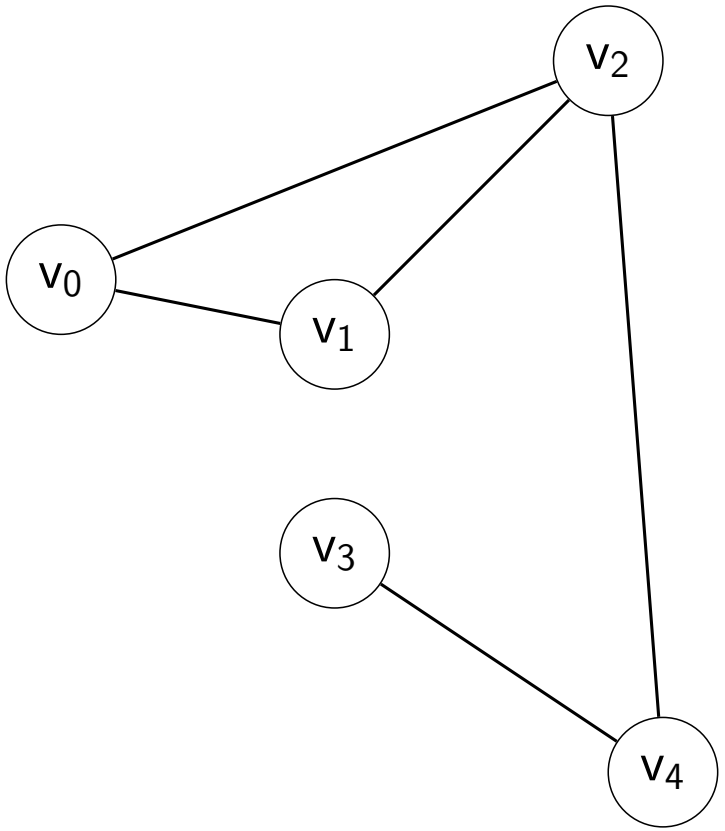
Algoritmos e Estruturas de Dados II
Aula 04 – Grafos: Matriz de Adjacência com Pesos

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

Grafos – Relembrando

- São representados como um conjunto de nós (vértices) conectados par a par por linhas (arestas)



Grafos - Relembrando

Podem ser representados utilizando:

Grafos - Relembrando

Podem ser representados utilizando:

- Matrizes de Adjacência

Grafos - Relembrando

Podem ser representados utilizando:

- Matrizes de Adjacência
- Listas de Adjacências

Grafos - Relembrando

Podem ser representados utilizando:

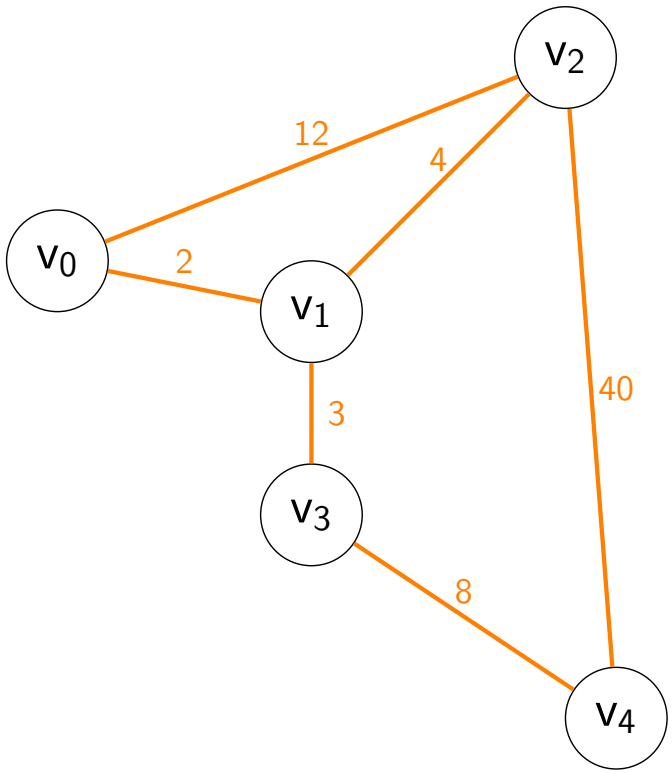
- **Matrizes de Adjacência com Peso**
- Listas de Adjacências

Grafos – Matriz de Adjacência com Pesos

- Grafos Ponderados Não Direcionados

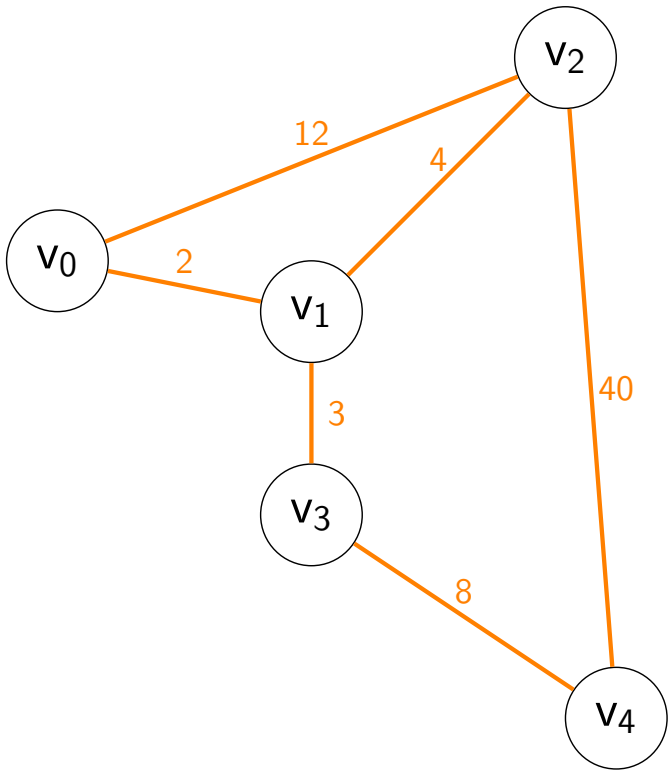
Grafos – Matriz de Adjacência com Pesos

- Grafos Ponderados Não Direcionados



Grafos – Matriz de Adjacência com Pesos

- Grafos Ponderados Não Direcionados



	v_0	v_1	v_2	v_3	v_4
v_0	0	2	12	0	0
v_1	2	0	4	3	0
v_2	12	4	0	0	40
v_3	0	3	0	0	8
v_4	0	0	40	8	0

Grafos – Matriz de Adjacência

- Grafos Não Direcionados Ponderados

- Cuidado!

	v_0	v_1	v_2	v_3	v_4
v_0	0	2	12	0	0
v_1	2	0	4	3	0
v_2	12	4	0	0	40
v_3	0	3	0	0	8
v_4	0	0	40	8	0

Grafos – Matriz de Adjacência

- Grafos Não Direcionados Ponderados

  - Cuidado!
  - Como diferenciar não aresta de um valor válido (como 0)?

	v_0	v_1	v_2	v_3	v_4
v_0	0	2	12	0	0
v_1	2	0	4	3	0
v_2	12	4	0	0	40
v_3	0	3	0	0	8
v_4	0	0	40	8	0

Grafos – Matriz de Adjacência

- Grafos Não Direcionados Ponderados

  - Cuidado!
  - Como diferenciar não aresta de um valor válido (como 0)?
  - Deve-se definir um padrão

	v_0	v_1	v_2	v_3	v_4
v_0	0	2	12	0	0
v_1	2	0	4	3	0
v_2	12	4	0	0	40
v_3	0	3	0	0	8
v_4	0	0	40	8	0

Matriz de Adjacência com Peso

Definiremos as estruturas de dados para representar um grafo, bem como algoritmos para:

Matriz de Adjacência com Peso

Definiremos as estruturas de dados para representar um grafo, bem como algoritmos para:

- Inicializar um Grafo
- Imprimir um Grafo
- Liberar a Memória de um Grafo
- Inserir uma Aresta
- Remover uma Aresta
- Verificar se uma Aresta Existe
- Retornar o Número de Vértices
- Retornar o Número de Arestas
- Verifica se um Vértice Possui Vizinhos
- Retornar o Grau de um Vértice

Matriz de Adjacência com Pesos

- Uma matriz de adjacências A de um grafo com n vértices é uma matriz $n \times n$ de **números**, em que:

Matriz de Adjacência com Pesos

- Uma matriz de adjacências A de um grafo com n vértices é uma matriz $n \times n$ de **números**, em que:
  - $A[i, j] =$ *valor* se houver uma aresta indo do vértice i para o vértice j no grafo.
  - $A[i, j] =$ *ARESTA_INVALIDA* se não houver aresta indo de i para j

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
   int numVertices;
   int numArestas;
   Peso ** matriz;
} Grafo;
```

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
   int numVertices;
   int numArestas;
   Peso ** matriz;
} Grafo;
```

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
   int numVertices;
   int numArestas;
   Peso** matriz;
} Grafo;
```

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
   int numVertices;
   int numArestas;
   Peso ** matriz;
} Grafo;
```

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
   int numVertices;
   int numArestas;
   Peso ** matriz;
} Grafo;
```

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
  int numVertices;
  int numArestas;
  Peso** matriz;
} Grafo;
```

Representação

```
#include <stdio.h>
#include <stdlib.h>

#define ARESTA_INVALIDA -1

#define true 1
#define false 0

typedef int bool;
typedef float Peso;
```

```
typedef struct {
   int numVertices;
   int numArestas;
   Peso** matriz;
} Grafo;
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){

}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
```

```
}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;




}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  int x, y;
  g->matriz = (Peso**) malloc(sizeof(Peso*)*vertices);




}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  int x, y;
  g->matriz = (Peso**) malloc(sizeof(Peso*)*vertices);
  for (x=0; x<vertices; x++){
    g->matriz[x]=(Peso*)malloc(sizeof(Peso)*vertices);



  }

}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  int x, y;
  g->matriz = (Peso**) malloc(sizeof(Peso*)*vertices);
  for (x=0; x<vertices; x++){
    g->matriz[x]=(Peso*)malloc(sizeof(Peso)*vertices);
    for (y=0; y<vertices; y++){
      g->matriz[x][y] = ARESTA_INVALIDA;
    }
  }

}
```

Inicialização

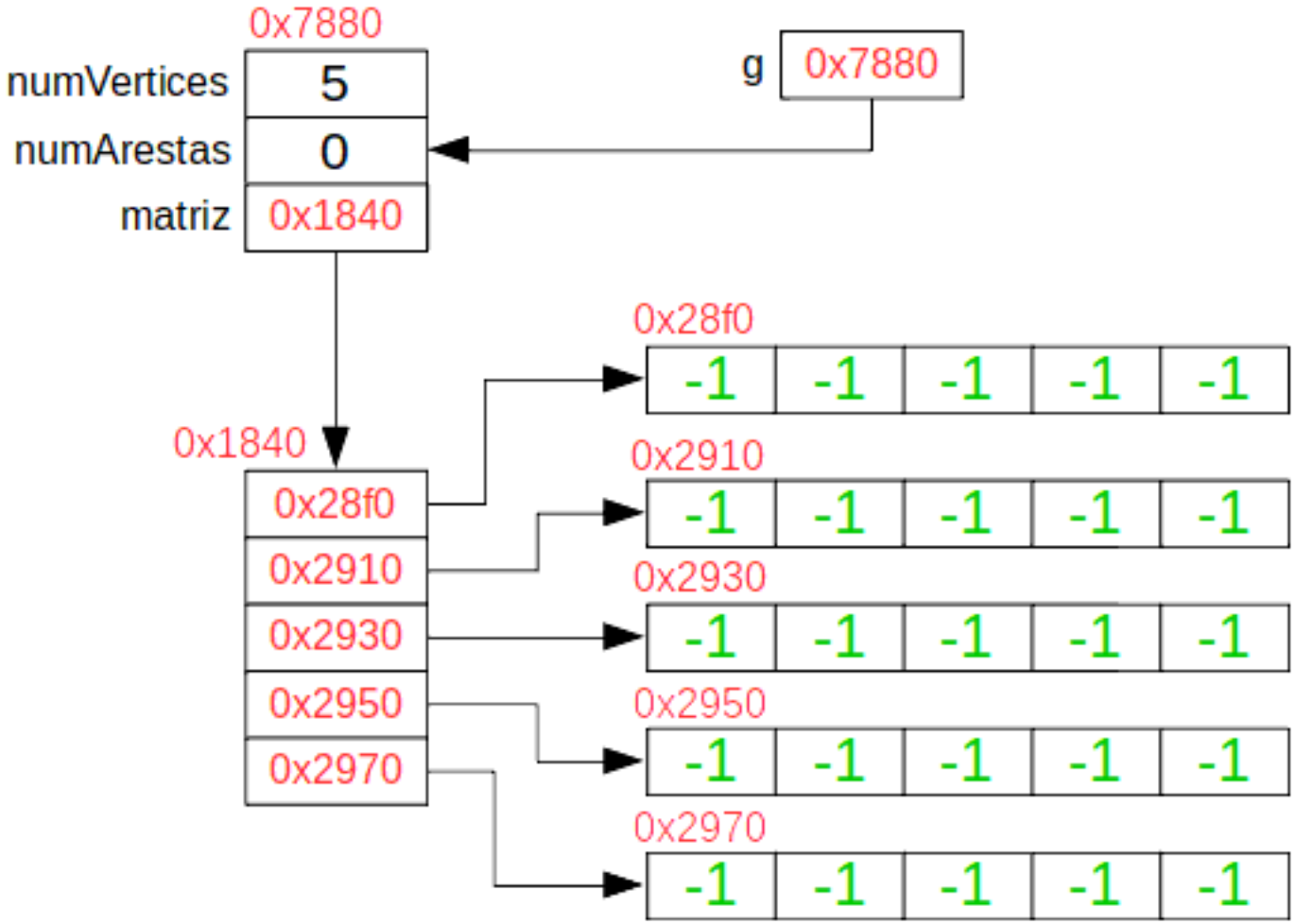
```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  int x, y;
  g->matriz = (Peso**) malloc(sizeof(Peso*)*vertices);
  for (x=0; x<vertices; x++){
    g->matriz[x]=(Peso*)malloc(sizeof(Peso)*vertices);
    for (y=0; y<vertices; y++){
      g->matriz[x][y] = ARESTA_INVALIDA;
    }
  }
  return true;
}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  int x, y;
  g->matriz = (Peso**) malloc(sizeof(Peso*)*vertices);
  for (x=0; x<vertices; x++){
    g->matriz[x]=(Peso*)malloc(sizeof(Peso)*vertices);
    for (y=0; y<vertices; y++){
      g->matriz[x][y] = ARESTA_INVALIDA;
    }
  }
  return true;
}
```

$O(V^2)$

Inicialização



Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){



}
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;




}
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  int x, y;
  printf("Imprimindo grafo
                (vertices: %i; arestas: %i).\n",
                    g->numVertices, g->numArestas);




}
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  int x, y;
  printf("Imprimindo grafo
                (vertices: %i; arestas: %i).\n",
                    g->numVertices, g->numArestas);
  for (x=0;x<g->numVertices;x++) printf("\t%5i",x);
  printf("\n");









}
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  int x, y;
  printf("Imprimindo grafo
                (vertices: %i; arestas: %i).\n",
                    g->numVertices, g->numArestas);
  for (x=0;x<g->numVertices;x++) printf("\t%5i",x);
  printf("\n");
  for (x=0;x<g->numVertices;x++){
    printf("%i",x);




  }
}
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  int x, y;
  printf("Imprimindo grafo
                (vertices: %i; arestas: %i).\n",
                    g->numVertices, g->numArestas);
  for (x=0;x<g->numVertices;x++) printf("\t%5i",x);
  printf("\n");
  for (x=0;x<g->numVertices;x++){
    printf("%i",x);
    for (y=0;y<g->numVertices;y++)
      if (g->matriz[x][y]==ARESTA_INVALIDA)
                                printf("\t     ");
      else printf("\t%5.2f",g->matriz[x][y]);
  }
}
```

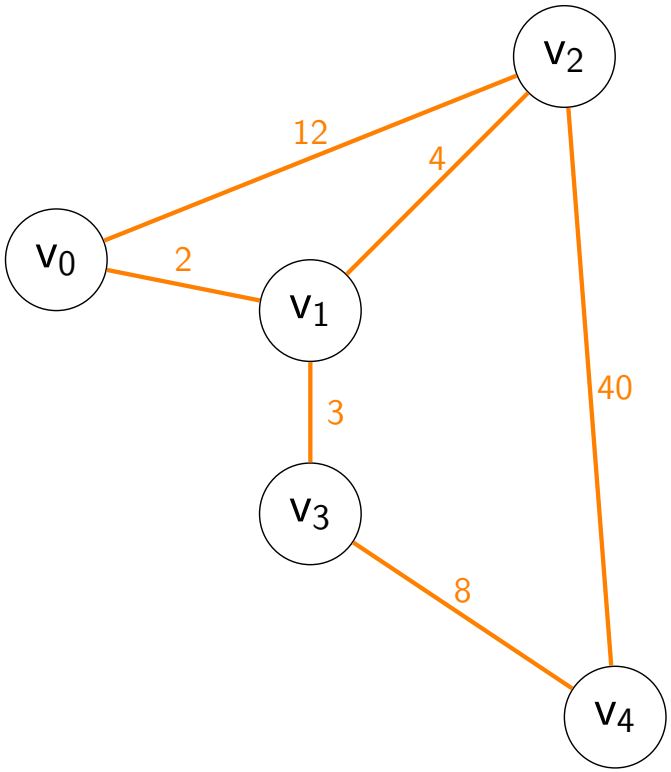
Imprimindo um Grafo

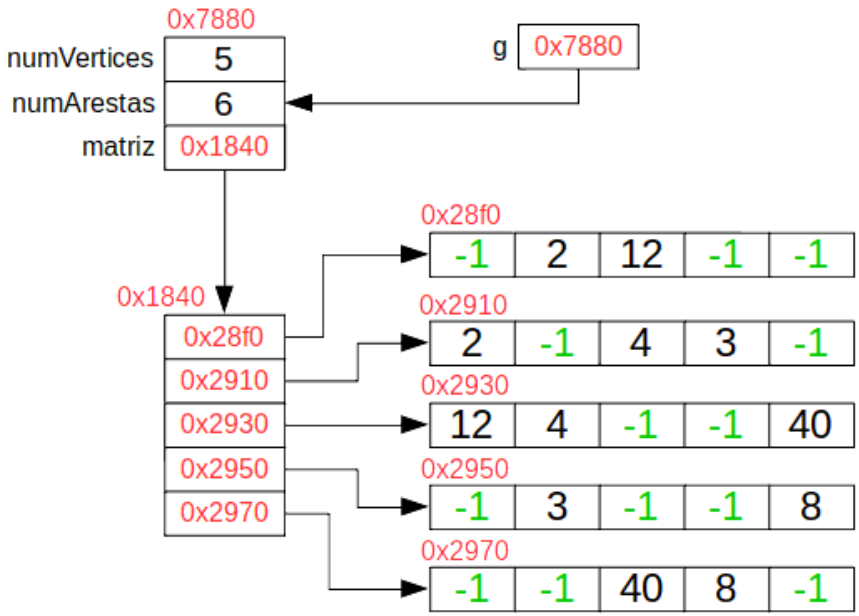
```
void exibeGrafo(Grafo* g){
  if (!g) return;
  int x, y;
  printf("Imprimindo grafo
                (vertices: %i; arestas: %i).\n",
                    g->numVertices, g->numArestas);
  for (x=0;x<g->numVertices;x++) printf("\t%5i",x);
  printf("\n");
  for (x=0;x<g->numVertices;x++){
    printf("%i",x);
    for (y=0;y<g->numVertices;y++)
      if ( g->matriz[x][y]==ARESTA_INVALIDA )
                                printf("\t     ");
      else printf(" \t%5.2f ",g->matriz[x][y]);
  }
}
```

$O(V^2)$

Imprimindo um Grafo

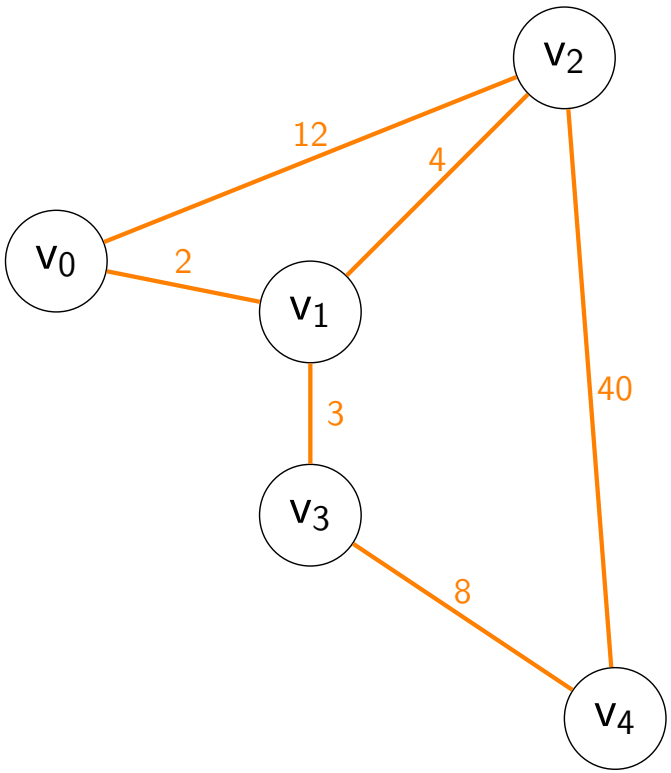
Grafo ponderado e não dirigido:





Imprimindo um Grafo

Grafo ponderado e não dirigido:




```
Imprimindo grafo (vertices: 5; arestas: 6).
    0     1     2     3     4
0         2.00 12.00
1  2.00         4.00  3.00
2 12.00  4.00               40.00
3         3.00                8.00
4               40.00  8.00
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  int x;
  for (x=0; x<g->numVertices; x++)
    free(g->matriz[x]);




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  int x;
  for (x=0; x<g->numVertices; x++)
    free(g->matriz[x]);
  free(g->matriz);




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  int x;
  for (x=0; x<g->numVertices; x++)
    free(g->matriz[x]);
  free(g->matriz);
  g->numVertices = 0;
  g->numArestas = 0;
  g->matriz = NULL;

}
```

Liberando a Memória de um Grafo

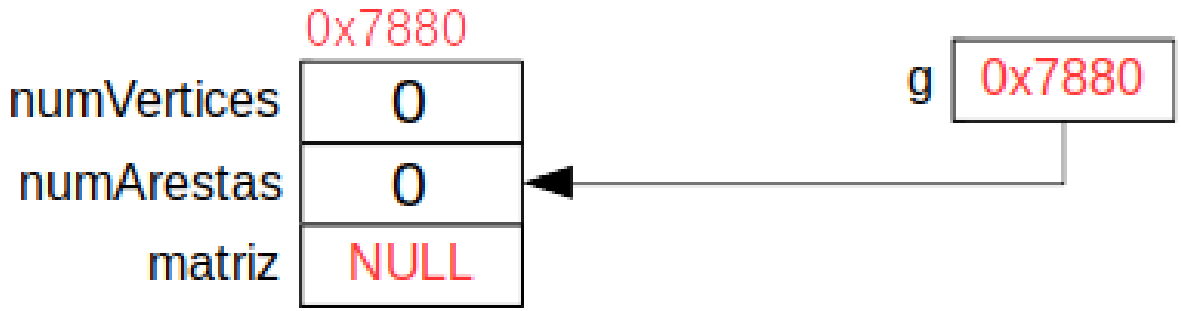
```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  int x;
  for (x=0; x<g->numVertices; x++)
    free(g->matriz[x]);
  free(g->matriz);
  g->numVertices = 0;
  g->numArestas = 0;
  g->matriz = NULL;
  return true;
}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  int x;
  for (x=0; x<g->numVertices; x++)
    free(g->matriz[x]);
  free(g->matriz);
  g->numVertices = 0;
  g->numArestas = 0;
  g->matriz = NULL;
  return true;
}
```

$O(V)$

Liberando a Memória de um Grafo



Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor){

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;









}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){




  }



}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){




    }
  }


}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }


}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }else{       // Já existia uma aresta válida



  }

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }else{       // Já existia uma aresta válida
    g->matriz[v1][v2] = valor;
    g->matriz[v2][v1] = valor;

  }

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }else{       // Já existia uma aresta válida
    g->matriz[v1][v2] = valor;
    g->matriz[v2][v1] = valor;
    if (valor == ARESTA_INVALIDA) g->numArestas--;
  }

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }else{       // Já existia uma aresta válida
    g->matriz[v1][v2] = valor;
    g->matriz[v2][v1] = valor;
    if (valor == ARESTA_INVALIDA) g->numArestas--;
  }
  return true;
}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor ){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }else{       // Já existia uma aresta válida
    g->matriz[v1][v2] = valor;
    g->matriz[v2][v1] = valor;
    if (valor == ARESTA_INVALIDA) g->numArestas--;
  }
  return true;
}
```

O(1)

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){



}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      g->matriz[v1][v2]== ARESTA_INVALIDA )return false;




}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      g->matriz[v1][v2]== ARESTA_INVALIDA )return false;
  g->matriz[v1][v2] =  ARESTA_INVALIDA ;
  g->matriz[v2][v1] =  ARESTA_INVALIDA ;


}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      g->matriz[v1][v2]== ARESTA_INVALIDA )return false;
  g->matriz[v1][v2] =  ARESTA_INVALIDA ;
  g->matriz[v2][v1] =  ARESTA_INVALIDA ;
  g->numArestas--;

}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      g->matriz[v1][v2]== ARESTA_INVALIDA )return false;
  g->matriz[v1][v2] =  ARESTA_INVALIDA ;
  g->matriz[v2][v1] =  ARESTA_INVALIDA ;
  g->numArestas--;
  return true;
}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      g->matriz[v1][v2]== ARESTA_INVALIDA )return false;
  g->matriz[v1][v2] =  ARESTA_INVALIDA ;
  g->matriz[v2][v1] =  ARESTA_INVALIDA ;
  g->numArestas--;
  return true;
}
```

O(1)

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){



}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices ||
      g->matriz[v1][v2]==ARESTA_INVALIDA )return false;

}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices ||
      g->matriz[v1][v2]==ARESTA_INVALIDA )return false;
  return true;
}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices ||
      g->matriz[v1][v2]==ARESTA_INVALIDA )return false;
  return true;
}
```

O(1)

Número de Vértices

```
int numeroDeVertices(Grafo* g){


}
```

Número de Vértices

```
int numeroDeVertices(Grafo* g){
  if (g!=NULL) return g->numVertices;

}
```

Número de Vértices

```
int numeroDeVertices(Grafo* g){
  if (g!=NULL) return g->numVertices;
  else return -1;
}
```

Número de Vértices

```
int numeroDeVertices(Grafo* g){
  if (g!=NULL) return g->numVertices;
  else return -1;
}
```

O(1)

Número de Aresta

```
int numeroDeArestas(Grafo* g){


}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}


int numeroDeArestas2(Grafo* g){




}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}


int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;









}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}


int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)



}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}


int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)
    for (y=x+1; y<g->numVertices; y++)


}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}


int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)
    for (y=x+1; y<g->numVertices; y++)
      if (g->matriz[x][y]!= ARESTA_INVALIDA ) arestas++;

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

```
int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)
    for (y=x+1; y<g->numVertices; y++)
      if (g->matriz[x][y]!= ARESTA_INVALIDA ) arestas++;
  return arestas;
}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

O(1)

```
int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)
    for (y=x+1; y<g->numVertices; y++)
      if (g->matriz[x][y]!= ARESTA_INVALIDA ) arestas++;
  return arestas;
}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

$O(1)$

```
int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)
    for (y=x+1; y<g->numVertices; y++)
      if (g->matriz[x][y]!= ARESTA_INVALIDA ) arestas++;
  return arestas;
}
```

$O(V^2)$

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){



}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return false;




}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return false;
  int x;
  for (x=0;x<g->numVertices;x++)
    if (g->matriz[v][x]!=ARESTA_INVALIDA) return true;

}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return false;
  int x;
  for (x=0;x<g->numVertices;x++)
    if (g->matriz[v][x]!= ARESTA_INVALIDA ) return true;
  return false;
}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return false;
  int x;
  for (x=0;x<g->numVertices;x++)
    if (g->matriz[v][x]!=ARESTA_INVALIDA) return true;
  return false;
}
```

$O(V)$

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){



}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;




}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int x, grau = 0;
  for (x=0;x<g->numVertices;x++)
    if (g->matriz[v][x]!= ARESTA_INVALIDA ) grau++;

}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int x, grau = 0;
  for (x=0;x<g->numVertices;x++)
    if (g->matriz[v][x]!= ARESTA_INVALIDA ) grau++;
  return grau;
}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int x, grau = 0;
  for (x=0;x<g->numVertices;x++)
    if (g->matriz[v][x]!= ARESTA_INVALIDA ) grau++;
  return grau;
}
```

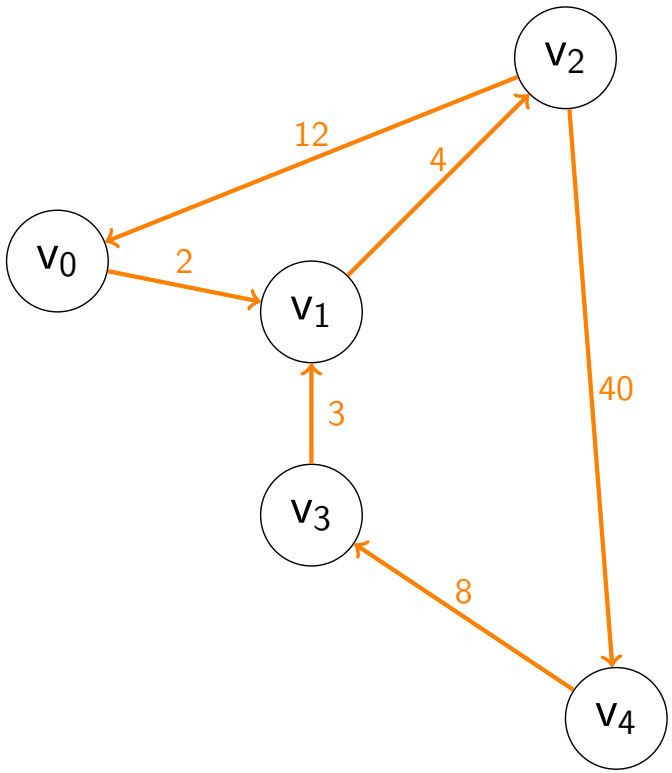
$O(V)$

Digrafos – Matriz de Adjacência com Peso

- E se o grafo for dirigido?

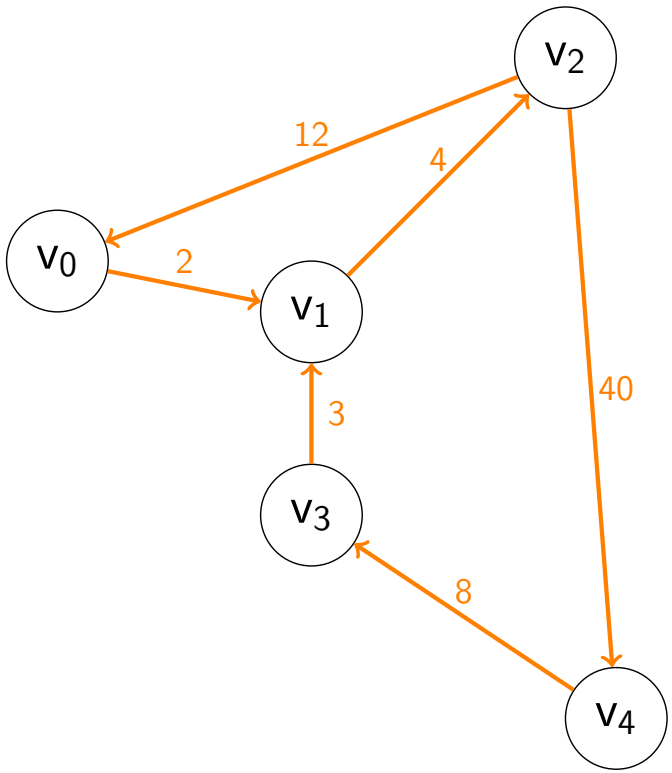
Digrafos – Matriz de Adjacência com Peso

- E se o grafo for dirigido?



Digrafos – Matriz de Adjacência com Peso

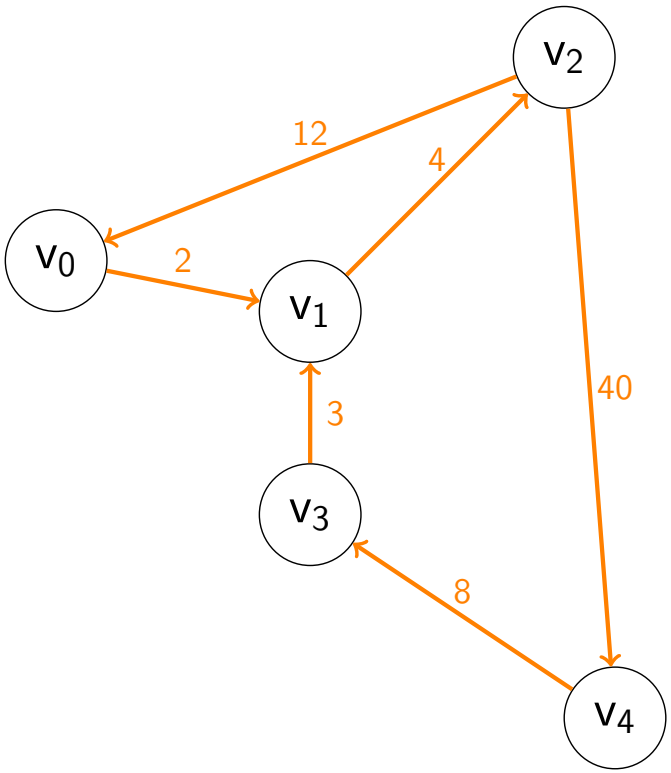
- E se o grafo for dirigido?

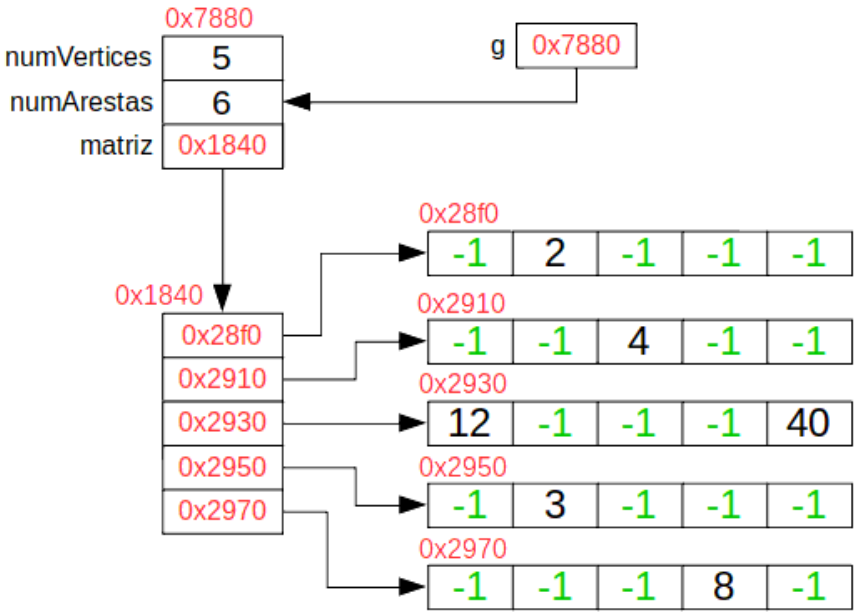


	v_0	v_1	v_2	v_3	v_4
v_0	0	2	0	0	0
v_1	0	0	4	0	0
v_2	12	0	0	0	40
v_3	0	3	0	0	0
v_4	0	0	0	8	0

Digrafos – Matriz de Adjacência com Pesos

- E se o grafo for dirigido?





Matriz de Adjacência - Grafo Dirigido

A estrutura de dados será a mesma, haverá pequenas mudanças em algumas funções:

Matriz de Adjacência - Grafo Dirigido

A estrutura de dados será a mesma, haverá pequenas mudanças em algumas funções:

- Inicializar um Grafo
- Imprimir um Grafo
- Liberar a Memória de um Grafo
- Inserir uma Aresta
- Remover uma Aresta
- Verificar se uma Aresta Existe
- Retornar o Número de Vértices
- Retornar o Número de Arestas*
- Verificar se um Vértice Possui Vizinhos
- Retornar o Grau de um Vértice

Inserindo uma Aresta - Digrafo

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor){

}
```

Inserindo uma Aresta - Digrafo

```
bool insereAresta(Grafo* g, int v1, int v2, Peso valor){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices
      || v1 == v2) return false;
  if (g->matriz[v1][v2] == ARESTA_INVALIDA){
    if (valor != ARESTA_INVALIDA){
      g->matriz[v1][v2] = valor;
      g->matriz[v2][v1] = valor;
      g->numArestas++;
    }
  }else{       // Já existia uma aresta válida
    g->matriz[v1][v2] = valor;
    g->matriz[v2][v1] = valor;
    if (valor == ARESTA_INVALIDA) g->numArestas--;
  }
  return true;
}
```

O(1)

Removendo uma Aresta - Digrafo

```
bool removeAresta(Grafo* g, int v1, int v2){




}
```

Removendo uma Aresta - Digrafo

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 || v1 >= g->numVertices ||
      v2 >= g->numVertices ||
      g->matriz[v1][v2]==ARESTA_INVALIDA) return false;
  g->matriz[v1][v2] = ARESTA_INVALIDA;
  g->matriz[v2][v1] = ARESTA_INVALIDA;
  g->numArestas--;
  return true;
}
```

O(1)

Número de Aresta - Digrafo

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

O(1)

```
int numeroDeArestas2(Grafo* g){




}
```

Número de Aresta - Digrafo

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

$O(1)$

```
int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  int x, y, arestas = 0;
  for (x=0; x<g->numVertices; x++)
    for (y= 0 ; y<g->numVertices; y++)
      if (g->matriz[x][y]!=ARESTA_INVALIDA) arestas++;
  return arestas;
}
```

$O(V^2)$

Grau de um Vértice - Digrafo

```
int retornaGrauDoVertice(Grafo* g, int v){




}
```

Grau de um Vértice - Digrafo

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int x, grau = 0;
  for (x=0;x<g->numVertices;x++){
     if (g->matriz[v][x]!=ARESTA_INVALIDA) grau++;
     if (g->matriz[x][v]!=ARESTA_INVALIDA) grau++;
  }
  return grau;
}
```

$O(V)$

Matrizes de Adjacências - Complexidade

Função	**Complexidade**
Inicializar um Grafo	$O(V^2)$
Imprimir um Grafo	$O(V^2)$
Liberar a Memória de um Grafo	$O(V)$
Inserir uma Aresta	$O(1)$
Remover uma Aresta	$O(1)$
Verificar se uma Aresta Existe	$O(1)$
Retornar o Número de Vértices	$O(1)$
Retornar o Número de Arestas	$O(1)$ ou $O(V^2)$
Verifica se um Vértice Possui Vizinhos	$O(V)$
Retornar o Grau de um Vértice	$O(V)$

Algoritmos e Estruturas de Dados II
Aula 04 – Grafos: Matriz de Adjacência com Pesos

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri