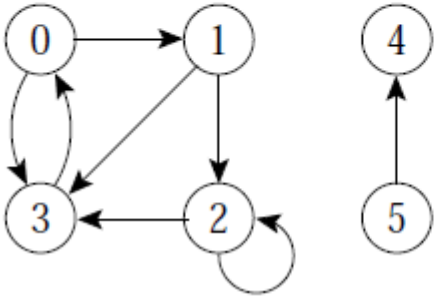
ACH2024

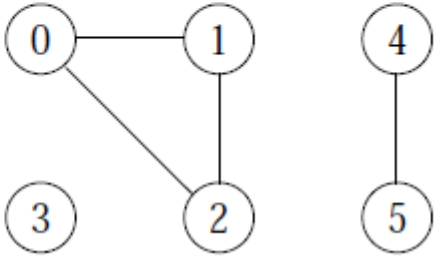
Aula 5

Grafos: Implementações
Matriz e Lista de Adjacência (cont.)

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)

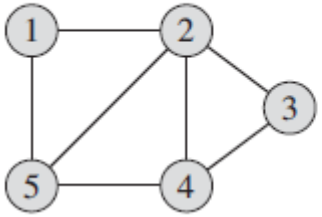
	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						1
5					1	

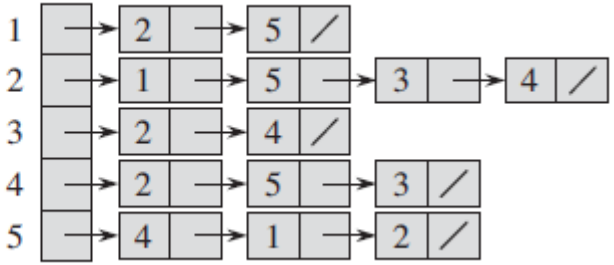
(b)

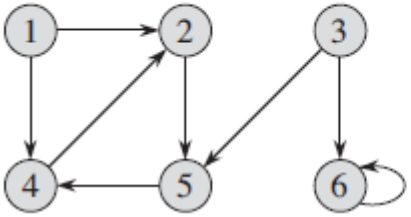
Matriz de Adjacência: Estrutura de Dados

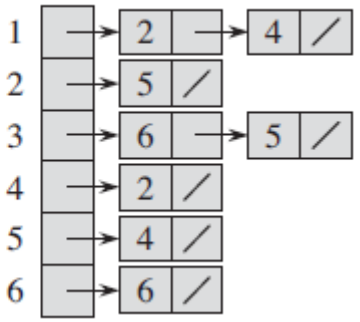
```c
#define MAXNUMVERTICES 100
#define AN -1                 /* aresta nula, ou seja, valor que representa ausencia de aresta */
#define VERTICE_INVALIDO -1   /* numero de vertice invalido ou ausente */

#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;
```

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;
```

	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)	

Lista de adjacência - operações

```
/*
  void liberaGrafo (TipoGrafo *grafo): Libera o espaço 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)	

	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)	

Matriz e listas de adjacência - escolhas

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

- O grafo é esparso? ($|A| << |V^2|$)
- 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

Cuidados para transparência de implementação

Incrementando o Makefile

Testando o “testa_grafo”

Cuidado especial com primeiroListaAdj e proxListaAdj

Pergunta

Como identificar se um grafo possui ou não ciclos?

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

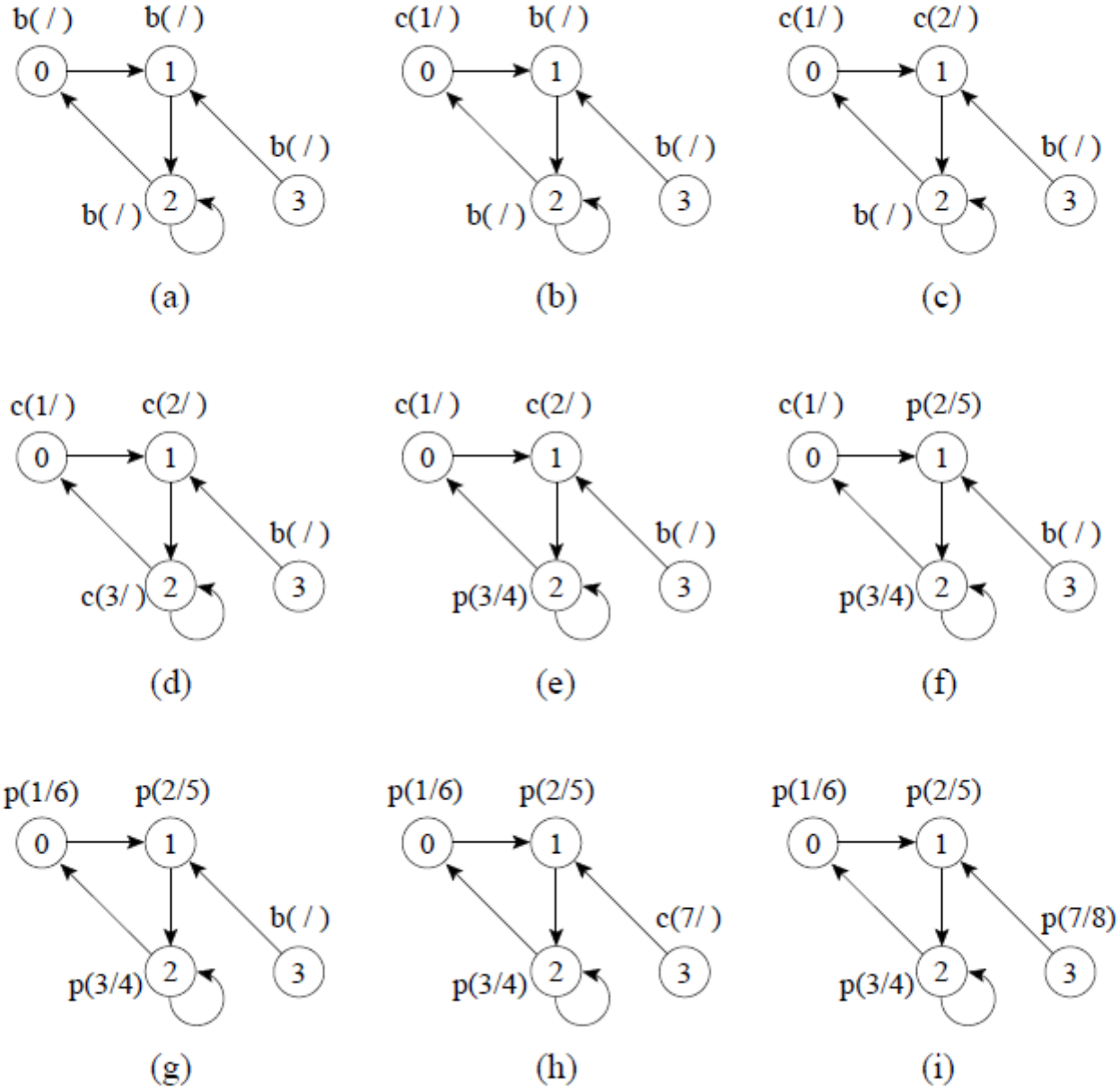
- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é *descoberto* pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada.

- $d[v]$: tempo de descoberta
- $t[v]$: tempo de término do exame da lista de adjacentes de v.
- 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.

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

Busca em Profundidade: Exemplo

Cada vértice tem: cor(d[v],t[v])



Slide do livro do Ziviani – cap 7

Exercícios

Implementar busca em profundidade utilizando as operações de interface dos grafos

Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
    Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
    tempo ← 0;
    Para cada vertice v
        cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
    Para cada vertice v
        Se cor[v] = branco visitaBP(v, grafo, &tempo, cor, tdesc, tterm, antecessor);
}


visitaBP(v, grafo, tempo, cor, tdesc, tterm, antecessor){
    cor[v] ← cinza; tdesc[v] ← ++(*tempo);
    Para cada vertice u da lista de adjacência de v
        Se u é branco
            antecessor[u] ← v;
            visitaBP(u, grafo, &tempo, cor, tdesc, tterm, antecessor);
    tterm ← ++(*tempo);
    cor[v] ← preto;
}
```

Busca em Profundidade: Análise

- Os dois anéis da *BuscaEmProfundidade* têm custo $O(|V|)$ cada um, a menos da chamada do procedimento *VisitaDfs*(u) no segundo anel.
- O procedimento *VisitaDfs* é chamado exatamente uma vez para cada vértice $u \in V$, desde que *VisitaDfs* é chamado apenas para vértices brancos e a primeira ação é pintar o vértice de cinza.
- Durante a execução de *VisitaDfs*(u) o anel principal é executado $|Adj[u]|$ vezes.
- Desde que $\sum_{u \in V} |Adj[u]| = O(|A|)$, o tempo total de execução de *VisitaDfs* é $O(|A|)$.
- Logo, a complexidade total da *BuscaEmProfundidade* é $O(|V| + |A|)$.

Busca em Profundidade: Análise

Essa complexidade vale para as duas implementações de grafos? Se não:

- qual a complexidade para matrizes e listas de adjacência?
- então deve-se sempre usar uma ao invés de outra? Se não, por quê?

Classificação de Arestas

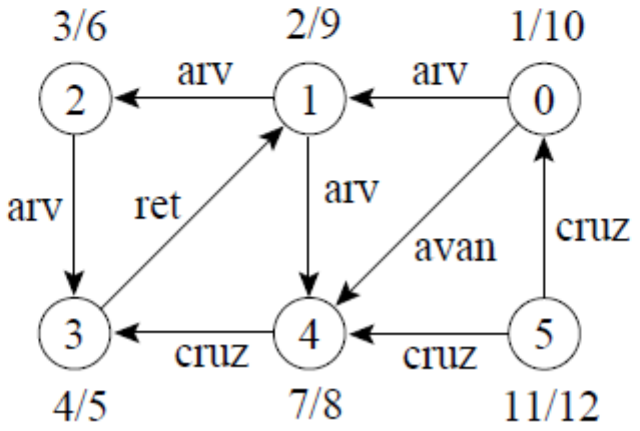
1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).
3. **Arestas de avanço**: não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.
4. **Arestas de cruzamento**: podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Neste caso (vértices da mesma árvore de busca), cruza ramos desta árvore, já que conecta um vértice a um outro que **não é seu antecessor nem descendente**

Slide do livro do Ziviani – cap 7

Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de retorno.
  - Preto indica uma aresta de avanço quando u é descoberto antes de v ou uma aresta de cruzamento caso contrário.



(u,v) é
avanço se tdesc[u] < tdesc[v]
cruzamento caso contrário

Observação

Se o grafo é não direcionado, a busca em profundidade produzirá apenas arestas de árvore e arestas de retorno

Por quê?

Algumas aplicações de busca em profundidade

Identificar se um grafo é acíclico ou não

Teste para Verificar se Grafo é Acíclico Usando Busca em Profundidade

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em profundidade em G, então o grafo tem ciclo.
- Um grafo direcionado G é acíclico se e somente se a busca em profundidade em G não apresentar arestas de retorno.
- O algoritmo BuscaEmProfundidade pode ser alterado para descobrir arestas de retorno. Para isso, basta verificar se um vértice v adjacente a um vértice u apresenta a cor cinza na primeira vez que a aresta (u, v) é percorrida.
- O algoritmo tem custo linear no número de vértices e de arestas de um grafo $G = (V, A)$ que pode ser utilizado para verificar se G é **a**cíclico.

Exercício

Implemente tal algoritmo.

E imprima tal ciclo?

Pergunta-Exercício

Por que a busca precisa ser em profundidade (não espalhada) para detectar um ciclo? (dica: veja por exemplo o grafo do slide 10).