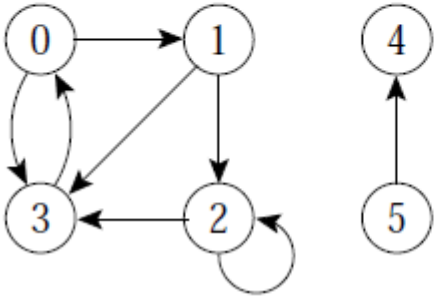
ACH2024

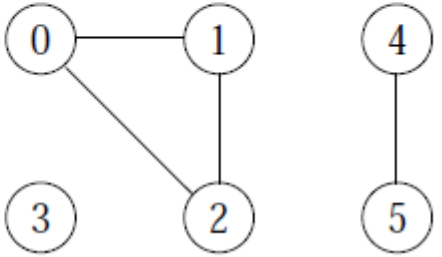
Aula 4

Grafos: Implementação por matriz e lista de adjacência

Prof. Helton Hideraldo Bíscaro

Matriz de Adjacência: Exemplo





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

(a)

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

(b)

Matriz de Adjacência: Estrutura de Dados

```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 TipoPeso;
typedef struct TipoGrafo {
  TipoPeso mat[MAXNUMVERTICES + 1][MAXNUMVERTICES + 1];
  int numVertices;
  int numArestas;
} TipoGrafo;
typedef int TipoApontador;
```

Matriz de Adjacência: Operações

```
/*
  bool inicializaGrafo(TipoGrafo* grafo, int nv): Inicializa um grafo com nv vertices.
  Vertices vao de 0 a nv-1.
  Preenche as celulas com AN (representando ausencia de aresta).
  Retorna true se inicializou com sucesso e false caso contrario
*/
bool inicializaGrafo(TipoGrafo* grafo, int nv);

/* int obtemNrVertices(TipoGrafo* grafo): retorna o numero de vertices do grafo */
int obtemNrVertices(TipoGrafo* grafo);

/* int obtemNrArestas(TipoGrafo* grafo): retorna o numero de arestas do grafo */
int obtemNrArestas(TipoGrafo* grafo);

/*
  bool verificaValidadeVertice(int v, TipoGrafo *grafo): verifica se o nr do vertice eh valido no grafo,
  ou seja, se ele é maior que zero e menor ou igual ao nr total de vertices do grafo.
*/
bool verificaValidadeVertice(int v, TipoGrafo *grafo);

/*
  void insereAresta(int v1, int v2, TipoPeso peso, TipoGrafo *grafo):
  Insere a aresta (v1, v2) com peso "peso" no grafo.
  Nao verifica se a aresta ja existia.
*/
void insereAresta(int v1, int v2, TipoPeso peso, TipoGrafo *grafo);
```

Matriz de Adjacência: Operações

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

/*
  TipoPeso obtemPesoAresta(int v1, int v2, TipoGrafo *grafo):
  Retorna o peso da aresta (v1, v2) no grafo se ela existir e AN caso contrário
*/
TipoPeso obtemPesoAresta(int v1, int v2, TipoGrafo *grafo);


/*
  bool removeArestaObtendoPeso(int v1, int v2, TipoPeso* peso, TipoGrafo *grafo);
  Remove a aresta (v1, v2) do grafo colocando AN em sua celula (representando ausencia de aresta).
  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, TipoPeso* peso, TipoGrafo *grafo);

/*
  bool removeAresta(int v1, int v2, TipoGrafo *grafo);
  Remove a aresta (v1, v2) do grafo colocando AN em sua celula (representando ausencia de aresta).
  Se a aresta existia, retorna true,
  caso contrario retorna false.
*/
bool removeAresta(int v1, int v2, TipoGrafo *grafo);
```

Matriz de Adjacência: Operações

```
/*
   bool listaAdjVazia(int v, TipoGrafo* grafo):
   Retorna true se a lista de adjacencia (de vertices adjacentes) do vertice v é vazia, e false caso contrário.
*/
bool listaAdjVazia(int v, TipoGrafo* grafo);

/*
   TipoApontador primeiroListaAdj(int v, TipoGrafo* grafo):
   Retorna o primeiro vertice da lista de adjacencia de v
   ou VERTICE_INVALIDO se a lista de adjacencia estiver vazia.
*/
TipoApontador primeiroListaAdj(int v, TipoGrafo* grafo);

/*
   TipoApontador proxListaAdj(int v, TipoGrafo* grafo, TipoApontador atual):
   Trata-se de um iterador sobre a lista de adjacência do vertice v.
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou VERTICE_INVALIDO se a lista de adjacencia tiver terminado sem um novo proximo.
*/
TipoApontador proxListaAdj(int v, TipoGrafo* grafo, TipoApontador atual);

/*
    void imprimeGrafo(TipoGrafo* grafo):
    Imprime a matriz de adjacencia do grafo.
    Assuma que cada vértice e cada peso de aresta são inteiros de até 2 dígitos.
*/
void imprimeGrafo(TipoGrafo* grafo);

/* Nao precisa fazer nada para matrizes de adjacencia */
void liberaGrafo(TipoGrafo* grafo);
```

Matriz de adjacência - complexidades

V = número de vértices

Operação	Complexidade
inicializaGrafo	
existeAresta	
insereAresta	
removeAresta	
listaAdjVazia	

Matriz de adjacência - complexidades

V = número de vértices

Operação	Complexidade
inicializaGrafo	$O(V^2)$
existeAresta	O(1)
insereAresta	O(1)
removeAresta	O(1)
listaAdjVazia	O(V)

Matriz de adjacência

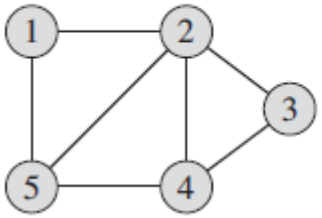
Essa representação por matriz adjacência é sempre eficiente?

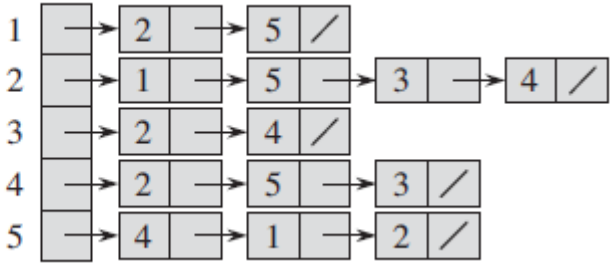
Matriz de Adjacência: Análise

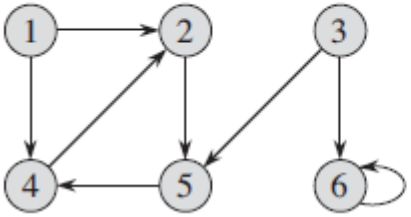
- Deve ser utilizada para grafos **densos**, onde $|A|$ é próximo de $|V|^2$.
- O tempo necessário para acessar uma aresta é independente de $|V|$ ou $|A|$.
- É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.
- A maior desvantagem é que a matriz necessita $\Omega(|V|^2)$ de espaço. Ler ou examinar a matriz tem complexidade de tempo $O(|V|^2)$.
- A inserção de ~~um novo vértice~~ uma nova aresta ou retirada de ~~um vértice~~ uma aresta já existente pode ser realizada com custo constante.

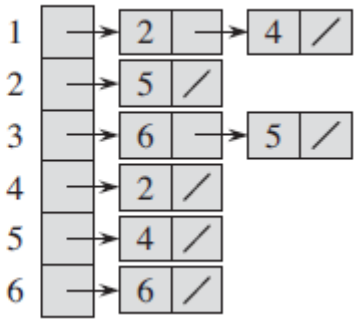
Outra sugestão de implementação de grafos para grafos não densos?

Listas de adjacência









Lista de adjacência - estrutura

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

typedef int TipoPeso;

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

typedef TipoAresta* TipoApontador;

/*
  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 {
  TipoApontador *listaAdj;
  int numVertices;
  int numArestas;
} TipoGrafo;
```

Lista de adjacência - operações

(as mesmas que as de matriz de adjacência)

Lista de adjacência - operações

```
/*
  inicializaGrafo(TipoGrafo* grafo, int nv): Cria um grafo com n vertices.
  Aloca espaco para o vetor de apontadores de listas de adjacencias e,
  para cada vertice, inicializa o apontador de sua lista de adjacencia.
  Retorna true se inicializou com sucesso e false caso contrario.
  Vertices vao de 1 a nv.
*/

bool inicializaGrafo(TipoGrafo *grafo, int nv);
```

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$		
existeAresta	O(1)		
insereAresta	O(1)		
removeAresta	O(1)		

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)		
insereAresta	O(1)		
removeAresta	O(1)		

Lista de adjacência - operações

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

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)		
insereAresta	O(1)		
removeAresta	O(1)		

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)		
removeAresta	O(1)		

Lista de adjacência - operações

```
/*
  void insereAresta(int v1, int v2, TipoPeso peso, TipoGrafo *grafo):
  Insere NÃO ORDENADO a aresta (v1, v2) com peso "peso" no grafo.
  Nao verifica se a aresta ja existe.
*/
void insereAresta(int v1, int v2, TipoPeso peso, TipoGrafo *grafo);
```

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)		Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)		

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)		

Lista de adjacência - operações

```
/*
  bool removeAresta(int v1, int v2, TipoPeso* peso, TipoGrafo *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 removeAresta(int v1, int v2, TipoPeso* peso, TipoGrafo *grafo);
```

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)		Tem que encontrar a aresta

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta

Lista de adjacência - operações

```
/*
   bool listaAdjVazia(int v, TipoGrafo* grafo):
   Retorna true se a lista de adjacencia (de vertices adjacentes) do vertice v é vazia, e false caso contrário.
*/
bool listaAdjVazia(int v, TipoGrafo *grafo);
```

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta

Lista de adjacência - operações

```
/*
   TipoApontador primeiroListaAdj(int v, TipoGrafo* grafo):
   Retorna o endereco do primeiro vertice da lista de adjacencia de v
   ou NULL se a lista de adjacencia estiver vazia.
*/
TipoApontador primeiroListaAdj(int v, TipoGrafo *grafo);
```

Lista de adjacência - operações

```
/*
   TipoApontador proxListaAdj(int v, TipoGrafo* grafo):
   Retorna o proximo vertice adjacente a v, partindo do vertice "prox" adjacente a
v
   ou NULL se a lista de adjacencia tiver terminado sem um novo proximo.
*/
TipoApontador proxListaAdj(int v, TipoGrafo *grafo, TipoApontador prox);
```

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta

Matriz e listas de adjacência - complexidades

V = número de vértices

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta

Lista de adjacência - operações

```
/*
    void imprimeGrafo(TipoGrafo* 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(TipoGrafo *grafo);
```

Matriz e listas de adjacência - complexidades

V = número de vértices, A = número de arestas

	Matriz	**Listas**	**Observação**
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta
listaAdjVazia	O(V)	O(1)	

Matriz e listas de adjacência - complexidades

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta
listaAdjVazia	O(V)	O(1)	

Lista de adjacência - operações

```
/*
  void liberaGrafo (TipoGrafo *grafo): Libera o espaco ocupado por um
grafo.
*/
void liberaGrafo (TipoGrafo *grafo);
```

	Matriz	Listas	Observação
Operação	**Complexidade**	**Complexidade**	
inicializaGrafo	$O(V^2)$	O(V)	
existeAresta	O(1)	O(V)	
insereAresta	O(1)	O(1)	Assumindo que não é verificada a pré-existência da aresta
removeAresta	O(1)	O(V)	Tem que encontrar a aresta
listaAdjVazia	O(V)	O(1)	
proxListaAdj	O(V)	O(1)	

Exercícios

Implementar a estrutura e operações de grafos utilizando listas de adjacência para:

- grafos direcionados
- grafos não direcionados