IX. GRAFURI.

Elemente de teoria grafurilor: definiții şi terminologie.

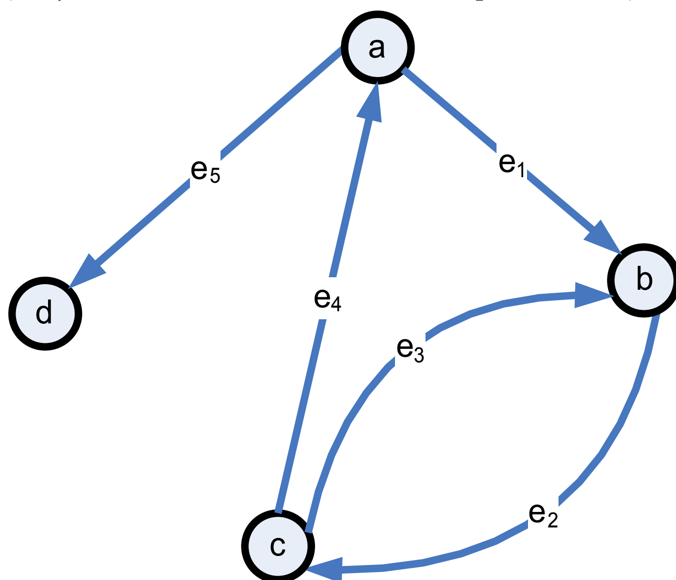
Relațiile între obiecte sunt descrise în mod natural prin intermediul grafurilor. Interconexiunea elementelor într-un circuit sau o rețea de drumuri pot fi modelate prin grafuri.

Un graf este un obiect matematic format din două mulțimi: **G = (V, E)**, în care **V** este o *mulțime de vârfuri*, iar **E** – o *mulțime de muchii* (sau *arce*) care conectează vârfurile. Mulțimea muchiilor **E** este o relație binară pe **V**, adică:

$$E \subseteq V \times V\ ,\ E = \{(u, v)\ |\ u \in V,\ v \in V\}$$

muchiile sunt reprezentate prin perechi de vârfuri conectate (obiecte aflate în relație).

Grafurile pot fi *grafuri orientate* (sau *digrafuri*) sau *grafuri neorientate*.
Un *graf orientat* este format din *arce* **(u, v)**, care reprezintă *perechi ordonate de vârfuri*. Vârful **u** este *originea* arcului **(u, v)** , iar **v** este *destinația* arcului. Arcele **(u, v)** şi **(v, u)** sunt distincte (relația *nu este simetrică*, **u R v** nu implică **v R u**).



G=(V,E)
V={a, b, c, d}
E=$\{e_1, e_2, e_3, e_4, e_5\}$
e_1**=(a,b)** , e_2**=(b,c)** , e_3**=(c,b)** , e_4**=(c,a)** , e_5**=(a,d)**

Două arce sunt *arce multiple* (sau *paralele*) dacă au aceeaşi origine şi aceeaşi destinație. Un *graf simplu* nu conține arce multiple şi nici bucle.

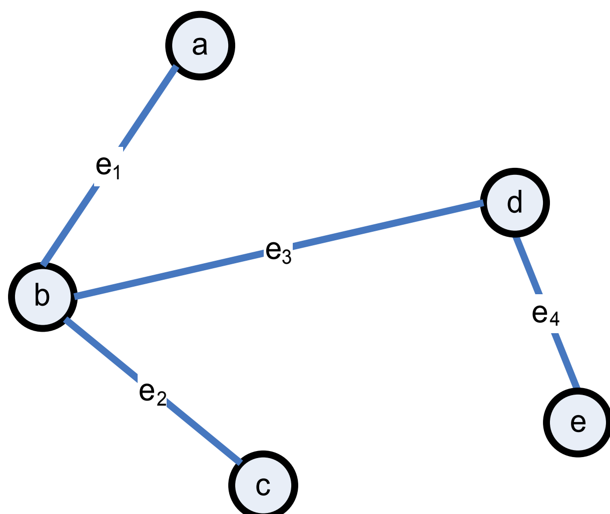
Un *graf neorientat* este format din *muchii* **{u, v}**, **u ≠ v**, care sunt *perechi neordonate de vârfuri*. Muchiile **{u, v}** şi **{v, u}** sunt identice (relația *este simetrică*, **u R v ⇒ v R u**).

Muchia **{u, v}** este *incidentă vârfurilor* **u** şi **v**. Extremitățile **u** şi **v** ale muchiei **{u, v}** sunt *adiacente*. Intr-un *graf complet* oricare două vârfuri sunt adiacente.
Arcul **(u, v)** este *incident din* (*pleacă din*) **u** şi este *incident în* (*intră în*) **v**.

Gradul unui vârf **deg(v)**, într-un graf neorientat, reprezintă numărul muchiilor incidente în acel vârf. Un *vârf izolat* este un vârf având gradul 0.

Gradul interior (sau *gradul de intrare*) **indeg(v)** a unui vârf a unui graf orientat este egal cu numărul arcelor care intră în acel vârf. Un vârf cu gradul interior 0 este un *vârf de ieşire* (*sursă*). *Gradul exterior* (sau *gradul de ieşire*) **outdeg(v)** a unui vârf a unui graf orientat este egal cu numărul arcelor care ies din acel vârf. Un vârf cu gradul exterior 0 este un *vârf de intrare* (*puț*).

Ordinul unui graf este egal cu numărul de vârfuri al grafului: **n = |V|.**
Dimensiunea unui graf este egală cu numărul de muchii (arce) ale grafului: **m = |E|.**



$G=(V,E)$
$V=\{a, b, c, d, e\}$
$E=\{e_1, e_2, e_3, e_4\}$
$E1=(a,b), e2=(b,c), e3=(b,d), e4=(d,e)$

Un *drum* de lungime p între două vârfuri a şi b este o succesiune de vârfuri $v_0, v_1, ..., v_p$, cu $v_0=a$ şi $v_p=b$ şi $\{v_{i-1}, v_i\} \in E$ pentru $i=1:p$. Drumul conţine atât muchiile $(v_0,v_1), ..., (v_{p-1}, v_p)$ cât şi vârfurile $v_0,...,v_p$. Dacă există un drum de la a la b, atunci b este *accesibil* din a $(a \rightarrow b)$.

Un *drum elementar* într-un graf orientat are toate vârfurile de pe el distincte. Un *ciclu* este un drum $(v_0, v_1, ..., v_p)$ în care $v_0 = v_p$. O *buclă* (a,a) este un ciclu de lungime 1. Un *ciclu elementar* are toate vârfurile distincte (exceptând vârful de plecare) şi nu conţine bucle. Un *graf aciclic* (*pădure*) nu conţine cicluri.
Un graf neorientat este *conex*, dacă oricare două vârfuri pot fi unite printr-un drum. Un graf conex aciclic este *arbore liber*.

Componentele conexe ale unui graf neorientat sunt clasele de echivalenţă ale vârfurilor prin relaţia "*este accesibil din*". Un graf neorientat este conex, dacă are o singură componentă conexă.
Un graf orientat este *tare conex* dacă, pentru oricare două vârfuri a şi b, a este accesibil din b şi b este accesibil din a.
Graful $G'=(V',E') \subseteq G$ este *subgraf* al grafului $G=(V,E)$, dacă $V' \subseteq V$ **şi** $E' \subseteq E$.
Subgraful indus de mulţimea de vârfuri $V' \subseteq V$ este graful: $G'=(V',E')$ în care:

$$E' = \{(u,v) \in E : u,v \in V'\}$$

Graful $G'=(V,E')$ este un *graf parţial* (sau *subgraf de acoperire*) al grafului $G=(V,E)$ dacă $E' \subseteq E$

Intr-un *graf bipartit*, mulţimea vârfurilor V se partiţionează $V = V_1 \vee V_2$, $V_1 \wedge V_2 = \varnothing$ astfel încât $(u, v) \in E \Rightarrow u \in V_1, v \in V_2$ sau $u \in V_1, v \in V_2$.
Pentru un graf neorientat cu m muchii:

- $0 \le E \le \binom{n}{2} = \frac{n \cdot (n-1)}{2} \in O(n^2)$
- $\sum_{v \in V} deg(v) = 2m$.

Intr-un graf orientat cu m arce

- $0 \le E \le n^2$
- $\sum_{v \in V} in\ deg(v) = \sum_{v \in V} out\ deg(v) = m$

Într-un *graf dens* $E \in \Theta(|V|^2)$, iar într-un *graf rar* $E \in \Theta(|V|)$

Oricărei muchii din graf i se asociază un *atribut* numit *ponderea* muchiei. În general, aceasta este o valoare reală şi pozitivă. Intern o muchie este referită printr-un indice.

Intr-un graf **G**, următoarele condiţii sunt echivalente:

1. **G** este conex aciclic
2. **G** este *aciclic maximal* (prin adăugarea unei muchii apare un ciclu)
3. **G** este *conex minimal* (prin ştergerea unei muchii graful îşi pierde conexitatea).

Intr-un graf **G** neorientat, cu **n** vârfuri şi **m** muchii:

1. dacă **G** este *conex*, $m \geq n - 1$
2. dacă **G** este *arbore liber*, $m = n - 1$
3. dacă **G** este *pădure*, $m < n - 1$

Operaţii asociate vârfurilor grafului.

Oricărui vârf i se asociază o *etichetă*; aceasta este de obicei un şir de caractere. Referirea la un vârf prin eticheta sa se face numai la creerea grafului, prin adăugarea de vârfuri şi la afişarea rezultatelor. Cheia poate fi modificată pe parcurs.

Intern, un vârf în graf este identificat printr-un indice întreg – *cheie,* reprezentând poziţia vârfului în tabelul de vârfuri al grafului. Acest atribut este setat de constructorul grafului şi poate fi numai interogat.

Pentru simplificarea operaţiilor de traversare vom asocia fiecărui vârf:

- *o etichetă* (şir de caractere).
- *o culoare* (cu una din valorile **ALB**, **GRI** sau **NEGRU** – întregii 0,1,2)
- *o distanţă* (faţă de un vârf reper)
- *un predecesor*

La traversarea unui graf unui vârf i se ataşează şi alte atribute întregi precum: momentul descoperirii vârfului – *start* şi momentul terminării prelucrării vârfului – *stop*.

```
// interfata TAD Varf (fisierul Varf.h)

struct varf;
typedef struct varf *Varf;
Varf   V_New(char *etv);          // constructor
char*  V_GetEt(Varf u);           // da eticheta varfului u
int    V_GetCol(Varf u);          // da culoarea varfului u
double V_GetDist(Varf u);         // da distanta varfului u
Varf   V_GetPred(Varf u);         // da varful predecesor varfului u
void   V_SetEt(Varf u, char *et);//schimba eticheta varfului u
void   V_SetCol(Varf u, int  c); //schimba culoare varfului u
void   V_SetDist(Varf u,double d);  //schimba distanta varfului u
void   V_SetPred(Varf u, Varf p);//schimba predecesorul varfului u
```

Operaţii asociate arcelor grafului..

Unui arc i se asociază următoarele atribute:

- o etichetă (un şir de caractere)
- o cheie (o valoare întreagă) reprezentând poziţia arcului în tabloul de arce
- capetele arcului (de tip **Varf**)
- ponderea arcului (o valoare reală)

```
// interfata TAD Arc (fisierul Arc.h)

struct arc;
typedef struct arc *Arc;
Arc A_New(Varf u1, Varf u2, double pond, char *et); //constructor
Varf V1(Arc a);              //selectare varf origine arc
Varf V2(Arc a);              //selectare varf destinatie arc
```

```
Varf Opus(Arc a, Varf u);   //selectie varf opus lui u in arcul a
double A_GetCost(Arc a);    //extrage ponderea arcului
char*  A_GetEt(Arc a);      //extrage eticheta arcului
```

TAD Graf

Vom presupune limitări pentru numărul maxim de vârfuri – **N** şi de muchii **M**.

```
#define N 100
#define M 1000
```

Vom considera separat grafurile orientate şi cele neorientate. Unificarea conduce la complicaţii inutile. De asemenea, grafurile neponderate vor fi tratate ca grafuri ponderate, cu costul muchiilor **1**.

```
// interfața TAD Graf (fisierul Graf.h)

struct graf;
typedef struct graf *Graf;
typedef enum{ALB,GRI,NEGRU} color;

Graf G_New();    //constructor - initializare graf vid

//operatii relative la varfuri
void  G_AddV (Graf G, char *etv);//adauga un varf cu eticheta data
void  G_DelV (Graf G, Varf u);   //sterge un varf specificat
int   G_IsV  (Graf G, Varf u);   //test existenta varf in graf
Varf  G_GetVK(Graf G, int ch);   //varful avand cheia ch
int   G_GetKV(Graf G, Varf u);   //da cheia varfului

//operatii relative la muchii
void  G_AddA (Graf G, Varf u, Varf v, double pond, char *eta);
void  G_DelA(Graf G, Varf u, Varf v);
void  G_DelA(Graf G, Arc a);
int   G_IsA (Graf G, Varf u, Varf v);
Arc   G_GetArc(Graf G, Varf u, Varf v);
Arc   G_GetAK(Graf G, int ch);   //arcul avand cheia ch
int   G_GetKA(Graf G,  Arc a);   //da cheia arcului

//operatii generale
int  G_Size(Graf G);             //ordin graf
int  G_Dim(Graf G);              //dimensiune graf
int  G_Deg(Graf G, Varf u);      //grad varf
int  G_OutDeg(Graf G, Varf u);   //grad exterior varf(graf orientat)
int  G_InDeg(Graf G, Varf u);    //grad interior varf(graf orientat)
```

Iterarea vârfurilor şi muchiilor unui graf.

Pentru a realiza enumerarea vârfurilor sau muchiilor unui graf printr-o iterare de tipul instrucţiunii **for**, vom preciza: primul element din enumerare, condiţia de terminare şi modul de trecere la următorul element.

Astfel iterarea tuturor vâfurilor unui graf, de tipul: **for(v∈V(G))** se reprezintă prin:

```
Varf v;
for(v=primV(G); !dultimV(G); v=avansV(G,v))
   if(G_IsV(G,v)){ ...}
```

Iterarea tuturor muchiilor unui graf de tipul: **for(e∈E(G))** reprezintă prin:

```
Arc e;
for(e=primA(G); !dultimA(G); e=avansA(G,e))
  if(G_IsA(G,e)){...}
```

Iterarea *vârfurilor adiacente* unui vârf **u**, de tipul: **for(v∈Vecini(u))** se reprezintă prin:

```
Varf u,v;
for(v=primVad(G,u); !dultimVad(G,u); v=avansVad(G,u,v))
   if(G_IsV(G,v){...}
```

Iterarea pe *muchiile incidente* unui vârf **u** se reprezintă prin:

```
Arc a;
for(a=primAinc(G,u);!dultAinc(G,u); a=avansAinc(G,a,u))
  if(G_IsA(G,a)){...}
```

cu operațiile primitive:

Arc primAinc(Graf G, Varf v) – prima muchie incidentă unui vârf **v**
Arc avansAinc(Graf G, Arc a, Varf v) – următoarea muchie incidentă vârfului **v**, după muchia **a**
int dultAinc(Graf G, Varf v) – 1 dacă s-a folosit şi ultima muchie incidentă vârfului **v**

Implementarea operațiilor independente de reprezentarea grafurilor.

```
// implementare TAD Varf (fisierul Varf.c)

struct varf{
   char   *etv; //eticheta
   int    col;  //culoare
   double dist; //distanta
   Varf   pred; //predecesor
};

Varf  V_New(char *etv){
   Varf v = (Varf)malloc(sizeof(struct varf));
   v->etv = strdup(etv);
   v->col = 0;
   v->dist = 0.;
   v->pred = NULL;
   return v;
}

char*  V_GetEt(Varf u)  { return u->etv;}
int    V_GetCol(Varf u) { return u->col;}
double V_GetDist(Varf u){ return u->dist;}
Varf   V_GetPred(Varf u){ return u->pred;}
void   V_SetEt(Varf u, char *et){u->etv = strdup(et);}
void   V_SetCol(Varf u,int  c) {u->col = c;}
void   V_SetDist(Varf u, double d){u->dist = d;}
void   V_SetPred(Varf u, Varf p){u->pred = p;}

// implementare TAD Arc (fisierul Arc.c)

struct arc{
   char   *eta; //eticheta muchie
   Varf   v1;   //un capat al muchiei
   Varf   v2;   //celalalt capat al muchiei
   double cost; //ponderea muchiei
};

Arc A_New(Varf u1, Varf u2, double pond, char *et){
   Arc a = (Arc)malloc(sizeof(struct arc));
   a->v1 = u1;
   a->v2 = u2;
   a->eta = strdup(et);
   a->cost = pond;
```

```
}
Varf V1(Arc a){
   if(a==NULL) return NULL;
   return a->v1;
}
Varf V2(Arc a){
   if(a==NULL) return NULL;
   return a->v2;
}
Varf Opus(Arc a, Varf u){
   assert(V1(a)==u || V2(a)==u);
   if(V1(a)==u) return V2(a);
   return V1(a);
}
double A_GetCost(Arc a){return a->cost;}
char*  A_GetEt(Arc a){return a->eta;}
```

Structura care descrie graful va conține:

- **n** = primul vârf liber
- **m** = primul arc liber

Din motive de eficiență, mulțimea vârfurilor va fi reprezentată printr-un tablou de pointeri la structuri vârfuri. În acest tablou (alocat de dimensiune maximă **N**) vor exista şi poziții cu valoarea **NULL**, corespunzător vârfurilor nealocate încă sau a celor care au fost şterse. Adăugara unui vârf leagă prima poziție liberă din tabloul de vârfuri **V** la structura vârf. Ştergerea unui vârf eliberează memoria ocupată de structura vârf şi face pointerul corespunzător **NULL**.

Mulțimea muchiilor este reprezentată printr.un tablou de pointeri la structuri arce (de dimensiune maximă **M**). Adăugara unui arc leagă prima poziție liberă din tabloul de arce **A**, la structura arc. Ştergerea unui arc eliberează memoria ocupată de structura arc şi face pointerul corespunzător **NULL**.

```
struct graf{
  int n;        // indice prim varf nealocat
  int m;        // indice prim arc nealocat
  Varf *V;      // tablou de pointeri la structuri varfuri
  Arc  *E;      // tablou de pointeri la structuri arce
    . . .       // parte dependenta de reprezentare
};
```

Urmează partea dependentă de modul de reprezentare: cu *matrice de adiacențe* sau cu *liste de adiacențe*, care va fi discutată după prezentarea *funcțiilor independente de reprezentare*.
Acestea sunt funcțiile generale: ordin şi dimensiune graf şi funcțiile relative la vârfuri.
Ordinul grafului se stabileşte numărând intrările nenule din tabloul **V** (care are goluri).

```
//functii independente de reprezentare.

//ordin graf(numar de noduri)
int G_Size(Graf G){
  int i, nn=0;
  for(i=0; i<G->n; i++)
    if(G->V[i]!=NULL) nn++;
  return nn;
}

//dimensiune graf(numar de muchii)
int G_Dim(Graf G){
  int i, na=0;
  for(i=0; i<G->m; i++)
```

```
    if(G->E[i]!=NULL) na++;
  return na;
}
```

Pentru adăugarea unui vârf cu etichetă precizată, se creează o structură cu atributele vârfului care se leagă în tabloul de vârfuri în prima poziție neocupată.

```
void G_AddV (Graf G, char *etv){
   int ch = G->n;              //cheia ce se asociaza varfului
   G->V[ch] = V_New(etv);      //creaza o structura varf si o leaga
                               //in tabloul de varfuri
   G->n++;                     //actualizare primul varf liber
}

int G_GetKV(Graf G, Varf u){
   return (u - G->V[0])/sizeof(Varf);
}

Varf G_GetVK()){ return G->V[k]); }
```

Implementarea grafurilor cu matrice de adiacențe.

Dacă folosim cheile asociate vârfurilor grafului (întregii **0, 1, …, n-1**) atunci graful se poate reprezenta printr-o matrice **MA** numită *matrice de adiacențe*, în care, în mod clasic:

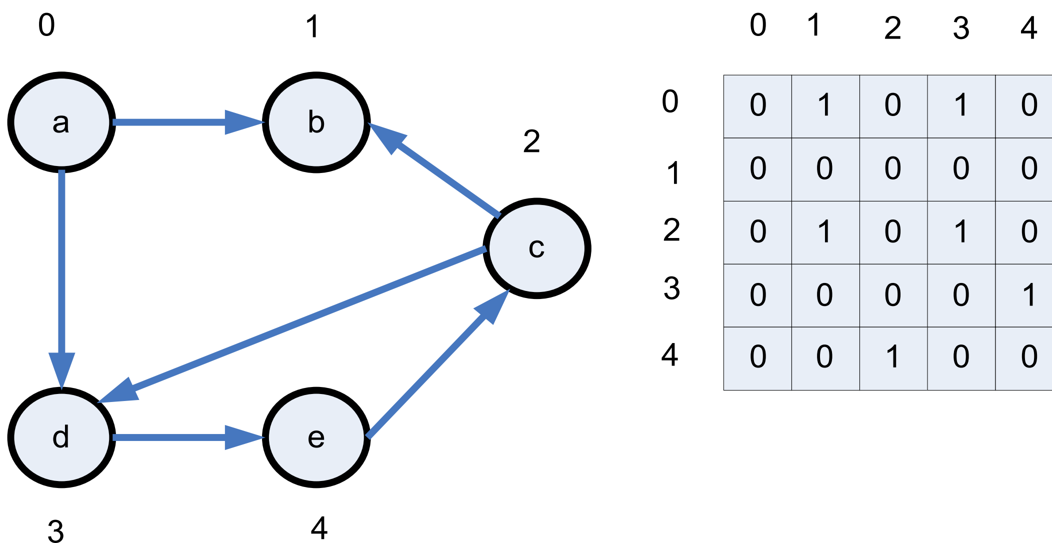
$$MA[i][j] = \begin{cases} 1 & \text{dacă} \quad (i, j) \in E \\ 0 & \text{dacă} \quad (i, j) \notin E \end{cases}$$

(**i** şi **j** sunt cheile a două vârfuri oarecare)

Pentru o referire mai simplă la muchiile grafului, în matricea de adiacențe, în locul marcării unei muchii prin 1 se va pune un pointer în vectorul de muchii.

Structura **Graf** va fi:

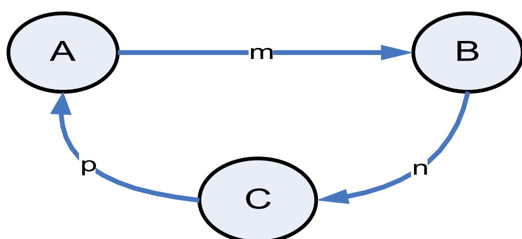
```
struct graf{
  int n;             // indice prim varf nealocat
  int m;             // indice prim arc nealocat
  Varf *V;           // tablou de pointeri la structuri varfuri
  Arc  *E;           // tablou de pointeri la structuri arce
  Arc  MA[N][N]  ;   // parte dependenta de reprezentare
};
```

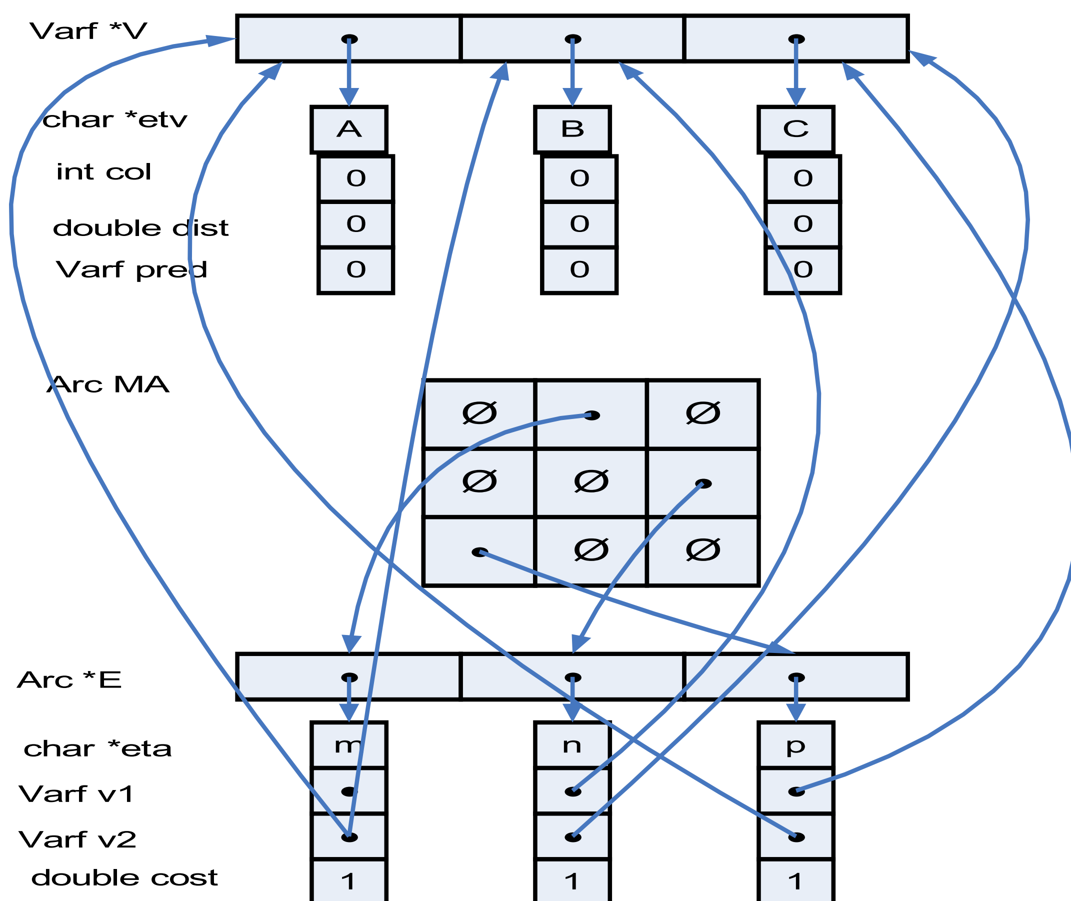


Reprezentarea prin matrice de adiacenţe asigură o simplitate a reprezentării, dar utilizează în mod ineficient memoria (consumul de memorie este **O(n²)**, iar matricea de adiacenţe este o matrice rară). Algoritmii dezvoltaţi au în general complexitate **O(n²)**.

Constructorul alocă memorie pentru tablourile de vârfuri **V** şi de muchii **E** şi pentru matricea de adiacenţe **MA**..Se iniţializează de asemenea, prima poziţie liberă din tablourile **V** şi **E** la 0.

```
Graf G_New (){
  Graf G=(Graf)malloc(sizeof(struct graf));
  int i,j;
  G->n = 0;
  G->m = 0;
  G->V = (Varf*)calloc(N, sizeof(Varf));
  G->E = (Arc*)calloc(M, sizeof(Arc));
  for(i=0; i<N; i++)
    for(j=0; j < N; j++)
      G->MA[i][j] = NULL;
  return G;
}
```





Ştergerea unui vârf eliberează memoria ocupată de atributele vârfului şi pune pe **NULL** pointerul la acel vârf (crează o poziţie liberă).

```
//sterge varful si muchiile incidente in varf
void G_DelV (Graf G, Varf v){
  int k = V_GetKV(G, v);
  assert(k < G->n && G->V[k]);
  int i;
  for(i=0; i<N; i++) //sterge referinte la arce incidente in v
    if(i!=k && G->V[k] && G->MA[i][k])
      G->MA[i][k]=G->MA[k][i]=NULL;
  G->V[k] =NULL; //sterge varful
  Varf u = v;
  free(u);
  //sterge muchiile incidente varfului
  for(i=0; i < M; i++)
    if(G->E[i].v1==v || G->E[i].v2==v){
      free(G->E[i]);
      G->E[i] = NULL;
    }
}

int G_IsV(Graf G, Varf u){
   int k = G_GetKV(G,u);
   if(k<0) return 0;
   return 1;
}

//adauga un arc; intoarce indicele arcului adaugat

void G_AddA(Graf G, Varf u1, Varf u2, int pond, char *eta){
   assert(G_IsV(G,u1) && G_IsV(G,u2));
   Arc a=A_New(u1,u2,pond,eta);
   G->E[G->m++]=a;
   G->MA[G_GetKV(G,u1)][G_GetKV(G,u2)]=a;
}

void G_DelA(Graf G, Varf u1, Varf u2){
   int ku1, ku2;
   assert(G_IsV(G,u1) && G_IsV(G,u2));
   ku1 = G_GetKV(G,u1);
   ku2 = G_GetKV(G,u2);
   Arc a=G_MA[ku1][ku2];
   free(a);
   G_MA[ku1][ku2]=NULL;
}

//test prezenta muchie intre varfurile u si v
void G_IsA(Graf G, Varf u1, Varf u2){
   if(!G_IsV(G,u1) || !G_IsV(G,u2)) return 0;
   return G_MA[G_GetKV(G,u1)][G_GetKV(G,u2)]!=NULL;
}
```

Pentru un graf neorientat gradul unui vârf **u** este egal cu numărul de referinţe nenule (numărul de muchii) din linia **u** (sau coloana **u**) a matricei de adiacenţe.

```
int G_Deg(Graf G, Varf u){
   int ku = G_GetKV(G,u);
   assert(G->V[ku]);
   int dg=0, i;
   for(i=0; i<G->n; i++)
```

```
      if(G->V[i] && G->MA[ku][i]!=NULL) dg++;
   return dg;
}
```

Pentru un graf orientat, gradul de ieşire `OutDeg(G, u)` al unui vârf **u** este numărul de referinţe nenule din linia **u** a matricei de adiacenţe, în timp ce gradul de intrare `InDeg(G, u)` este numărul de referinţe nenule din coloana **u** a matricei de adiacenţe

În reprezentarea cu matrice de adiacenţă, iterarea pe muchiile incidente unui vârf **v** se realizează prin:

```
Arc primAinc(Graf G, Varf v){
   Arc a;
   int i, kv;
   kv = G_GetKV(G,v);
   for(i=0; i<G->n; i++)
      if((a=G->MA[kv][i])!=NULL)
         return A_New(v, G_GetVK(G,i), A_GetCost(a), A_GetEt(a));
   return NULL;
}
```

```
Arc avansAinc(Graf G, Arc a, Varf v){
   int i, j;
   Varf u1, u2, u;
   u1 = V1(a);
   u2 = V2(a);
   assert(u1==v || u2==v);
   if(u1==v)  //mai simplu u = Opus(G,a,u)
      u = u2;
   else
      u = u1;
   j = G_GetKV(G, u);
   for(i=j+1; i<G->n; i++)
      if((e=G->MA[j][i])!=NULL)
         return A_New(u, G_GetVK(G,i), A_GetCost(a), A_GetEt(a));
   return NULL;
}
```

Reprezentarea grafurilor prin liste de adiacenţe.

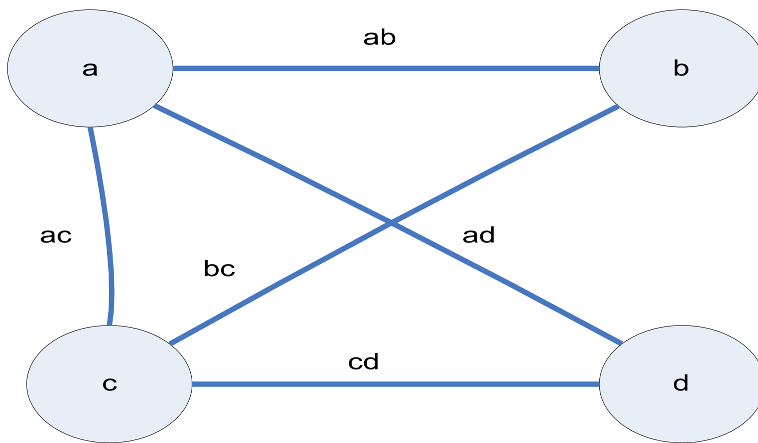
Pentru grafuri neorientate, fiecărui vârf i se asociază o listă de muchii incidente vârfului. Graful va fi reprezentat printr-un tablou de liste de muchii.
Pentru a itera mai uşor pe muchiile grafului, într-o implementare mai bună, elementele listei de adiacenţe a unui vârf nu sunt muchii, ci referinţe la muchiile incidente acelui vârf.

```
struct graf{
   int n;
   int m;
   Varf  *V;
   Arc   *E;
   Lista *LA;
};
```

Constructorul iniţializează listele de adiacenţe cu liste vide.

```
Graf G_New (){
  Graf G=(Graf)malloc(sizeof(struct graf));
  int i,j;
  G->n = 0;
  G->m = 0;
  for(i=0; i < N; i++) G->V[i] = NULL;
  for(i=0; i < M; i++) G->E[i] = NULL;
```

```
  for(i=0; i < N; i++) G->LA[i] = L_New();
  return G;
}
```
Astfel pentru graful neorientat:



avem reprezentarea:

LA

E

eta ab ac ad bc cd
V1
v2
pond

V

etv a b c d
dist
pred
col

```
//test prezenta arc intre u si v (Complexitate O(deg(u))
int G_IsA(Graf G, Varf u, Varf v){
  int ku = G_GetKV(G,u);
```

```
  int kv = G_GetKV(G,v);
  assert(G->V[ku] && G->V[kv]);
  List_Iter p;
  for(p=L_Begin(G->LA[ku]); p!=L_End(G->LA[ku]);p=L_Next(G->LA[ku],p))
    if(V2((Arc)L_Get(G->LA[ku],p)) ==v) return 1;
  return 0;
}
```

Gradul unui vârf dintr-un graf neorientat este lungimea listei de adiacențe a vârfului.

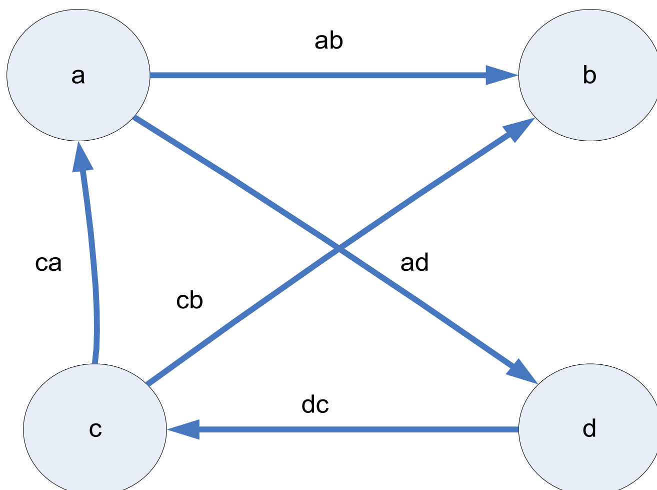
```
int G_Deg(Graf G, Varf u){ return L_Size(G->LA[G_GetKV(G,u)];}
```

Gradul de ieşire al unui vârf dintr-un graf orientat este numărul muchiilor având ca origine acel vârf, adică lungimea listei de adiacențe a succesorilor. Dacă am folosi numai această listă de adiacențe, determinarea gradului de intrare al vârfului, ar căuta ineficient în toate listele de adiacențe a muchiilor având ca vârf destinație vârful dat. Pentru a face eficientă această operație vom introduce pentru grafuri orientate o a doua listă de adiacențe, asociată predecesorilor unui vârf (pentru fiecare vârf lista muchiilor având ca destinație acel vârf).

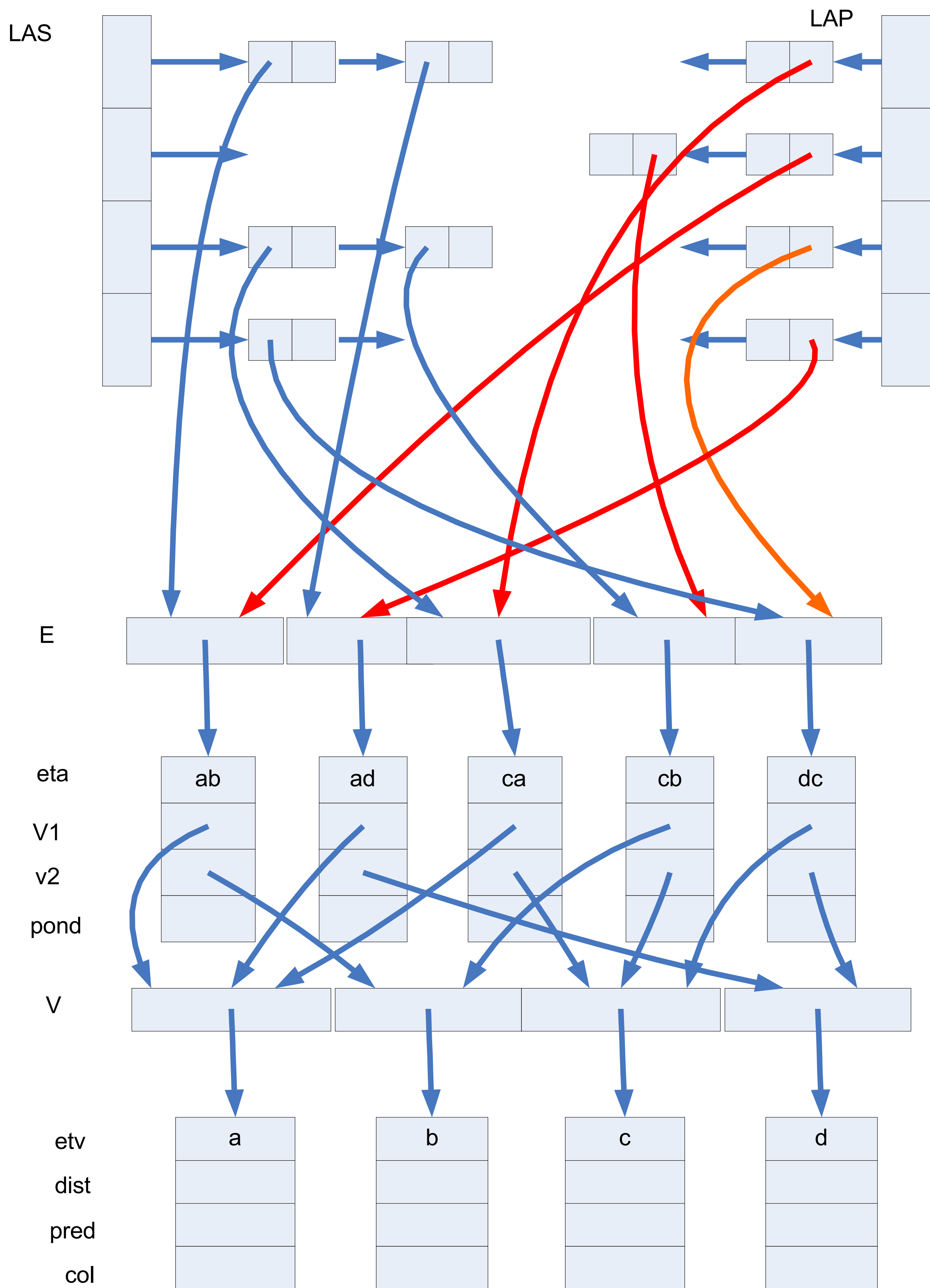
```
struct graf{
   int n;
   int m;
   Varf  *V;
   Arc   *E;
   Lista *LAS;
   Lista *LAP;
};
```

```
int OutDeg(Graf G, Varf u){ return L_Size(G->LAS[G_GetKV(G,u)];}
int InDeg(Graf G, Varf u) { return L_Size(G->LAP[G_GetKV(G,u)];}
```

Astfel pentru graful orientat:



Reprezentarea cu liste de adiacențe




```
void G_DelA(Graf G, Arc a){
   int ka=G_GetKA(G, a);
   Varf u, v;
   int ku, kv;
   u = V1(a);
   v = V2(a);
   ku = G_GetKV(G, u);
   kv = G_GetKV(G, v);
```

```
   List_Iter p;
   //cauta arcul in lista de adiacente a varfului u
   p = L_Find(G->LA[ku], a, comp);
   if(p) L_Modif(G_LA[ku], p, NULL);
   p = L_Find(G->LA[kvu], a, comp);
   if(p) L_Modif(G_LA[kv], p, NULL);
   free(G->E[ka];
   G->E[ka] = NULL;
}
```

Implementarea iteratorilor.

```
Varf primV(Graf G){ return G->V[0]; }
int  dultimV(Graf G, Varf v){ return v >= G->V[G->n]; }
Varf avansV(Graf G, Varf v) { return ++v; }
Arc  primA(Graf G){ return G->E[0]; }
int  dultimA(Graf G, Arc a){ return a >= G->E[G->m]; }
Arc  avansA(Graf G, Arc a) { return ++a; }
```

Implementarea iteratorilor pe vârfurile **v**, adiacente unui vârf **u** consideră modul de reprezentare al grafului:

```
//implementare iteratori pe varfurile adiacente unui varf
//in reprezentarea cu liste de adiacente
Varf primVad(Graf G, Varf u){
   Arc a;
   List_Iter p=L_Begin(G->LA[G_GetKV(G,u)]);
   if(p==NULL) return NULL;
   a = (Arc)(L_Get(G->LA[G_GetKV(G,u)], p))
   if(V1(a)==u)return V2(a);
   return V1(a);
}

//implementare iteratori pe muchiile imcidente unui varf
//in reprezentarea cu liste de adiacente
Arc primAinc(Graf G, Varf u){
   Arc a;
   List_Iter p=L_Begin(G->LA[u]);
   if(p==NULL) return NULL;
   a = (Arc)(L_Get(G->LA[u], p))
   return A_New(V1(a), V2(a), A_GetCost(a), A_GetEt(a));
}
Arc avansAinc(Graf G, Arc a, Varf u){
   Arc e;
   List_Iter p = L_Find(G_LA[u], (void*)a, comp);
   if(p==NULL) return NULL;
   p = L_Next(G_LA[u],p);
   if(p==NULL) return NULL;
   e = (Arc)L_Get(G_LA[u],p);
   return A_New(V1(e), V2(e), A_GetCost(e), A_GetEt(e));
   return NULL;
}
int dultimAinc(Graf G, Arc a, Varf u){
   List_Iter p =L_Find(G->LA[u],(void*)a,comp);
   if(p==NULL || L_Next(G->LA[u],p)==NULL) return 1;
   return 0;
}
```