Grafe şi arbori

1 Definiţii, glosar

graf orientat $G = (X, E)$ unde X este o mulţime nevidă, finită, de vârfuri, iar $E \subseteq X \times X$ este o mulţime de arce = perechi de vârfuri ce relaţionează;

graf neorientat $G = (X, E)$ unde X este o mulţime nevidă, finită, de vârfuri, iar $E \subseteq \{\{x, y\} \| x, y \in X, x \neq y\}$ este o mulţime de muchii = perechi de vârfuri ce relaţionează;

incidenţă o muchie / arc e incident(ă) vîrfurilor care reprezintă capetele sale;

adiacenţă două vârfuri sunt adiacente dacă sunt unite printr-o muchie;

grad / grad interior / grad exterior numărul de varfuri legate prin muchii neorientate de varful curent / dinspre care sunt arce spre vârful curent / spre care există arce dinspre vârful curent

drum / lanţ o succesiune de 1 sau mai multe vârfuri $s = x_0, x_1, \ldots, x_{l-1}, x_l = t$, cu $(x_{i-1}, x_i) \in E$, $\forall i \in \{1, \ldots, l\}$; l = lungimea drumului = nr. de muchii/arce;

accesibilitate y e accesibil din x dacă există un drum de la x la y; accesibilitatea e reflexivă şi tranzitivă, iar în grafele neorientate, e şi simetrică;

drum elementar un drum în care nu se repetă niciun vârf (şi, ca urmare, nicio muchie);

ciclu un drum care începe şi se termină în acelaşi vârf;

ciclu elementar un ciclu, de lungime cel puţin 1, în care nu se repetă vârfuri, cu excepţia faptului că primul vârf e acelaşi cu ultimul, şi nu se repetă muchii (ultima condiţie e importantă doar pentru grafuri neorientate şi poate fi înlocuită cu condiţia ca lungimea să fie cel puţin 3);

graf (tare) conex un graf în care fiecare vârf este accesibil din fiecare vârf;

2 Reprezentare grafe

- listă de vecini (pentru grafe rare);
- matrice de adiacenţă (pentru grafe dense).

```
1  struct Vecini {
2      int* vecini;
3      int nrVecini;
4  };
5  struct GrafOrientat {
6      int nrVarfuri;
7      Vecini* succesori;
8      Vecini* predecesori;
9  };
10 struct GrafNeorientat {
11     int nrVarfuri;
12     Vecini* vecini;
13 };
```

3 Arbori

Un arbore este un graf neorientat. Există 6 caracterizări echivalente; de obicei prima este considerată definiţia unui arbore, celelalte sunt proprietăţi echivalente:

1. graf conex şi fără cicluri elementare;
2. graf conex minimal (prin eliminarea oricărei muchii devine neconex);
3. graf aciclic maximal (adăugarea unei muchii crează un ciclu elementar);
4. graf conex cu cel mult $n-1$ muchii (unde n e numărul de vârfuri);
5. graf aciclic cu cel puţin $n-1$ muchii;
6. graf în care, între oricare două vârfuri, există un unic drum elementar.

4 Arbori cu rădăcină

Un arbore cu rădăcină este un arbore în care s-a distins un vârf care este numit *rădăcină*. De obicei privim un arbore cu rădăcină ca având muchiile orientate dinspre vârful mai apropiat de rădăcină spre cel mai îndepărtat.

În plus, în structurile arborescente, adesea contează ordinea fiilor fiecărui vârf. Adiţional, în cazul arborilor binari nestricţi, un unic fiu se distinge dacă este fiu stâng sau drept.

5 Probleme

5.1 Verificare arbore

Se dă un graf neorientat. Se cere să se determine dacă este arbore.

Soluţia 1: Efectuăm o parcurgere, de exemplu, în lăţime, a vârfurilor accesibile din primul vârf, verificăm dacă toate vârfurile grafului sunt accesibile, şi, în plus, verificăm dacă numărul de muchii este egal cu $n - 1$, unde n este numărul de vârfuri.

```
1  bool eArbore(GrafNeorientat const& g) {
2      int* q = new int[g.nrVarfuri];
3      bool* visited = new bool[g.nrVarfuri];
4      for(int i=0 ; i<g.nrVarfuri ; ++i) {
5          visited[i] = false;
6      }
7      q[0] = 0;
8      int cap = 1;
9      int coada = 0;
10     visited[0] = true;
11     int nrSemiMuchii = 0;
12     while (cap > coada) {
13         int const x = q[coada];
14         ++coada;
15         for(int j=0 ; j<g.vecini[x].nrVecini ; ++j) {
16             int const y = g.vecini[x].vecini[j];
17             ++nrSemiMuchii;
18             if (!visited[y]) {
19                 q[cap] = y;
20                 ++cap;
21                 visited[y] = true;
22             }
23         }
24     }
25     return cap == g.nrVarfuri && cap*2 == nrSemiMuchii + 2;
26 }
27
```

Soluţia 2: Efectuăm o parcurgere, de exemplu, în lăţime, a vârfurilor accesibile din primul vârf, verificăm dacă toate vârfurile grafului sunt accesibile, şi, în plus, verificăm dacă nu există vreun vârf accesibil prin două drumuri distincte.

```
1  bool eArbore2(GrafNeorientat const& g) {
2      int* q = new int[g.nrVarfuri];
3      bool* visited = new bool[g.nrVarfuri];
4      int* parent = new int[g.nrVarfuri];
5      for(int i=0 ; i<g.nrVarfuri ; ++i) {
6          visited[i] = false;
7          parent[i] = -1;
8      }
9      q[0] = 0;
10     int cap = 1;
11     int coada = 0;
12     visited[0] = true;
```

```
13      while (cap > coada) {
14          int const x = q[coada];
15          ++coada;
16          for(int j=0 ; j<g.vecini[x].nrVecini ; ++j) {
17              int const y = g.vecini[x].vecini[j];
18              if (!visited[y]) {
19                  q[cap] = y;
20                  ++cap;
21                  visited[y] = true;
22                  parent[y] = x;
23              } else if(y != parent[x]) {
24                  return false;
25              }
26          }
27      }
28      return cap == g.nrVarfuri;
29  }
30
```

5.2 Căutare ciclu elementar

Problema: Se dă un graf neorientat; se cere să se găsească un ciclu elementar, dacă există.

Folosind metoda a doua de mai sus, de verificare a existenţei a două drumuri distincte de la vârful de plecare, apoi trebuie găsit ultimul strămoş comun şi, în final, să obţinem ciclul elementar.

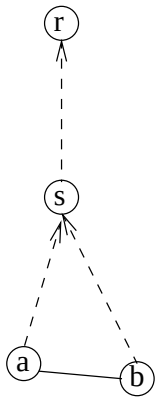

Figure 1: Cautare ciclu într-un graf neorientat

```
1   void findPrintCycle(GrafNeorientat const& g) {
2       int* q = new int[g.nrVarfuri];
3       bool* visited = new bool[g.nrVarfuri];
4       int* parent = new int[g.nrVarfuri];
5       for(int i=0 ; i<g.nrVarfuri ; ++i) {
6           visited[i] = false;
7           parent[i] = -1;
8       }
9       q[0] = 0;
```

```
10      int cap = 1;
11      int coada = 0;
12      visited[0] = true;
13      bool foundCycle = false;
14      int v1, v2; // varfurile descoperite ca parte din ciclu
15      while (!foundCycle && cap > coada) {
16          int const x = q[coada];
17          ++coada;
18          for(int j=0 ; j<g.vecini[x].nrVecini ; ++j) {
19              int const y = g.vecini[x].vecini[j];
20              if (!visited[y]) {
21                  q[cap] = y;
22                  ++cap;
23                  visited[y] = true;
24                  parent[y] = x;
25              } else if(y != parent[x]) {
26                  foundCycle = true;
27                  v1 = x;
28                  v2 = y;
29                  break;
30              }
31          }
32      }
33      if (!foundCycle) {
34          printf("Nu exista ciclu\n");
35          return;
36      }
37      // Gaseste lanturi de la radacina
38      int* lant1 = new int[g.nrVarfuri];
39      int* lant2 = new int[g.nrVarfuri];
40      int pos1 = 1;
41      lant1[0] = v1;
42      while(lant1[pos1-1] != 0) {
43          lant1[pos1] = parent[lant1[pos1-1]];
44          ++pos1;
45      }
46      int pos2 = 1;
47      lant2[0] = v2;
48      while(lant1[pos2-1] != 0) {
49          lant2[pos2] = parent[lant2[pos2-1]];
50          ++pos2;
51      }
52      // elimina partea comuna
53      while(pos1 > 0 && pos2 > 0 && lant1[pos1-1] == lant2[pos2-1]) {
54          --pos1;
55          --pos2;
56      }
57      // tipareste
58      for(int i=0 ; i<=pos1 ; ++i) {
59          printf("%d ", lant1[i]);
60      }
61      for(int i=pos2-1 ; i>=0 ; --i) {
62          printf("%d ", lant2[i]);
63      }
64      printf("\n");
65  }
```

5.3 Reconstituirea unui arbore din parcurgerile în preordine şi postordine

Idee: Mai întâi, rădăcina e primul vârf ce apare în preordine şi ultimul ce apare în postordine. Apoi, pentru fiecare sub-arbore al rădăcinii, rădăcina subarborelui e primul vârf în preordine şi ultimul în postordine. Asta permite să extragem lista de vîrfuri corespunzătoare fiecărui subarbore: luăm rădăcina subarborelui din parcurgerea în preordine, o căutăm în parcurgerea în postordine, lista vârfurilor până acolo este lista vârfurilor din subarbore, de unde obţinem numărul de vârfuri din subarbore şi de aici putem selecta din ambele liste sub-listele corespunzătoare subarborelui.

Apoi procedăm similar pentru a extrage listele corespunzătoare celorlalţi subarbori şi, recursiv, subarborilor lor.

Exemplu:

```
5 1 4 7 2 3 6
4 2 7 1 3 6 5
```

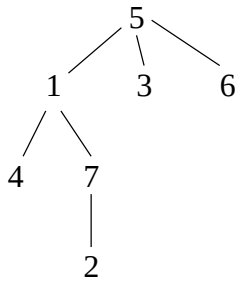

Figure 2: Arborele reconstituit

- rădăcina = `5`;
- primul fiu = `1`; listă postordine = `4 2 7 1`, listă preordine = `1 4 7 2`
  - rădăcina subarbore = `1`
  - primul fiu = `4`, listă postordine = `4`, listă preordine = `4`
  - al doilea fiu = `7`, listă postordine = `2 7`, listă preordine = `7 2`
    - rădăcina subarbore = `7`
    - primul fiu = `2`, listă postordine = `2`, listă preordine = `2`
- al doilea fiu = `3`; listă postordine = `3`, listă preordine = `3`

- al treilea fiu = 6; listă postordine = 6, listă preordine = 6

```
1  void proceseaza(int const preord[], int const postord[], int n) {
2          int* children = new int[n];
3          int nrChildren = 0;
4          int i=1;
5          while(i<n) {
6              int child = preord[i];
7              children[nrChildren] = child;
8              ++nrChildren;
9              int j = i-1;
10             while(j<n && postord[j] != child) {
11                 ++j;
12             }
13             if(j >= n) {
14                     fprintf(stderr, "Varful %d nu apare in postordine\n", child);
15                     return;
16             }
17             proceseaza(preord+i, postord+i-1, j+2-i);
18             i=j+2;
19         }
20         printf("%d :", *preord);
21         for(i=0 ; i<nrChildren ; ++i) {
22             printf(" %d", children[i]);
23         }
24         printf("\n");
25         delete[] children;
26 }
```