ACH2024

Aula 6

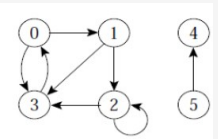
Implementação de Grafos por Lista de Adjacências (parte 2)

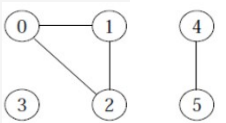
Profa. Ariane Machado Lima

Aulas passadas...

grafo_matrizadj.h

```c
#include <stdbool.h>

#define MAXNUMVERTICES 100
#define AN -1                  /* aresta nula (representa ausencia de aresta) */
#define VERTICE_INVALIDO -1  /* vertice inexistente */

typedef int Peso;
typedef struct {
  Peso mat[MAXNUMVERTICES][MAXNUMVERTICES];
  int numVertices;
  int numArestas;
} Grafo;

void inicializaGrafo(Grafo* grafo, int nv);
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo);
bool existeAresta(int v1, int v2, Grafo *grafo);
void removeAresta(int v1, int v2, Peso* peso, Grafo *grafo);
bool listaAdjVazia(int v, Grafo* grafo);
int primeiroListaAdj(int v, Grafo* grafo);
int proxListaAdj(int v, Grafo* grafo, int prox);
void imprimeGrafo(Grafo* grafo);
void liberaGrafo(Grafo* grafo);
```





	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

Matriz de adjacência - Reflexões

Essa representação por matriz adjacência é sempre eficiente?

	Matriz de adj.
inicializaGrafo	$O(v^2)$
imprimeGrafo	$O(v^2)$
insereAresta	O(1)
existeAresta	O(1)
removeAresta	O(1)
listaAdjVazia	O(v)
proxListaAdj	O(v)
liberaGrafo	O(1)

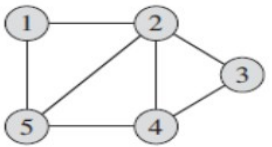
- ✓ Acesso instantâneo a uma aresta (tempo constante) – consulta, inserção e remoção
- ✓ **Para grafos densos OK !!!**

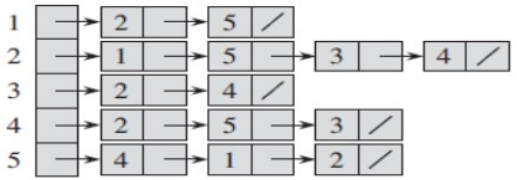
Mesmo que o grafo tenha poucas arestas (**esparso**):

- ✗ Utilização da lista de adjacência em O(v)
- ✗ Espaço Ω (v^2)

Lista de adjacência (grafo_listaadj.h)

```c
#include <stdbool.h>   /* variaveis bool assumem valores "true" ou "false" */

#define VERTICE_INVALIDO NULL  /* numero de vertice invalido ou ausente */
#define AN -1                  /* aresta nula */

typedef int Peso;

/*
  tipo estruturado taresta:
       vertice destino, peso, ponteiro p/ prox. aresta
*/
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;

typedef Aresta* Apontador;

/*
  tipo estruturado grafo:
       vetor de listas de adjacencia (cada posicao contem o ponteiro
                    para o inicio da lista de adjacencia do vertice)
       numero de vertices
*/
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```





Por conta do `proxListaAdj`

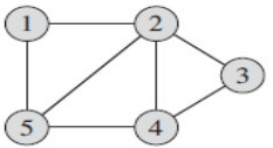
Novo grafo_matrizadj.h

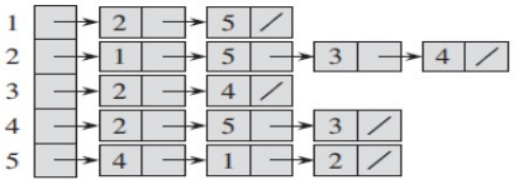
```c
#define MAXNUMVERTICES 100
#define AN -1                 /* aresta nula, ou seja, valor que representa ausencia de aresta */
#define VERTICE_INVALIDO -1   /* numero de vertice invalido ou ausente */

#include <stdbool.h>   /* variaveis bool assumem valores "true" ou "false" */

typedef int Peso;
typedef struct  {
  Peso mat[MAXNUMVERTICES][MAXNUMVERTICES];
  int numVertices;
  int numArestas;
} Grafo;
typedef int Apontador;

bool inicializaGrafo(Grafo* grafo, int nv);
int obtemNrVertices(Grafo* grafo);
int obtemNrArestas(Grafo* grafo);
bool verificaValidadeVertice(int v, Grafo *grafo);
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo);
bool existeAresta(int v1, int v2, Grafo *grafo);
Peso obtemPesoAresta(int v1, int v2, Grafo *grafo);
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
bool removeAresta(int v1, int v2, Grafo *grafo);
bool listaAdjVazia(int v, Grafo* grafo);
Apontador primeiroListaAdj(int v, Grafo* grafo);
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual);
void imprimeGrafo(Grafo* grafo);
void liberaGrafo(Grafo* grafo);
int verticeDestino(Apontador p, Grafo* grafo);
```

Essa parte (ou seja, os protótipos) devem ser idênticos em grafo_matrizadj.h e grafo_listaadj.h

Lista de adjacência (grafo_listaadj.h)

```c
#include <stdbool.h>   /* variaveis bool assumem valores "true" ou "false" */

#define VERTICE_INVALIDO NULL  /* numero de vertice invalido ou ausente */
#define AN -1                  /* aresta nula */

typedef int Peso;

/*
  tipo estruturado taresta:
      vertice destino, peso, ponteiro p/ prox. aresta
*/
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;

typedef Aresta* Apontador;

/*
  tipo estruturado grafo:
      vetor de listas de adjacencia (cada posicao contem o ponteiro
                    para o inicio da lista de adjacencia do vertice)
      numero de vertices
*/
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```






```c
bool inicializaGrafo(Grafo* grafo, int nv);
int obtemNrVertices(Grafo* grafo);
int obtemNrArestas(Grafo* grafo);
bool verificaValidadeVertice(int v, Grafo *grafo);
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo);
bool existeAresta(int v1, int v2, Grafo *grafo);
Peso obtemPesoAresta(int v1, int v2, Grafo *grafo);
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
bool removeAresta(int v1, int v2, Grafo *grafo);
bool listaAdjVazia(int v, Grafo* grafo);
Apontador primeiroListaAdj(int v, Grafo* grafo);
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual);
void imprimeGrafo(Grafo* grafo);
void liberaGrafo(Grafo* grafo);
int verticeDestino(Apontador p, Grafo* grafo);
```

Complexidades
(considerando um grafo de v vértices e a arestas)

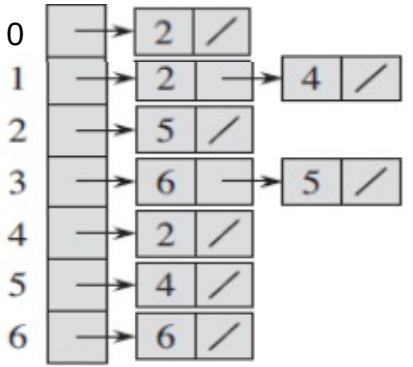
	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

Verificação se a lista de adjacência de um vértice é vazia

```
/*
   bool listaAdjVazia(int v, Grafo* grafo):
   Retorna true se a lista de adjacencia (de vertices adjacentes)
   do vertice v é vazia, e false caso contrário.
*/
bool listaAdjVazia(int v, Grafo* grafo){
  if (! verificaValidadeVertice(v, grafo))
    return false;
  return (grafo->listaAdj[v]==NULL);
}
```




```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

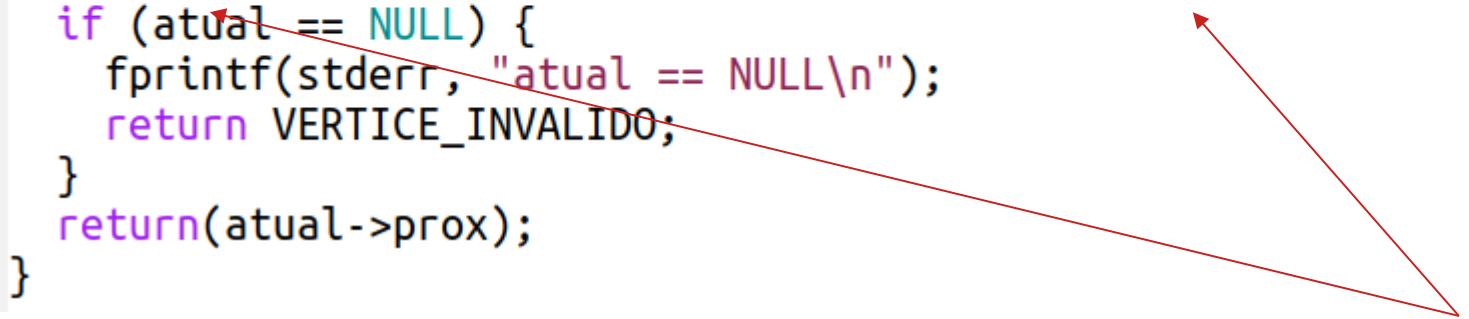
```
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	
existeAresta	O(1)	
removeAresta	O(1)	
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	
liberaGrafo	O(1)	

Próximo da lista de adjacência

```
/*
   Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual):
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou NULL se a lista de adjacencia tiver terminado sem um novo proximo.
*/
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual){
  if (atual == NULL) {
    fprintf(stderr, "atual == NULL\n");
    return VERTICE_INVALIDO;
  }
  return(atual->prox);
}
```

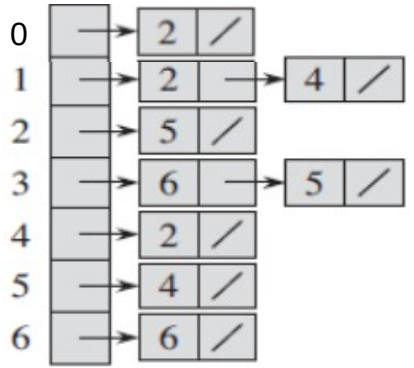


Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	$O(1)$
liberaGrafo	$O(1)$	

Primeiro da lista de adjacência

```c
/*
  primeiroListaAdj(v, Grafo): retorna o endereco do primeiro vertice
  adjacente a v.
*/
Apontador primeiroListaAdj(int v, Grafo *grafo) {
  return(grafo->listaAdj[v]);
}
```




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

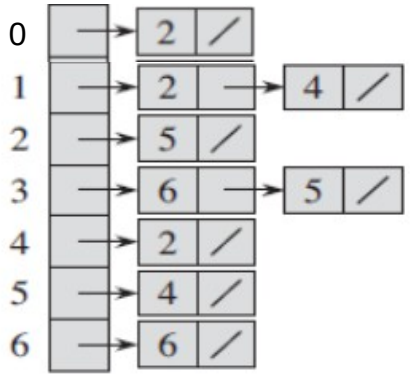
```c
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Existência de aresta

```
/*
  bool existeAresta(int v1, int v2, Grafo *grafo):
  Retorna true se existe a aresta (v1, v2) no grafo e false caso contrário
*/
bool existeAresta(int v1, int v2, Grafo *grafo)
{
  Apontador q;

  if (! (verificaValidadeVertice(v1, grafo) && verificaValidadeVertice(v2, grafo)))
      return false;

  q = grafo->listaAdj[v1];
  while ((q != NULL) && (q->vdest != v2))
    q = q->prox;
  if (q != NULL) return true;
  return false;
}
```

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```



Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	
existeAresta	O(1)	O(v)
removeAresta	O(1)	
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	
liberaGrafo	O(1)	

Obtenção do peso da aresta

```
/*
  Peso obtemPesoAresta(int v1, int v2, Grafo *grafo):
  Retorna o peso da aresta (v1, v2) no grafo se ela existir e AN caso contrário
*/
Peso obtemPesoAresta(int v1, int v2, Grafo *grafo)
{
```

O(v)

Aula de hoje

Demais operações

Juntando tudo

Buscas em grafos

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	$O(v)$
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	$O(1)$
liberaGrafo	$O(1)$	

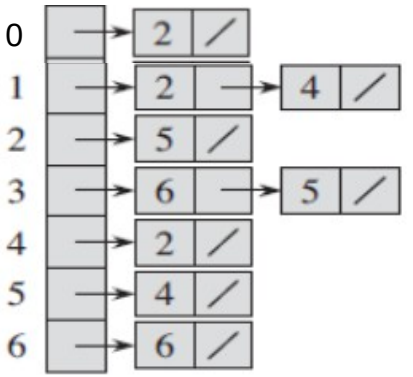
Inserção de aresta

```
/*
  void insereAresta(int v1, int v2, Peso peso, Grafo *grafo):
  Insere a aresta (v1, v2) com peso "peso" no grafo.
  Nao verifica se a aresta ja existia (isso deve ser feito pelo usuario antes, se necessario).
*/
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo){
```

(**não** verifica se ela já existe; sem ordenação dos adjacentes)

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

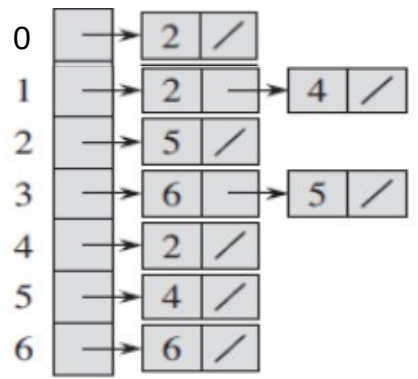


Inserção de aresta

```c
/*
  void insereAresta(int v1, int v2, Peso peso, Grafo *grafo):
  Insere a aresta (v1, v2) com peso "peso" no grafo.
  Nao verifica se a aresta ja existia.
*/
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo) {
  Apontador p;

  if (! (verificaValidadeVertice(v1, grafo) && verificaValidadeVertice(v2, grafo)))
      return;

  if(!(p = (Apontador) calloc(1, sizeof(Aresta)) )){
    fprintf(stderr, "ERRO: Falha na alocacao de memoria na funcao insereAresta\n");
    return;
  }
  p->vdest = v2;
  p->peso = peso;
  p->prox = grafo->listaAdj[v1]; /* insere no início! */
  grafo->listaAdj[v1] = p;
  grafo->numArestas++;
}
```

```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```



Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	
existeAresta	O(1)	O(v)
removeAresta	O(1)	
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	

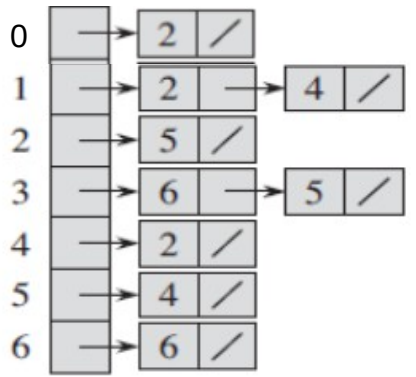
Sem verificar existência!!!
E sem ordenar!!!

Remoção de aresta

```
/*
  bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
  Remove a aresta (v1, v2) do grafo.
  Se a aresta existia, coloca o peso dessa aresta em "peso" e retorna true,
  caso contrario retorna false (e "peso" é inalterado).
*/
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo)
{
```

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

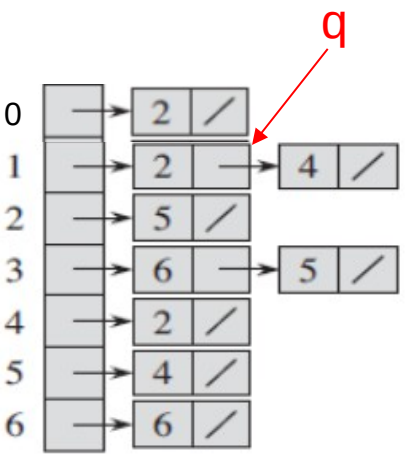


Remoção de aresta

```
/*
  bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
  Remove a aresta (v1, v2) do grafo.
  Se a aresta existia, coloca o peso dessa aresta em "peso" e retorna true,
  caso contrario retorna false (e "peso" é inalterado).
*/
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo)
{
  Apontador q, ant;

  if (! (verificaValidadeVertice(v1, grafo) && verificaValidadeVertice(v2, grafo)))
      return false;

  q = grafo->listaAdj[v1];
  while ((q != NULL) && (q->vdest != v2)){
    ant = q;
    q = q->prox;
  }
  //aresta existe
  if (q != NULL){
    if (grafo->listaAdj[v1] == q)
      grafo->listaAdj[v1] = q->prox;
    else ant->prox = q->prox;
    *peso = q->peso;
    q->prox = NULL;
    free(q);
    q = NULL;
    return true;
  }
  //aresta nao existe
  return false;
}
```

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

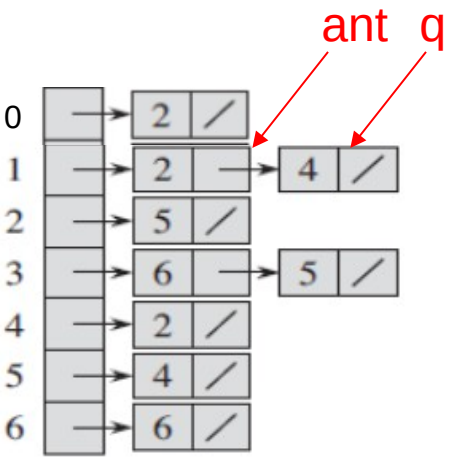


Remoção de aresta

```
/*
  bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
  Remove a aresta (v1, v2) do grafo.
  Se a aresta existia, coloca o peso dessa aresta em "peso" e retorna true,
  caso contrario retorna false (e "peso" é inalterado).
*/
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo)
{
  Apontador q, ant;

  if (! (verificaValidadeVertice(v1, grafo) && verificaValidadeVertice(v2, grafo)))
      return false;

  q = grafo->listaAdj[v1];
  while ((q != NULL) && (q->vdest != v2)){
    ant = q;
    q = q->prox;
  }
  //aresta existe
  if (q != NULL){
    if (grafo->listaAdj[v1] == q)
      grafo->listaAdj[v1] = q->prox;
    else ant->prox = q->prox;
    *peso = q->peso;
    q->prox = NULL;
    free(q);
    q = NULL;
    return true;
  }
  //aresta nao existe
  return false;
}
```

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```



Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	

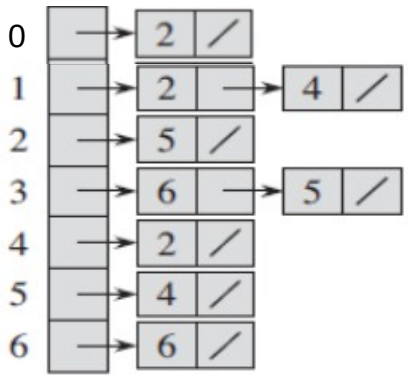
Sem verificar existência!!!
E sem ordenar!!!

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	O(v)
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	

Sem verificar existência!!!
E sem ordenar!!!

Liberação do espaço em memória

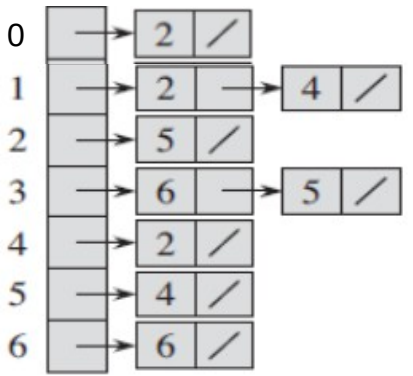



```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
   Apontador* listaAdj;
   int numVertices;
   int numArestas;
} Grafo;
```

Liberação do espaço em memória

```c
/*
  void liberaGrafo (Grafo *grafo): Libera o espaco ocupado por um grafo.
*/
void liberaGrafo (Grafo *grafo) {
  int v;
  Apontador p;

  // libera a lista de adjacencia de cada vertice
  for (v = 0; v <  grafo->numVertices; v++) {
    while ((p = grafo->listaAdj[v]) != NULL)  {
      grafo->listaAdj[v] = p->prox;
      p->prox=NULL;
      free(p);
    }
  }
  grafo->numVertices=0;
  // Libera o vetor de ponteiros para as listas de adjacencia
  free(grafo->listaAdj);
  grafo->listaAdj = NULL;
}
```




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	O(v)
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	

Sem verificar existência!!!
E sem ordenar!!!

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	O(v)
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	O(v+a)

Sem verificar existência!!!
E sem ordenar!!!

Impressão do grafo

```
/*
    void imprimeGrafo(Grafo* grafo):
    Imprime os vertices e arestas do grafo no seguinte formato:
    v1: (adj11, peso11); (adj12, peso12); ...
    v2: (adj21, peso21); (adj22, peso22); ...
    Assuma que cada vértice é um inteiro de até 2 dígitos.
*/
void imprimeGrafo(Grafo *grafo) {
```

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	O(v)
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	O(v+a)

Sem verificar existência!!!
E sem ordenar!!!

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	O(v)
imprimeGrafo	$O(v^2)$	O(v+a)
insereAresta	O(1)	O(1)
existeAresta	O(1)	O(v)
removeAresta	O(1)	O(v)
listaAdjVazia	O(v)	O(1)
proxListaAdj	O(v)	O(1)
liberaGrafo	O(1)	O(v+a)

Sem verificar existência!!!
E sem ordenar!!!

Matriz e listas de adjacência - escolhas

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	$O(v+a)$
insereAresta	$O(1)$	$O(1)$
existeAresta	$O(1)$	$O(v)$
removeAresta	$O(1)$	$O(v)$
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	$O(1)$
liberaGrafo	$O(1)$	$O(v+a)$

Matriz e listas de adjacência - escolhas

Para escolher entre uma representação e outra, devem ser considerados:

- O grafo é esparso? ($|a| << |V^2|$) (a: arestas)
- Economia de espaço é fundamental?
  - Cuidado: ponteiros também ocupam espaço...
- Prioridade para economia de tempo em algumas dessas operações:
  - Acesso a arestas específicas
  - Iterar sobre os adjacentes de um vértice

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	$O(v+a)$
insereAresta	$O(1)$	$O(1)$
existeAresta	$O(1)$	$O(v)$
removeAresta	$O(1)$	$O(v)$
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	$O(1)$
liberaGrafo	$O(1)$	$O(v+a)$

Juntando tudo

Juntando tudo

Temos então 6 arquivos:

- **grafo_matrizadj.h**: tipos e protótipos para matriz
- **grafo_matrizadj.c**: implementações das funções prototipadas em grafo_matrizadj.h, de acordo com matriz
- **grafo_listaadj.h**: tipos e protótipos para lista
- **grafo_listaadj.c**: implementações das funções prototipadas em grafo_listaadj.h, de acordo com lista
- **testa_grafo.c**: onde tem o main que chama as funções prototipadas nos .h (são idênticas nos dois .h)
- **Makefile**: rege como eles serão compilados e ligados, gerando um executável usando matriz ou lista

grafo_matrizadj.h

```c
#define MAXNUMVERTICES 100
#define AN -1                 /* aresta nula, ou seja, valor que representa ausencia de aresta */
#define VERTICE_INVALIDO -1   /* numero de vertice invalido ou ausente */

#include <stdbool.h>   /* variaveis bool assumem valores "true" ou "false" */

typedef int Peso;
typedef struct  {
  Peso mat[MAXNUMVERTICES][MAXNUMVERTICES];
  int numVertices;
  int numArestas;
} Grafo;
typedef int Apontador;

bool inicializaGrafo(Grafo* grafo, int nv);
int obtemNrVertices(Grafo* grafo);
int obtemNrArestas(Grafo* grafo);
bool verificaValidadeVertice(int v, Grafo *grafo);
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo);
bool existeAresta(int v1, int v2, Grafo *grafo);
Peso obtemPesoAresta(int v1, int v2, Grafo *grafo);
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
bool removeAresta(int v1, int v2, Grafo *grafo);
bool listaAdjVazia(int v, Grafo* grafo);
Apontador primeiroListaAdj(int v, Grafo* grafo);
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual);
void imprimeGrafo(Grafo* grafo);
void liberaGrafo(Grafo* grafo);
int verticeDestino(Apontador p, Grafo* grafo);
```

Essa parte (ou seja, os protótipos) devem ser idênticos em grafo_matrizadj.h e grafo_listaadj.h

Arquivo testa_grafo.c

```c
//#include "grafo_matrizadj.h"
#include "grafo_listaadj.h"
#include <stdio.h>

int main()
{
  Grafo g1;
  int numVertices;

  //inicializaGrafo(&g1, 10);

  do {
   printf("Digite o número de vértices do grafo\n");
   scanf("%d", &numVertices);
  } while (!inicializaGrafo(&g1, numVertices));


  //imprimeGrafo(&g1);

  return 0;
}
```

Mudança apenas nessas duas linhas !!!!

Makefile

```
testa_matriz: grafo_matrizadj.o testa_grafo.o
	gcc -o testa_grafo_matriz.exe grafo_matrizadj.o testa_grafo.o

grafo_matrizadj.o: grafo_matrizadj.c grafo_matrizadj.h
	gcc -c grafo_matrizadj.c

clean:
	rm -f *.o *.exe



testa_lista: grafo_listaadj.o testa_grafo.o
	gcc -o testa_grafo_lista.exe grafo_listaadj.o testa_grafo.o


grafo_listaadj.o: grafo_listaadj.c grafo_listaadj.h
	gcc -c grafo_listaadj.c

testa_grafo.o: testa_grafo.c grafo_matrizadj.h  grafo_listaadj.h
	gcc -c testa_grafo.c
```

Exercícios

1) Implementar a estrutura e operações de grafos utilizando lista de adjacências para:

   - grafos direcionados

   - grafos não direcionados

2) Você pode implementar outras operações que julgar relevantes também, como por exemplo obtemPeso

3) Capriche no testa_grafo.c! Inclua a leitura de um grafo (prox slide)

```
/*
  LeGrafo(nomearq, Grafo)
  Le o arquivo nomearq e armazena na estrutura Grafo
  Layout do arquivo:
    A 1a linha deve conter o número de vertices e o numero de arestas do grafo,
    separados por espaço.
    A 2a linha em diante deve conter a informacao de cada aresta, que consiste
    no indice do vertice de origem, indice do vertice de destino e o peso da
    aresta, tambem separados por espacos.
    Observações:
      Os vertices devem ser indexados de 0 a |V|-1
      Os pesos das arestas sao numeros racionais nao negativos.

  Exemplo: O arquivo abaixo contem um grafo com 4 vertices (0,1,2,3) e
  7 arestas.

  4 7
  0 3 6.3
  2 1 5.0
  2 0 9
  1 3 1.7
  0 1 9
  3 1 5.6
  0 2 7.2

  Codigo de saida:
    1: leitura bem sucedida
    0: erro na leitura do arquivo
*/
int leGrafo(char* nomearq, Grafo *grafo) {
```

```c
int leGrafo(char* nomearq, Grafo *grafo) {
  FILE *fp;
  int nvertices, narestas;
  int v1, v2;
  Peso peso;
  int idar;

  fp = fopen(nomearq, "r");
  if (fp==NULL)
     return(0);

  if (fscanf(fp, "%d %d", &nvertices, &narestas)!=2)
    return(0);

  inicializaGrafo(grafo, nvertices);
```

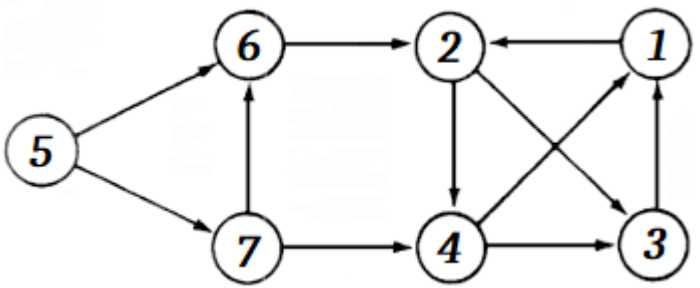
Coloque aqui antes uma msg de erro

Coloque aqui antes uma msg de erro

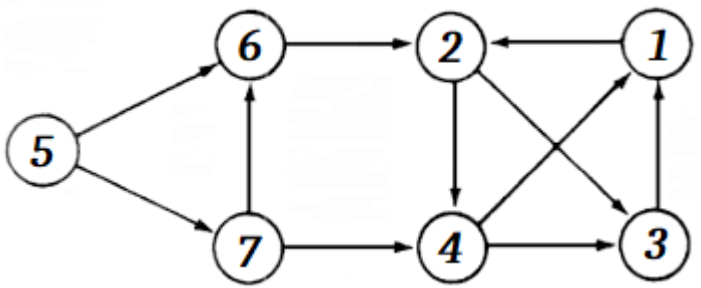
(...)

Formas de percorrer um grafo

Como podemos percorrer esse grafo?

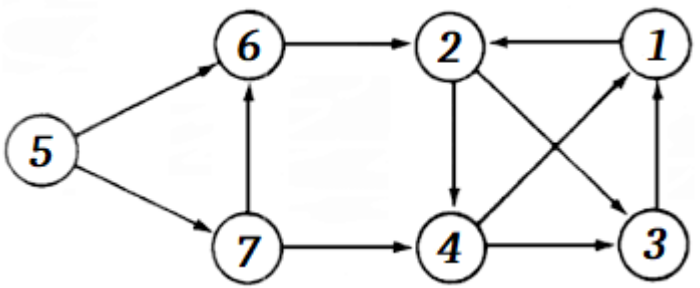


Como podemos percorrer esse grafo?

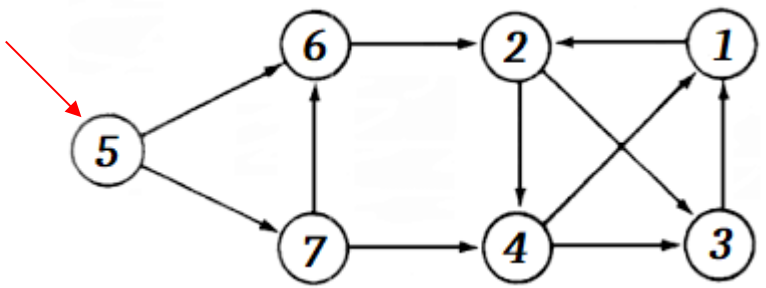


Se simplesmente queremos saber quem são todos os vértices e/ou arestas, logicamente podemos varrer a estrutura de dados de alguma forma.
Mas a forma como percorremos o grafo de um nó a outro pode ser parte da solução de diferentes problemas envolvendo grafos...

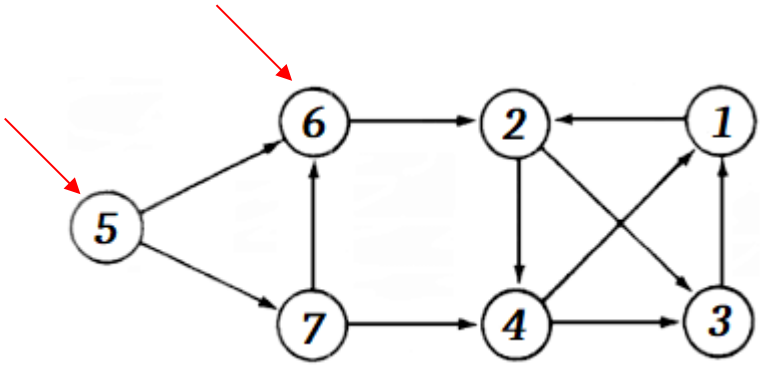
Como podemos identificar se esse grafo possui um ciclo?



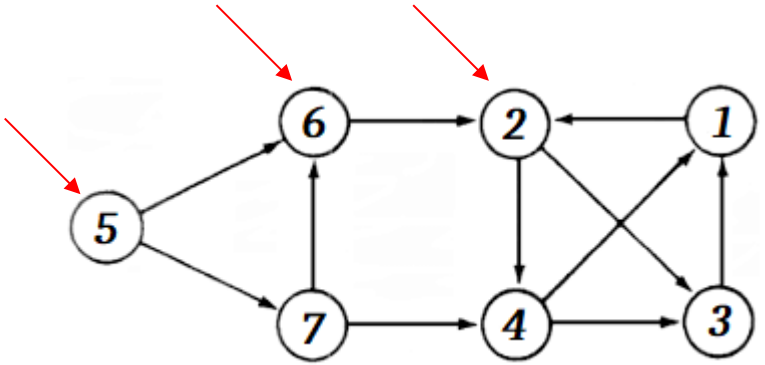
Como podemos identificar se esse grafo possui um ciclo?



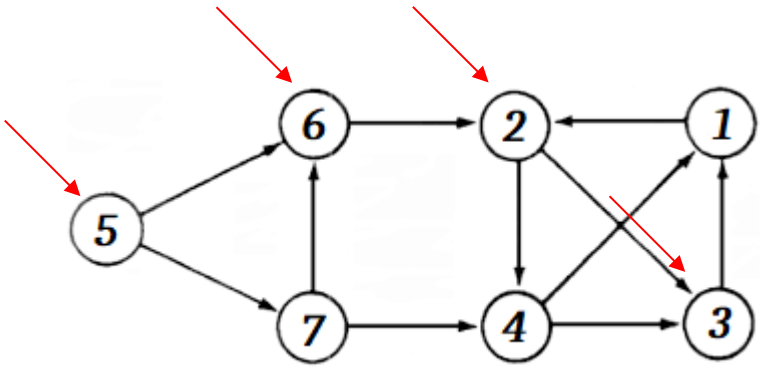
Como podemos identificar se esse grafo possui um ciclo?



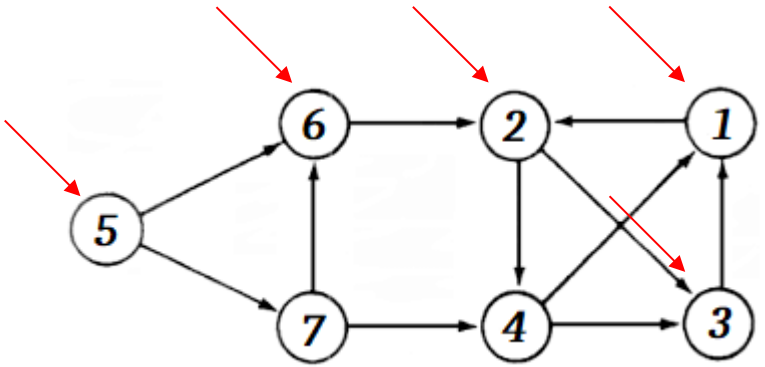
Como podemos identificar se esse grafo possui um ciclo?



Como podemos identificar se esse grafo possui um ciclo?

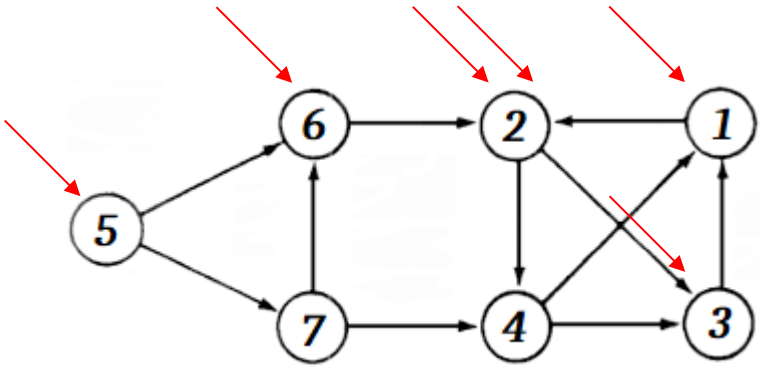


Como podemos identificar se esse grafo possui um ciclo?



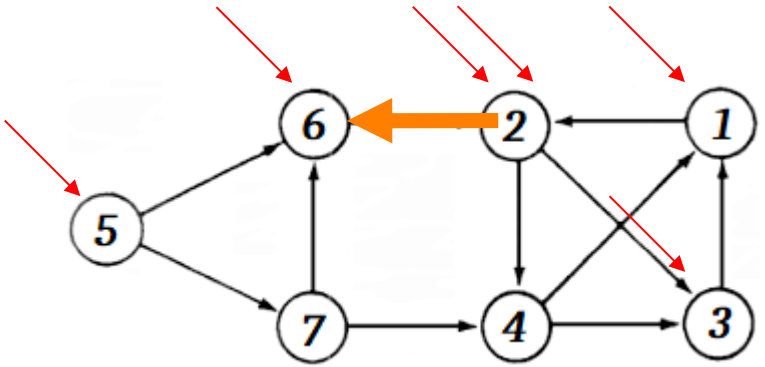
Como podemos identificar se esse grafo possui um ciclo?

BINGO!!!



Como podemos identificar se esse grafo possui um ciclo?

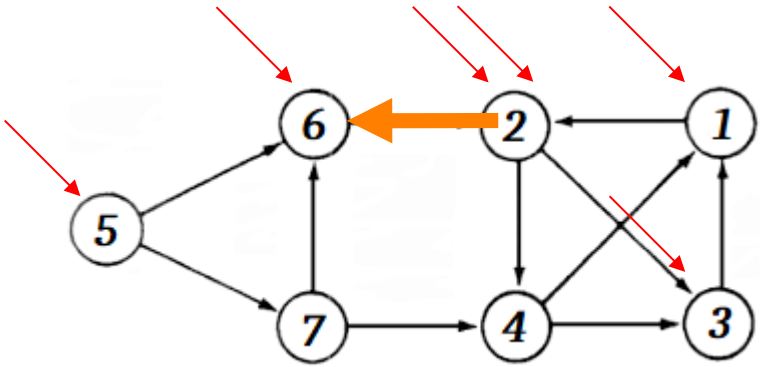
BINGO!!!



E se esse ciclo não fosse alcançável a partir do vértice 6?

Como podemos identificar se esse grafo possui um ciclo?

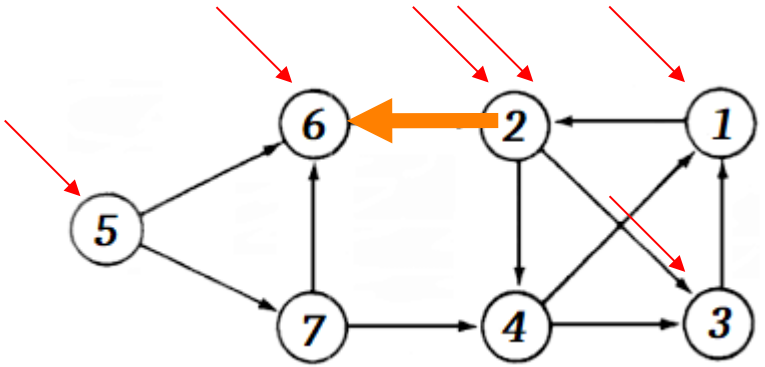
BINGO!!!



E se esse ciclo não fosse alcançável a partir do vértice 6?
Teria que testar também os vários caminhos… (por ex pelo vértice 7)

Como podemos identificar se esse grafo possui um ciclo?

BINGO!!!



Essa forma de percorrer o grafo (sempre indo adiante a cada vértice alcançado) chama-se **busca em profundidade**

Busca em profundidade

- O algoritmo geral tem a finalidade de passar por TODOS OS VÉRTICES seguindo arestas do grafo
- Pode ser utilizado/adaptado para várias aplicações em grafos

Busca em Profundidade

- A busca em profundidade, do inglês *depth-first search*), é um algoritmo para caminhar no grafo.
- A estratégia é buscar o mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.
- Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual v foi descoberto. (antecessor de v)
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

Busca em Profundidade

- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é *descoberto* pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada.
- $d[v]$: tempo de descoberta
- $t[v]$: tempo de término do exame da lista de adjacentes de v.

Medidores de tempo: úteis para
- acompanhar a evolução da busca
- utilizados em vários algoritmos de grafos

- Estes registros são inteiros entre 1 e $2|V|$ pois existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices.

Busca em Profundidade: Exemplo

Cada vértice tem:
cor(d[v],t[v]) e
antecessor (→)