Grafos: definições básicas

▷ $G = (V, E)$: grafo (simples)

(a) V: conjunto de *vértices*

(b) E: conjunto de *arestas*; $E \subset \binom{V}{2}$

Códigos!

1. Sedgewick, Algorithms in C, Part 5

2. Knuth, The Stanford GraphBase

Programa 17.1, Interface para o TAD Graph

```
typedef struct { int v; int w; } Edge;
Edge EDGE(int, int);

typedef struct graph *Graph;
Graph GRAPHinit(int);
 void GRAPHinsertE(Graph, Edge);
 void GRAPHremoveE(Graph, Edge);
  int GRAPHedges(Edge [], Graph G);
Graph GRAPHcopy(Graph);
 void GRAPHdestroy(Graph);

 void GRAPHshow(Graph);
Graph GRAPHrand(int, int);

Graph GRAPHscan(int, int);
```

Programa 17.2, Um cliente simples

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

main(int argc, char *argv[])
  { int V = atoi(argv[1]), E = atoi(argv[2]);
    Graph G = GRAPHrand(V, E);
    if (V < 20)
         GRAPHshow(G);
    else {
      printf("%d vertices, %d edges, ", V, E);
      printf("Too big!\n");
    }
  }
```

Prog. 17.3, TAD Graph, Implemenção, matriz de adjacências

```
#include <stdio.h>
#include <stdlib.h>
#include "GRAPH.h"
struct graph { int V; int E; int **adj; };
```

```
/* prog17.4 */
int **MATRIXint(int r, int c, int val)
  { int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i = 0; i < r; i++)
      t[i] = malloc(c * sizeof(int));
    for (i = 0; i < r; i++)
      for (j = 0; j < c; j++)
        t[i][j] = val;
    return t;
  }
```

```
/* prog17.3.c */
Edge EDGE(int v, int w)
{
  Edge p;
  p.v = v; p.w = w;
  return p;
}

Graph GRAPHinit(int V)
  { Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = MATRIXint(V, V, 0);
    return G;
  }
```

```
void GRAPHinsertE(Graph G, Edge e)
  { int v = e.v, w = e.w;
  if ((G->adj[v][w] == 0) && (v != w))
    {
      G->E++;
      G->adj[v][w] = 1;
      G->adj[w][v] = 1;
    }
  }
```

```
void GRAPHremoveE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    if (G->adj[v][w] == 1) G->E--;
    G->adj[v][w] = 0;
    G->adj[w][v] = 0;
  }
int GRAPHedges(Edge a[], Graph G)
  { int v, w, E = 0;
    for (v = 0; v < G->V; v++)
      for (w = v+1; w < G->V; w++)
        if (G->adj[v][w] == 1)
          a[E++] = EDGE(v, w);
    return E;
  }
```

```
/* prog17.5 */

void GRAPHshow(Graph G)
  { int i, j;
    printf("%d vertices, %d edges\n", G->V, G->E);
    for (i = 0; i < G->V; i++)
      {
        printf("%2d:", i);
        for (j = 0; j < G->V; j++)
          if (G->adj[i][j] == 1) printf(" %2d", j);
        printf("\n");
      }
  }
```

```
/* prog17.7.c */

int randV(Graph G)
  { return G->V * (rand() / (RAND_MAX + 1.0)); }
Graph GRAPHrand(int V, int E)
  { Graph G = GRAPHinit(V);
    while (G->E < E)
      GRAPHinsertE(G, EDGE(randV(G), randV(G)));
    return G;
  }
```

Prog. 17.6, TAD Graph, Implemenção, listas de adjacências

```
#include <stdio.h>
#include <stdlib.h>
#include "GRAPH.h"
typedef struct node *link;
struct node { int v; link next; };
struct graph { int V; int E; link *adj; };

Edge EDGE(int v, int w)
{
  Edge p;
  p.v = v; p.w = w;
  return p;
}
```

```
int GRAPHadj(Graph G, int v, int w)
  {
    link x;
    for (x = G->adj[v]; x != NULL; x = x->next)
      if (x->v == w)
        return 1;
    return 0;
  }
```

```
link NEW(int v, link next)
  { link x = malloc(sizeof *x);
    x->v = v; x->next = next;
    return x;
  }
Graph GRAPHinit(int V)
  { int v;
    Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = malloc(V*sizeof(link));
    for (v = 0; v < V; v++) G->adj[v] = NULL;
    return G;
  }
```

```
void GRAPHinsertE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    if (!GRAPHadj(G, v, w) && (v != w))
      {
        G->adj[v] = NEW(w, G->adj[v]);
        G->adj[w] = NEW(v, G->adj[w]);
        G->E++;
      }
  }
int GRAPHedges(Edge a[], Graph G)
  { int v, E = 0; link t;
    for (v = 0; v < G->V; v++)
      for (t = G->adj[v]; t != NULL; t = t->next)
        if (v < t->v) a[E++] = EDGE(v, t->v);
    return E;
  }
```

Grafos a partir de pares de símbolos

```
#include <stdio.h>
#include "GRAPH.h"
#include "ST.h"
Graph GRAPHscan(int Vmax, int Emax)
  { char v[100], w[100];
    Graph G = GRAPHinit(Vmax);
    STinit();
    while (scanf("%99s %99s", v, w) == 2)
      GRAPHinsertE(G, EDGE(STindex(v), STindex(w)));
    return G;
  }
```

Um cliente simples

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

main(int argc, char *argv[])
  { int V = atoi(argv[1]), E = atoi(argv[2]);
    Graph G = GRAPHscan(V, E);
    if (V < 20)
         GRAPHshow(G);
    else {
      printf("%d vertices, %d edges, ", V, E);
      printf("Too big!\n");
    }
  }
```

Interface da tabela de símbolos

```
void STinit();
int STindex(char *);
```

A seguir, consideramos uma adaptação da estrutura de TSTs, *ternary search tries* (existenciais). Você pode usar qualquer implementação para esta tabela de símbolos, mas as TSTs são muito eficientes. Para detalhes, veja Sedgewick, Partes 1–4, Capítulo 15 (Seção 15.4). [Não discutiremos essas estruturas de dados.]

Implementação da tabela de símbolos

TST: *(existence) ternary search tries*

```
#include <stdlib.h>
typedef struct STnode* link;
struct STnode { int index, d; link l, m, r; };
static link head;
static int val, N;
void STinit()
  { head = NULL; N = 0; }
```

```
link stNEW(int d)
  { link x = malloc(sizeof *x);
    x->index = -1; x->d = d;
    x->l = NULL; x->m = NULL; x->r = NULL;
    return x;
  }
```

```
link indexR(link h, char* v, int w)
  { int i = v[w];
    if (h == NULL) h = stNEW(i);
    if (i == 0)
      {
        if (h->index == -1) h->index = N++;
        val = h->index;
        return h;
      }
    if (i < h->d) h->l = indexR(h->l, v, w);
    if (i == h->d) h->m = indexR(h->m, v, w+1);
    if (i > h->d) h->r = indexR(h->r, v, w);
    return h;
  }

int STindex(char* key)
  { head = indexR(head, key, 0); return val; }
```