ciclos e
  - tenha peso total mínimo



Problema - Modelagem com grafos

- O nosso objetivo é, então, encontrar um subconjunto de arestas de *G* que
  - conecte todos os vértices de *G*,
  - não contenha ciclos e
  - tenha peso total mínimo

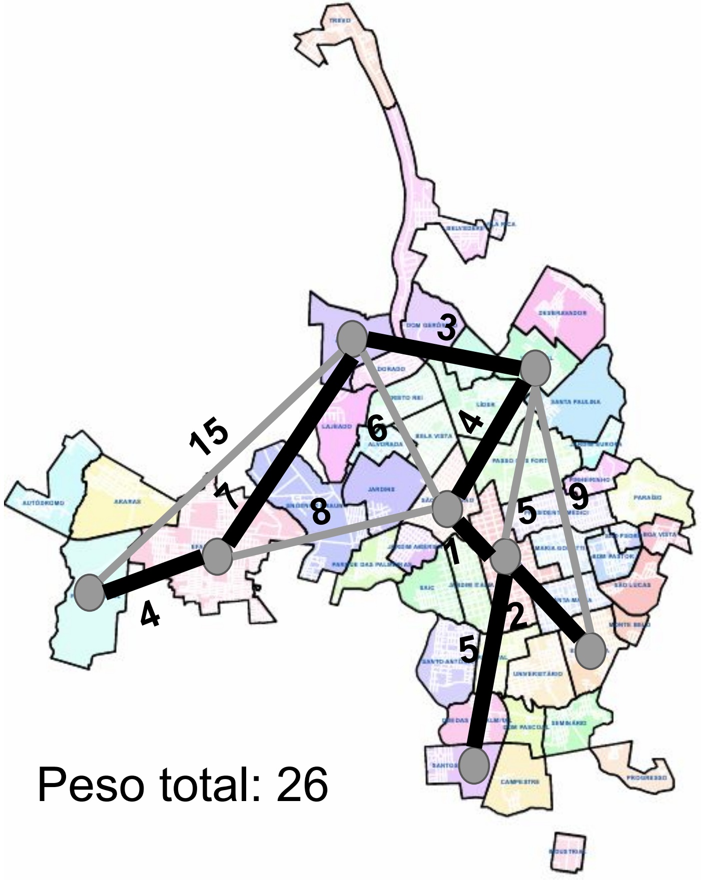


Peso total: 26

Problema - Modelagem com grafos

- O nosso objetivo é, então, encontrar um subconjunto de arestas de *G* que
  - conecte todos os vértices de *G*,
  - não contenha ciclos e
  - tenha peso total mínimo

- Podemos definir este objetivo de outra maneira



Problema - Modelagem com grafos

- O nosso objetivo é, então, encontrar um subconjunto de arestas de *G* que
    - conecte todos os vértices de *G*,
    - não contenha ciclos e
    - tenha peso total mínimo

- O nosso objetivo é, então, encontrar um subgrafo *T* de *G* que
    - seja gerador (contenha todos os vértices de *G*) e conexo,
    - seja acíclico e
    - tenha peso mínimo,
      com o **peso do subgrafo** *T* sendo igual à soma dos pesos das suas arestas

Problema - Modelagem com grafos

- O nosso objetivo é, então, encontrar um subgrafo *T* de *G* que
  - seja gerador (contenha todos os vértices de *G*) e conexo,
  - seja acíclico e
  - tenha peso mínimo,
    com o peso do subgrafo *T* sendo igual à soma dos pesos das suas arestas

- Um subgrafo de *G* que é gerador, conexo e acíclico é denominado uma **árvore geradora** de *G*

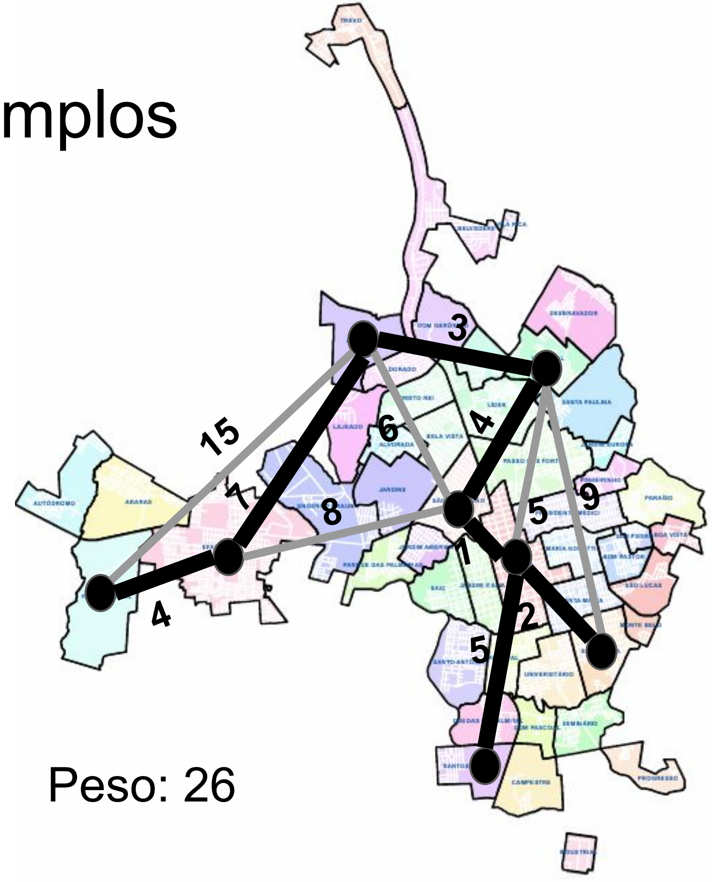
- O nosso objetivo é, então, encontrar uma árvore geradora *T* de *G* que tenha peso mínimo, com o **peso da árvore geradora** *T* sendo igual à soma dos pesos das suas arestas

Problema - Modelagem com grafos

- O nosso objetivo é, então, encontrar uma árvore geradora *T* de *G* que tenha peso mínimo, com o peso da árvore geradora *T* sendo igual à soma dos pesos das suas arestas

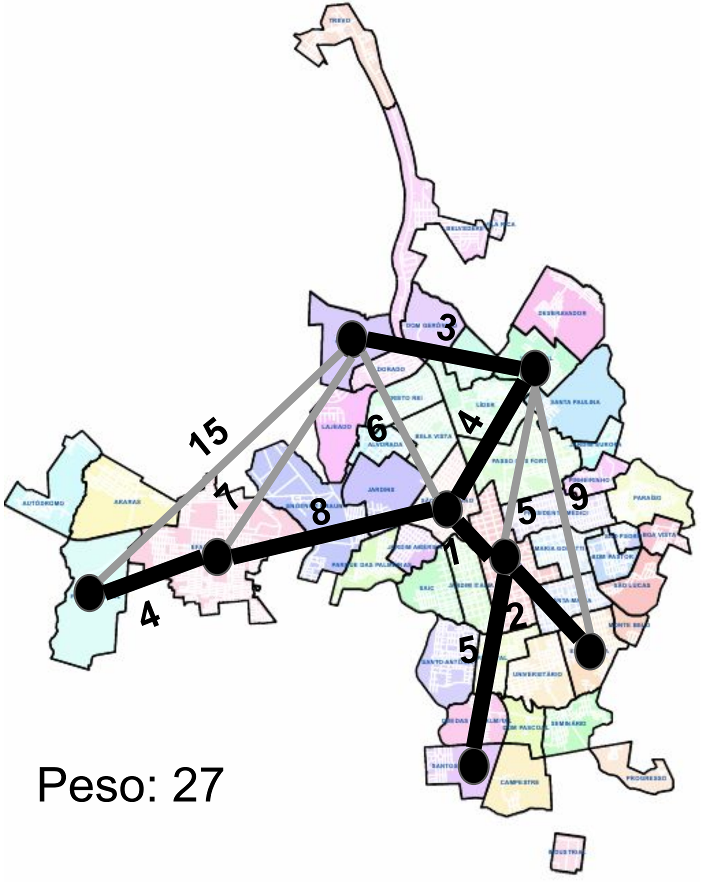
- O nosso objetivo é, então, encontrar uma **árvore geradora de peso mínimo** de *G*

Exemplos



Peso: 26

Árvore geradora de
peso mínimo



Peso: 27

Árvore geradora de
peso mínimo

Algoritmos

- O problema de encontrar uma árvore geradora de peso mínimo de um grafo com pesos nas arestas tem sido estudado (pelo menos) desde os anos 1920
- Para este problema, vamos considerar que o **grafo** recebido como **entrada** é **conexo** (caso contrário, o problema não admite solução)
- Veremos dois algoritmos para resolver o problema: o **Algoritmo de Prim** e o **Algoritmo de Kruskal**

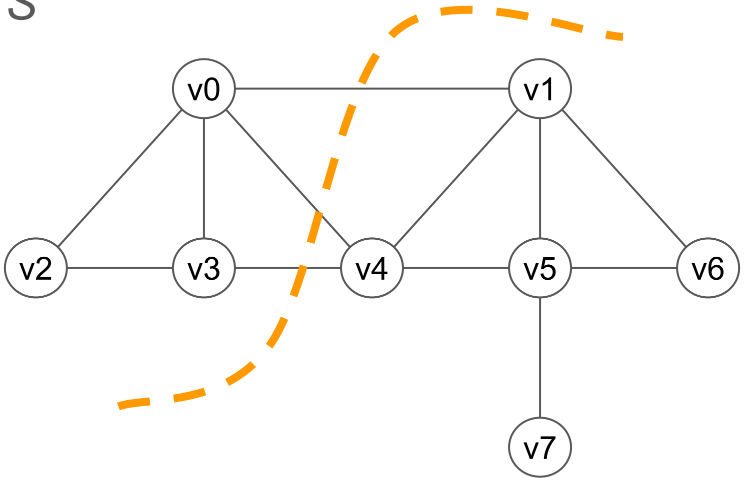
Conteúdo

Corte

- Um **corte** em um grafo *G* é uma partição (*S*, *V*(*G*) \ *S*) do conjunto de vértices de *G* em dois subconjuntos disjuntos (e não vazios): o subconjunto *S* e o subconjunto *V*(*G*) \ *S*

- Exemplo:

G:

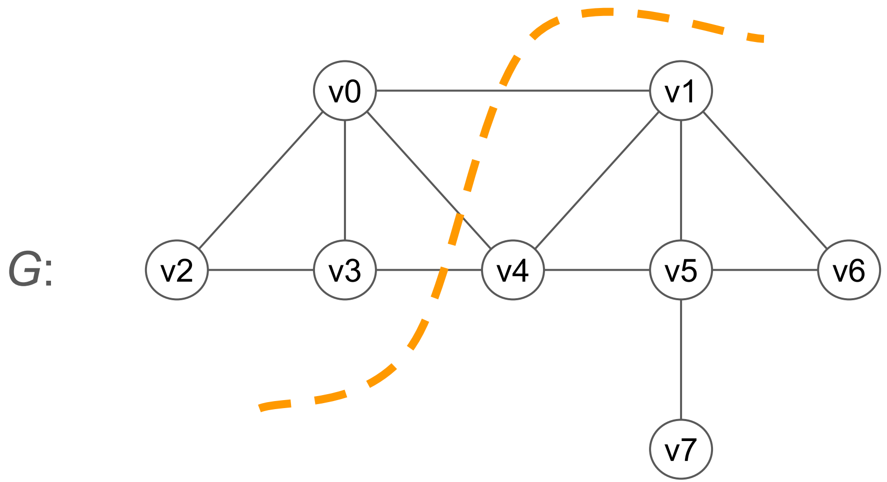


({ v0, v2, v3 }, { v1, v4, v5, v6, v7 }) é um corte de *G*

Corte

- Dado um corte (S, $V(G) \setminus S$) em um grafo G, uma **aresta do corte** é uma aresta que tem um extremo em S e um extremo em $V(G) \setminus S$

- Exemplo:

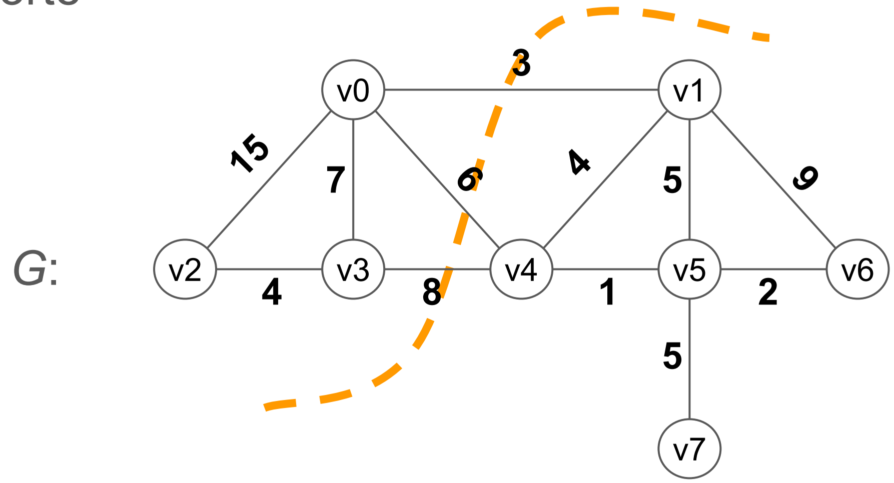


  - v0v4 e v3v4 são arestas do corte
  - v0v2 e v5v6 **não** são arestas do corte

Corte

- Dado um corte (S, $V(G) \setminus S$) em um grafo G com pesos nas arestas, uma **aresta de peso mínimo do corte** é uma aresta de peso mínimo dentre as arestas do corte

- Exemplo:



- v0v1 é uma aresta de peso mínimo do corte
- v3v4 **não** é uma aresta de peso mínimo do corte

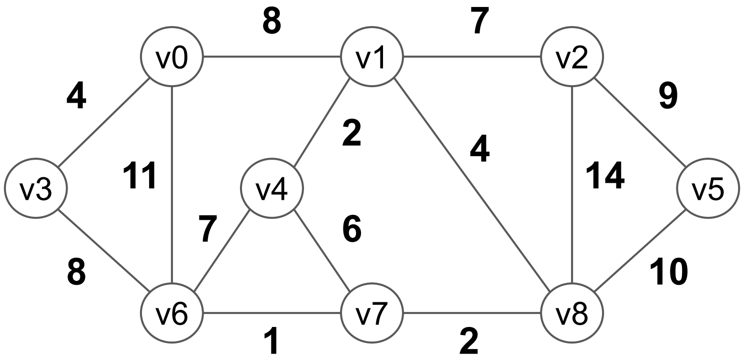
Algoritmo de Prim

Inicialmente, T é uma árvore que consiste apenas no vértice 0 de G

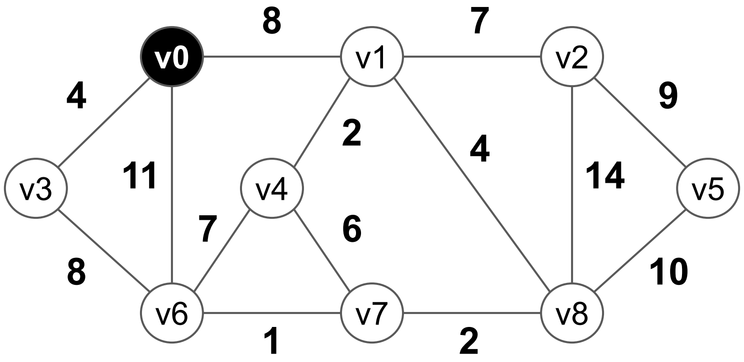
Prim(G conexo)

1. $T = (\{ 0 \}, \varnothing)$
2. Enquanto T não é uma árvore geradora de G:
3.   Encontre uma aresta uv de peso mínimo do corte $(V(T), V(G) \setminus V(T))$
4.   Adicione uv a T
5. Retorne T

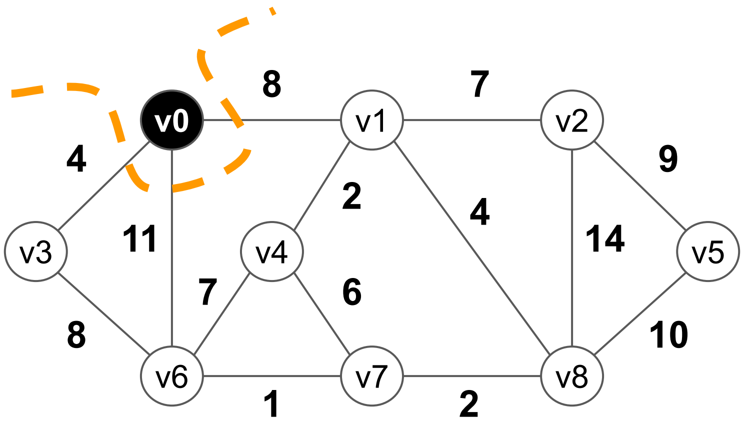
Algoritmo de Prim



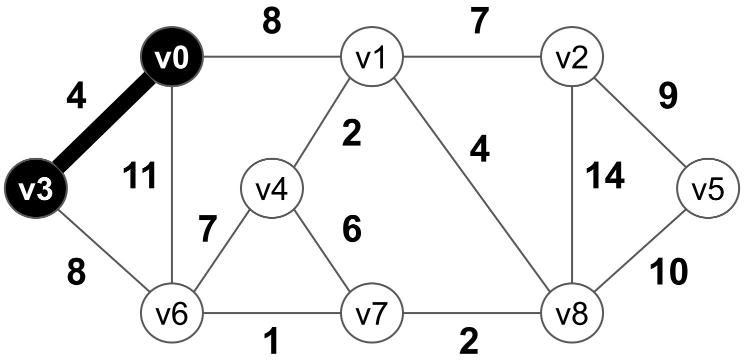
Algoritmo de Prim



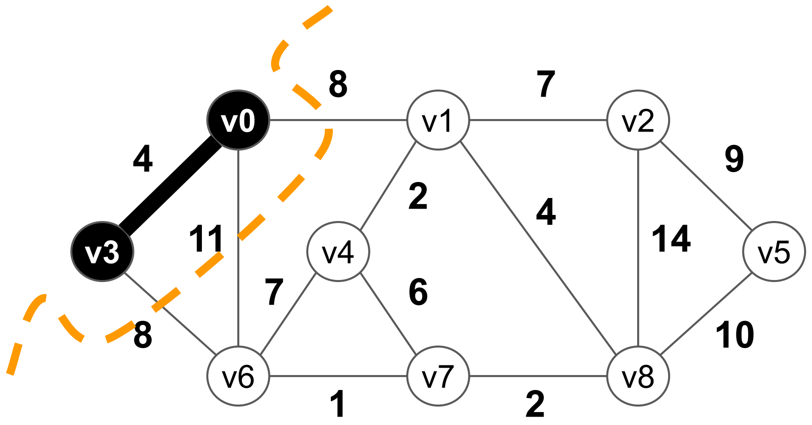
Algoritmo de Prim



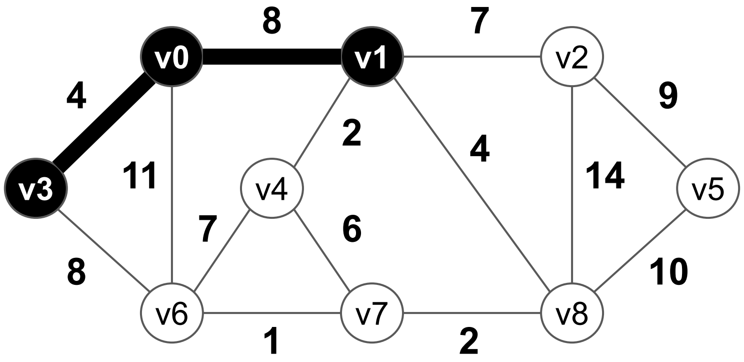
Algoritmo de Prim



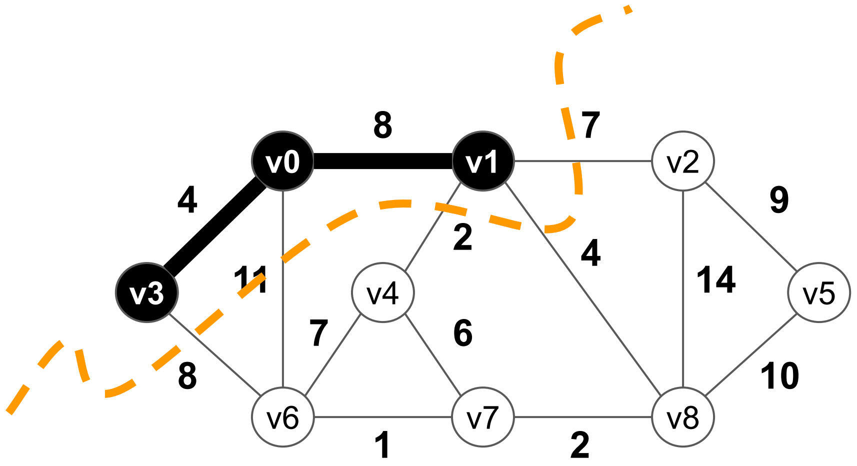
Algoritmo de Prim



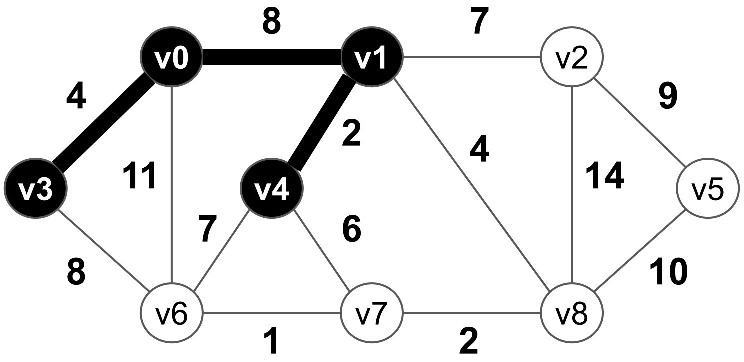
Algoritmo de Prim



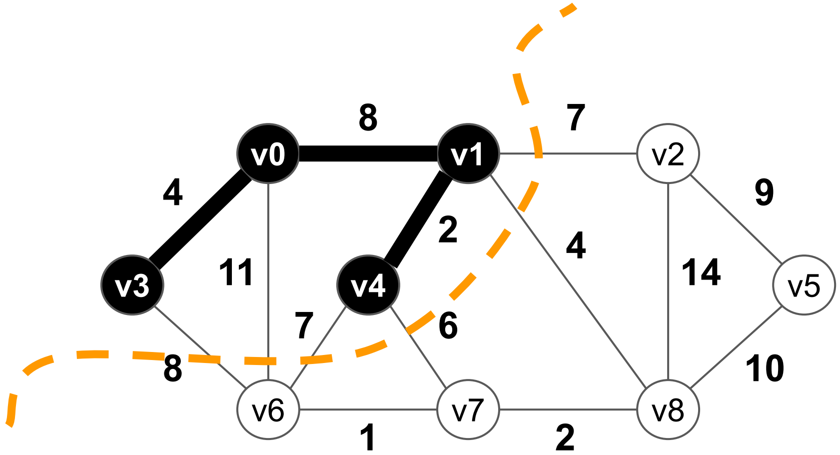
Algoritmo de Prim



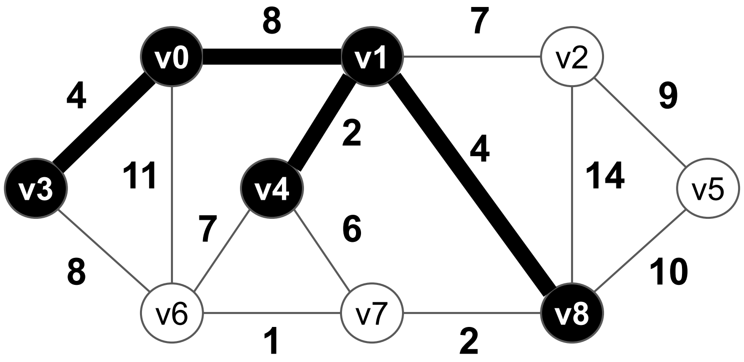
Algoritmo de Prim



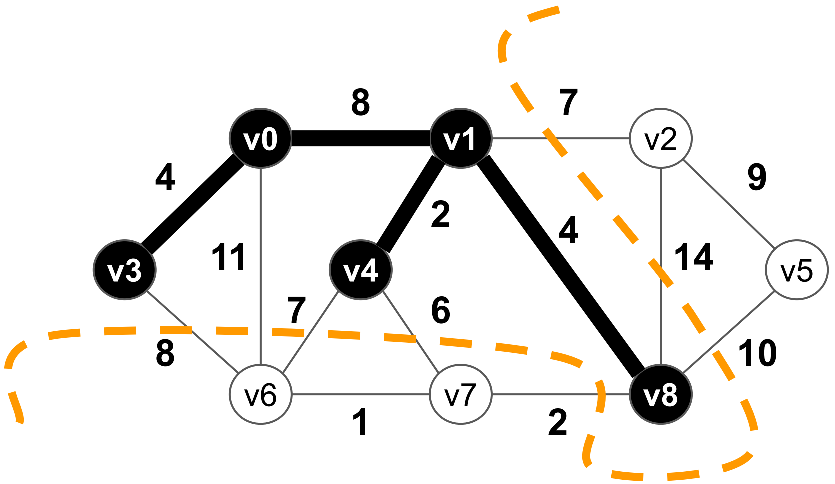
Algoritmo de Prim



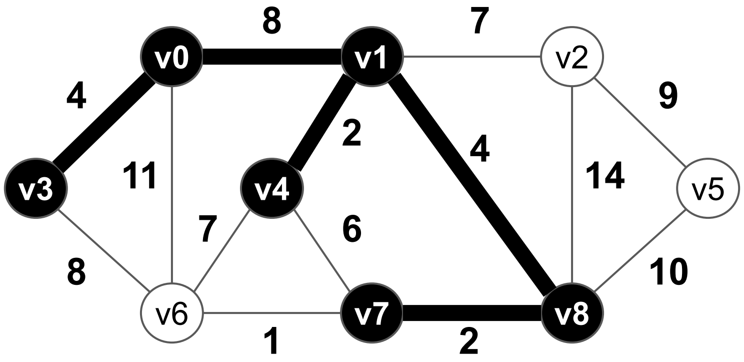
Algoritmo de Prim



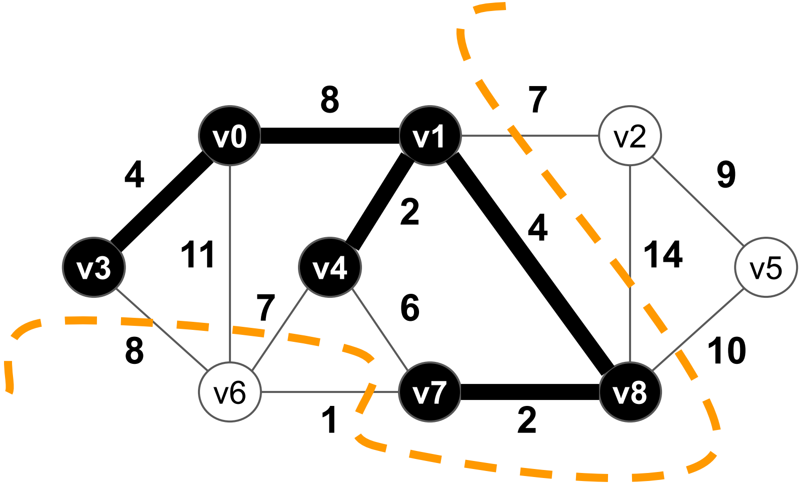
Algoritmo de Prim



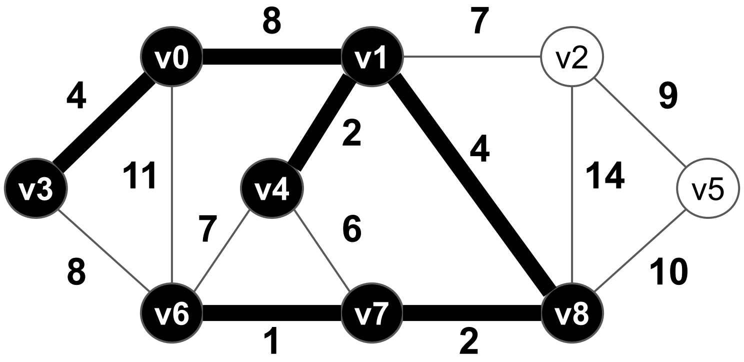
Algoritmo de Prim



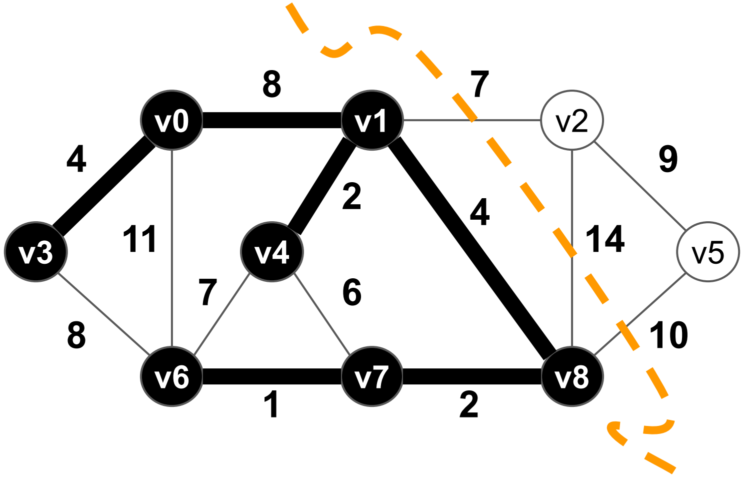
Algoritmo de Prim



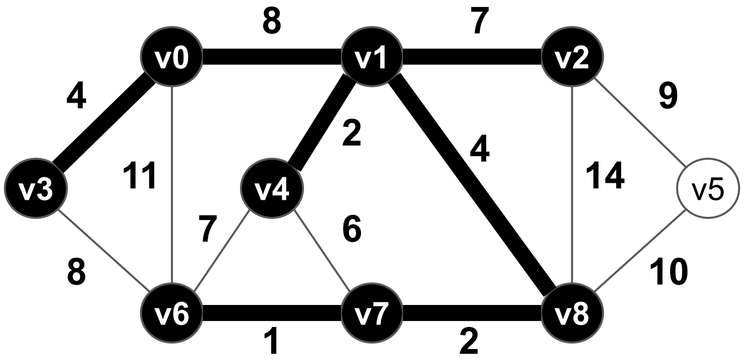
Algoritmo de Prim



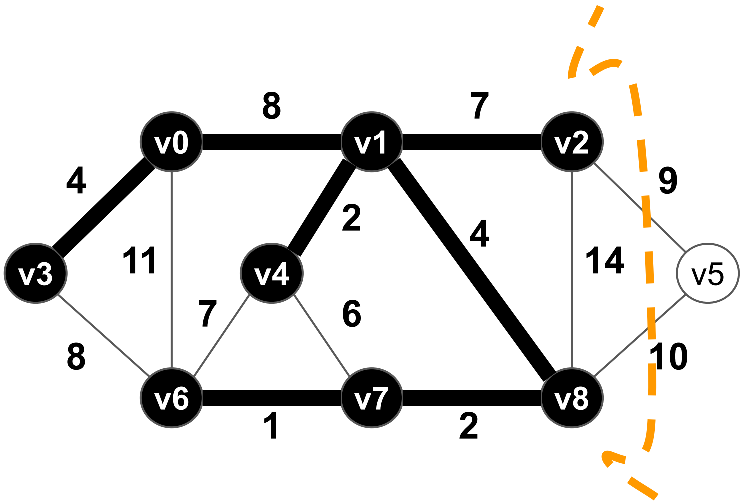
Algoritmo de Prim



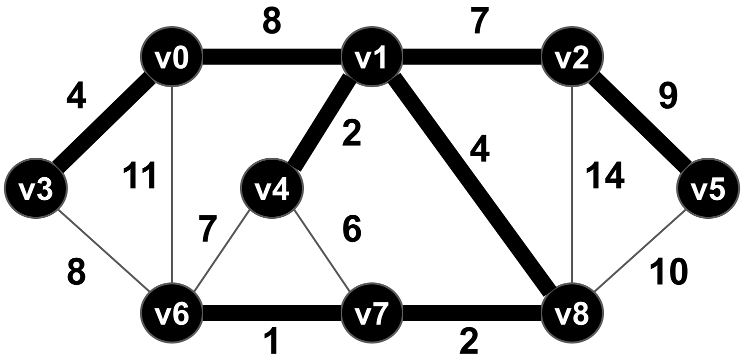
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim

Inicialmente, T é uma árvore que consiste apenas no vértice 0 de G

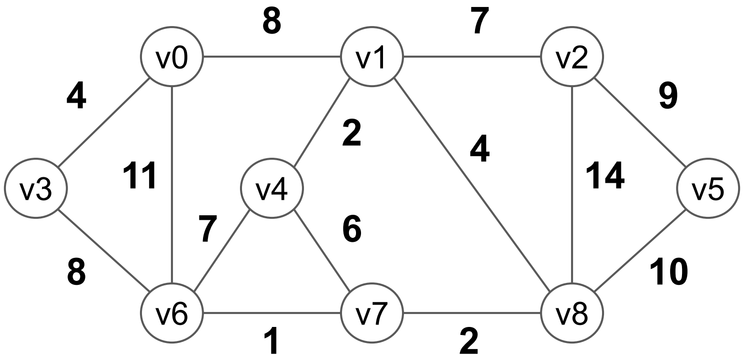
Prim(G conexo)

1. $T = (\{ 0 \}, \varnothing)$
2. Enquanto T não é uma árvore geradora de G:
3. Encontre uma aresta uv de peso mínimo do corte $(V(T), V(G) \setminus V(T))$
4. Adicione uv a T
5. Retorne T

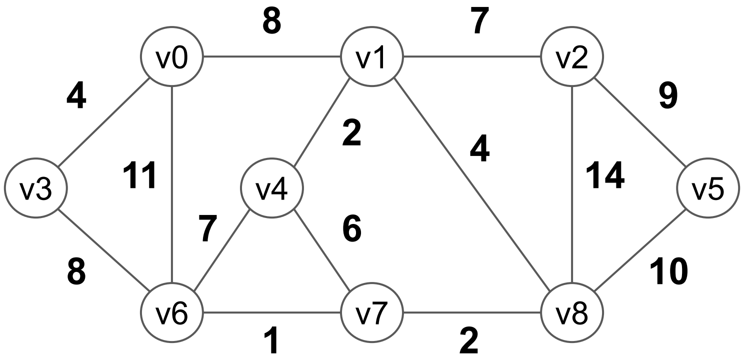
Algoritmo de Prim - Implementação

- Na implementação do Algoritmo de Prim, vamos usar uma fila de prioridade para executar de maneira eficiente o passo de determinar a próxima aresta a ser adicionada à árvore que estamos construindo
- Além disso, vamos usar um vetor *na_arvore* para registrar os vértices que já foram adicionados à árvore

Algoritmo de Prim - Implementação



Algoritmo de Prim - Implementação

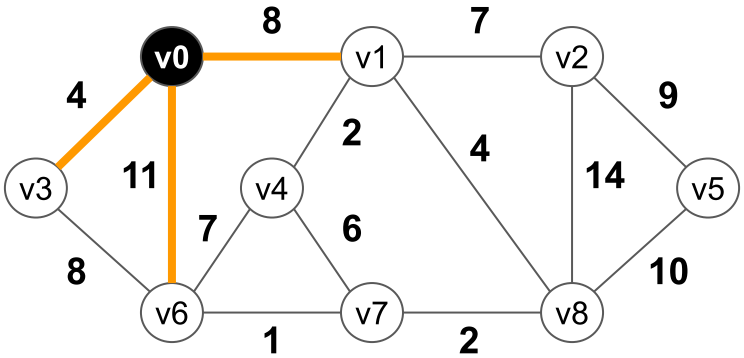


Fila de prioridade:

na_arvore:

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

Algoritmo de Prim - Implementação



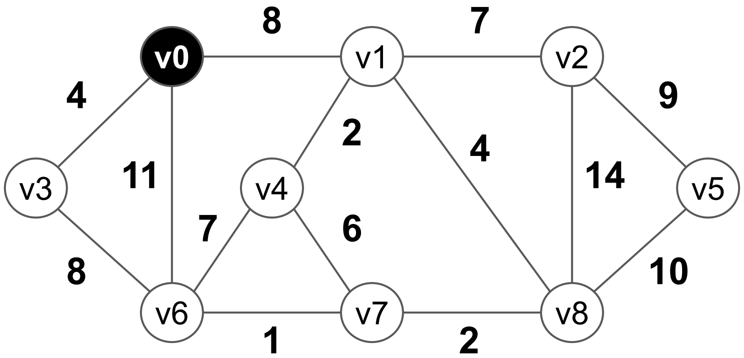
Fila de prioridade:

v3	v1	v6
4	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0

Algoritmo de Prim - Implementação



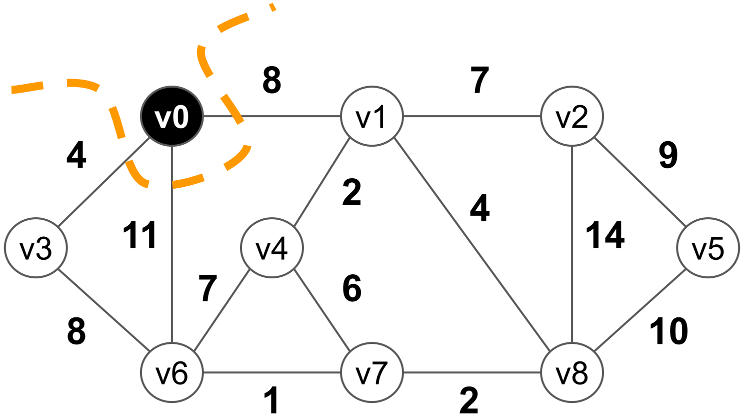
Fila de prioridade:

v3	v1	v6
4	8	11

na_arvore:

1	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



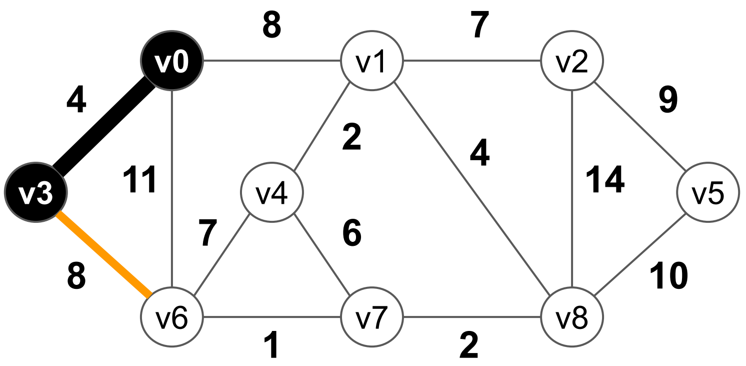
Fila de prioridade:

v3	v1	v6
4	8	11

na_arvore:

1	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação

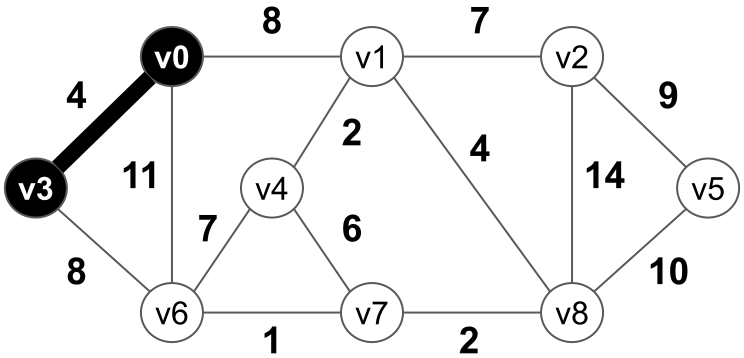


Fila de prioridade:

na_arvore:

0	1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	0	0

Algoritmo de Prim - Implementação



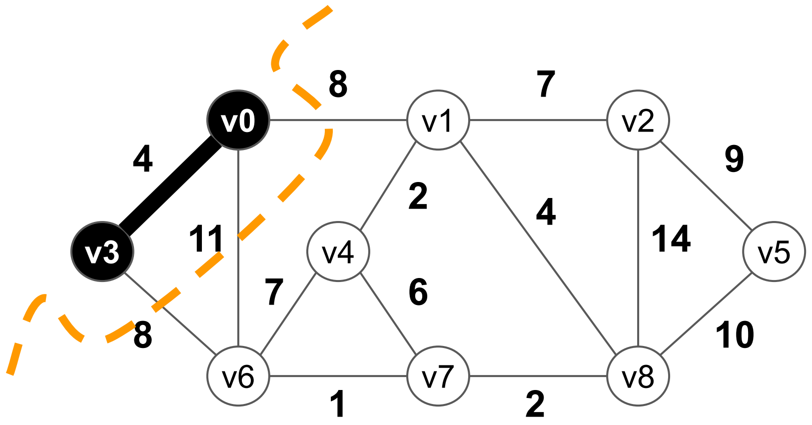
Fila de prioridade:

v1	v6	v6
8	8	11

na_arvore:

1	0	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



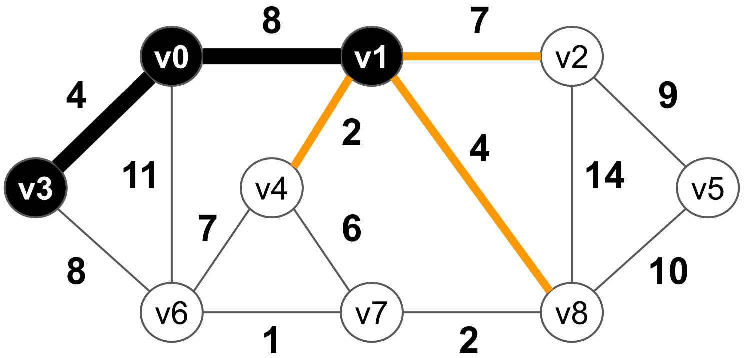
Fila de prioridade:

v1	v6	v6
8	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	0	0

Algoritmo de Prim - Implementação



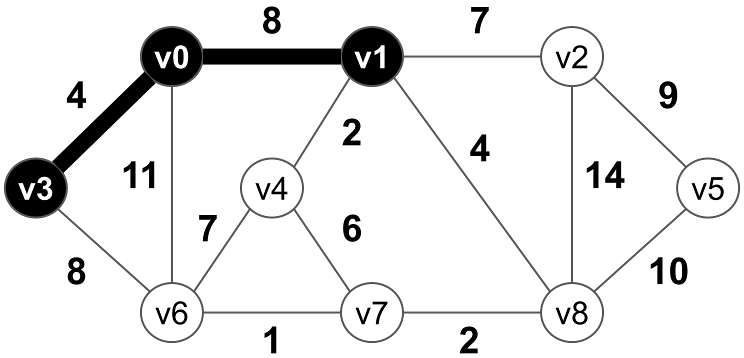
Fila de prioridade:

v4	v8	v2	v6	v6
2	4	7	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	0	0	0	0	0

Algoritmo de Prim - Implementação



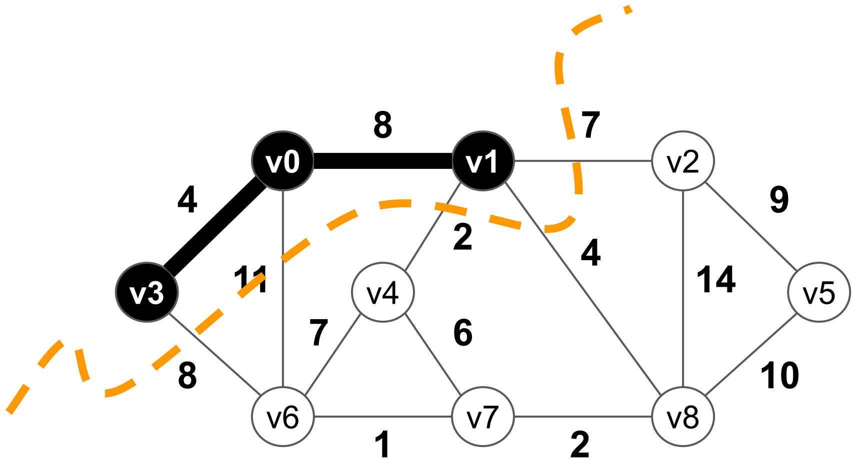
Fila de prioridade:

v4	v8	v2	v6	v6
2	4	7	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	0	0	0	0	0

Algoritmo de Prim - Implementação



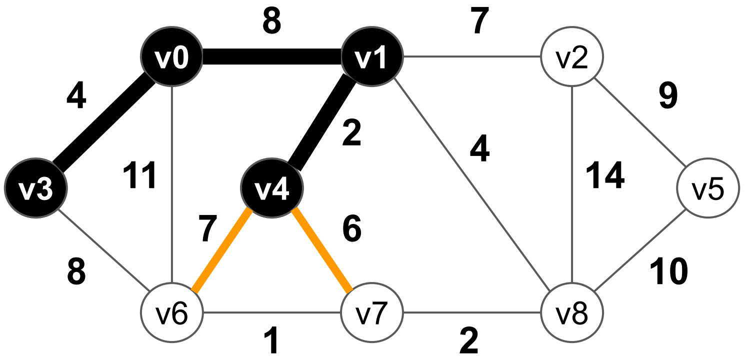
Fila de prioridade:

v4	v8	v2	v6	v6
2	4	7	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	0	0	0	0	0

Algoritmo de Prim - Implementação



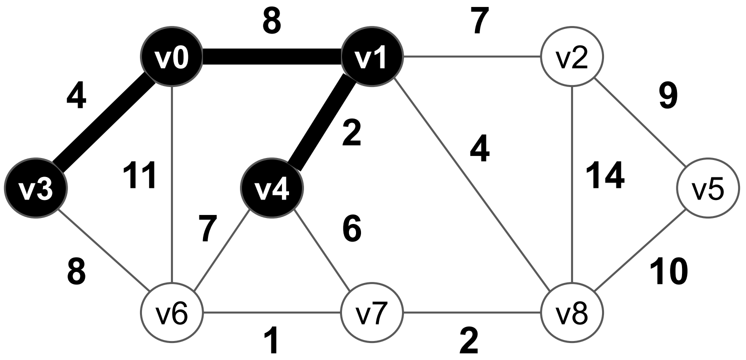
Fila de prioridade:

v8	v7	v2	v6	v6	v6
4	6	7	7	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	0	0

Algoritmo de Prim - Implementação



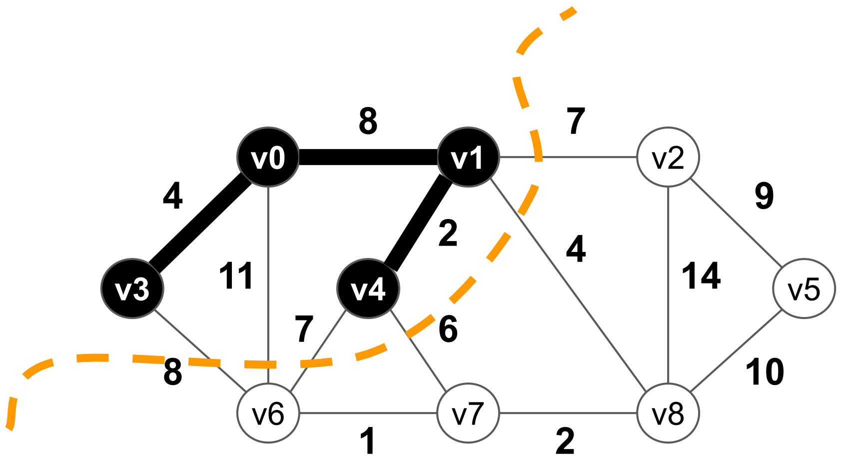
Fila de prioridade:

v8	v7	v2	v6	v6	v6
4	6	7	7	8	11

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	0	0

Algoritmo de Prim - Implementação



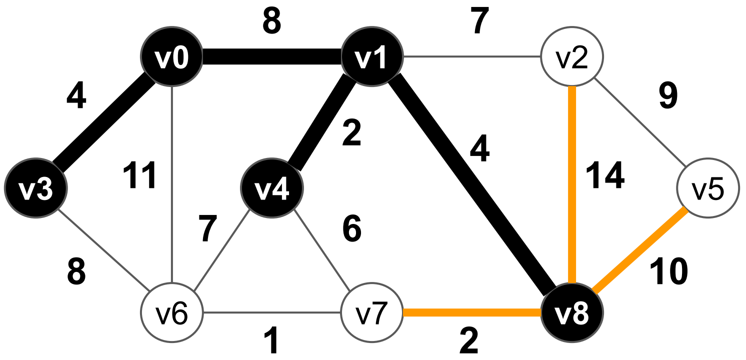
Fila de prioridade:

v8	v7	v2	v6	v6	v6
4	6	7	7	8	11

na_arvore:

1	1	0	1	1	0	0	0	0
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



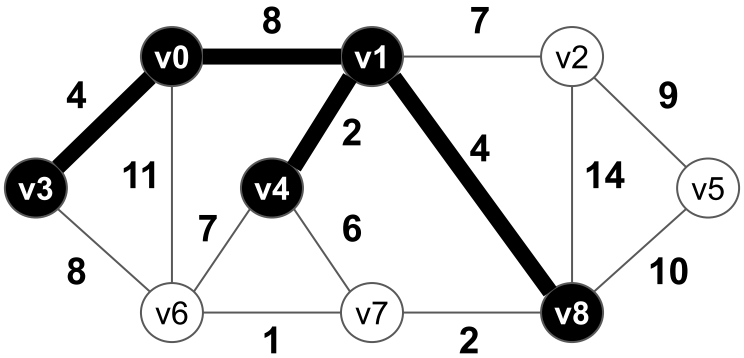
Fila de prioridade:

v7	v7	v2	v6	v6	v5	v6	v2
2	6	7	7	8	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	0	1

Algoritmo de Prim - Implementação



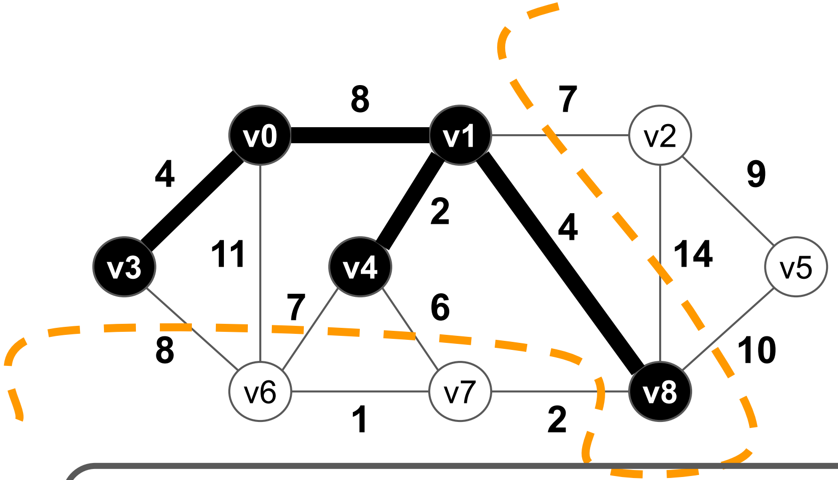
Fila de prioridade:

v7	v7	v2	v6	v6	v5	v6	v2
2	6	7	7	8	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	0	1

Algoritmo de Prim - Implementação



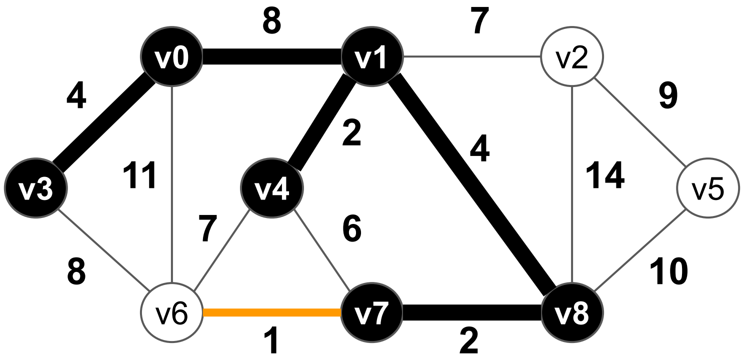
Fila de prioridade:

v7	v7	v2	v6	v6	v5	v6	v2
2	6	7	7	8	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	0	1

Algoritmo de Prim - Implementação



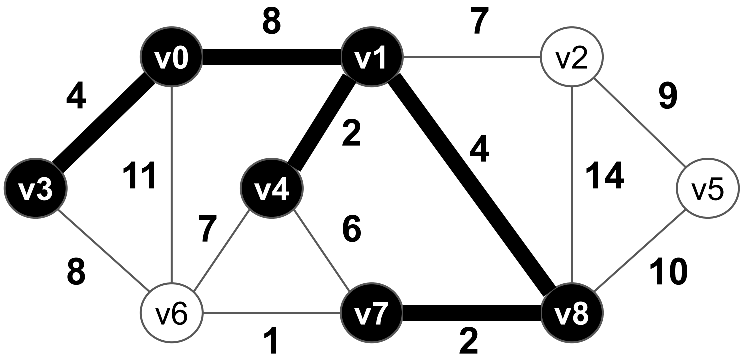
Fila de prioridade:

v6	v7	v2	v6	v6	v5	v6	v2
1	6	7	7	8	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1	1

Algoritmo de Prim - Implementação



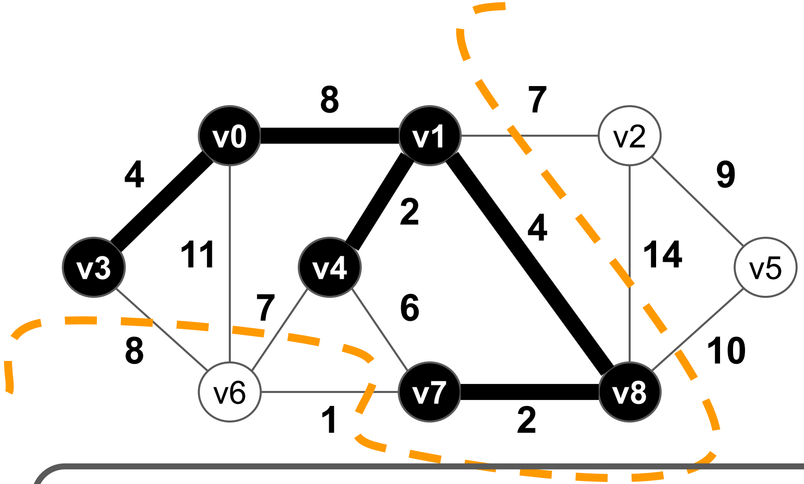
Fila de prioridade:

v6	v7	v2	v6	v6	v5	v6	v2
1	6	7	7	8	10	11	14

na_arvore:

1	1	0	1	1	0	0	1	1
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



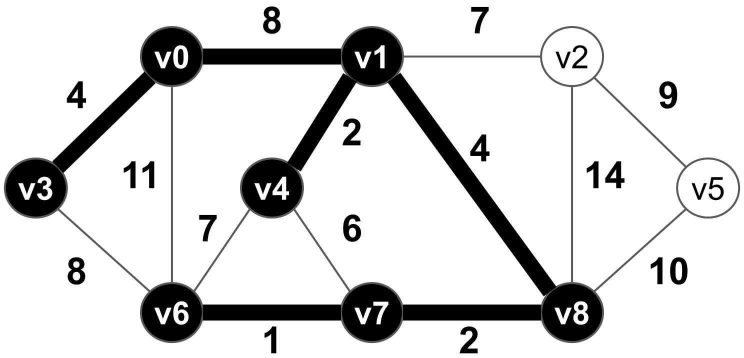
Fila de prioridade:

v6	v7	v2	v6	v6	v5	v6	v2
1	6	7	7	8	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1	1

Algoritmo de Prim - Implementação



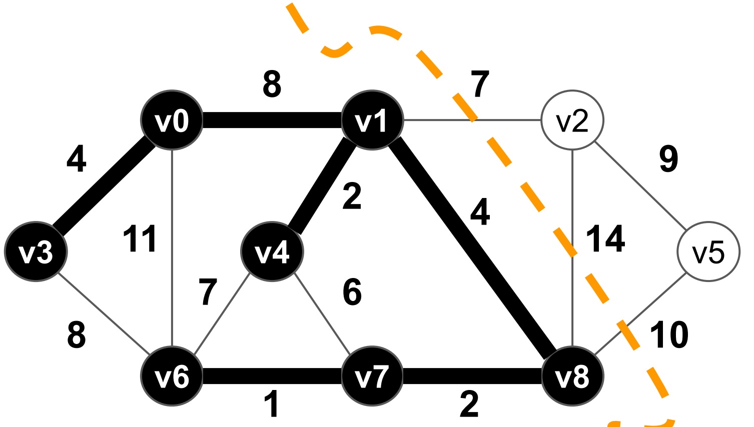
Fila de prioridade:

v7	v2	v6	v6	v5	v6	v2
6	7	7	8	10	11	14

na_arvore:

1	1	0	1	1	0	1	1	1
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



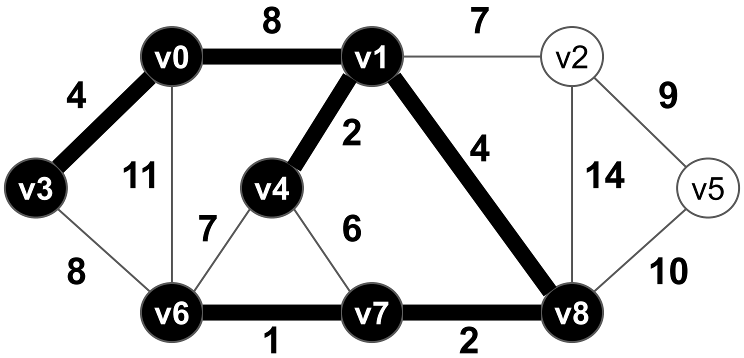
Fila de prioridade:

v7	v2	v6	v6	v5	v6	v2
6	7	7	8	10	11	14

na_arvore:

1	1	0	1	1	0	1	1	1
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



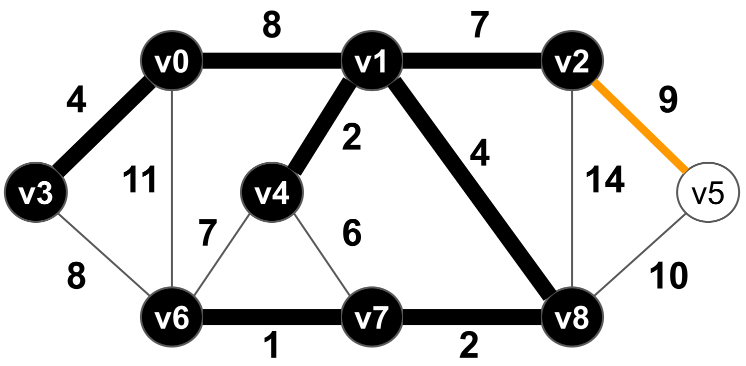
Fila de prioridade:

v2	v6	v6	v5	v6	v2
7	7	8	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	0	1	1	0	1	1	1

Algoritmo de Prim - Implementação



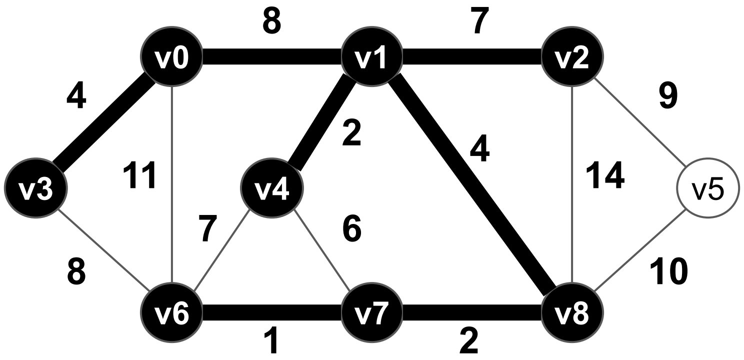
Fila de prioridade:

v6	v6	v5	v5	v6	v2
7	8	9	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	0	1	1	1

Algoritmo de Prim - Implementação



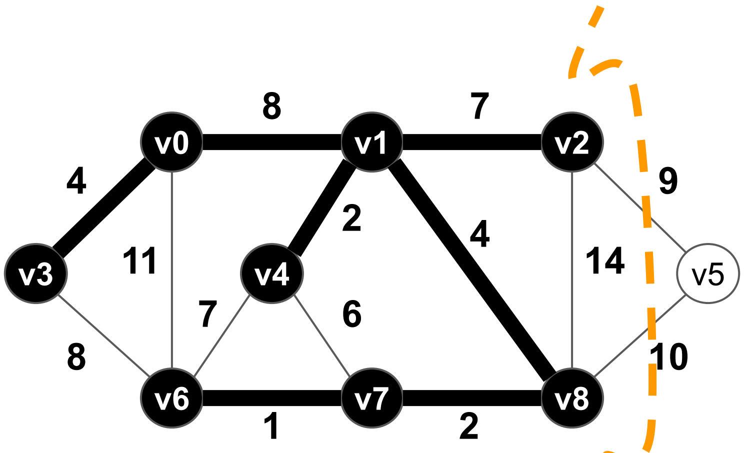
Fila de prioridade:

v6	v6	v5	v5	v6	v2
7	8	9	10	11	14

na_arvore:

1	1	1	1	1	0	1	1	1
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação



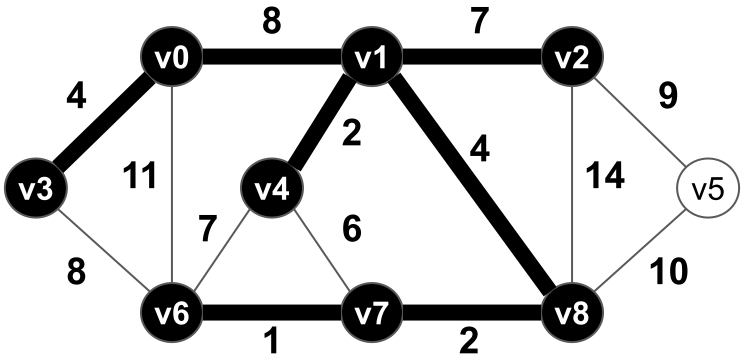
Fila de prioridade:

v6	v6	v5	v5	v6	v2
7	8	9	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	0	1	1	1

Algoritmo de Prim - Implementação



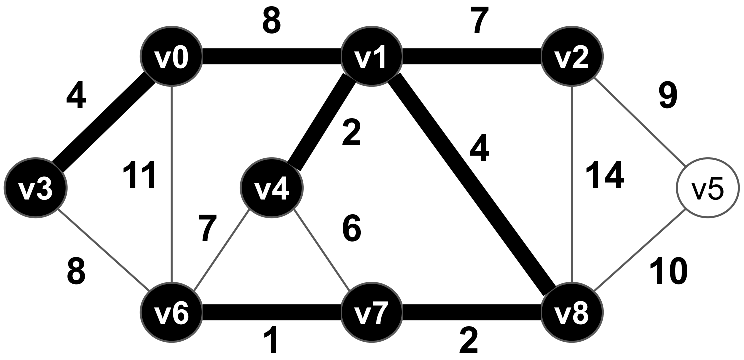
Fila de prioridade:

v6	v5	v5	v6	v2
8	9	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	0	1	1	1

Algoritmo de Prim - Implementação



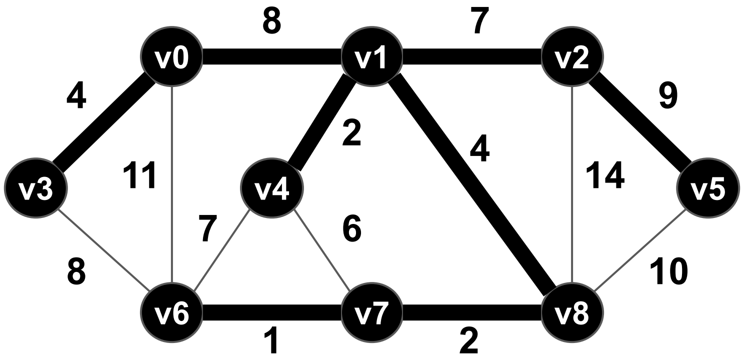
Fila de prioridade:

v5	v5	v6	v2
9	10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	0	1	1	1

Algoritmo de Prim - Implementação



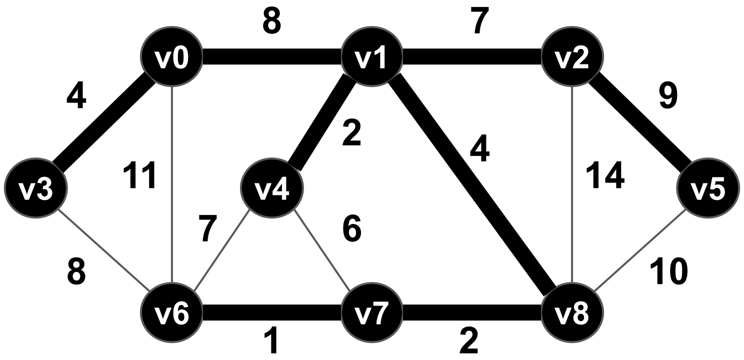
Fila de prioridade:

v5	v6	v2
10	11	14

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1

Algoritmo de Prim - Implementação



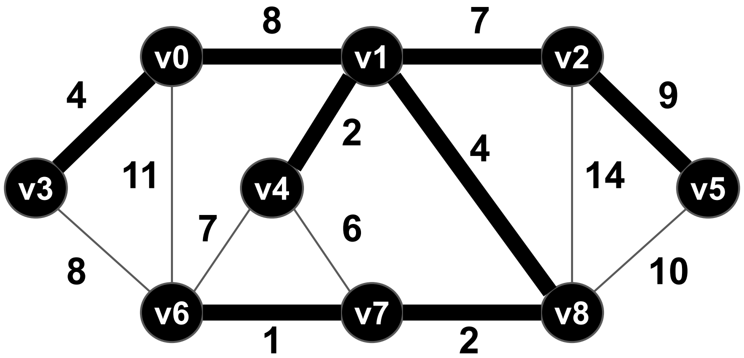
Fila de prioridade:

v6	v2
11	14

na_arvore:

1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8

Algoritmo de Prim - Implementação

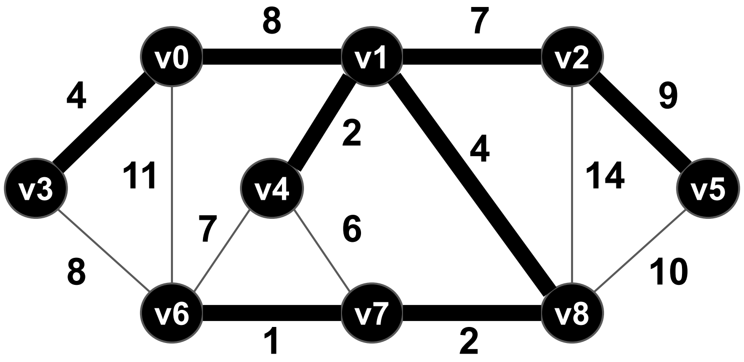


Fila de prioridade:

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1

Algoritmo de Prim - Implementação



Fila de prioridade:

na_arvore:

0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1

Algoritmo de Prim - Implementação

Prim(*G* conexo)

1. Para cada vértice *w* de *G*:
2.   *na_arvore*[*w*] = 0
3. *peso_arvore* = 0
4. Crie uma fila de prioridade *Q*
5. Adicione a *Q* o vértice 0 com prioridade 0
6. Enquanto *Q* não está vazia:
7.   Remova o item de menor prioridade de *Q*; seja *v* o vértice do item removido e *prio* a sua prioridade
8.   Se *na_arvore*[*v*] == 0:
9.   *na_arvore*[*v*] = 1
10.   *peso_arvore* = *peso_arvore* + *prio*
11.   Para cada vizinho *w* de *v* em *G*:
12.   Se *na_arvore*[*w*] == 0:
13.   Adicione a *Q* o vértice *w* com prioridade igual ao peso da aresta *vw*
14. Retorne *peso_arvore*

> O laço dos Passos 6 a 13 pode ser encerrado após todos os vértices de *G* terem sido adicionados à árvore que estamos construindo

Conteúdo

- Árvores geradoras de peso mínimo
- Árvores geradoras de peso mínimo - Algoritmo de Prim
- **Árvores geradoras de peso mínimo - Algoritmo de Kruskal**
- Caminhos de peso mínimo
- Caminhos de peso mínimo - Algoritmo de Dijkstra
- Referências

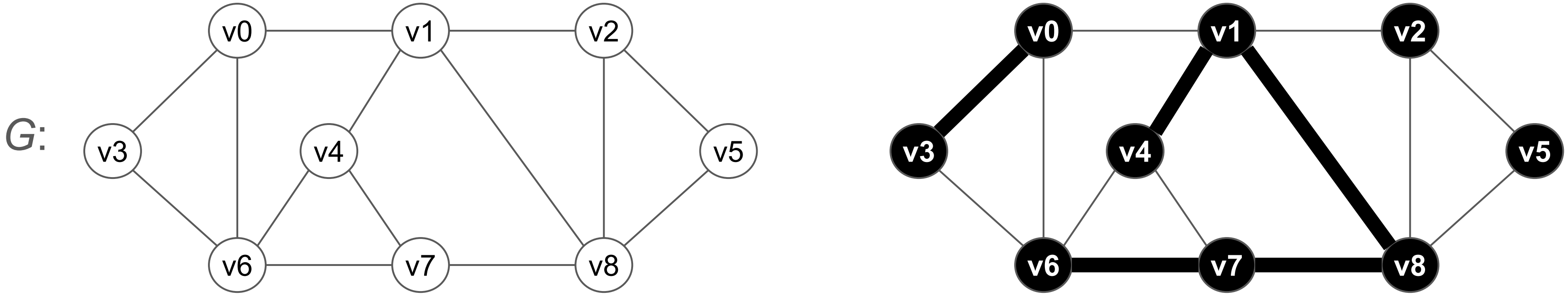
Como encontrar uma árvore geradora de peso mínimo

- Vimos o Algoritmo de Prim para encontrar uma árvore geradora de peso mínimo
- Este algoritmo usa a estratégia de adicionar arestas (e vértices) a uma árvore até que seja formada uma árvore geradora de peso mínimo
- Agora, veremos outra estratégia para obter uma árvore geradora de peso mínimo

Floresta geradora

- Um subgrafo de um grafo *G* que é gerador e acíclico é denominado uma **floresta geradora** de *G*

- Exemplo:



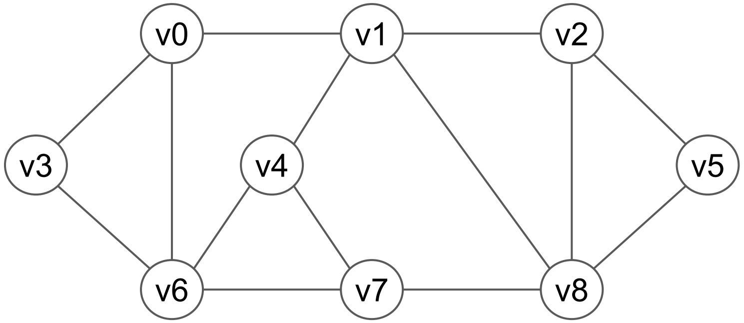
Floresta geradora de *G*

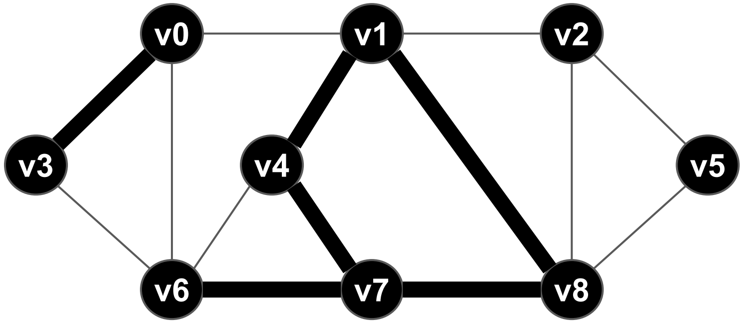
Floresta geradora

- Um subgrafo de um grafo *G* que é gerador e acíclico é denominado uma **floresta geradora** de *G*

- Exemplo:

G:





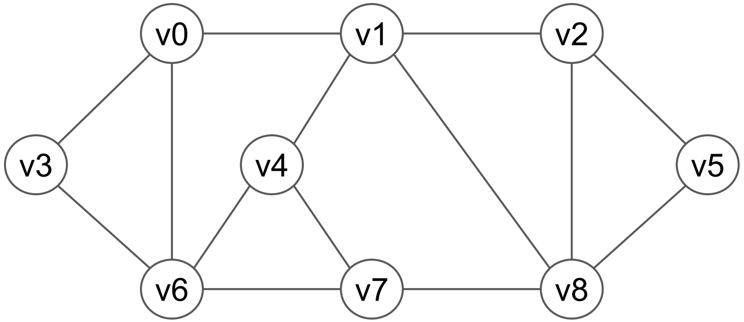
Floresta geradora de *G*

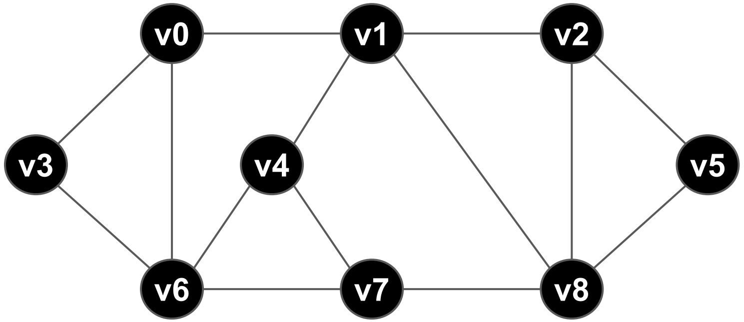
Floresta geradora

- Um subgrafo de um grafo *G* que é gerador e acíclico é denominado uma **floresta geradora** de *G*

- Exemplo:

G:





Floresta geradora de *G*

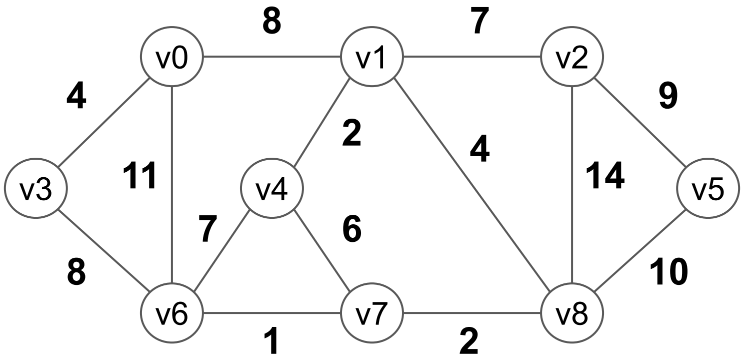
Algoritmo de Kruskal

Kruskal(G conexo)

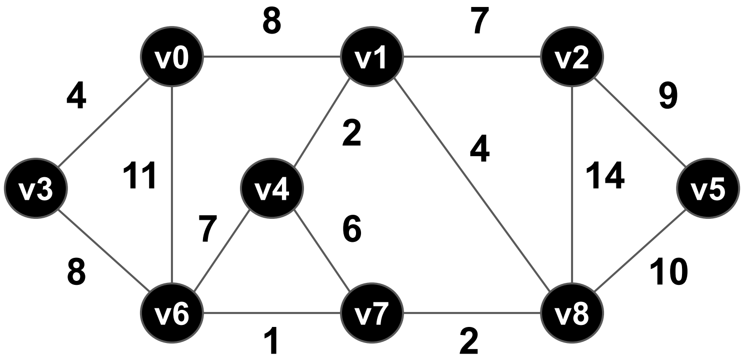
Inicialmente, F é a floresta formada por todos os vértices de G e por nenhuma aresta. Em outras palavras, F é formada por $V(G)$ árvores que contêm apenas um vértice

1. $F = (V(G), \varnothing)$
2. Ordene as arestas de G em ordem crescente de peso
3. Para cada aresta uv de G considerando as arestas de G em ordem crescente de peso:
4. Se uv conecta duas componentes conexas de F (duas árvores de F) diferentes:
5. Adicione uv a F
6. Retorne F

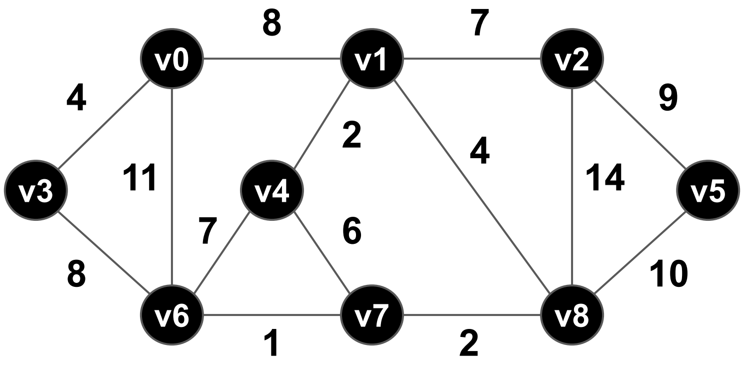
Algoritmo de Kruskal



Algoritmo de Kruskal



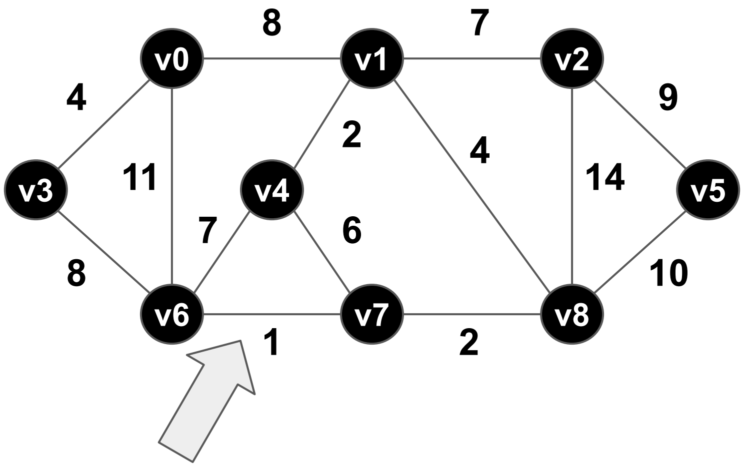
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

v6 v7
v1 v4
v7 v8
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

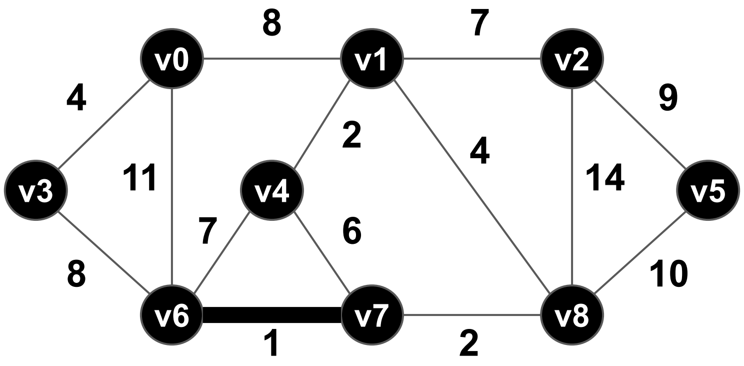
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

v6 v7
v1 v4
v7 v8
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

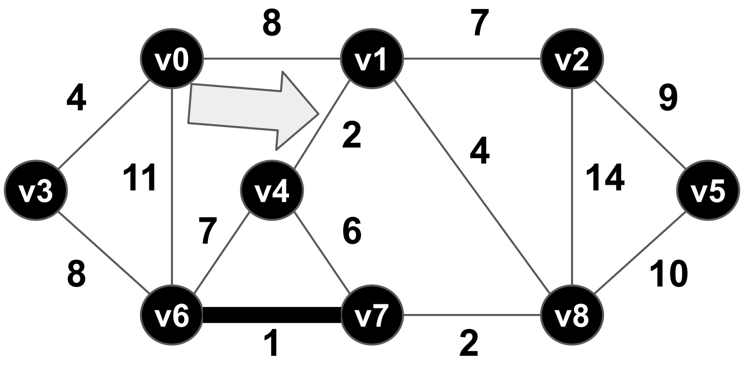
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
v1 v4
v7 v8
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

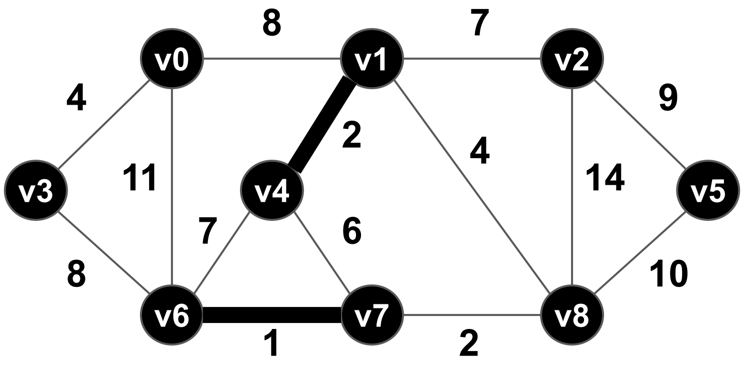
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
v1 v4
v7 v8
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

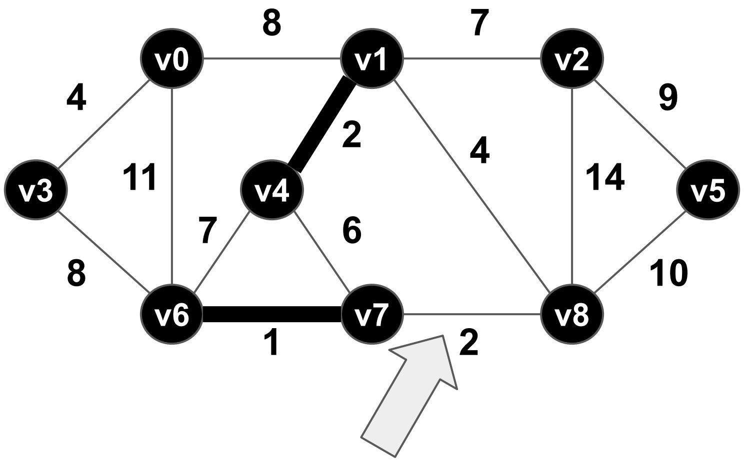
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
v7 v8
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

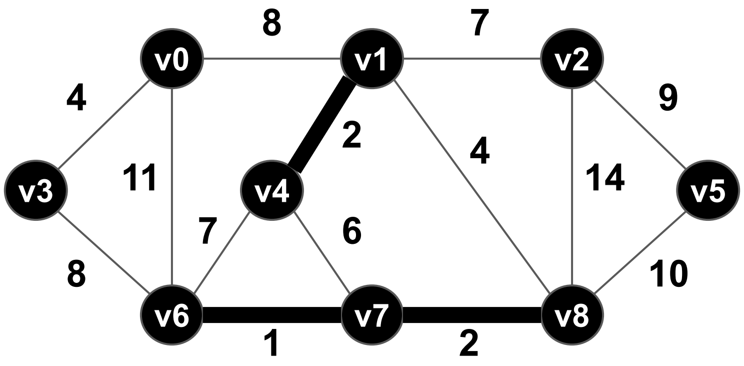
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
v7 v8
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

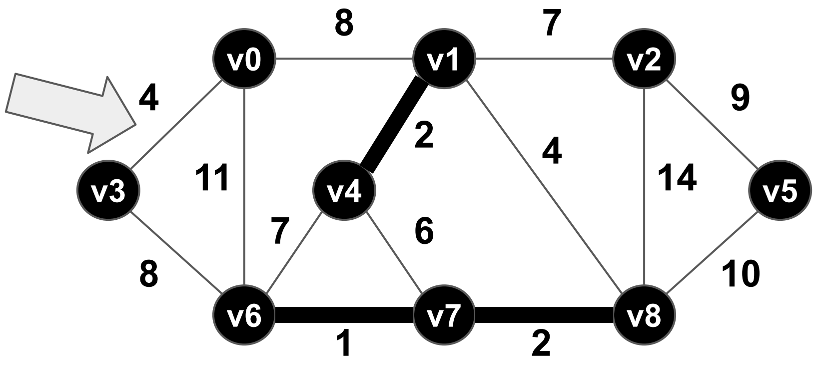
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

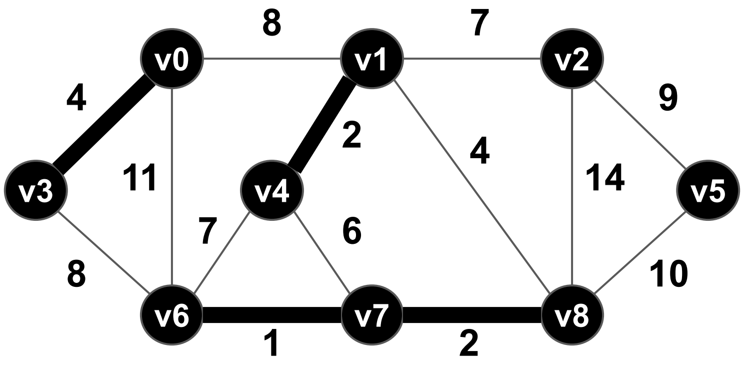
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
v0 v3
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

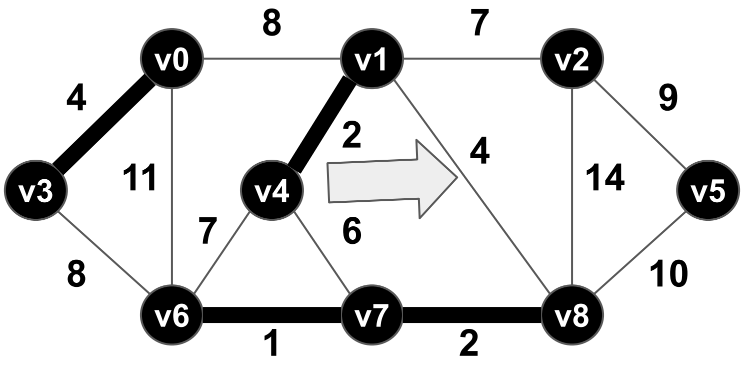
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

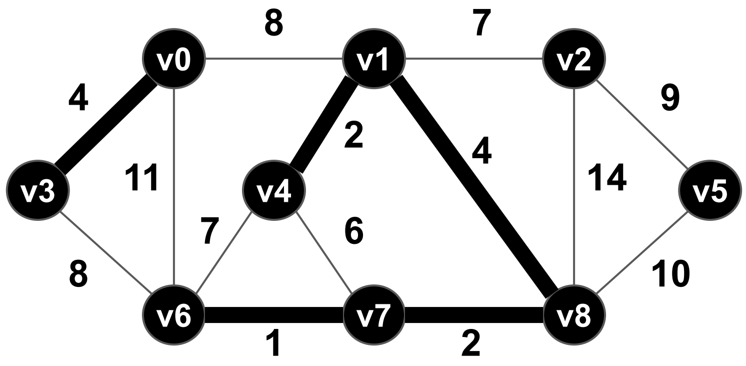
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
v1 v8
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

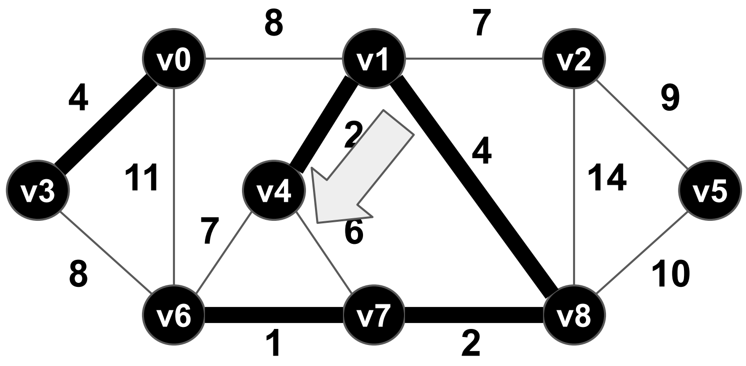
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

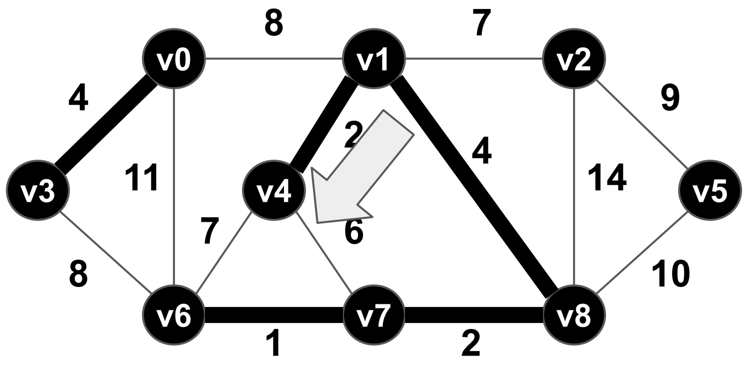
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

Algoritmo de Kruskal

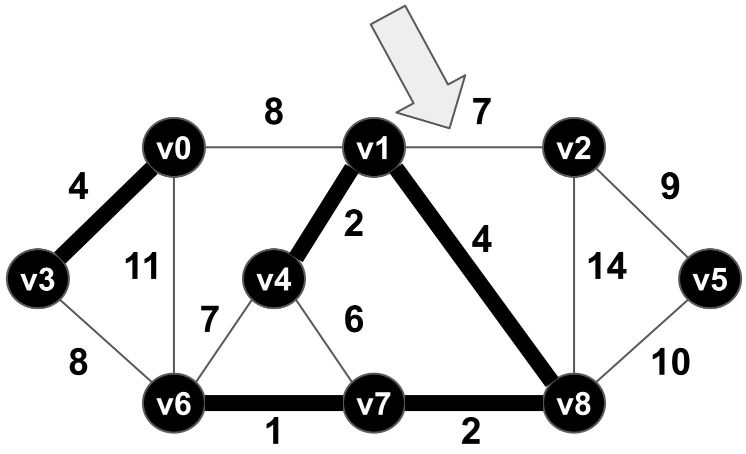


v4 v7 **não conecta duas árvores diferentes**.
Portanto, não é adicionada à floresta

Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
v4 v7
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

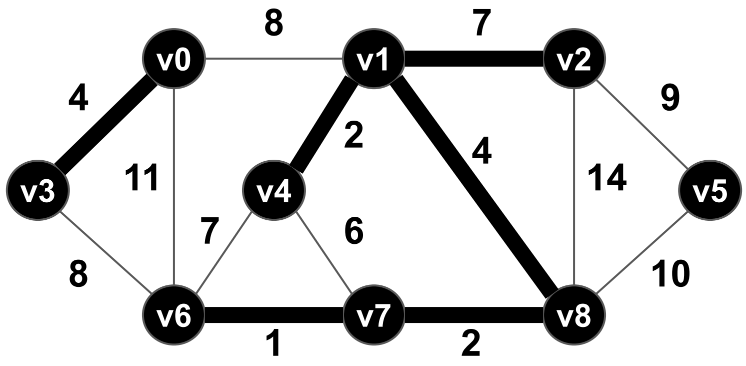
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
v1 v2
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

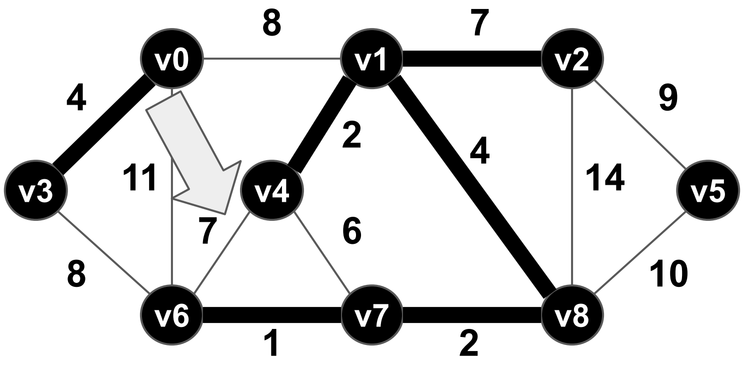
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

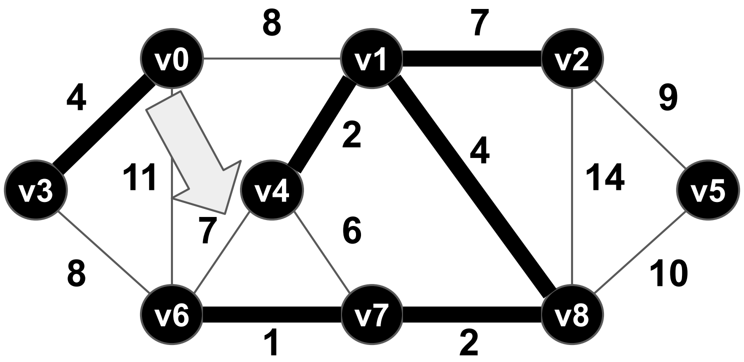
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

Algoritmo de Kruskal

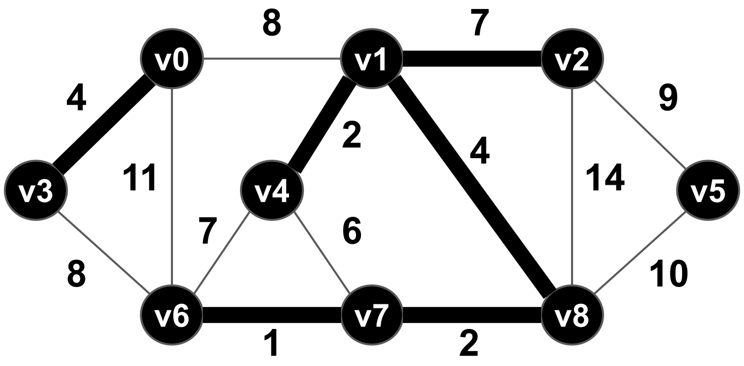


v4 v6 **não conecta duas árvores diferentes**.
Portanto, não é adicionada à floresta

Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
v4 v6
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

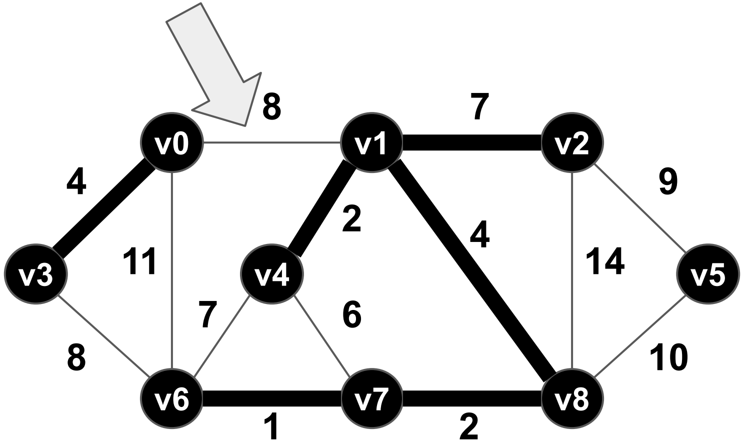
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

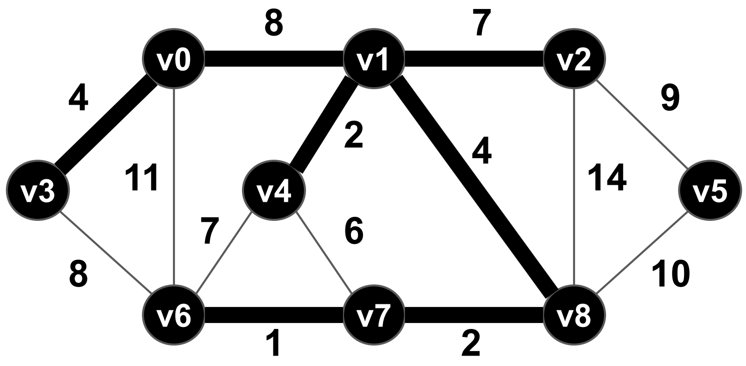
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
v0 v1
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

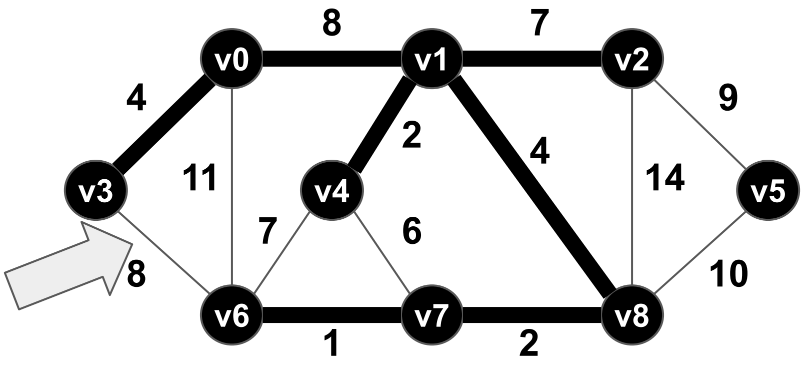
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

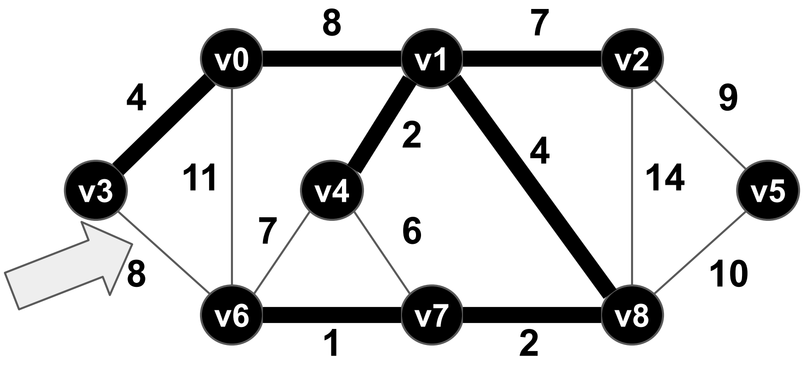
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

Algoritmo de Kruskal

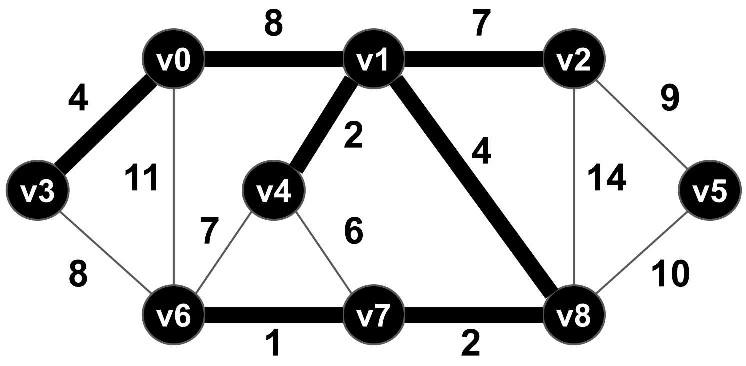


v3 v6 **não conecta duas árvores diferentes**.
Portanto, não é adicionada à floresta

Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
v3 v6
v2 v5
v5 v8
v0 v6
v2 v8

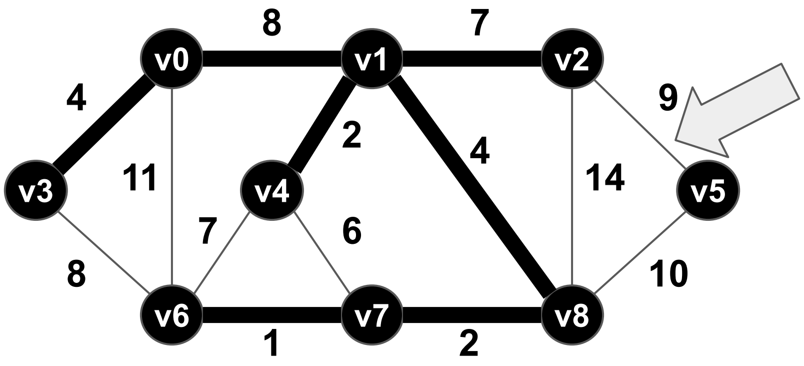
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
v2 v5
v5 v8
v0 v6
v2 v8

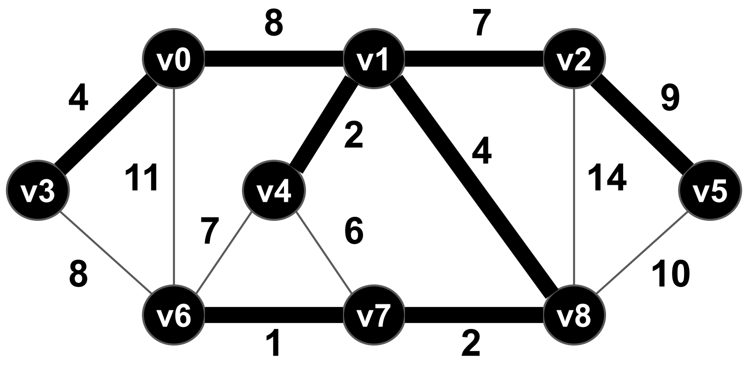
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
v2 v5
v5 v8
v0 v6
v2 v8

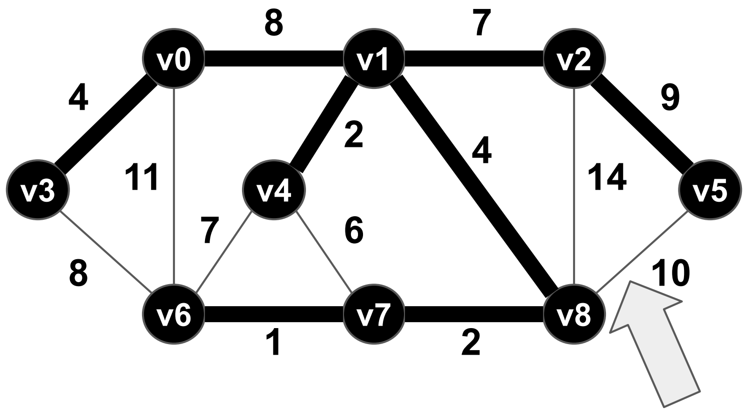
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
v5 v8
v0 v6
v2 v8

Algoritmo de Kruskal



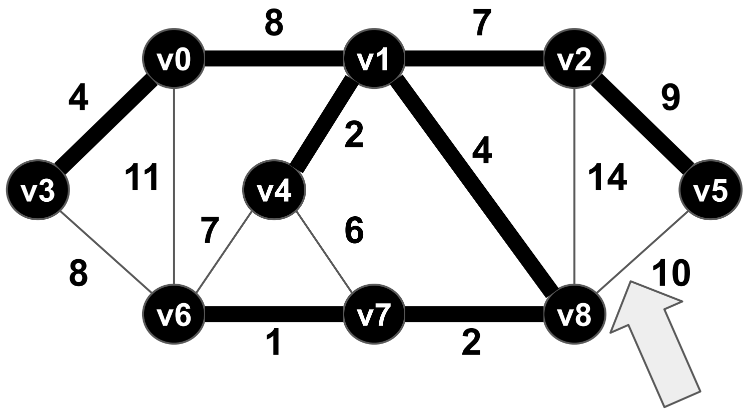
Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
v5 v8
v0 v6
v2 v8

Algoritmo de Kruskal

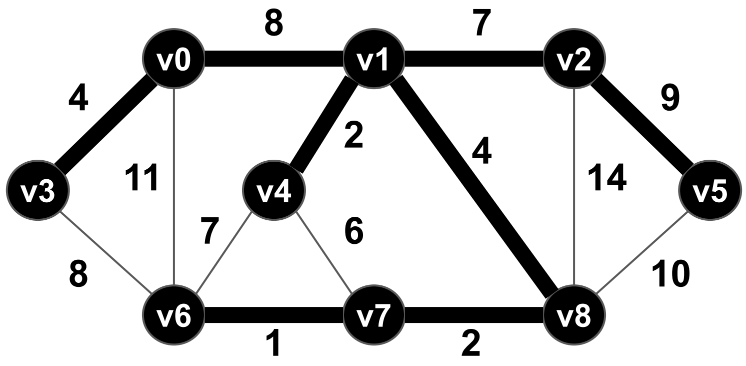
Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
v5 v8
v0 v6
v2 v8



v5 v8 **não conecta duas árvores diferentes**.
Portanto, não é adicionada à floresta

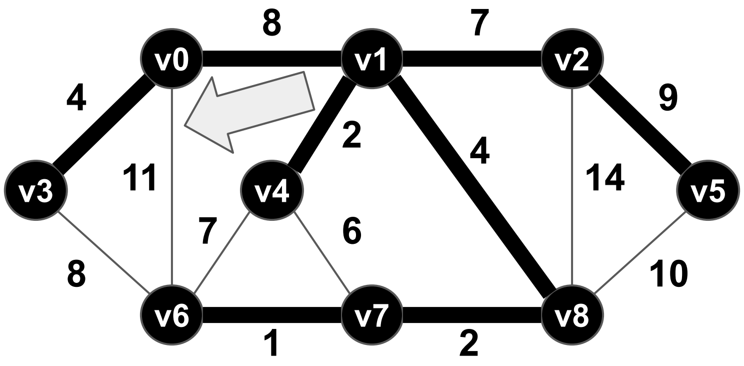
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
v0 v6
v2 v8

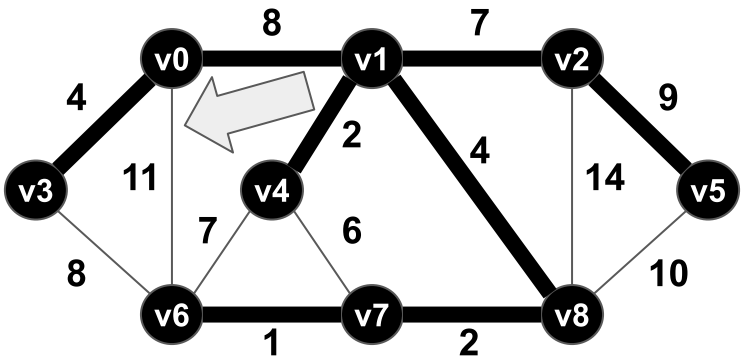
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
v0 v6
v2 v8

Algoritmo de Kruskal

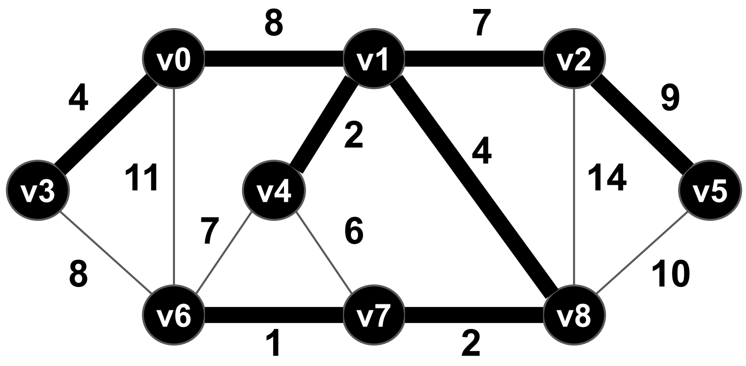


v0 v6 **não conecta duas árvores diferentes**.
Portanto, não é adicionada à floresta

Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
v0 v6
v2 v8

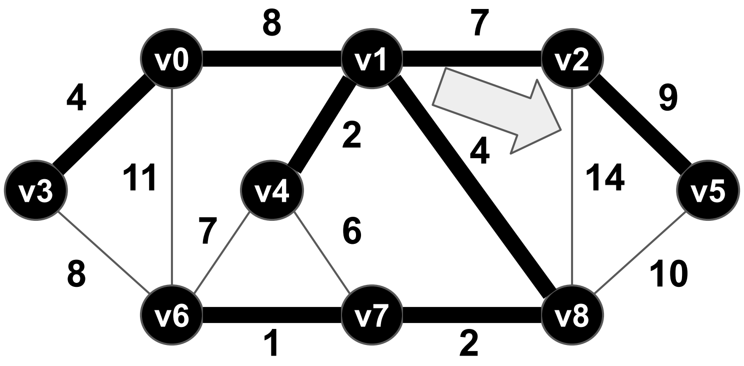
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
~~v0 v6~~
v2 v8

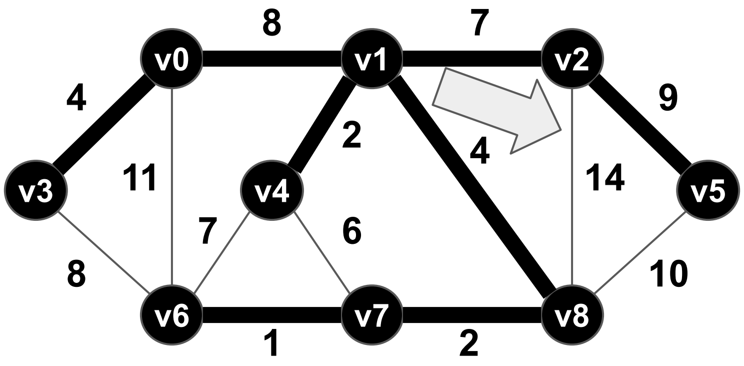
Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
~~v0 v6~~
v2 v8

Algoritmo de Kruskal

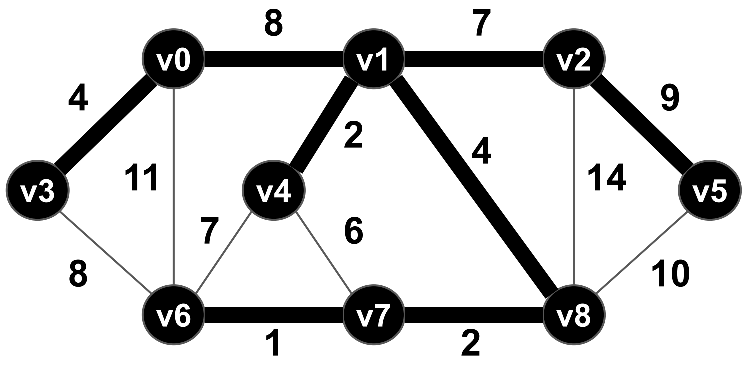


v2 v8 **não conecta duas árvores diferentes**.
Portanto, não é adicionada à floresta

Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
~~v0 v6~~
v2 v8

Algoritmo de Kruskal



Arestas em ordem crescente de peso:

~~v6 v7~~
~~v1 v4~~
~~v7 v8~~
~~v0 v3~~
~~v1 v8~~
~~v4 v7~~
~~v1 v2~~
~~v4 v6~~
~~v0 v1~~
~~v3 v6~~
~~v2 v5~~
~~v5 v8~~
~~v0 v6~~
~~v2 v8~~

Algoritmo de Kruskal

- Para implementar o Algoritmo de Kruskal, podemos usar uma estrutura de dados conhecida como **conjuntos-disjuntos** (*disjoint-sets* ou *union-find*)
- Uma estrutura de dados deste tipo mantém uma coleção de conjuntos disjuntos, associando a cada conjunto um **representante**, que é um dos elementos do conjunto
- Tipicamente, podemos realizar três operações em uma estrutura de dados de conjuntos disjuntos:
  - CriaConjuntos(n): Para $i = 0, 1, \ldots, n - 1$, cria um conjunto que contém apenas i
  - UneConjuntos(x, y): Faz a união do conjunto que contém x com o conjunto que contém y
  - EncontraConjunto(x): Retorna o representante do conjunto que contém x

Algoritmo de Kruskal

- Para implementar o Algoritmo de Kruskal, podemos usar uma estrutura de dados conhecida como **conjuntos-disjuntos** (*disjoint-sets* ou *union-find*)
- Uma implementação em C++ do Algoritmo de Kruskal pode ser encontrada nas Refs. 1 e 2 (veja o último slide)

Algoritmo de Kruskal

O laço dos Passos 4 a 7 pode ser encerrado após *n* - 1 arestas terem sido adicionadas à floresta que estamos construindo

Kruskal(*G* conexo)

1. *peso_floresta* = 0
2. CriaConjuntos(*n*); sendo *n* o número de vértices de *G*
3. Ordene as arestas de *G* em ordem crescente de peso
4. Para cada aresta *uv* de *G* considerando as arestas de *G* em ordem crescente de peso:
5.   Se EncontraConjunto(u) ≠ EncontraConjunto(v):
6.   *peso_floresta* = *peso_floresta* + peso da aresta *uv*
7.   UneConjuntos(*u*, *v*)
8. Retorne *peso_floresta*

Conteúdo

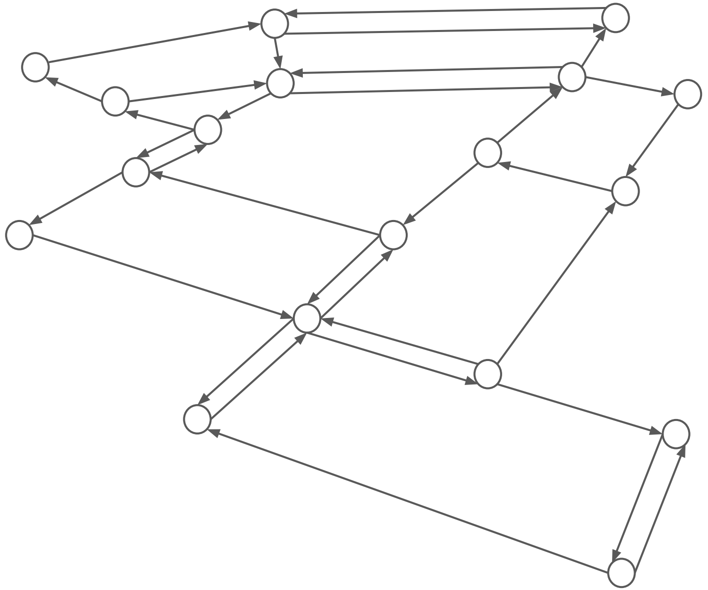
- Árvores geradoras de peso mínimo
- Árvores geradoras de peso mínimo - Algoritmo de Prim
- Árvores geradoras de peso mínimo - Algoritmo de Kruskal
- **Caminhos de peso mínimo**
- Caminhos de peso mínimo - Algoritmo de Dijkstra
- Referências

Grafos dirigidos (Digrafos)

- Vimos situações que podem ser modeladas com grafos dirigidos (digrafos) – grafos cujas arestas têm uma **direção** (ou **orientação** ou **sentido**)
- Exemplo:
  - Temos um mapa de vias (ruas ou rodovias) e estamos interessados nos caminhos que podemos percorrer neste mapa
  - Uma via que conecta um ponto *x* a um ponto *y* pode ter apenas a mão de *x* para *y*, apenas a mão de *y* para *x* ou ambas as mãos
  - Podemos representar este mapa como um grafo onde cada **aresta** tem uma direção e representa **uma mão de uma via**

Grafos dirigidos (Digrafos)

- Vimos situações que podem ser modeladas com grafos dirigidos (digrafos) – grafos cujas arestas têm uma **direção** (ou **orientação** ou **sentido**)
- Exemplo:
  - Temos um mapa de vias (ruas ou rodovias) e estamos interessados nos caminhos que podemos percorrer neste mapa
  - Uma via que conecta um ponto *x* a um ponto *y* pode ter apenas a mão de *x* para *y*, apenas a mão de *y* para *x* ou ambas as mãos
  - Podemos representar este mapa como um grafo onde cada **aresta** tem uma direção e representa **uma mão de uma via**

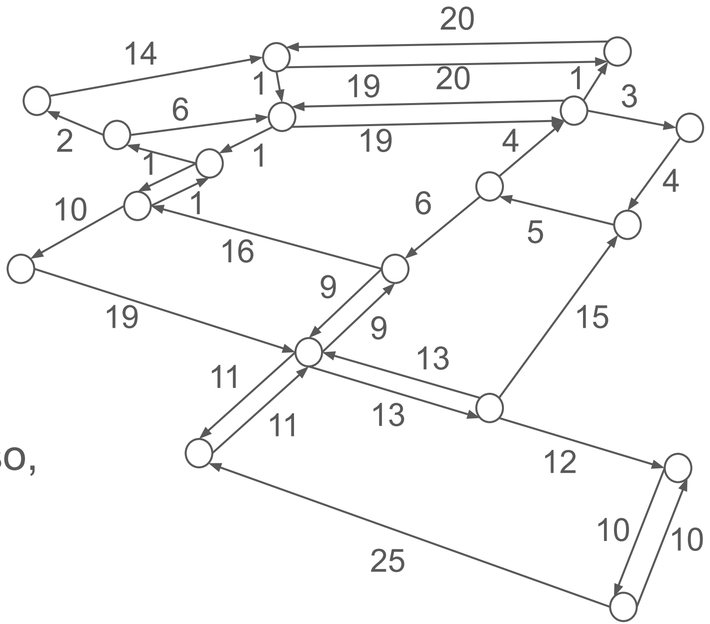


Digrafos com pesos nas arestas

- Em situações como estas, também pode fazer sentido considerarmos pesos nas arestas do digrafo
- Exemplo:
  - Temos um mapa de vias (ruas ou rodovias) e estamos interessados em caminhos **curtos** (considerando as **distâncias no mapa**) que podemos percorrer neste mapa
  - Podemos representar este mapa como um grafo onde cada aresta tem uma direção, que representa uma mão de uma via, e tem um peso, que representa a distância no mapa entre os pontos conectados

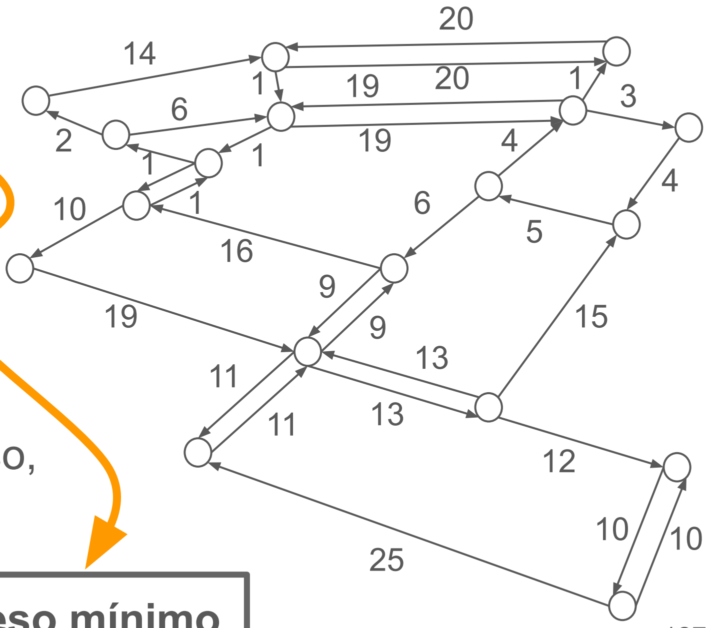
Digrafos com pesos nas arestas

- Em situações como estas, também pode fazer sentido considerarmos pesos nas arestas do digrafo
- Exemplo:
  - Temos um mapa de vias (ruas ou rodovias) e estamos interessados em caminhos **curtos** (considerando as **distâncias no mapa**) que podemos percorrer neste mapa
  - Podemos representar este mapa como um grafo onde cada aresta tem uma direção, que representa uma mão de uma via, e tem um peso, que representa a distância no mapa entre os pontos conectados



Digrafos com pesos nas arestas

- Em situações como estas, também pode fazer sentido considerarmos pesos nas arestas do digrafo
- Exemplo:
  - Temos um mapa de vias (ruas ou rodovias) e estamos interessados em caminhos **curtos** (considerando as **distâncias no mapa**) que podemos percorrer neste mapa
  - Podemos representar este mapa como um grafo onde cada aresta tem uma direção, que representa uma mão de uma via, e tem um peso, que representa a distância no mapa entre os pontos conectados

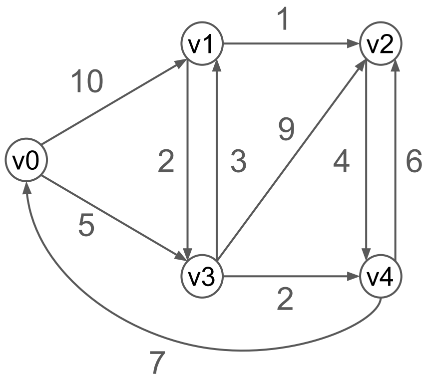


caminhos de **peso mínimo**

Peso de um caminho

- O **peso** de um caminho em um digrafo *G* é a soma dos pesos das arestas do caminho (o mesmo vale para um passeio e um ciclo)

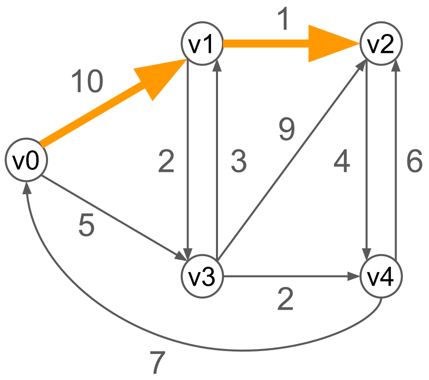
- Exemplo:
    - No digrafo ao lado,
        - O peso do caminho $v_0v_1v_2$



Peso de um caminho

- O **peso** de um caminho em um digrafo *G* é a soma dos pesos das arestas do caminho (o mesmo vale para um passeio e um ciclo)

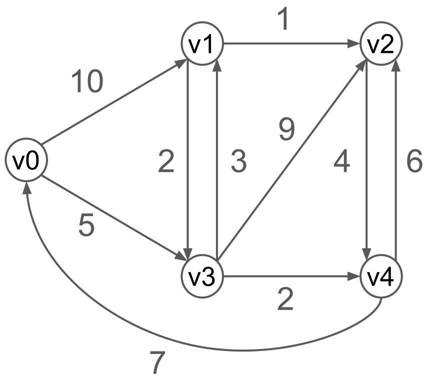
- Exemplo:
  - No digrafo ao lado,
    - O peso do caminho $v_0v_1v_2$ é 11



Peso de um caminho

- O **peso** de um caminho em um digrafo *G* é a soma dos pesos das arestas do caminho (o mesmo vale para um passeio e um ciclo)

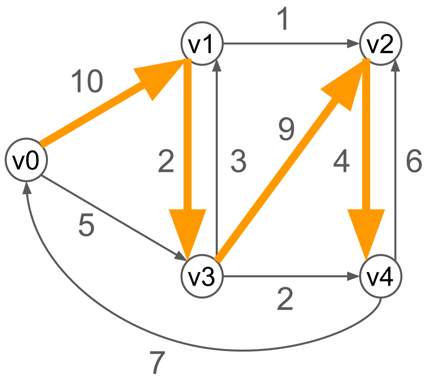
- Exemplo:
  - No digrafo ao lado,
    - O peso do caminho $v_0 v_1 v_2$ é 11
    - O peso do caminho $v_0 v_1 v_3 v_2 v_4$



Peso de um caminho

- O **peso** de um caminho em um digrafo G é a soma dos pesos das arestas do caminho (o mesmo vale para um passeio e um ciclo)

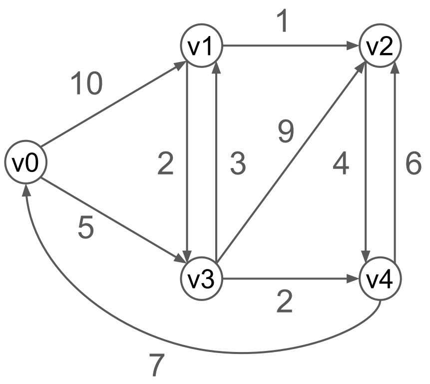
- Exemplo:
  - No digrafo ao lado,
    - O peso do caminho $v_0 v_1 v_2$ é 11
    - O peso do caminho $v_0 v_1 v_3 v_2 v_4$ é 25



Distância ponderada

- A **distância ponderada** de um vértice v_i para um vértice v_j em um digrafo G, denotada por $dp(v_i , v_j)$, é
  - o menor peso de um v_iv_j-caminho em G ou
  - ∞ (infinita) caso não exista um v_iv_j-caminho em G
- Note que, em geral, $dp(v_i , v_j) \neq dp(v_j , v_i)$

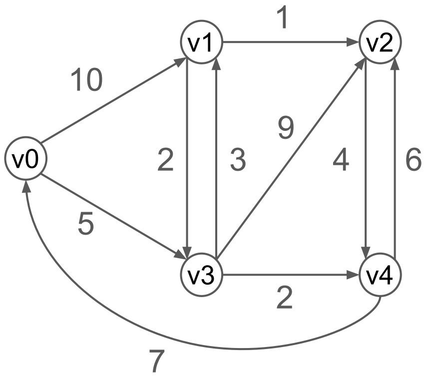
- Exemplo:
  - No digrafo ao lado,
    - $dp(v_0, v_1) =$ e $dp(v_1, v_0) =$,
    - $dp(v_3, v_2) =$ e $dp(v_2, v_3) =$ e
    - $dp(v_4, v_4) =$



Distância ponderada

- A **distância ponderada** de um vértice v_i para um vértice v_j em um digrafo G, denotada por $dp(v_i\ , v_j)$, é
  - o menor peso de um v_iv_j-caminho em G ou
  - ∞ (infinita) caso não exista um v_iv_j-caminho em G
- Note que, em geral, $dp(v_i\ , v_j) \neq dp(v_j\ , v_i)$

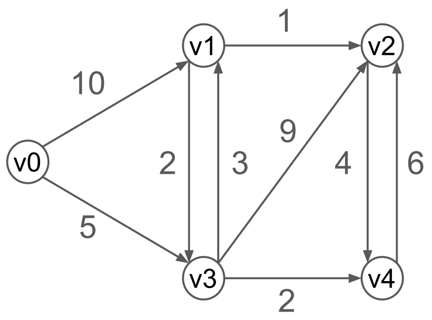
- Exemplo:
  - No digrafo ao lado,
    - $dp(v_0, v_1) = 8$ e $dp(v_1, v_0) = 11$,
    - $dp(v_3, v_2) = 4$ e $dp(v_2, v_3) = 16$ e
    - $dp(v_4, v_4) = 0$



Distância ponderada

- A **distância ponderada** de um vértice v_i para um vértice v_j em um digrafo G, denotada por $dp(v_i, v_j)$, é
  - o menor peso de um v_iv_j-caminho em G ou
  - ∞ (infinita) caso não exista um v_iv_j-caminho em G
- Note que, em geral, $dp(v_i, v_j) \neq dp(v_j, v_i)$

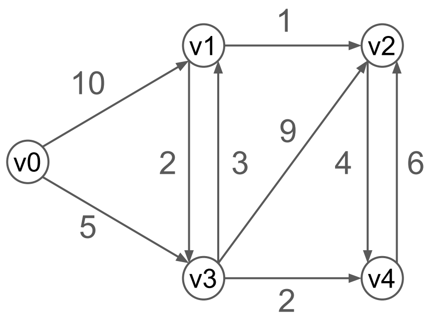
- Exemplo:
  - No digrafo ao lado,
    - $dp(v_1, v_0) =$



Distância ponderada

- A **distância ponderada** de um vértice v_i para um vértice v_j em um digrafo G, denotada por $dp(v_i , v_j)$, é
  - o menor peso de um $v_i v_j$-caminho em G ou
  - ∞ (infinita) caso não exista um $v_i v_j$-caminho em G
- Note que, em geral, $dp(v_i , v_j) \neq dp(v_j , v_i)$

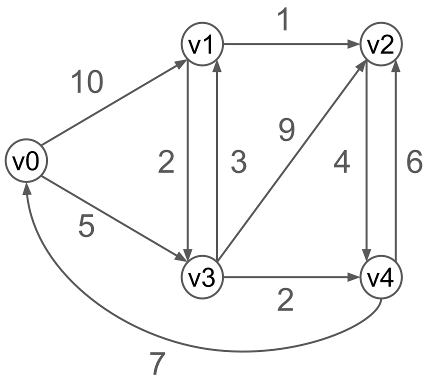
- Exemplo:
  - No digrafo ao lado,
    - $dp(v_1, v_0) = \infty$



Problema dos caminhos de peso mínimo

- **Problema:** Dado um digrafo *G* e um vértice *s* de *G*, encontre, para cada vértice *v* de *G*, um *sv*-caminho de peso mínimo em *G*

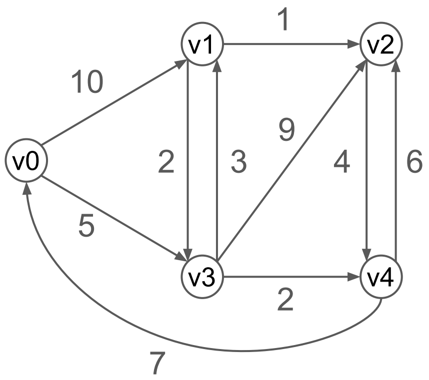
- Exemplo:
  - Sendo *G* dado pelo digrafo ao lado e *s* igual a v_0, uma solução para o problema é
    - $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
    - $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
    - v_0v_3 – $dp(v_0, v_3) = 5$ e
    - $v_0v_3v_4$ – $dp(v_0, v_4) = 7$



Problema dos caminhos de peso mínimo

- **Problema:** Dado um digrafo G e um vértice s de G, encontre, para cada vértice v de G, um sv-caminho de peso mínimo em G
- Existem algumas variações interessantes deste problema

- Exemplo:
  - Sendo G dado pelo digrafo ao lado e s igual a v_0, uma solução para o problema é
    - $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
    - $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
    - v_0v_3 – $dp(v_0, v_3) = 5$ e
    - $v_0v_3v_4$ – $dp(v_0, v_4) = 7$



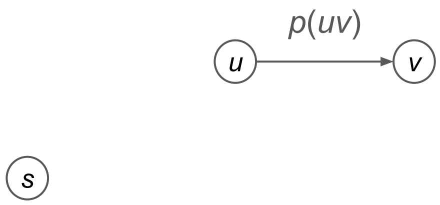
Conteúdo

Relaxação de uma aresta

- O algoritmo que veremos para resolver o problema dos caminhos de peso mínimo se baseia em uma operação chamada de **relaxação de uma aresta**
- Antes de ver o algoritmo, vamos entender esta operação

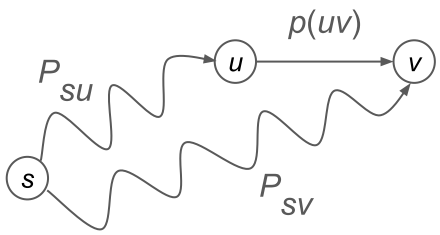
Relaxação de uma aresta

- Considere um digrafo G, um vértice s de G, e uma aresta uv de G com peso $p(uv)$



Relaxação de uma aresta

- Considere um digrafo G, um vértice s de G, e uma aresta uv de G com peso $p(uv)$
- Seja P_{su} um su-caminho de peso mínimo em G e P_{sv} um sv-caminho de peso mínimo em G



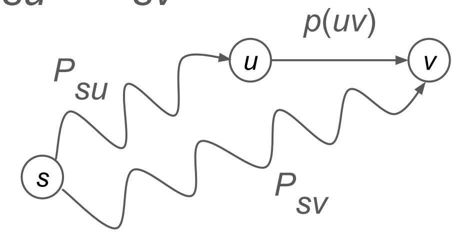
Relaxação de uma aresta

- Considere um digrafo G, um vértice s de G, e uma aresta uv de G com peso $p(uv)$
- Seja P_{su} um su-caminho de peso mínimo em G e P_{sv} um sv-caminho de peso mínimo em G
- O que podemos dizer sobre os pesos de P_{su} e P_{sv}?

  peso de P_{sv} **?** peso de P_{su} + $p(uv)$

  $\leq$ **?**

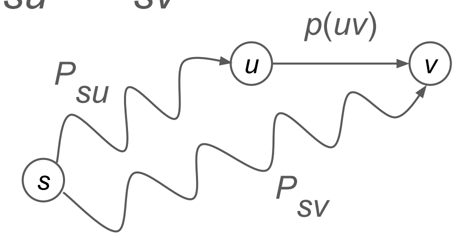
  $\geq$ **?**



Relaxação de uma aresta

- Considere um digrafo G, um vértice s de G, e uma aresta uv de G com peso $p(uv)$
- Seja P_{su} um su-caminho de peso mínimo em G e P_{sv} um sv-caminho de peso mínimo em G
- O que podemos dizer sobre os pesos de P_{su} e P_{sv}?

peso de P_{sv} $\leq$ peso de P_{su} + $p(uv)$



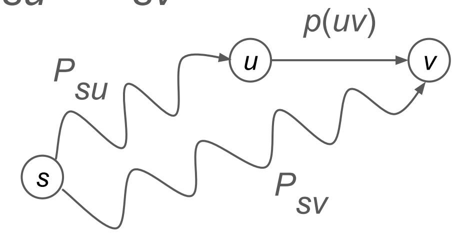
Relaxação de uma aresta

- Considere um digrafo G, um vértice s de G, e uma aresta uv de G com peso $p(uv)$
- Seja P_{su} um su-caminho de peso mínimo em G e P_{sv} um sv-caminho de peso mínimo em G
- O que podemos dizer sobre os pesos de P_{su} e P_{sv}?

  peso de P_{sv} $\leq$ peso de P_{su} + $p(uv)$

- Consequentemente,

  $dp(s,v) \leq dp(s,u) + p(uv)$

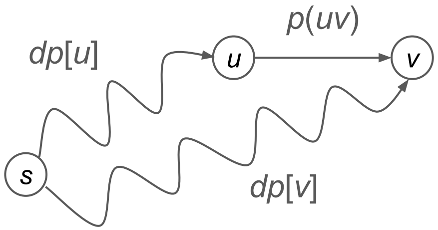


Relaxação de uma aresta

- O algoritmo que veremos a seguir usa um vetor *dp* sobre o qual podemos falar o seguinte:
  - Durante a execução do algoritmo,
    - *dp*[*u*] contém o menor peso de um *su*-caminho encontrado até o momento e
    - *dp*[*v*] contém o menor peso de um *sv*-caminho encontrado até o momento
  - Ao fim da execução do algoritmo,
    - *dp*[*u*] contém o peso mínimo de um *su*-caminho, ou seja, *dp*[*u*] = *dp*(*s*,*u*) e
    - *dp*[*v*] contém o peso mínimo de um *sv*-caminho, ou seja, *dp*[*v*] = *dp*(*s*,*v*)
- O que podemos dizer sobre *dp*[*u*] e *dp*[*v*]?

Relaxação de uma aresta

- O algoritmo que veremos a seguir usa um vetor *dp* sobre o qual podemos falar o seguinte:
  - Durante a execução do algoritmo,
    - *dp*[*u*] contém o menor peso de um *su*-caminho encontrado até o momento e
    - *dp*[*v*] contém o menor peso de um *sv*-caminho encontrado até o momento
  - Ao fim da execução do algoritmo,
    - *dp*[*u*] contém o peso mínimo de um *su*-caminho, ou seja, *dp*[*u*] = *dp*(*s*,u) e
    - *dp*[*v*] contém o peso mínimo de um *sv*-caminho, ou seja, *dp*[*v*] = *dp*(*s*,v)
- O que podemos dizer sobre *dp*[*u*] e *dp*[*v*]?

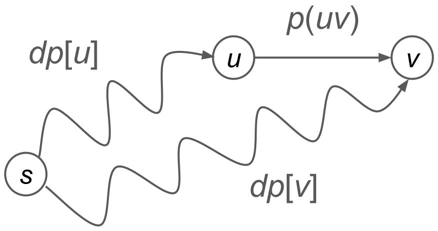


Relaxação de uma aresta

- O algoritmo que veremos a seguir usa um vetor *dp* sobre o qual podemos falar o seguinte:
  - Durante a execução do algoritmo,
    - *dp*[*u*] contém o menor peso de um *su*-caminho encontrado até o momento e
    - *dp*[*v*] contém o menor peso de um *sv*-caminho encontrado até o momento
  - Ao fim da execução do algoritmo,
    - *dp*[*u*] contém o peso mínimo de um *su*-caminho, ou seja, *dp*[*u*] = *dp*(*s*,u) e
    - *dp*[*v*] contém o peso mínimo de um *sv*-caminho, ou seja, *dp*[*v*] = *dp*(*s*,v)

- O que podemos dizer sobre *dp*[*u*] e *dp*[*v*]?

  Se *dp*[*v*] > *dp*[*u*] + *p*(*uv*), então *dp*[*v*] ainda não contém o peso mínimo de um *sv*-caminho

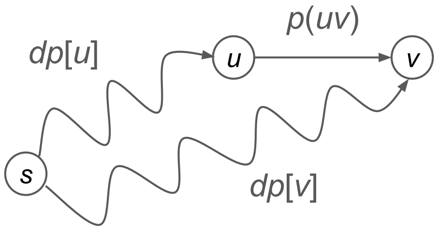


Relaxação de uma aresta

- O algoritmo que veremos a seguir usa um vetor *dp* sobre o qual podemos falar o seguinte:
    - Durante a execução do algoritmo,
        - *dp*[*u*] contém o menor peso de um *su*-caminho encontrado até o momento e
        - *dp*[*v*] contém o menor peso de um *sv*-caminho encontrado até o momento
    - Ao fim da execução do algoritmo,
        - *dp*[*u*] contém o peso mínimo de um *su*-caminho, ou seja, *dp*[*u*] = *dp*(*s*,u) e
        - *dp*[*v*] contém o peso mínimo de um *sv*-caminho, ou seja, *dp*[*v*] = *dp*(*s*,v)

- O que podemos dizer sobre *dp*[*u*] e *dp*[*v*]?

  Se *dp*[*v*] > *dp*[*u*] + *p*(*uv*), então *dp*[*v*] ainda não contém o peso mínimo de um *sv*-caminho e podemos fazer *dp*[*v*] = dp[*u*] + *p*(*uv*)

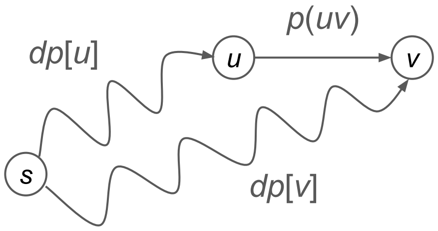


Relaxação de uma aresta

- O algoritmo que veremos a seguir usa um vetor *dp* sobre o qual podemos falar o seguinte:
  - Durante a execução do algoritmo,
    - *dp*[*u*] contém o menor peso de um *su*-caminho encontrado até o momento e
    - *dp*[*v*] contém o menor peso de um *sv*-caminho encontrado até o momento
  - Ao fim da execução do algoritmo,
    - *dp*[*u*] contém o peso mínimo de um *su*-caminho, ou seja, *dp*[*u*] = *dp*(*s*,u) e
    - *dp*[*v*] contém o peso mínimo de um *sv*-caminho, ou seja, *dp*[*v*] = *dp*(*s*,v)

- O que podemos dizer sobre *dp*[*u*] e *dp*[*v*]?

  Ao fazer com que *dp*[*v*] ≤ *dp*[*u*] + *p*(*uv*), podemos dizer que esta **restrição** está satisfeita ou **relaxada**, que **relaxamos** esta **restrição**, ou ainda que **relaxamos** esta **aresta**



Relaxação de uma aresta

- Assim, a operação de relaxação da aresta *uv* consiste no seguinte:
    1. Se $dp[v] > \mathrm{dp}[u] + p(uv)$:
    2. $\quad dp[v] = dp[u] + p(uv)$

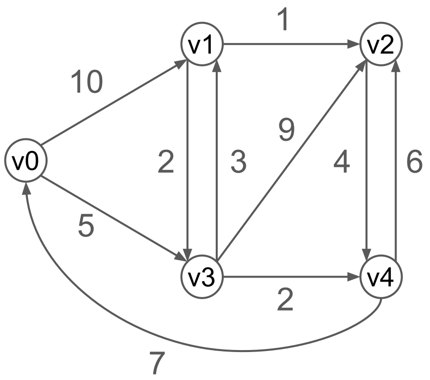
Representação dos caminhos de peso mínimo

- O algoritmo que veremos também se baseia em uma árvore que vai representar os caminhos de peso mínimo
- Esta árvore terá as seguintes propriedades:
  - O vértice *s* será a raiz da árvore
  - Para todo vértice *v*, o caminho entre *s* e *v* na árvore no sentido de *s* para *v* corresponderá a um *sv*-caminho de peso mínimo no digrafo *G*

Representação dos caminhos de peso mínimo

- Esta árvore terá as seguintes propriedades:
  - O vértice *s* será a raiz da árvore
  - Para todo vértice *v*, o caminho entre *s* e *v* na árvore no sentido de *s* para *v* corresponderá a um *sv*-caminho de peso mínimo no digrafo *G*

- Exemplo:



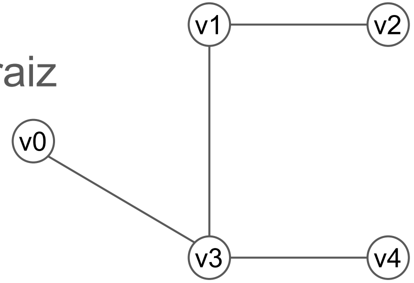
Caminhos de peso mínimo

s é igual a v_0

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

Árvore

raiz



Arestas com pesos não-negativos

- Vamos considerar o problema dos caminhos de peso mínimo no caso em que **todas as arestas** do digrafo possuem **peso não-negativo**
- Neste caso, o problema pode ser resolvido usando o Algoritmo de Dijkstra

- O Algoritmo de Dijkstra pode ser descrito de maneira semelhante ao Algoritmo de Prim

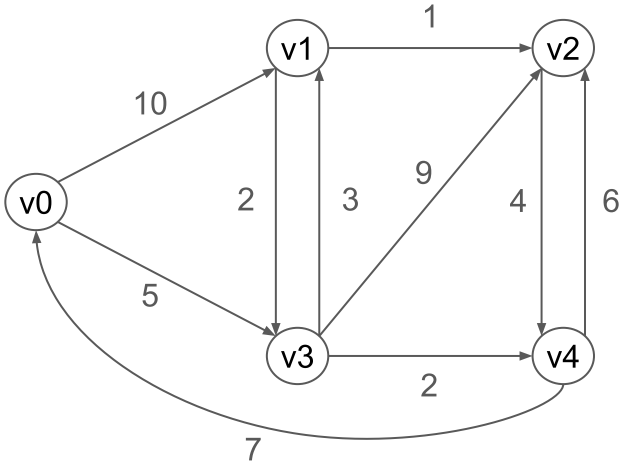
Algoritmo de Dijkstra

Inicialmente, T é uma árvore que consiste apenas no vértice s de G

Dijkstra(G, s)

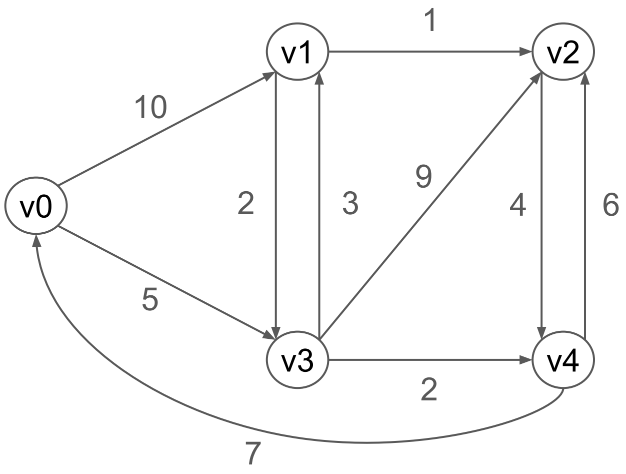
1. T = ({ s }, ∅)
2. Enquanto é possível aumentar T:
3. Encontre um vértice w de G tal que w não está em T e a distância ponderada de s para w é mínima
4. Encontre uma aresta xw de G tal que x está em T e xw completa um caminho de peso mínimo de s para w
5. Adicione xw a T
6. Retorne T

Algoritmo de Dijkstra



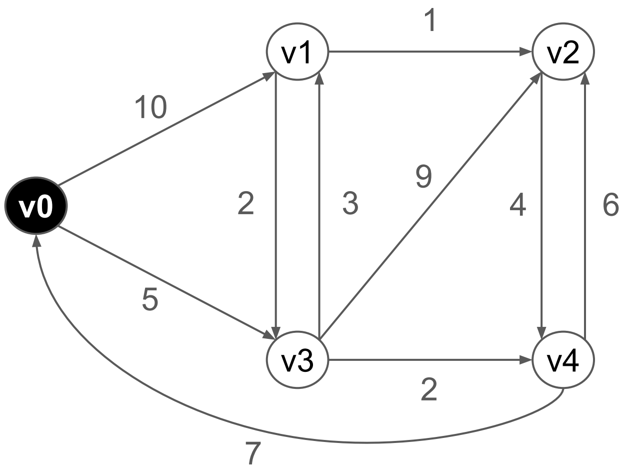
Algoritmo de Dijkstra

$s = v_0$



Algoritmo de Dijkstra

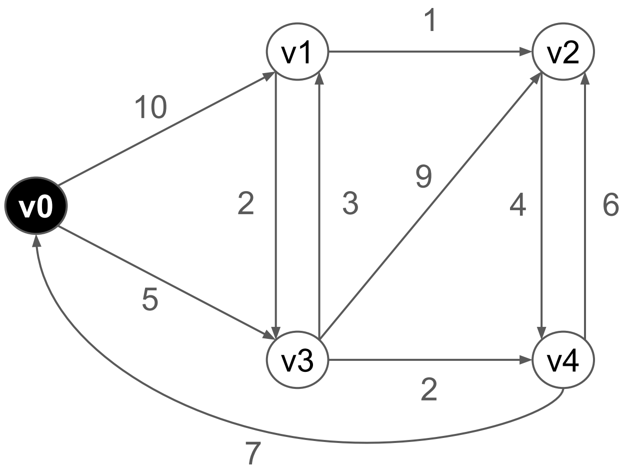
$s = v_0$



Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



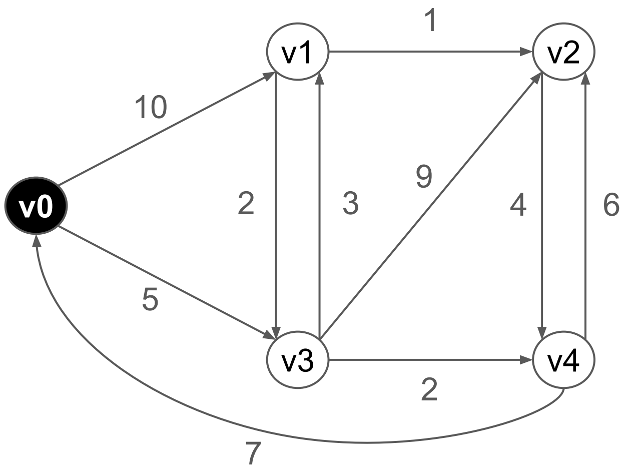
$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

3. Encontre um vértice *w* de *G* tal que *w* não está em *T* e a distância ponderada de *s* para *w* é mínima

$\boxed{v_3}$

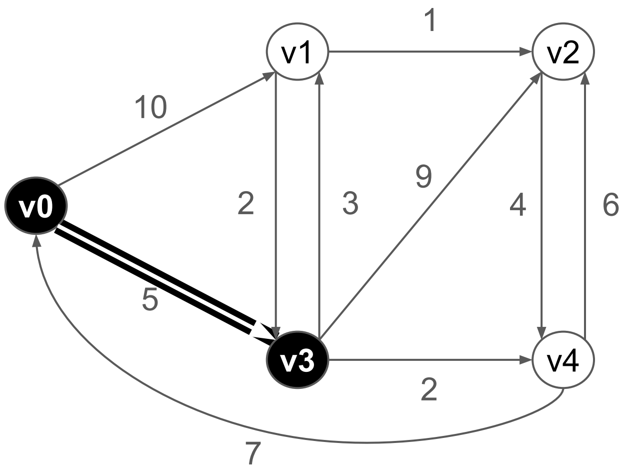
4. Encontre uma aresta *xw* de *G* tal que *x* está em *T* e *xw* completa um caminho de peso mínimo de *s* para *w*

$\boxed{v_0v_3}$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



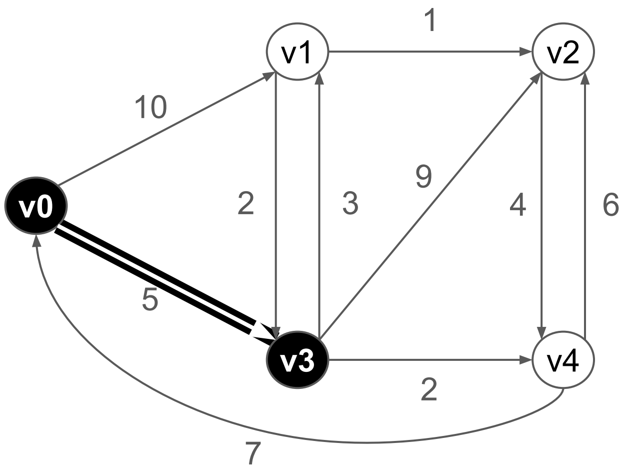
$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$
  e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

3. Encontre um vértice *w* de *G* tal que *w* não está em *T* e a distância ponderada de *s* para *w* é mínima

$\boxed{v_4}$

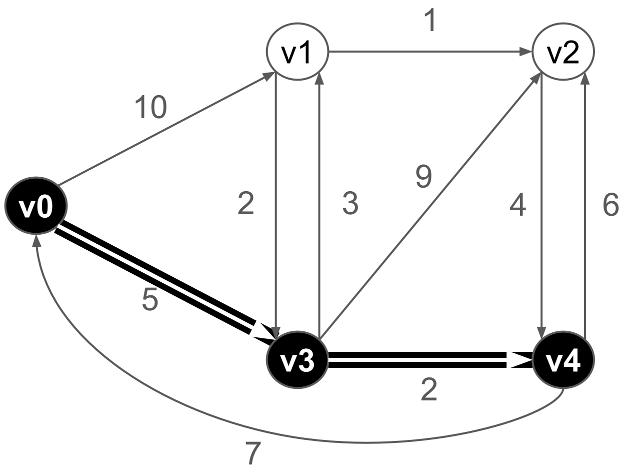
4. Encontre uma aresta *xw* de *G* tal que *x* está em *T* e *xw* completa um caminho de peso mínimo de *s* para *w*

$\boxed{v_3v_4}$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



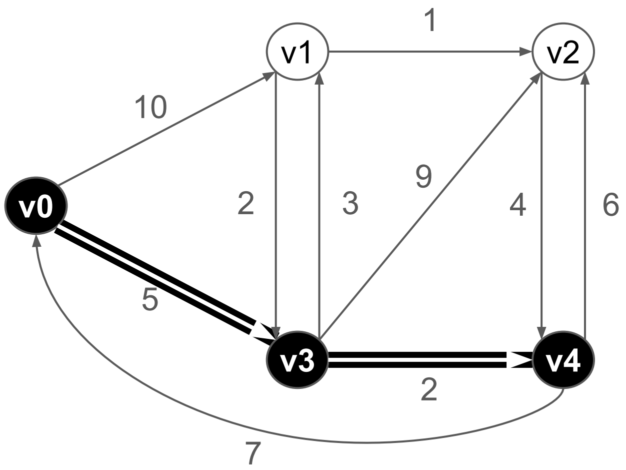
$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

3. Encontre um vértice *w* de *G* tal que *w* não está em *T* e a distância ponderada de *s* para *w* é mínima

$\boxed{v_1}$

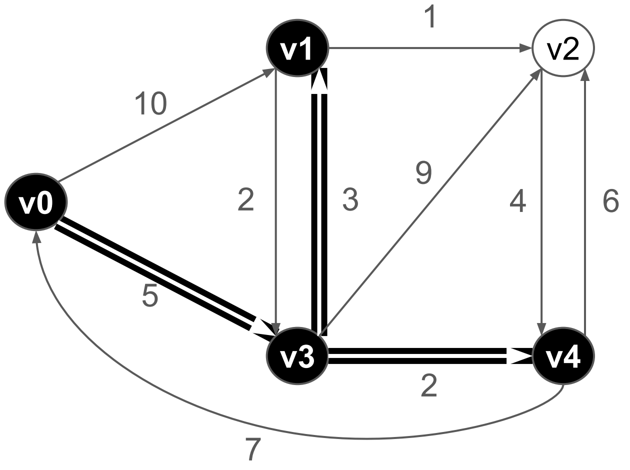
4. Encontre uma aresta *xw* de *G* tal que *x* está em *T* e *xw* completa um caminho de peso mínimo de *s* para *w*

$\boxed{v_3v_1}$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

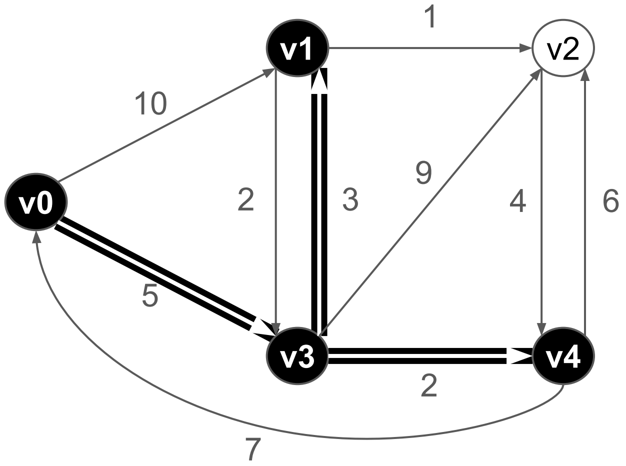
Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$

- $v_0 v_3 v_1$ – $dp(v_0, v_1) = 8$,
- $v_0 v_3 v_1 v_2$ – $dp(v_0, v_2) = 9$,
- $v_0 v_3$ – $dp(v_0, v_3) = 5$ e
- $v_0 v_3 v_4$ – $dp(v_0, v_4) = 7$

$s = v_0$



3. Encontre um vértice w de G tal que w não está em T e a distância ponderada de s para w é mínima

$\boxed{v_2}$

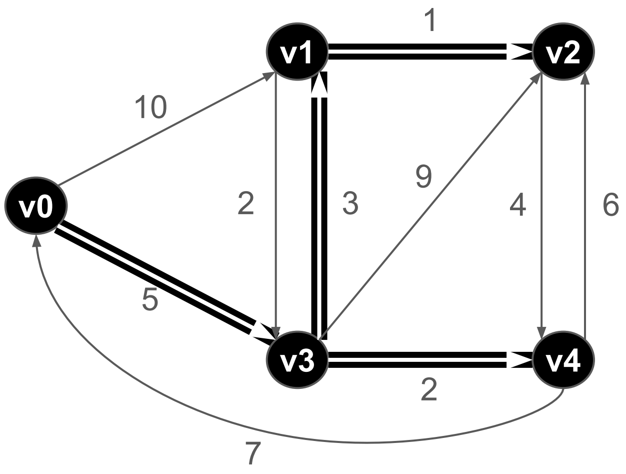
4. Encontre uma aresta xw de G tal que x está em T e xw completa um caminho de peso mínimo de s para w

$\boxed{v_1 v_2}$

Algoritmo de Dijkstra

Caminhos de peso mínimo

$s = v_0$



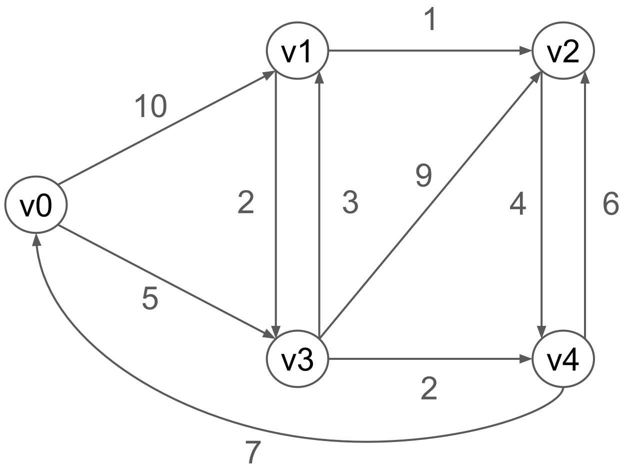
$s = v_0$

- $v_0v_3v_1$ – $dp(v_0, v_1) = 8$,
- $v_0v_3v_1v_2$ – $dp(v_0, v_2) = 9$,
- v_0v_3 – $dp(v_0, v_3) = 5$ e
- $v_0v_3v_4$ – $dp(v_0, v_4) = 7$

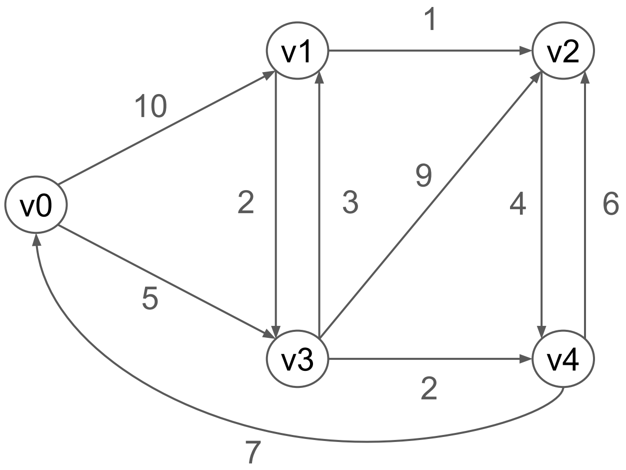
Algoritmo de Dijkstra - Implementação

- Na implementação do Algoritmo de Dijkstra, vamos usar uma fila de prioridade para executar de maneira eficiente o passo de determinar a próxima aresta a ser adicionada à árvore de caminhos de peso mínimo que estamos construindo
- Além disso, vamos usar um vetor *na_arvore* para registrar os vértices que já foram adicionados à árvore

Algoritmo de Dijkstra - Implementação



Algoritmo de Dijkstra - Implementação



dp:

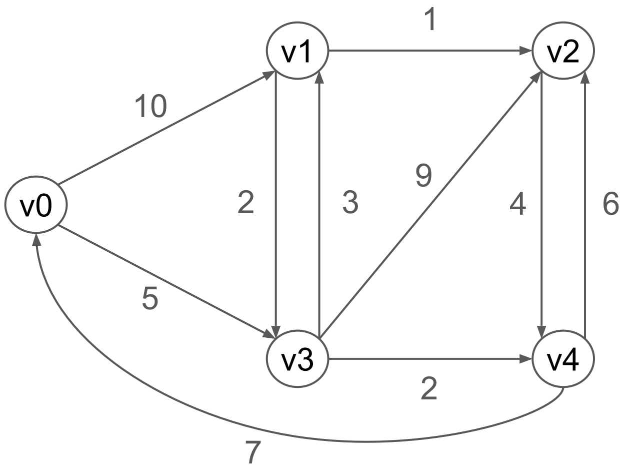
∞	∞	∞	∞	∞
0	1	2	3	4

na_arvore:

0	0	0	0	0
0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$



dp:

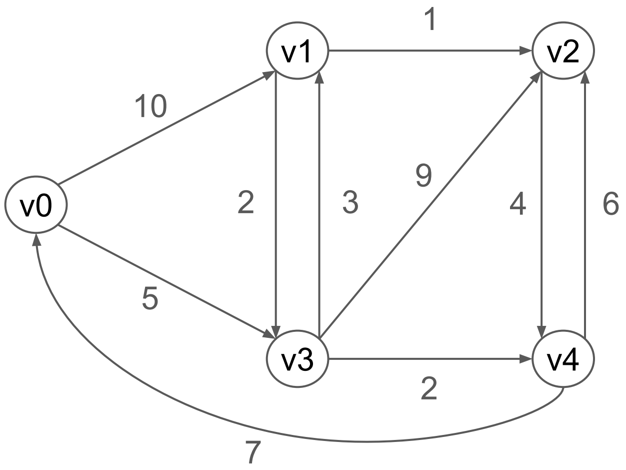
∞	∞	∞	∞	∞
0	1	2	3	4

na_arvore:

0	0	0	0	0
0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$

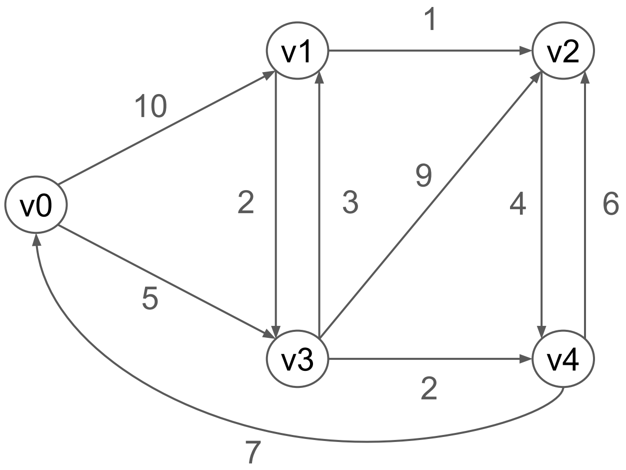


dp:	0	∞	∞	∞	∞
	0	1	2	3	4

na_arvore:	0	0	0	0	0
	0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:

dp:

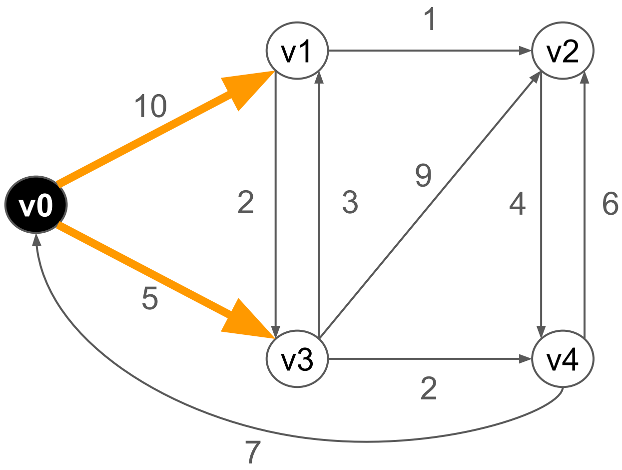
0	1	2	3	4
0	∞	∞	∞	∞

na_arvore:

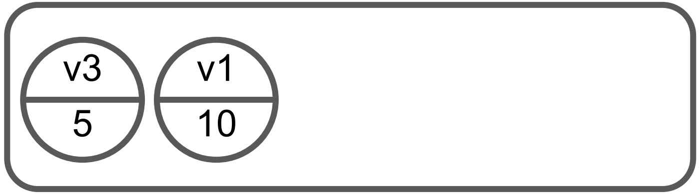
0	1	2	3	4
0	0	0	0	0

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

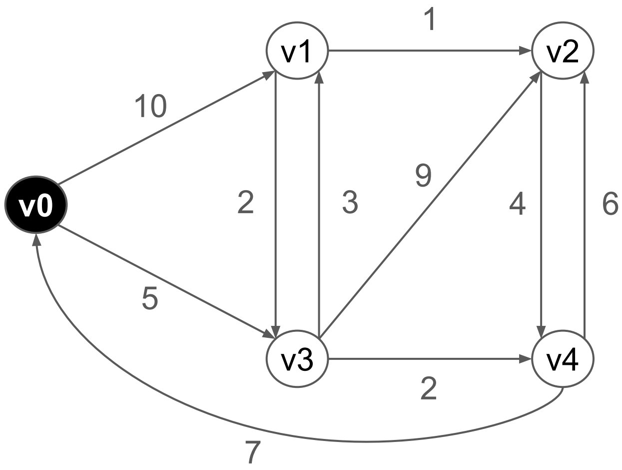
0	10	∞	5	∞
0	1	2	3	4

na_arvore:

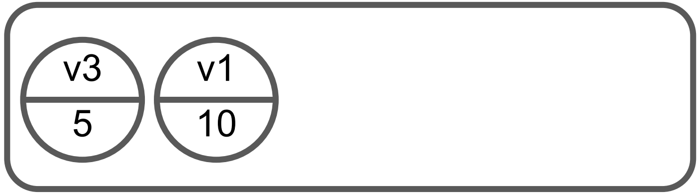
1	0	0	0	0
0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

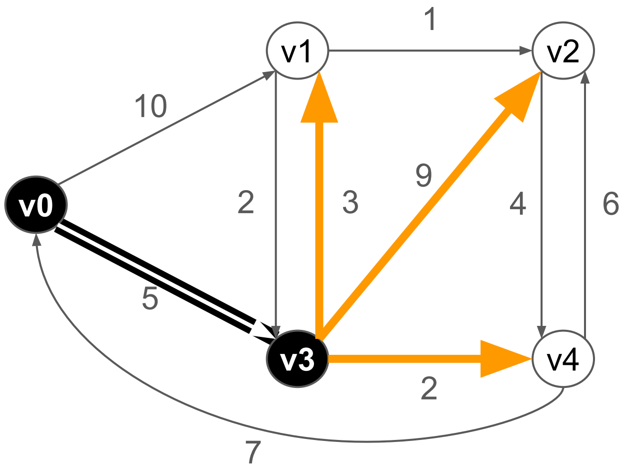
0	10	∞	5	∞
0	1	2	3	4

na_arvore:

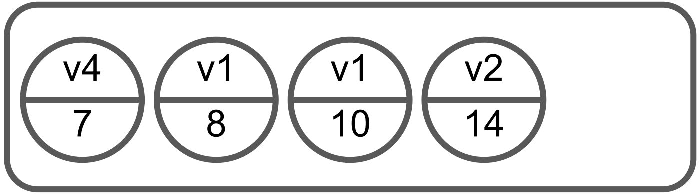
1	0	0	0	0
0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

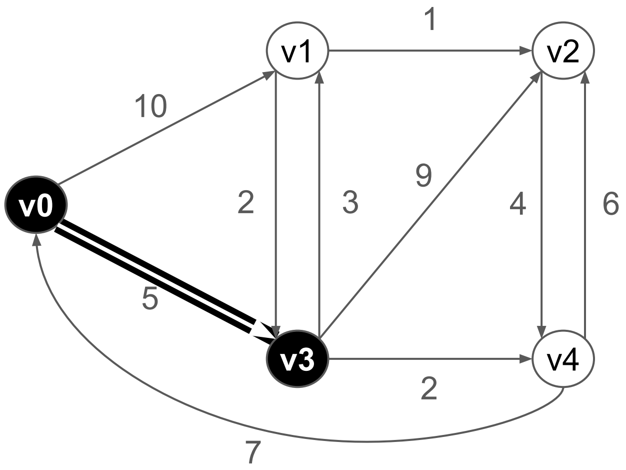
0	8	14	5	7
0	1	2	3	4

na_arvore:

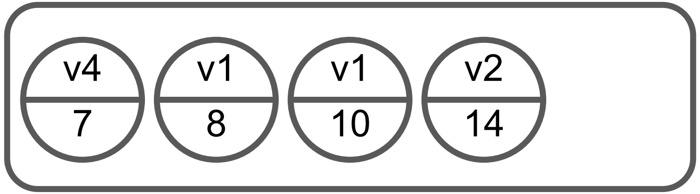
1	0	0	1	0
0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

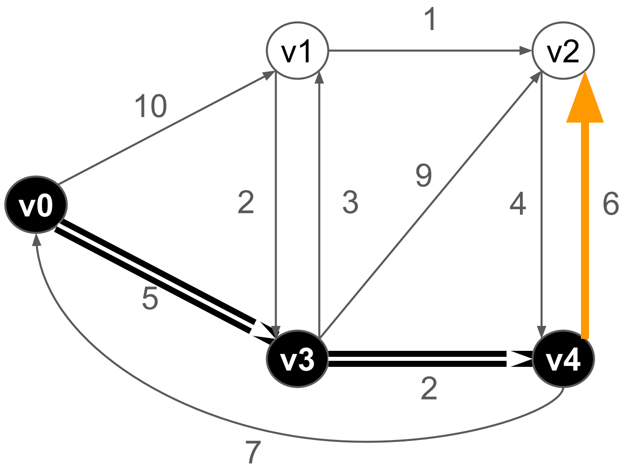
0	1	2	3	4
0	8	14	5	7

na_arvore:

0	1	2	3	4
1	0	0	1	0

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:

v1	v1	v2	v2
8	10	13	14

dp:

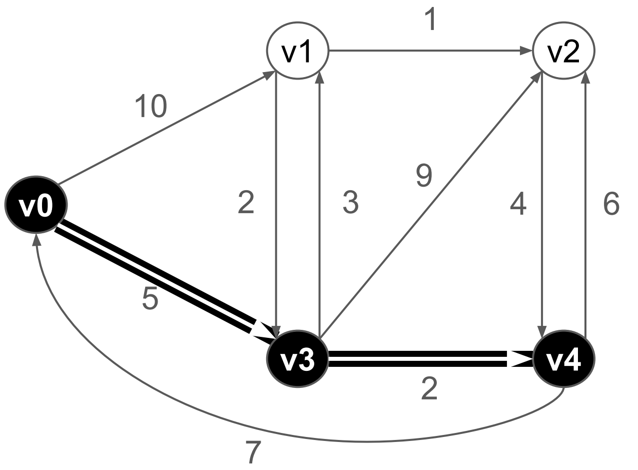
0	8	13	5	7
0	1	2	3	4

na_arvore:

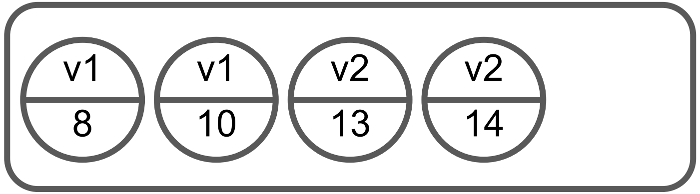
1	0	0	1	1
0	1	2	3	4

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

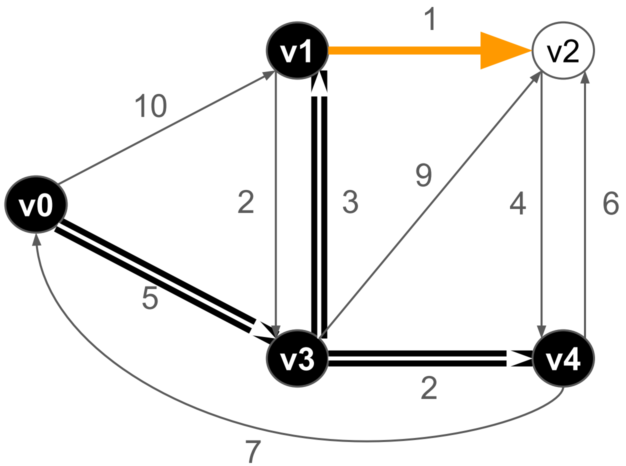
0	1	2	3	4
0	8	13	5	7

na_arvore:

0	1	2	3	4
1	0	0	1	1

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:

v2	v1	v2	v2
9	10	13	14

dp:

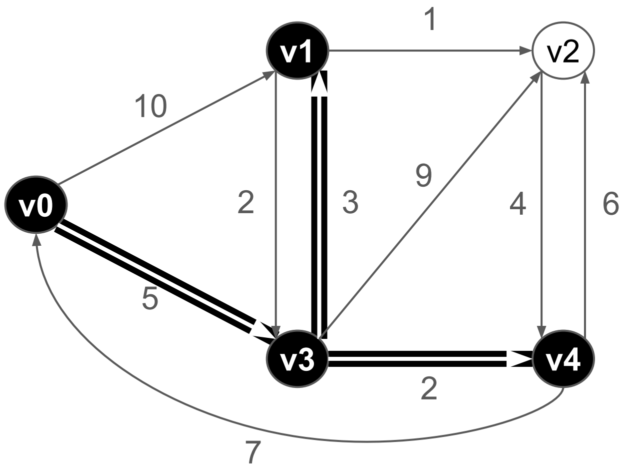
0	1	2	3	4
0	8	9	5	7

na_arvore:

0	1	2	3	4
1	1	0	1	1

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:

v2	v1	v2	v2
9	10	13	14

dp:

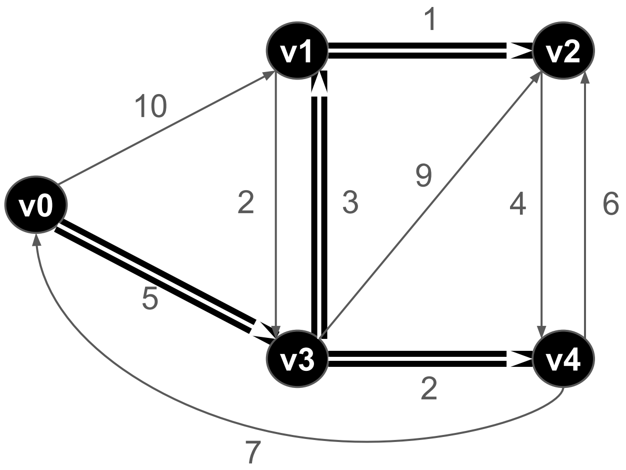
0	1	2	3	4
0	8	9	5	7

na_arvore:

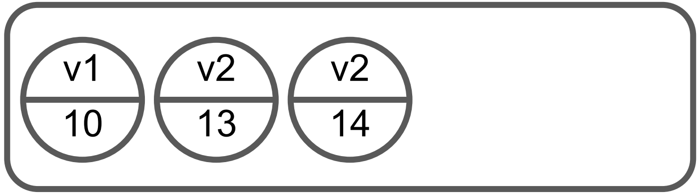
0	1	2	3	4
1	1	0	1	1

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

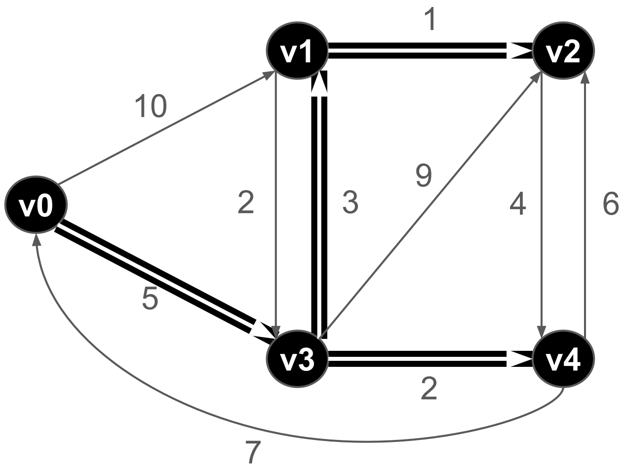
0	1	2	3	4
0	8	9	5	7

na_arvore:

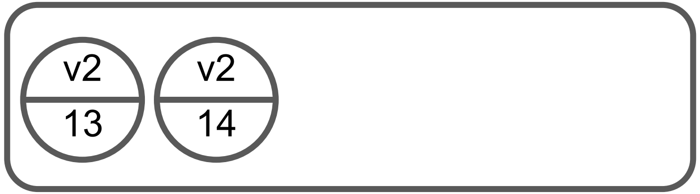
0	1	2	3	4
1	1	1	1	1

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:



dp:

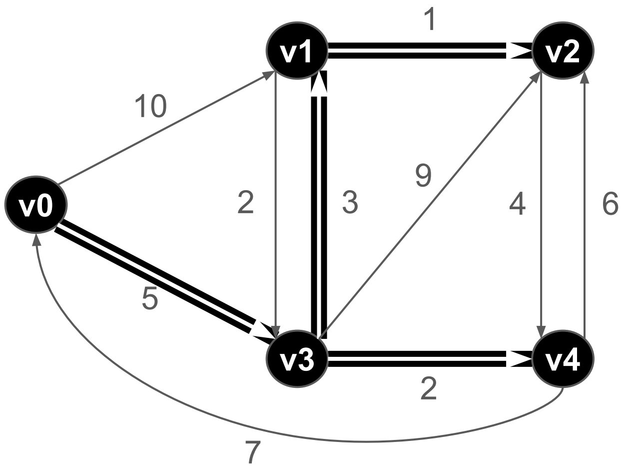
0	1	2	3	4
0	8	9	5	7

na_arvore:

0	1	2	3	4
1	1	1	1	1

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:

dp:

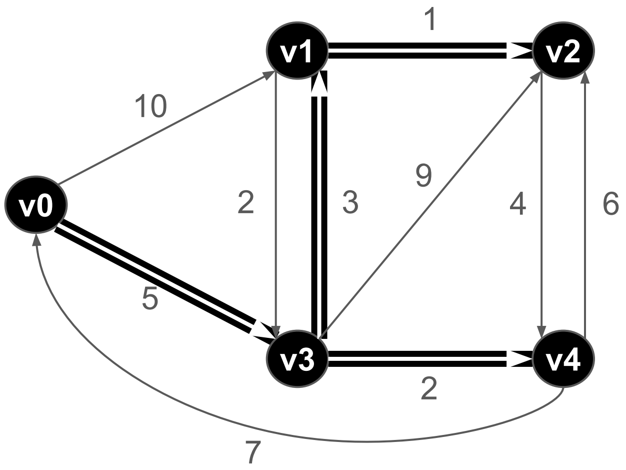
0	1	2	3	4
0	8	9	5	7

na_arvore:

0	1	2	3	4
1	1	1	1	1

Algoritmo de Dijkstra - Implementação

$s = v_0$



Fila de prioridade:

dp:

0	1	2	3	4
0	8	9	5	7

na_arvore:

0	1	2	3	4
1	1	1	1	1

Algoritmo de Dijkstra - Implementação

Dijkstra(G, s, dp)

1. Para cada vértice w de G:
2. $dp[w] = \infty$
3. $na\_arvore[w] = 0$
4. $dp[s] = 0$
5. Crie uma fila de prioridade Q
6. Adicione a Q o vértice s com prioridade $dp[s]$
7. Enquanto Q não está vazia:
8. Remova o item de menor prioridade de Q; seja u o vértice do item removido
9. Se $na\_arvore[u] == 0$:
10. $na\_arvore[u] = 1$
11. Para cada vizinho de saída v de u em G:
12. Se $dp[v] > dp[u] + p(uv)$: // $p(uv)$ é o peso da aresta uv
13. $dp[v] = dp[u] + p(uv)$
14. Adicione a Q o vértice v com prioridade igual a (o novo valor de) $dp[v]$

Relaxação da aresta *uv*

O laço dos Passos 7 a 14 pode ser encerrado após todos os vértices de G terem sido adicionados à árvore que estamos construindo

Referências

- Esta apresentação é baseada nos seguintes materiais:

1. Capítulo 4 do livro
   HALIM, S.; HALIM, F.; EFFENDY, S. Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s, book 1, chs. 1-4. Lulu, 2018.
2. Capítulo 7 do livro
   LAAKSONEN, A. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests, 2. ed. Springer, 2020.
3. Capítulo 23 do livro
   Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 3rd. ed. MIT Press, 2009.
4. Capítulo 20 do livro
   Sedgewick, R. Algorithms in C++ – Part 5. Graph Algorithms. 3rd. ed. Addison-Wesley, 2002.