Grafos

Estruturas de Dados e Implementação em linguagem C

Maria Adriana Vidigal de Lima

FACOM - UFU

Sumário

- Definição de Grafo
- Representação
- Algoritmos
  - Inserção de arestas e vértices
  - Exclusão de arestas e vértices
  - Busca

Grafo: Definição e Terminologia

Definição de Grafo

Um grafo é uma estrutura de dados não-linear constituída de um conjunto V (não vazio) de vértices ou nós, e um conjunto A (possivelmente vazio) de arestas ou arcos, conectando pares de vértices. Cada arco é especificado por um par de nós.

De maneira formal:

$$G = (V, A)$$

$|V|$ é a quantidade de vértices de G

$|A|$ é a quantidade de arestas de G

Ordem de um Grafo

A ordem de um grafo é o número de vértices que ele possui:

$$ordem(G) = |V|$$

Organização dos dados

Grafos podem ser representados fisicamente utilizando-se:

- Matriz de adjacência
- Lista de adjacência

Matriz de Adjacência

Uma **matriz de adjacência** $M(n \times n)$ de um grafo G de ordem n, é uma matriz em que cada célula m_{ij} é:

- Em grafos direcionados:

  $m_{ij} = 1$ se $(v_i, v_j) \in G(A)$

  $m_{ij} = 0$ se $(v_i, v_j) \notin G(A)$

- Em grafos não-direcionados: $m_{ij} = m_{ji}$

  $m_{ij} = 1$ se $\{v_i, v_j\} \in G(A)$

  $m_{ij} = 0$ se $\{v_i, v_j\} \notin G(A)$

Matriz de Adjacência

A **matriz de adjacência** é uma forma de representação simples e adequada para muitos problemas cuja solução se baseia na estrutura de grafo.

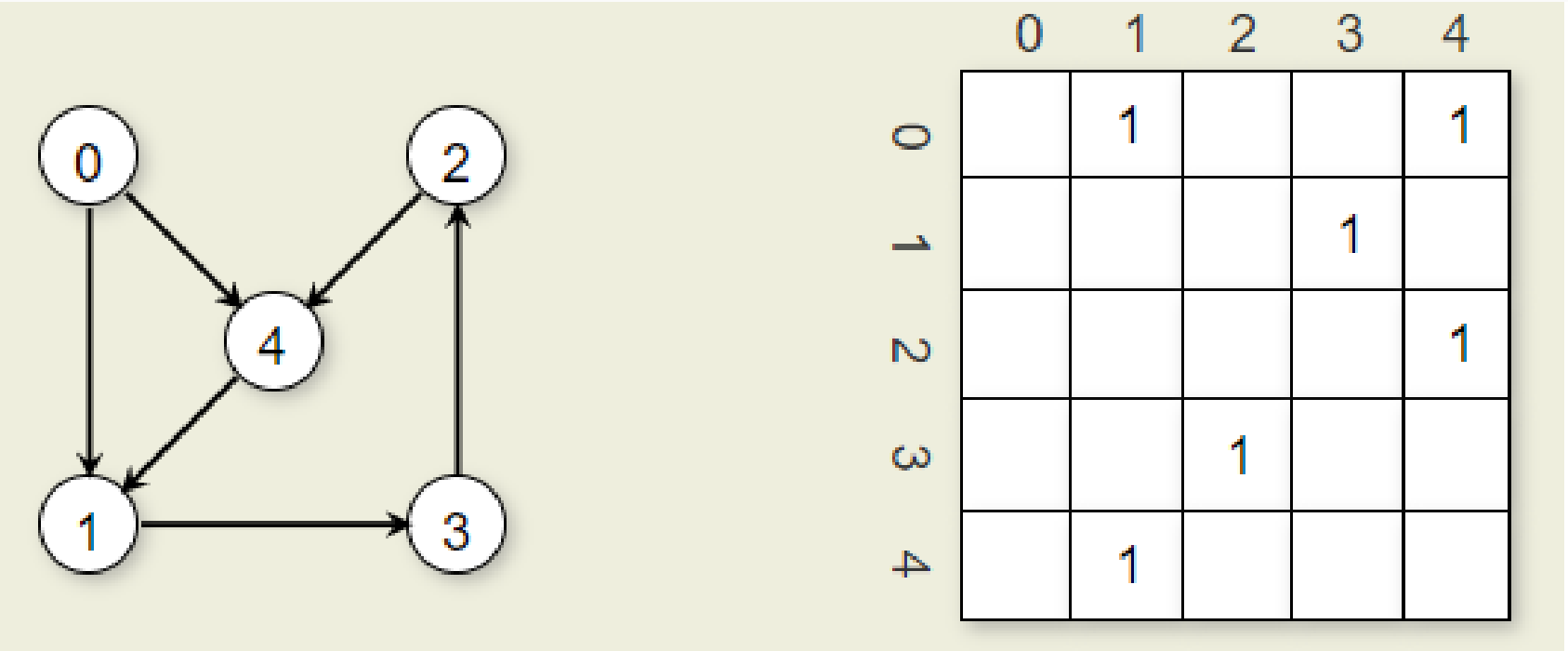


Figura 1: Matriz para grafo direcionado[1]

[1] https://opendsa-server.cs.vt.edu/

Matriz de Adjacência

Para grafos não-direcionados, a matriz é **simétrica**.

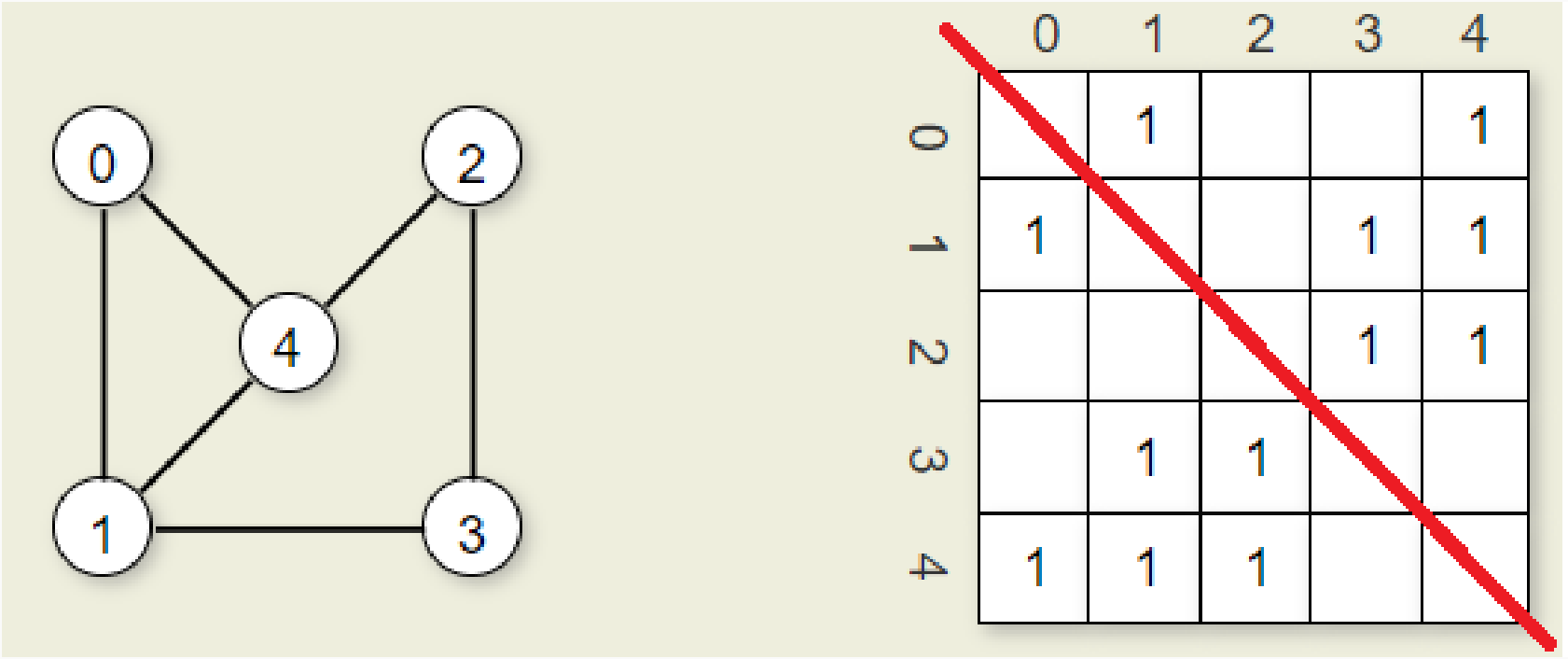
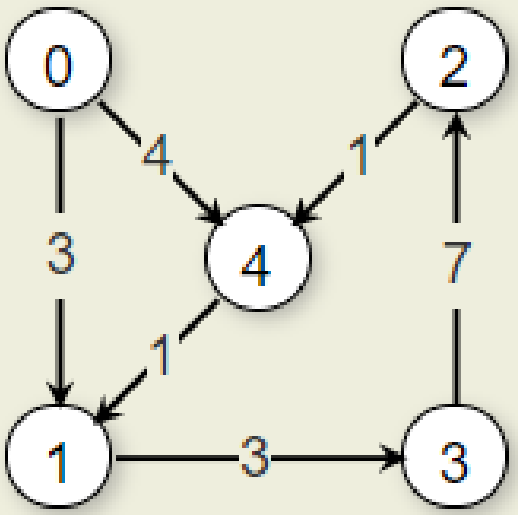


Figura 2: Matriz para grafo não-direcionado

Matriz de Adjacência

Para **grafos ponderados** com pesos nas arestas, a matriz armazena os valores contidos nas arestas:

$$m_{ij} = \text{k se } (v_i, v_j, \text{k}) \in G(A)$$



	0	1	2	3	4
0		3			4
1				3	
2					1
3			7		
4		1			

Figura 3: Matriz para grafo ponderado e direcionado

Matriz de Adjacência

- A construção da matriz e sua manipulação são operações simples.
- É fácil determinar se $(v_i, v_j) \in G(A)$.
- É fácil encontrar os vértices adjacentes a um determinado vértice v.
- Quando o grafo é não orientado, a matriz é simétrica (mais econômica).
- A inserção de novas arestas é fácil.
- A inserção de novos vértices é muito difícil.

Matriz de Adjacência: Análise

- Indicada para grafos densos, quando $|A|$ é um valor próximo de $|V^2|$.
- O tempo para acessar um elemento na matriz é independente de $|A|$ ou $|V|$ (acesso direto $m[i][j]$).
- Muito útil para problemas em que necessitamos saber rapidamente se existe uma ligação entre dois vértices.
- Necessita de $S(|V^2|)$ de espaço para armazenamento. Da mesma forma, ler ou examinar a matriz de adjacência tem complexidade $O(|V^2|)$.

Lista de Adjacência

A **lista de adjacência** consiste em criar uma lista para cada vértice. Esta lista contém cada vértice que o vértice tem ligação.

Considerando o **grafo direcionado**, para cada vértice v é associada a lista de vértices u tais que $(v, u) \in G(A)$:

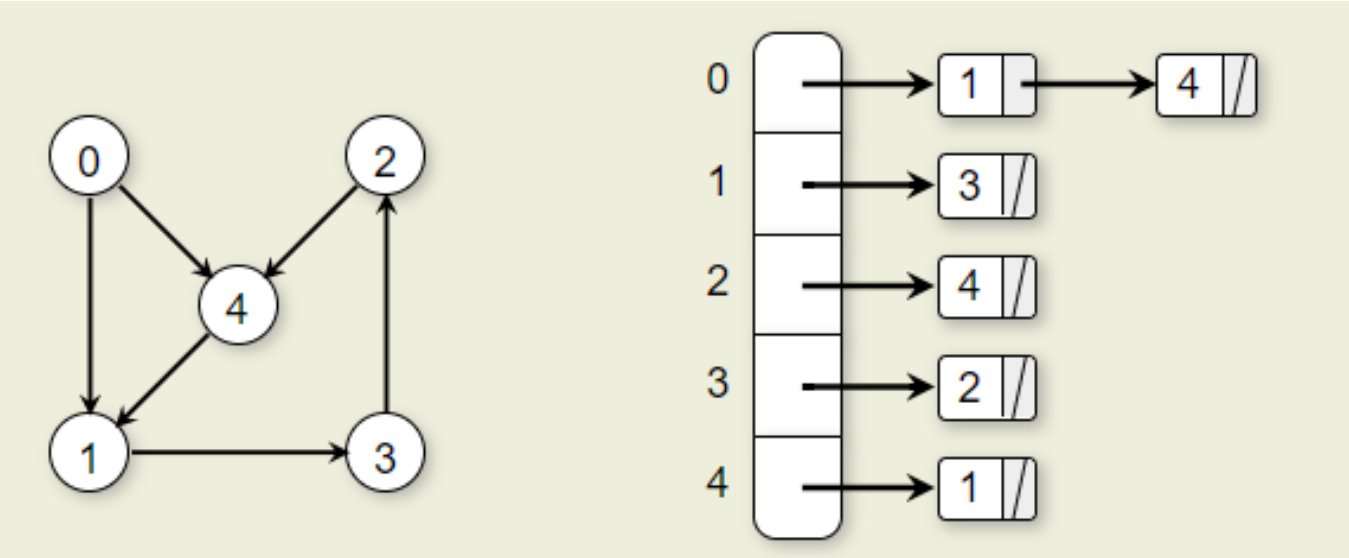


Figura 4: Lista de adjacência para grafo direcionado

Lista de Adjacência

Para um **grafo não-direcionado**, para cada vértice v é associada a lista de vértices u tais que $\{v, u\} \in G(A)$:

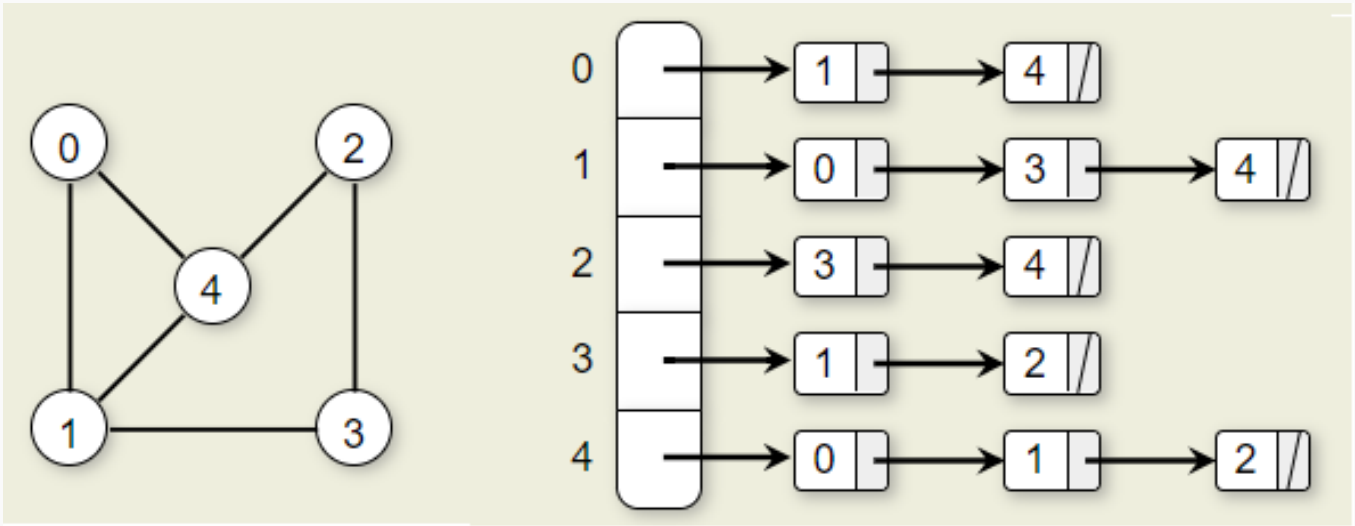


Figura 5: Lista de adjacência para grafo não-direcionado

Lista de Adjacência

Para um **grafo ponderado**, para cada vértice v é associada a lista de vértices u com seus pesos correspondentes, tais que $(v, u, k) \in G(A)$:

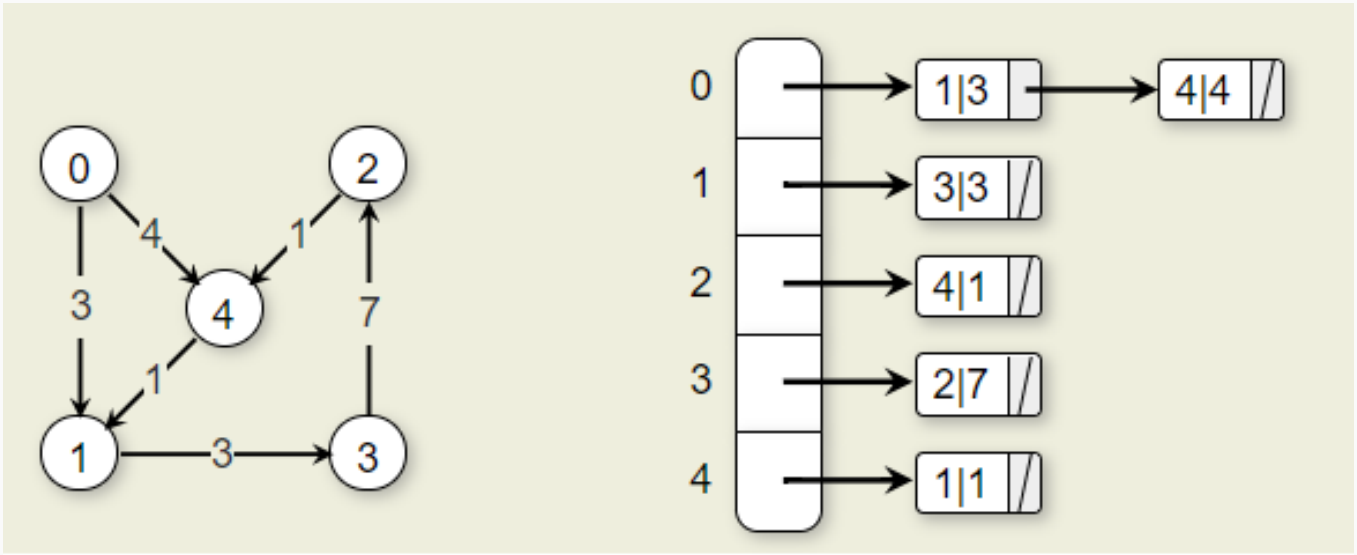


Figura 6: Lista de adjacência para grafo ponderado e direcionado

Lista de Adjacência

- Possíveis formas de armazenamento: vetores, vetores e listas encadeadas, listas encadeadas.
- Melhor forma de representação: listas encadeadas.
  - Uso otimizado do espaço.
  - Flexibilidade para inserção de novos vértices/arestas.

Lista de Adjacência: Análise

- Os vértices de uma lista de adjacência são, em geral, armazenados em uma ordem arbitrária.
- Possui complexidade de espaço $S(|V| + |A|)$.
- Indicada para grafos esparsos, em que $|A|$ é bem menor que $|V^2|$.
- A lista fica compacta e é normalmente utilizada na maioria das aplicações.
- Desvantagem: pode ter tempo $O(|V|)$ para verificar se existe uma ligação entre dois vértices v e u, pois podem existir V vértices na lista de v.

Matriz X Lista de Adjacência

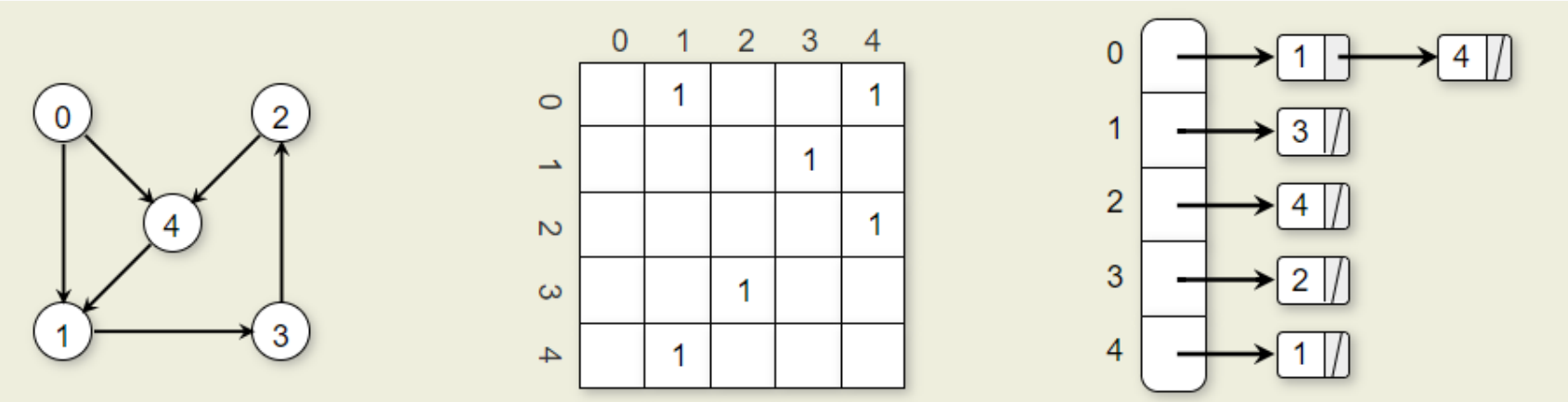


Figura 7: Adjacência para grafo direcionado

Matriz X Lista de Adjacência

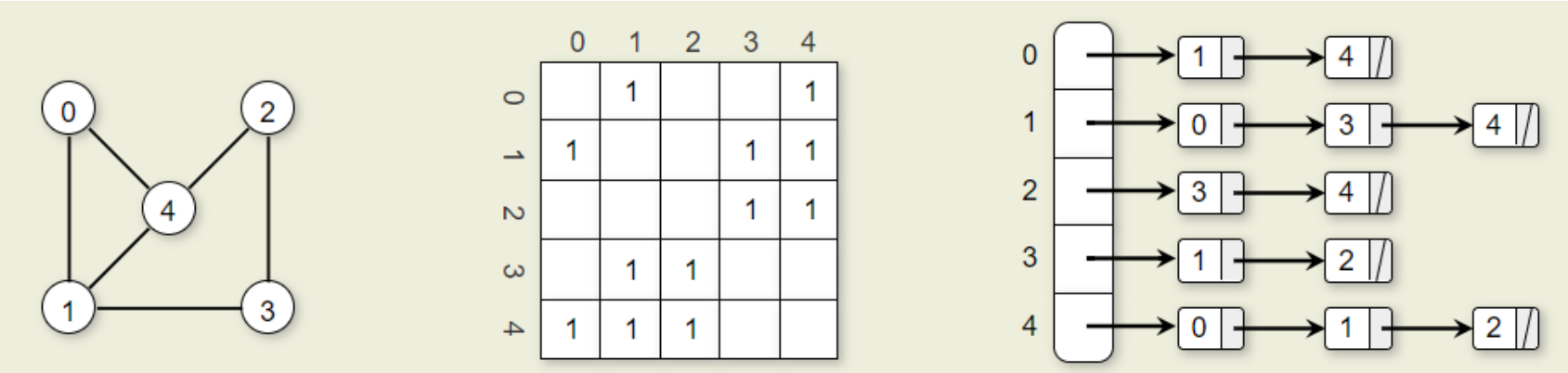


Figura 8: Adjacência para grafo não-direcionado

Matriz X Lista de Adjacência

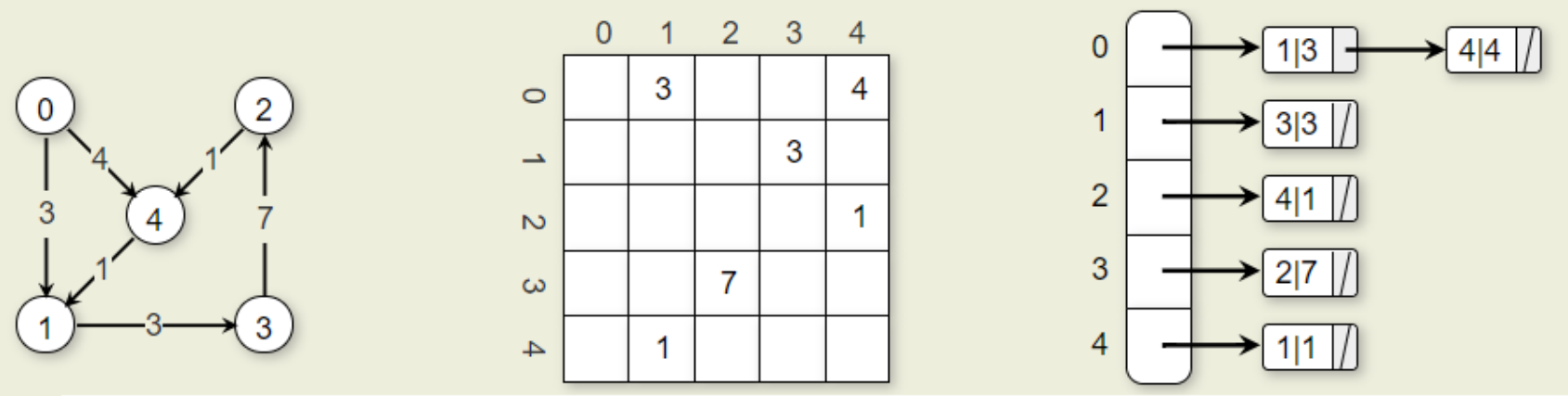


Figura 9: Adjacência para grafo ponderado e direcionado

Matriz de Adjacência - Implementação em C

Para a implementação de grafos **direcionados** usando matriz de adjacência pode-se ter a seguinte **definição de tipos**:

```
1  struct grafo {
2      int NumVertices;
3      int NumArestas;
4      int **Mat;
5  };
6  typedef struct grafo TipoGrafo;
```

Matriz de Adjacência - Implementação em C

As seguintes funções são fundamentais para a construção/destruição do grafo e inclusão/remoção de arestas:

```
1  TipoGrafo* CriarGrafo(int NVertices);
2  int inserirAresta(TipoGrafo *G, int v1, int v2);
3  int retirarAresta(TipoGrafo *Grafo, int v1, int v2);
4  void exibirGrafo(TipoGrafo *G);
5  void exibirMatriz(TipoGrafo *G);
6  TipoGrafo* liberarGrafo(TipoGrafo* G);
```

Matriz de Adjacência - Implementação em C

A função de criação do grafo deve:

- alocar área para a variável grafo (`TipoGrafo`)
- receber a quantidade de vértices do grafo (N) para que possa ser feita a alocação de memória para a matriz ($N \times N$)
- devolver a posição de memória de início para esta alocação.

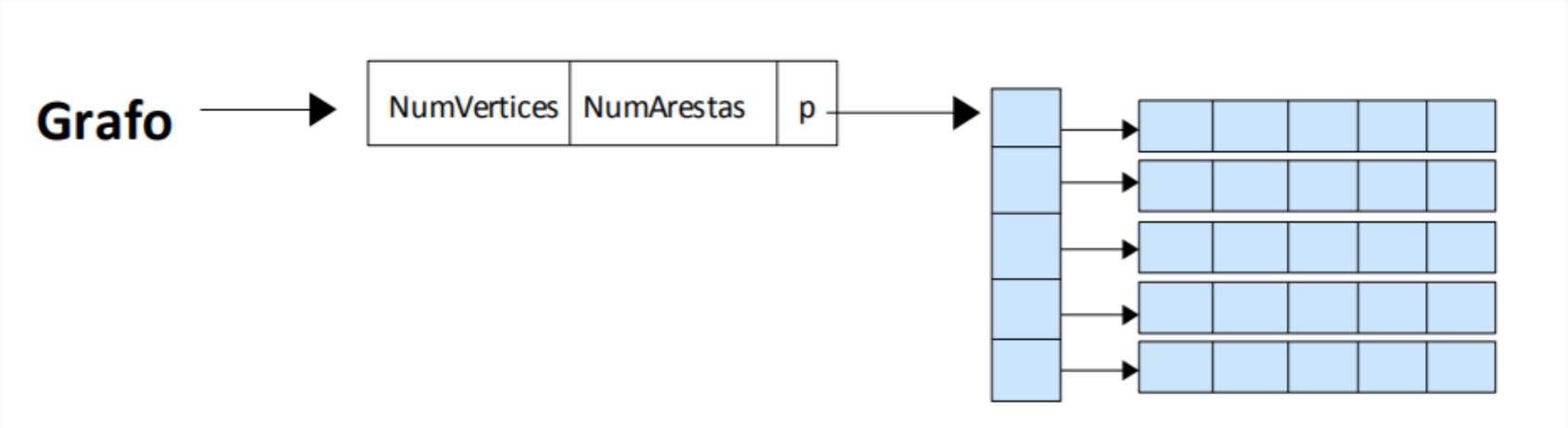


Figura 10: Alocação de memória para a Matriz

Matriz de Adjacência - Implementação em C

Criação de um grafo:

```
1  TipoGrafo* CriarGrafo(int NVertices){
2     int i, k;
3     TipoGrafo *Grafo;
4     if (NVertices <= 0) return NULL;
5     Grafo = (TipoGrafo*) malloc (sizeof(TipoGrafo));
6     if (Grafo == NULL) return NULL;
7
8     Grafo->Mat = (int **) malloc(NVertices * sizeof(int*));
9     if (Grafo->Mat == NULL) {
10       free(Grafo);
11       return NULL;
12    }
```

Matriz de Adjacência - Implementação em C

Criação de um grafo: continuação

```
13      for(i=0; i<NVertices; i++) {
14          Grafo->Mat[i] = (int*) calloc (NVertices,sizeof(int));
15          if (Grafo->Mat[i] == NULL) {
16              for (k=0; k<i; k++)
17                  free(Grafo->Mat[k]);
18              free(Grafo);
19              return NULL;
20          }
21      }
22      Grafo->NumVertices = NVertices;
23      Grafo->NumArestas = 0;
24      return Grafo;
25  }
```

Matriz de Adjacência - Implementação em C

Inserir uma aresta no grafo:

```
1  int inserirAresta(TipoGrafo *G, int v1, int v2) {
2    if (G == NULL)
3       return -1; // grafo nao existe
4    if (v1<0 || v1>= G-> NumVertices || v2 < 0 || v2 >= G->NumVertices)
5       return -1; // nao eh possivel criar aresta: intervalo
6    if (G->Mat[v1][v2] == 0) {
7          G->Mat[v1][v2] = 1;
8          G->NumArestas++;
9     }
10 }
```

Matriz de Adjacência - Implementação em C

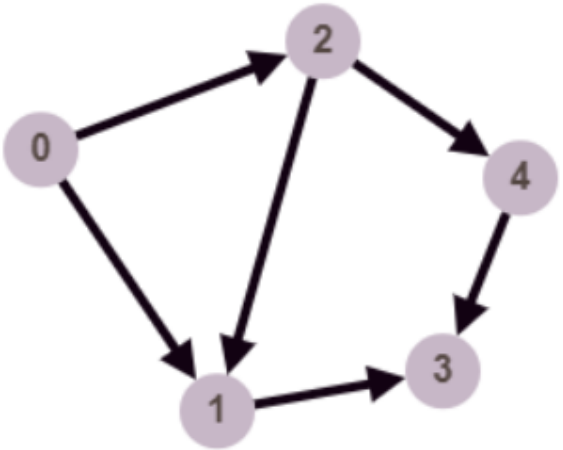
Retirar uma aresta do grafo:

```
1  int retirarAresta(TipoGrafo *G, int v1, int v2){
2     if (G == NULL)
3        return -1; // grafo nao existe
4     if (v1<0 || v1>=G->NumVertices || v2<0 || v2>=G-> NumVertices)
5        return -1; // nao eh possivel retirar aresta: intervalo
6     if( G->Mat[v1][v2] == 0)
7        return 0; // aresta nao existe
8     G->Mat[v1][v2] = 0; //remove aresta
9     G->NumArestas--;
10    return 1;
11 }
```

Matriz de Adjacência - Implementação em C

Imprimir a matriz do grafo:

```
1  void exibirMatriz(TipoGrafo *G){
2     int v, w;
3     printf("\n Grafo - Matriz:\n");
4     for (v = 0; v < G->NumVertices; v++) {
5         printf("%d:", v);
6         for (w = 0; w < G->NumVertices; w++)
7              printf(" %d",G->Mat[v][w]);
8         printf("\n");
9     }
10 }
```

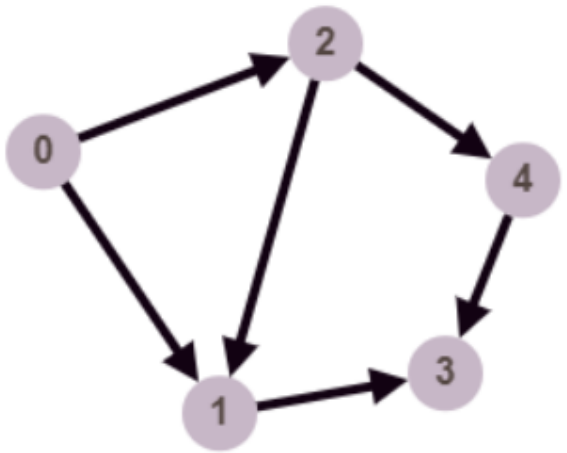



```
Grafo - Matriz:
0: 0 1 1 0 0
1: 0 0 0 1 0
2: 0 1 0 0 1
3: 0 0 0 0 0
4: 0 0 0 1 0
```

Matriz de Adjacência - Implementação em C

Imprimir as adjacências do grafo:

```
1  void exibirGrafo(TipoGrafo *G) {
2     int v, w;
3     printf("\nGrafo - Resumo:\n");
4     for (v = 0; v < G->NumVertices; ++v) {
5         printf("%d:", v);
6         for (w = 0; w < G->NumVertices; ++w)
7            if (G->Mat[v][w] == 1)
8                  printf(" %d",w );
9         printf( "\n");
10    }
11 }
```




```
Grafo - Resumo:
0: 1 2
1: 3
2: 1 4
3:
4: 3
```

Matriz de Adjacência - Implementação em C

Destruição do grafo e liberação da memória alocada:

```
1  TipoGrafo* liberarGrafo(TipoGrafo* G){
2     int i;
3     if (G == NULL)
4        return NULL;     // grafo nao existe
5     for(i=0; i< G->NumVertices; i++)
6        free(G->Mat[i]);   // liberar cada linha da matriz
7     free(G->Mat);
8     free(G);
9     G = NULL;
10    return G;
11 }
```

Matriz de Adjacência - Implementação em C

Exercício: Fazer uma função que verifica a existência de uma dada aresta no grafo, de acordo com o protótipo definido na linha 7:

```
1  TipoGrafo* CriarGrafo(int NVertices);
2  int inserirAresta(TipoGrafo *G, int v1, int v2);
3  int retirarAresta(TipoGrafo *Grafo, int v1, int v2);
4  void exibirGrafo(TipoGrafo *G);
5  void exibirMatriz(TipoGrafo *G);
6  TipoGrafo* liberarGrafo(TipoGrafo* G);
7  int verificaAresta(TipoGrafo *Grafo, int v1, int v2);
```

Lista de Adjacência - Implementação em C

- Dado um grafo $G = (V, A)$, as listas de adjacências são um conjunto de listas, uma para cada vértice $v \in V$. Cada lista contém os vértices w adjacentes à v em G.
- As listas de adjacências consistem tradicionalmente em um vetor para os $|V|$ vértices que são capazes de apontar, cada um, para uma lista linear.
- A implementação a seguir utiliza a representação de **lista dinâmica encadeada** para o armazenamento dos vértices, permitindo maior flexibilidade na inserção e remoção dos mesmos.

Lista de Adjacência - Implementação em C

Para a implementação de grafos usando lista de adjacência, pode-se ter a seguinte **definição de tipos**:

```
1  struct grafo {
2    int NumVert;
3    int NumArco;
4    struct noVert *vertices;
5  };
6  typedef struct grafo *Grafo;
7
8  struct noVert {
9    int vert;
10   struct noVert *prox;
11   struct noAdj *ladj;
12 };
13
14 struct noAdj {
15   int vert;
16   struct noAdj *prox;
17 };
```

Lista de Adjacência - Implementação em C

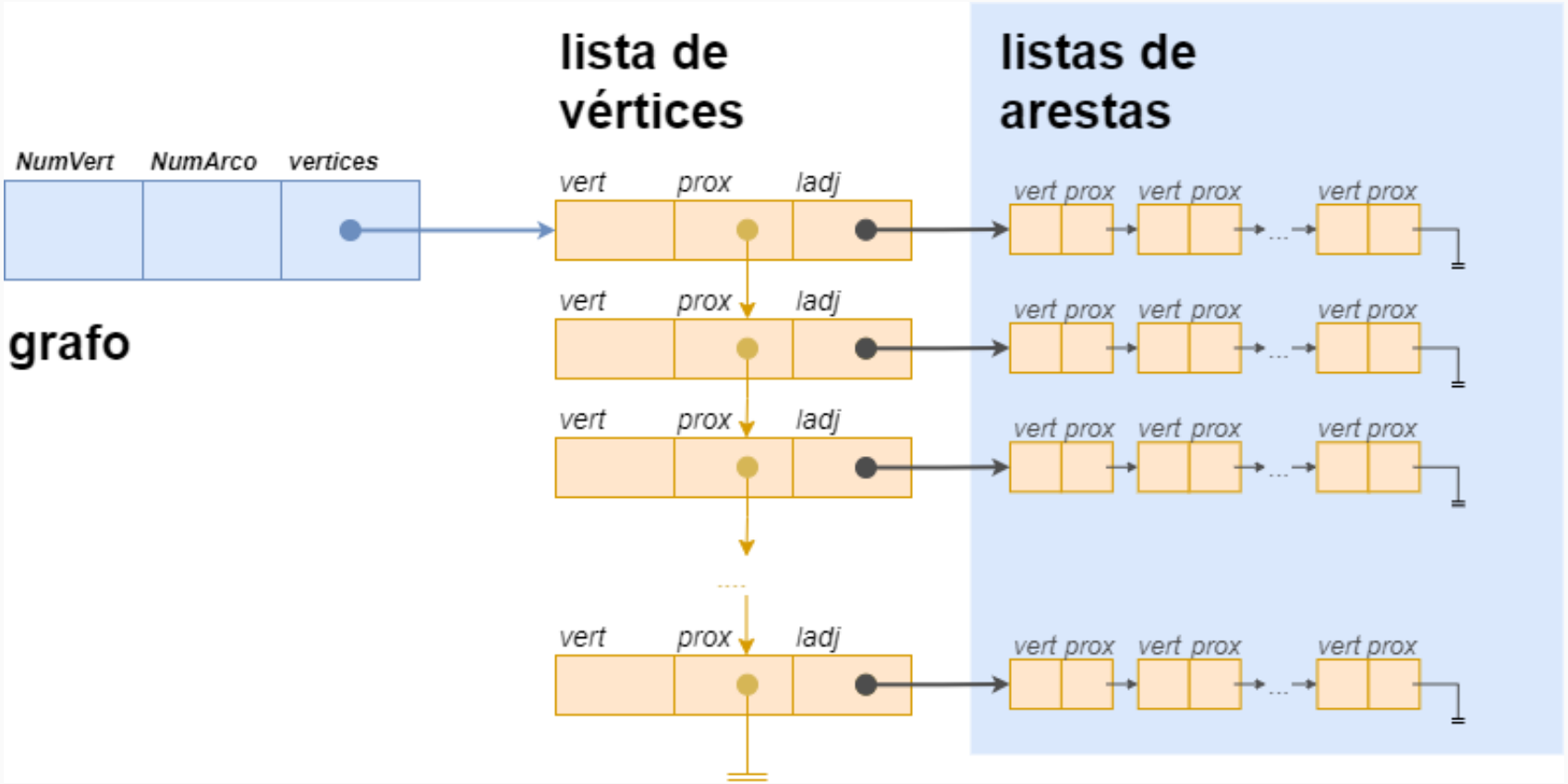


Figura 11: Visão geral da Lista de Adjacência

Lista de Adjacência - Implementação em C

Considerando a utilização de lista encadeada para o armazenamento dos vértices, as seguintes funções estão propostas para iniciar esta implementação para **grafo direcionado**:

```
1 Grafo criarGrafo(int nVert);
2 void inserirArco(Grafo G, int v1, int v2);
3 void inserirNovoVertice(Grafo G, int nv);
4 void imprimirListaAdj(Grafo G);
```

Lista de Adjacência - Implementação em C

A função de criação do grafo deve:

- Alocar área para a estrutura `grafo` (`NumVert, NumArco, vertices`) e iniciar `NumVert` e `NumArco` com 0 (zero).
- Para cada novo vértice: alocar memória, iniciar `ladj` com `NULL`, inserir na lista encadeada `vertices`, e por fim, e atualizar `NumVert`.

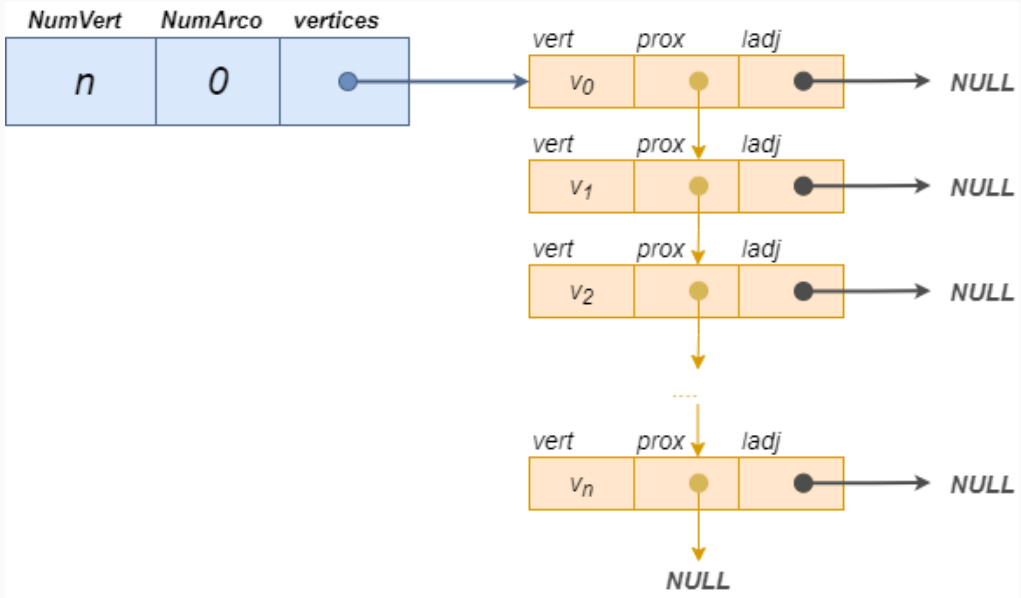


Figura 12: Lista de Adjacência para um grafo com n vértices

Lista de Adjacência - Implementação em C

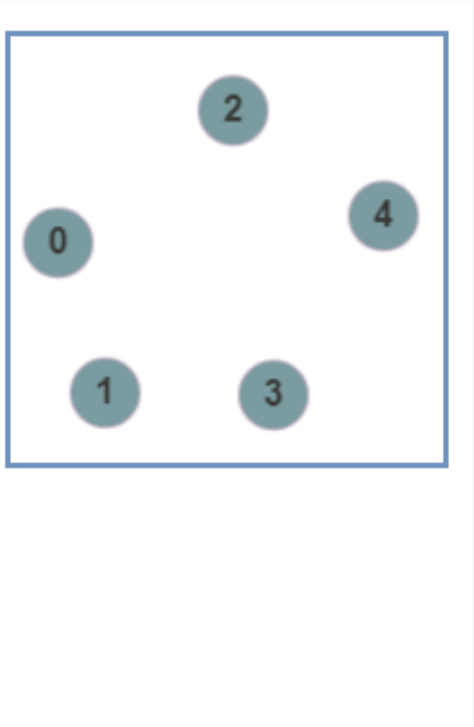
Criação de um grafo com N vértices:

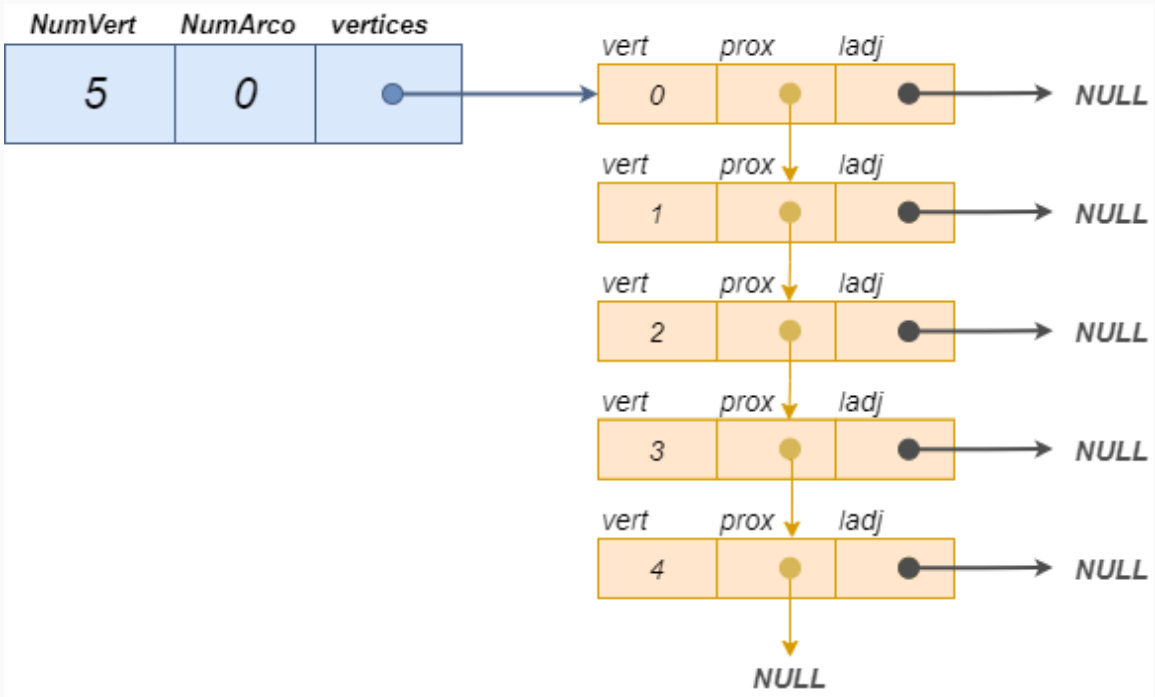
```
1  Grafo criarGrafo(int nVert) {
2    int v;
3    Grafo G;
4    G = (Grafo) malloc (sizeof (Grafo));
5    G->NumArco = 0;
6    G->NumVert = 0;
7    for (v = nVert-1; v >= 0; v--) {
8       G->vertices = inserirVertice(G->vertices,v);
9       G->NumVert++;
10   }
11   return G;
12 }
13
14 struct noVert* inserirVertice(struct noVert *ini, int num){
15   struct noVert* novoVertice;
16   novoVertice = (struct noVert*) malloc (sizeof(struct noVert));
17   novoVertice->vert = num;
18   novoVertice->prox = ini;  // inserção no início da lista
19   novoVertice->ladj = NULL;
20   return novoVertice;
21 }
```

Lista de Adjacência - Implementação em C

```
1  int main(void) {
2   Grafo g;
3   g = criarGrafo(5);
4  }
```

Cada um dos vértices é inserido na lista:

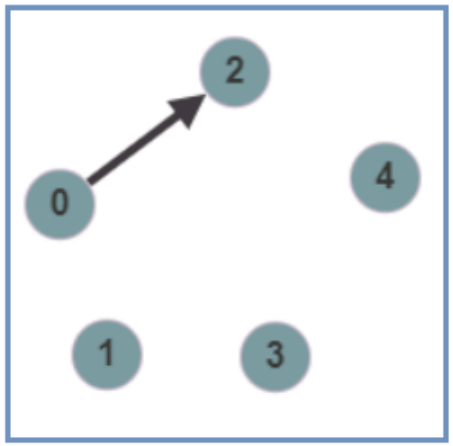


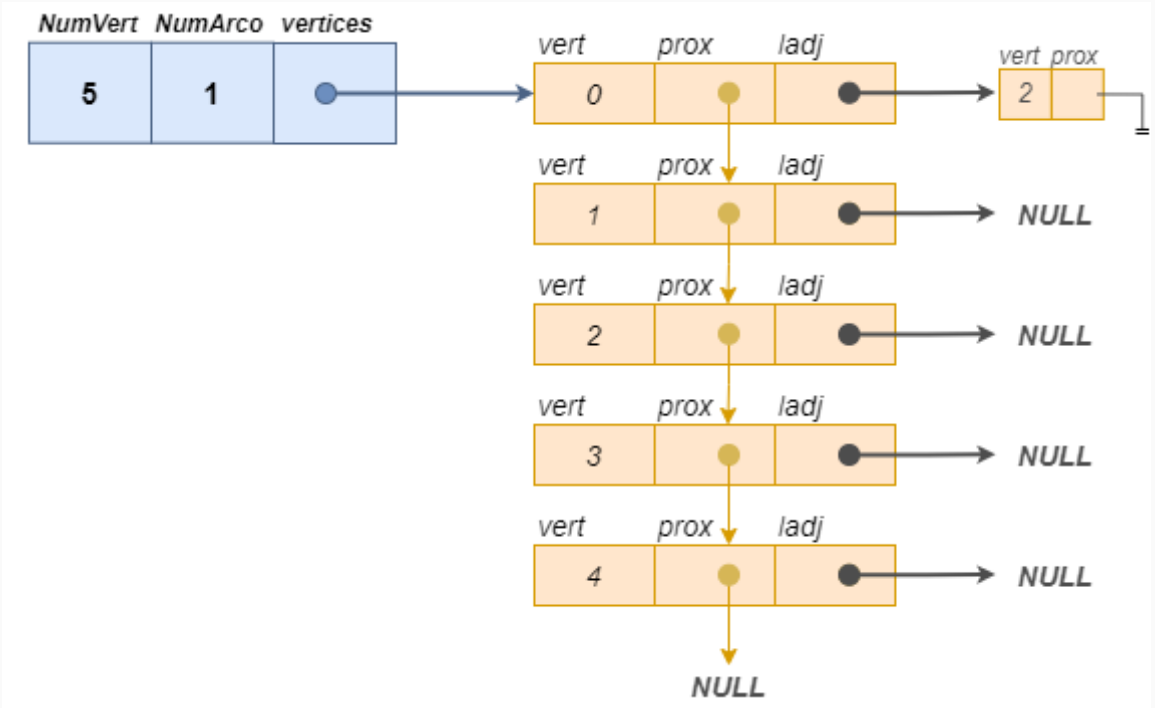


Lista de Adjacência - Implementação em C

Para a **inserção** de um novo Arco entre dois vértices deve-se:

- encontrar o vértice de origem na lista de vértices;
- verificar se o arco já existe;
- caso não exista, inserir o novo arco e atualizar o campo `NumArco` do grafo.





Lista de Adjacência - Implementação em C

Inserção de um novo Arco:

```
1  void inserirArco(Grafo G, int v1, int v2){
2    struct noVert *v;
3    struct noAdj *z;
4    if (G == NULL) return;
5    for (v = G->vertices; v != NULL; v = v->prox)
6       if (v->vert == v1) { // achou o vértice para inserir a adjacência
7          for(z = v->ladj; z!= NULL; z = z->prox)
8              if (z->vert == v2) return;  // o arco já existe, retornar!
9          v->ladj = inserirAdjacencia(v2,v->ladj);
10         G->NumArco++;
11      }
12 }
13
14 struct noAdj* inserirAdjacencia(int vdest, struct noAdj *l){
15   struct noAdj *novo = (struct noAdj*) malloc (sizeof (struct noAdj));
16   novo->vert = vdest;
17   novo->prox = l;     // inserção no início da lista
18   return novo;
19 }
```

Lista de Adjacência - Implementação em C

Inserção dos arcos no grafo: ordem arbitrária (entram sempre no início das listas `ladj`)

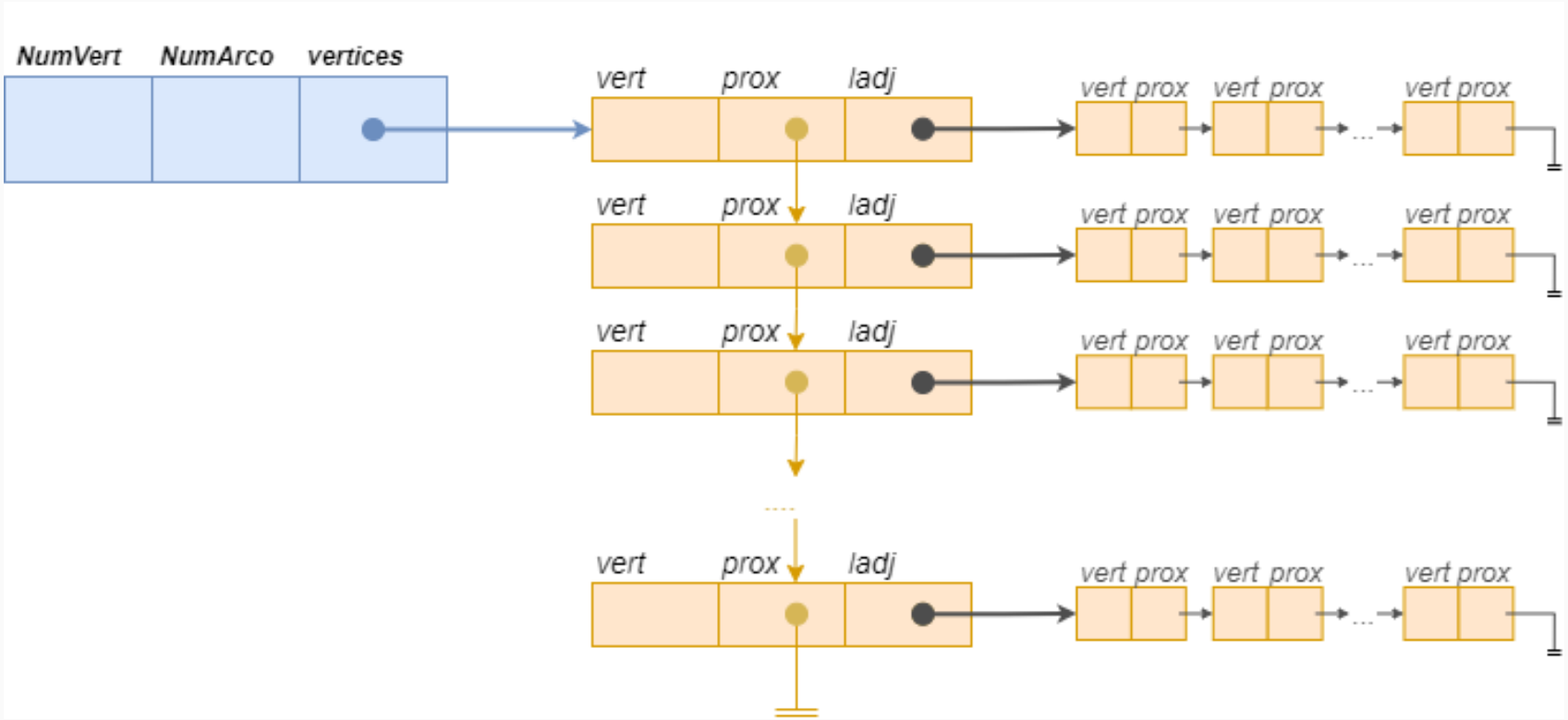


Figura 13: Visão geral da Lista de Adjacência após a inserção dos arcos, considerando um grafo denso

Lista de Adjacência - Implementação em C

Inserção de um novo Vértice:

```
1  void inserirNovoVertice(Grafo G, int nv){
2    G->vertices = inserirVertice(G->vertices,nv);
3    G->NumVert++;
4  }
5
6  struct noVert* inserirVertice(struct noVert *ini, int num){
7    struct noVert* novoVertice;
8    novoVertice = (struct noVert*) malloc (sizeof(struct noVert));
9    novoVertice->vert = num;
10   novoVertice->prox = ini;  // inserção no início da lista
11   novoVertice->ladj = NULL;
12   return novoVertice;
13 }
```

Lista de Adjacência - Implementação em C

Inserção de um novo vértice: ordem arbitrária (entrada no início da lista `vertices`)

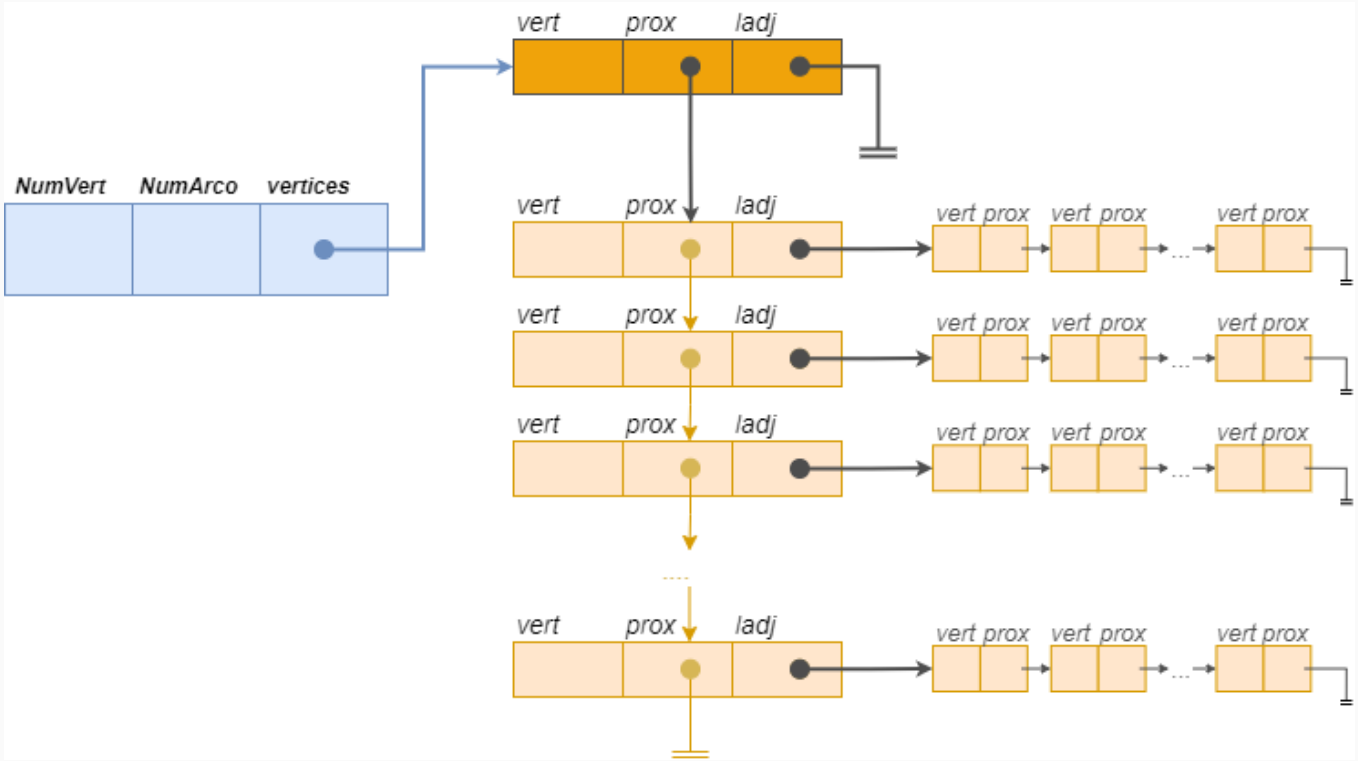
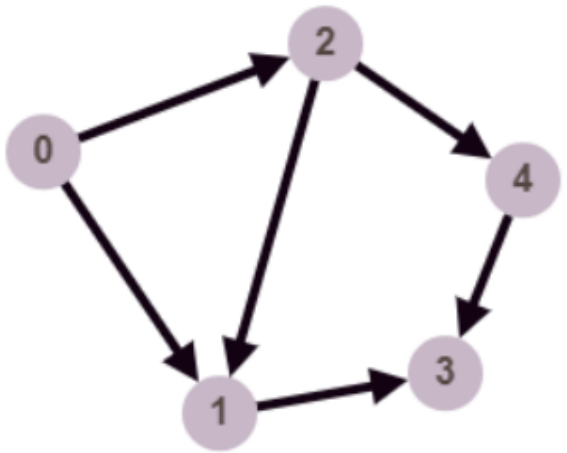


Figura 14: Visão geral da Lista de Adjacência após a inserção de um novo vértice

Lista de Adjacência - Implementação em C

Percurso completo no Grafo:

```
1  void imprimirListaAdj(Grafo G) {
2    struct noVert *nv;
3    struct noAdj *na;
4    if (G == NULL) return;
5    printf("\n\nLista de Adjacencias:");
6    for (nv = G->vertices; nv!=NULL; nv = nv->prox) {
7       printf("\nVertice %d:",nv->vert);
8       for (na = nv->ladj; na != NULL; na = na->prox)
9           printf("(%d)",na->vert);
10   }
11 }
```




```
Lista de Adjacencias:
Vertice 0:(2)(1)
Vertice 1:(3)
Vertice 2:(4)(1)
Vertice 3:
Vertice 4:(3)
```

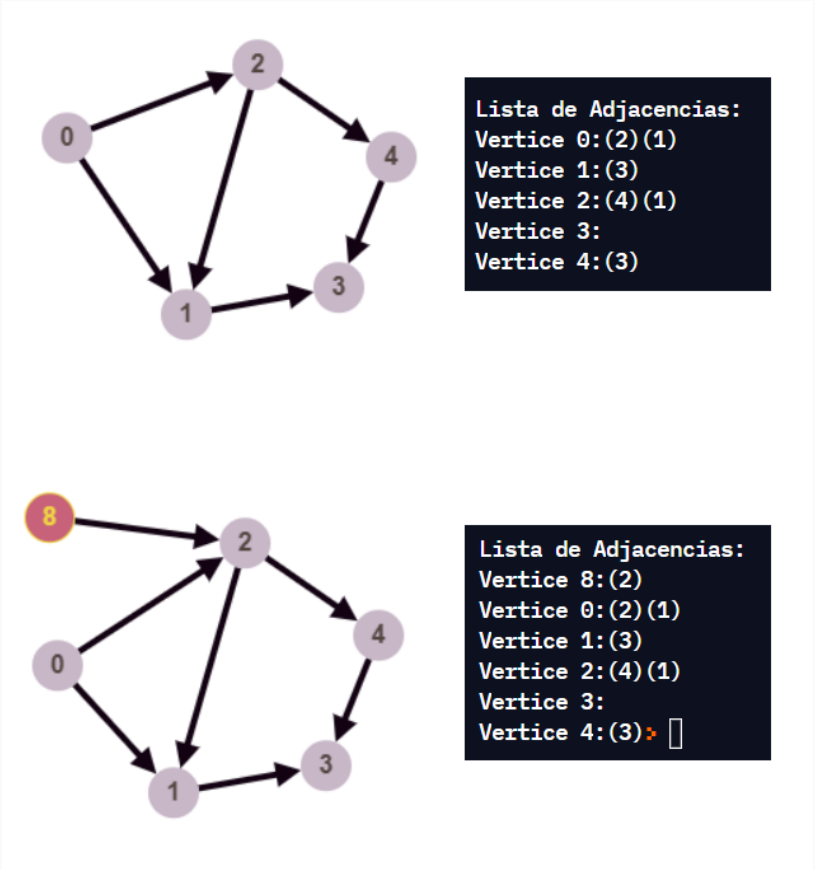
Lista de Adjacência - Implementação em C

Exemplo de execução:

```c
int main(void) {
  Grafo g;
  g = criarGrafo(5);

  inserirArco(g,0,1);
  inserirArco(g,0,2);
  inserirArco(g,2,1);
  inserirArco(g,2,4);
  inserirArco(g,4,3);
  inserirArco(g,1,3);
  imprimirListaAdj(g);

  inserirNovoVertice(g,8);
  inserirArco(g,8,2);
  imprimirListaAdj(g);
}
```



Lista de Adjacência - Implementação em C

Navegação pelo grafo usando recursão:

```
1 void imprimirListaAdjRec(struct noVert *nv){
2    struct noAdj *na;
3    if (nv == NULL) return;
4    printf("\nVertice %d:",nv->vert);
5    for (na = nv->ladj; na != NULL; na = na->prox)
6           printf("(%d)",na->vert);
7    imprimirListaAdjRec(nv->prox);
8 }
```

Lista de Adjacência - Implementação em C

Exercício: Completar a implementação de Lista de Adjacência para um **grafo direcionado** com as funções:

- Remover um Arco (v,u)
- Remover um Vértice v
- Verificar a existência de um Arco (v,u)
- Calcular o grau de um vértice v
- Destruir o Grafo

```
1 Grafo criarGrafo(int nVert);
2 void inserirArco(Grafo G, int v1, int v2);
3 void inserirNovoVertice(Grafo G, int nv);
4 void imprimirListaAdj(Grafo G);
```

Referência e Material Extra

- Celes, W.; Cerqueira, R.; Rangel, J.L. *Introdução a Estruturas de Dados com Técnicas de Programação em C*. 2a. ed. Elsevier, 2016.

* Backes, A. *Programação Descomplicada - Estruturas de Dados*: Vídeo-aulas de 56 a 61. `https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/`