16. GRAFURI

16.1 Structura de date de tip graf

Proiectarea unui sistem informatic care să gestioneze reţeaua naţională de căi ferate presupune crearea unui mediu de lucru capabil să utilizeze baza de date a staţiilor şi pe cea a liniilor de cale ferate pentru a oferi soluţii optime şi reale care să minimizeze costurile şi timpul de folosire a reţelei. Luând de exemplu situaţia reală din figura 16.1, se pune problema parcurgerii traseului Arad – Bucureşti cu cost redus, acest lucru implicând parcurgerea distanţei cea mai scurtă.

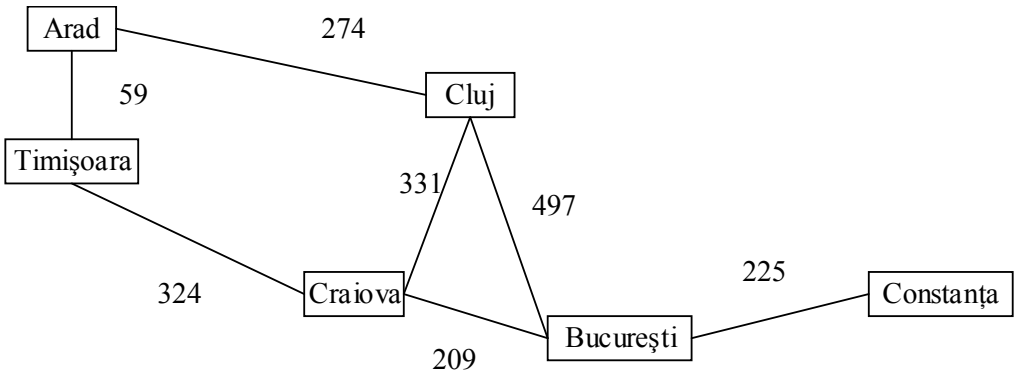


Figura 16.1 *O parte a reţelei de cale ferată*

Sistemul prelucrează informaţiile iniţiale, aflate în două mulţimi N şi A, unde N = {Arad, Bucureşti, Cluj, Craiova, Constanţa, Timişoara} este mulţimea oraşelor iar A = {Arad – Timişoara = 59 km, Timişoara – Craiova = 324 km, Craiova – Bucureşti = 209 km, Bucureşti – Constanţa = 225 km, Arad – Cluj = 274 km, Cluj – Craiova = 331 km, Cluj – Bucureşti = 497 km} este mulţimea distanţelor dintre două oraşe şi oferă soluţia căutată.

Reprezentarea în memorie a acestor informaţii care au multiple legături între ele, astfel încât să permită parcurgerea completă a zonelor sau localizarea unui element din structură, se face în situaţia de faţă utilizând graful.

Graful, asemenea arborelui, este o structură în care relaţia dintre nodul părinte şi nodul fiu este una ierarhică, dar care este mai puţin restrictivă în sensul că un nod are mai mulţi succesori dar şi mai mulţi predecesori. El este definit ca o colecţie de date reunite în două mulţimi: mulţimea $N = \{N_1, N_2, \ldots, N_n \mid n$ – numărul de noduri al grafului$\}$ ce conţine toate nodurile grafului şi mulţimea $A = \{(N_i, N_j) = A_{ij} \mid N_i, N_j \subset N$ şi $i,j = 1, \ldots, n$ cu $i \neq j\}$ care conţine arcele dintre două noduri vecine.

Graful este larg utilizat în domeniile: ciberneticii, matematicii, cercetărilor operaţionale în vederea optimizării diferitelor activităţi economice, chimiei pentru descrierea structurii cristalelor, reţelelor de transport de toate tipurile pentru optimizarea traseelor, circuitelor electrice pentru simularea funcţionări corecte, inteligenţei artificiale şi nu în ultimul rând în domeniul analizei aplicaţiilor software.

Graful este de mai multe tipuri, fiind clasificat în funcţie de :

- direcţia arcelor – în cazul în care arcele dintre nodurile grafului sunt nedirecţionate atunci graful este unul neorientat; când există

sens între două noduri N_i, N_j şi arcul este direcţionat ($N_i \rightarrow N_j$ sau $N_i \leftarrow N_j$ sau $N_i \leftrightarrow N_j$) atunci graful este unul orientat;

Figura 16.2 *Graf orientat şi neorientat*

- greutatea arcelor – dacă oricare arc dintre două noduri al grafului are asociată o valoare numerică (care reprezintă de cele mai multe ori distanţa, durata de timp sau costul) atunci graful este cu greutate; în cazul în care arcele nu au asociate valori numerice, graful este unul fără greutate;

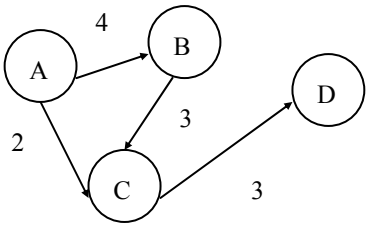


Figura 16.3 *Graf orientat cu greutate*

- existenţa arcelor; dacă într-un graf nu există nici un nod izolat, altfel spus pentru oricare nod N_i cu i = 1..n există cel puţin un nod N_j cu $1 \le j \le n$ şi $i \neq j$ pentru care există arcul A_{ij} asociat, atunci graful este conectat; un graf este neconectat dacă există cel puţin un nod izolat.

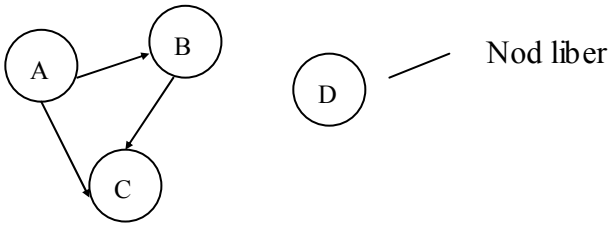
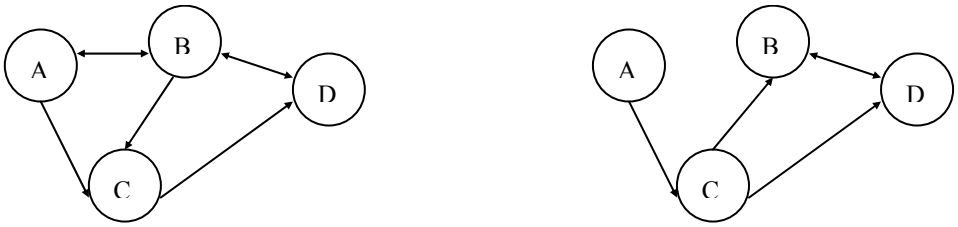


Figura 16.4 *Graf orientat neconectat*

La rândul lui graful orientat este puternic conectat dacă între oricare două noduri N_i şi N_j cu i, j = 1..n există drum (un drum este format din unul sau mai multe arce) orientat de la i la j, $N_i \rightarrow N_j$. Graful orientat este slab conectat dacă între oricare două noduri N_i şi N_j cu i, j = 1..n există drum orientat de la i la j, $N_i \rightarrow N_j$ **sau** de la j la i, $N_i \leftarrow N_j$ (doar unul dintre ele).



Graf orientat puternic conectat Graf orientat slab conectat

Figura 16.5 *Tipuri de graf orientat*

De exemplu în figura 16.5 se observă că al doilea graf orientat este slab conectat pentru că între nodul A şi D există drum orientat, însă de la D la A nu există drum.

16.2 Implementarea grafului

Există numeroase metode de reprezentare în memoria calculatorului a grafului, fiecare cu avantajele şi dezavantajele ei:

- matricea de adiacenţă;
- liste înlănţuite;
- un vector de pointeri la liste simple sau dublu înlănţuite de noduri adiacente;
- o listă simplu sau dublu înlănţuită de pointeri la liste simple sau dublu înlănţuite de noduri adiacente;
- un vector de pointeri la liste simple sau dublu înlănţuite de arce.

Dintre toate, cele mai utilizate metode sunt primele două.

Reprezentarea prin *matrice de adiacenţă* a grafului este eficientă când se cunoaşte numărul nodurilor şi numărul mediu al arcelor; acesta din urmă trebuie să fie mare pentru ca gradul de umplere al matricei de adiacenţă să fie scăzut. Cum crearea de software optimizat înseamnă şi generalizarea problemei, lucru care dă puţine şanse să se cunoască numărul nodurilor şi cel al arcelor, singurul argument pro pentru această metodă este dat doar de uşurinţă implementării şi utilizării matricelor. În cele mai multe situaţii reale, cea mai mare parte a memoriei necesară stocării matricei de adiacenţă este nefolosită.

Pentru un graf cu *n* noduri este necesară o matrice pătratică M de dimensiuni nxn, care pentru un *n* foarte mare ocupă mult spaţiu.

Iniţial matricea de adiacenţă are toate elementele egale cu valoarea 0. Pentru a reprezenta arcul A_{ij} dintre nodurile N_i şi N_j (orientat de la N_i la N_j) la intersecţia liniei *i* cu coloana *j* se trece valoarea 1, în cazul grafului cu fără greutate, sau greutatea arcului, pentru graful cu greutate.

$$\text{Aşadar: } M[i][j] = \begin{cases} 1, \text{ dacă există arc între nodul } N_i \text{ şi } N_j; \\ 0, \text{ dacă nu există arc între nodul } N_i \text{ şi } N_j; \end{cases}$$

în cazul grafului fără greutate şi:

$$M[i][j] = \begin{cases} a_{ij}, \text{ dacă există arc între nodul } N_i \text{ şi } N_j, \text{ iar } a_{ij} \\ \quad \text{reprezintă greutatea arcului;} \\ 0, \text{ dacă nu există arc între nodul } N_i \text{ şi } N_j; \end{cases}$$

În cazul unui graf neorientat, matricea de adiacenţă asociată este simetrică. Pentru graful cu 5 noduri din figura 16.6, Matricele de adiacenţă corespunzătoare grafurilor sunt prezentate în figura 16.7.

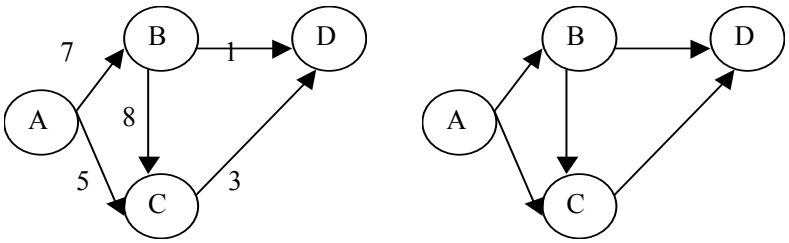


Figura 16.6 *Graf orientat cu greutate şi fără greutate*

0	7	5	0
0	0	8	1
0	0	0	3
0	0	0	0

0	1	1	0
0	0	1	1
0	0	0	1
0	0	0	0

Figura 16.7 *Matrice de adiacenţă*

Lungimea drumului dintre două noduri ale unui graf orientat fără greutate este dată de numărul de arce. Astfel în cazul grafului din figura 16.6, între nodul A şi D există trei drumuri posibile, unul de lungime 3 (A-B-C-D) şi două de lungime 2 (A-C-D şi A-B-D).

Uşurinţa lucrului cu matricea de adiacenţă constă şi în faptul că prin simple ridicări la putere se verifică dacă există drum între două noduri şi care este lungimea acestuia (doar în cazul grafului orientat fără greutate).

Fie M[4][4] matricea de adiacenţă din figura 16.7 asociată grafului orientat fără greutate, atunci obţinem:

$$M^2 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad M^3 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad M^4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad (16.1)$$

După cum se observă, dacă $M^k[i][j] = val \neq 0$ atunci între nodurile N_i şi N_j există *val* drumuri pe direcţia $N_i \rightarrow N_j$, drumuri de lungime *k*.

În cazul lui M^2 există 2 drumuri de lungime 2 de la nodul A la nodul D (A-B-D şi A-C-D), un drum de lungime 2 de la nodul A la nodul C (A-B-C) şi un drum de lungime 2 de la nodul B la nodul D (B-C-D).

Pentru *k = 3* există drumul de lungime 3 de la nodul a la nodul D (A-B-C-D).

Procesul de ridicare la putere se opreşte când se ajunge la *k = n*, unde *n* este dimensiunea matricei.

Clasa *graf* care implementează diferite operaţii cu grafuri reprezentate prin intermediul matricei de adiacenţă este:

```
#ifndef GRAF_MATRICE_H
#define GRAF_MATRICE_H

#include <iostream>
using namespace std;

class graf_matrice
{
public:
	static const int MaxN = 50;          // numarul maxim de noduri
	static const int Infinit = INT_MAX;  // nu exista drum intre noduri

	enum TipMatrice
	{
		MatriceAdiacenta = 1,
		MatriceCosturi = 2
	};

private:

	//matricea de adiacenta (1=exista, 0=nu exista)
	int matm[MaxN][MaxN];

	//matricea de costuri (Infinit=nu e drum)
	long matc[MaxN][MaxN];
	int nr_noduri;
	int nr_arce;

protected:
	int vect_rez[MaxN];     //vectorul în care se țin minte drumuri
	int nr_rez;             //nr. nodurilor din rezultat
	int matr[MaxN][MaxN];   //matricea rezultatelor obținute din
prelucrări
	int valid(int);   //testez daca int se afla deja in vect soluție
	int suma(int);    //calculează lungimea drumului

public:
	graf_matrice();
	void init(TipMatrice tip); //inițializare graf

	void vect_r();    //afişează vectorul rezultat
	void matr_r();    //afişează matricea rezultat
	void matr_r1();
	void drum_minim();/* se calculează drumul minim intre oricare 2
noduri */
	void este_drum(); //verif. existenta drumului intre 2 noduri
	void drum_minim(int, int);   //drumul minim între 2 noduri date
	void toate_drm(int,int);     //toate drumurile dintre 2 noduri
	bool este_arbore();          //verifică daca este sau nu arbore
	void componente_conexe();    //determină componentele conexe
	bool graf_matrice::este_eulerian();
	void parcurgere_bf(int);     //parcurgerea grafului în lățime
	void parcurgere_df(int);     //parcurgerea grafului în adâncime
```

```
	void hamiltonian();		//verif. exist. cicluri hamiltoniene
	void prim();			//det. arbore parțial de cost minim
};

graf_matrice::graf_matrice()
{
	nr_noduri = 0;
	nr_arce = 0;
	nr_rez = 0;
	for(int i = 0; i < MaxN; i++)
	{
		for(int j = 0; j < MaxN; j++)
		{
			matm[i][j] = 0;
			matc[i][j] = Infinit;
		}
	}
}

void graf_matrice::init(TipMatrice tip)
{
	cout << "Introduceti numarul de noduri: ";
	Do
	{
		cin >> nr_noduri;
		if (nr_noduri <= 0)
		{
			cout << "EROARE: Numar invalid de noduri" << endl;
		}
	} while(nr_noduri <= 0);

	if (tip == MatriceAdiacenta)
	{
		cout << "Se va introduce matricea de adiacenta" << endl;
		cout << "Se introduc valorile (1-e drum; 0- nu e drum\n";
	}
	else
	{
		cout << "Se va introduce matricea de costuri" << endl;
	}

	for (int j = 0; j < nr_noduri; j++)
	{
		for (int i = 0; i < j; i++)
		{
			cout << "m[" << i + 1 << "][" << j + 1 << "]=";

			if (tip == MatriceAdiacenta)
			{
				do
				{
					cin >> matm[i][j];
					matm[j][i] = matm[i][j];
				} while (matm[i][j] != 0 && matm[i][j] != 1);

				if (matm[i][j] != 0)
				{
					nr_arce++;
					matc[i][j] = matc[j][i] = 1;
				}
				else
```

```
                        {
                              matc[i][j]=Infinit;
                              matc[j][i]=Infinit;
                        }
                  }
                  else
                  {
                        do
                        {
                              cin >> matc[i][j];
                              matc[j][i] = matc[i][j];
                        } while (matc[i][j] < 0);

                        if (matc[i][j] != 0)
                        {
                              nr_arce++;
                              matm[j][i] = matm[i][j] = 1;
                        }
                        else
                        {
                              matc[i][j] = matc[j][i] = Infinit;
                              matm[i][j] = matm[j][i] = 0;
                        }
                  }
            } // for j
      } // for i
}

void graf_matrice::vect_r()
{
      cout << "Vectorul rezultat este:" << endl;
      for (int i = 0; i < nr_rez; i++)
      {
            cout << vect_rez[i] << " " << endl;
      }
}

void graf_matrice::matr_r()
{
      for (int i = 0; i < nr_noduri; i++)
      {
          for (int j = 0; j < i; j++)
            {
                  cout << "Intre " << i + 1 << " si " << j + 1 <<
                    (matr[i][j] != 0 ? "" : " nu") << "exista drum.\n";
            }
      }
}

void graf_matrice::matr_r1()
{
      for (int i = 0; i < nr_noduri; i++)
      {
          for (int j = 0; j < i; j++)
            {
                  if (matr[i][j] != Infinit)
                  {
                        cout << "Intre " << i + 1 << " si " << j + 1 <<
                            "exista drum de cost minim " << matr[i][j] <<
endl;
                  }
```

```
                  else
                  {
                        Cout << "Intre " << i + 1 << " si " << j + 1 <<
                              "nu exista drum." << endl;
                  }
            }
      }
}

void graf_matrice::drum_minim()
{
      for (int i = 0; i < nr_noduri; i++)
      {
            for (int j = 0; j < nr_noduri; j++)
            {
                  matr[i][j] = matc[i][j];
            }
      }

      for (int k = 0; k < nr_noduri; k++)
      {
            for (int i = 0; i < nr_noduri; i++)
            {
                  for (int j=0; j < nr_noduri; j++)
                  {
                     matr[i][j] = min(matr[i][j],matr[i][k]+matr[k][j]);
                  }
            }
      }
}


void graf_matrice::este_drum()
{
      for (int i = 0; i < nr_noduri; i++)
      {
            for (int j = 0; j < nr_noduri; j++)
            {
                  matr[i][j] = matm[i][j];
            }
      }

      for (int k = 0; k < nr_noduri; k++)
      {
            for (int i = 0; i < nr_noduri; i++)
            {
                  for (int j = 0; j < nr_noduri; j++)
                  {
                        matr[i][j]=matr[i][j]||(matr[i][k]&&matr[k][j]);
                  }
            }
      }
}

int graf_matrice::valid(int k)
{
      if (matc[vect_rez[k-1]][vect_rez[k]] == Infinit)
      {
            return 0;
      }
```

```
	for (int i = 0; i < k; i++)
	{
		if (vect_rez[i] == vect_rez[k])
		{
			return 0;
		}
	}

	return 1;
}

int graf_matrice::suma(int k)
{
	int suma = 0;
	for (int i=0; i<k; i++)
	{
		suma += matc[vect_rez[i]][vect_rez[i+1]];
	}

	return suma;
}

void graf_matrice::drum_minim(int a, int b)
{
	int h, vect[MaxN], lgdrum, min = Infinit;

	for (int f = 0; f < MaxN; f++)
	{
		vect_rez[f] = -1;
	}

	vect_rez[0] = a;
	h = 1;

	while(h >= 1)
	{
		while(vect_rez[h] < nr_noduri)
		{
			vect_rez[h]++;
			if(valid(h))
			{
				if(vect_rez[h] == b)
				{
					lgdrum = suma(h);
					if(lgdrum < min)
					{
						for (int e = 0; e <= h; e++)
						{
							vect[e] = vect_rez[e];
						}

						min = lgdrum;
						nr_rez = h + 1;
					}
				}
				else
				{
					h++;
					vect_rez[h] = -1;
				}
			} // if (valid(h))
```

```
            }

            h--;
      }

      for (int e = 0; e < nr_rez; e++)
      {
            vect_rez[e] = vect[e];
      }
}

bool graf_matrice::este_arbore()
{
      int b[MaxN], i, j, v, s;

      v = 0;

      for (i = 0; i < nr_noduri; i++)
      {
            for(j = i + 1; j < nr_noduri; j++)
            {
                  v += (matm[i][j] == 1);
            }
      }

      for (int i = 0; i < nr_noduri; b[i] = 0, i++);
      b[0]=1;

      for(i=0;i<nr_noduri;i++)
      {
            for(j=0;j<nr_noduri;j++)
            {
                  if((matm[i][j] == 1) && (b[i] == 1))
                  {
                        b[j] = 1;
                  }
            }
      }

      s=0;
      for (i = 0; i < nr_noduri; i++)
      {
            s += (b[i] == 0);
      }

      return (s == 0) && (v == nr_noduri - 1);
}

bool graf_matrice::este_eulerian()
{
      int b[MaxN],i,j,k,v,s;

      for (int i = 0; i < nr_noduri; b[i] = 0, i++);
      b[0] = 1;

      for(i=0;i<nr_noduri;i++)
      {
            for(j=0;j<nr_noduri;j++)
            {
                  if((matm[i][j] == 1) && (b[i] == 1))
                  {
```

```
                        b[j] = 1;
                  }
            }
      }

      s=0;
      for (i = 0; i < nr_noduri; i++)
      {
            s += (b[i] == 0);
      }

      for(i = 0; i < nr_noduri; i++)
      {
            k = 0;
            for(j = 0; j < nr_noduri; j++)
            {
                  k += (matm[i][j]==1);
                  v = k / 2;

                  if ((k - 2 * v) > 0)
                        s = 1;
            }
      }

      return s <= 0;
}

void graf_matrice::parcurgere_bf(int i)
{
      int j,u,p,v,c[MaxN],vizitat[MaxN];

      for (int i = 0; i < nr_noduri; vizitat[i] = 0, i++);
      c[1] = i; u = 1; vizitat[i] = 1;

      for(p = 1; p <= u; p++)
      {
            v = c[p];
            for(j = 0; j < nr_noduri; j++)
            {
                  if ((matm[v][j] == 1) && (vizitat[j] == 0))
                  {
                        u++;
                        c[u] = j;
                        vizitat[j] = 1;
                  }
            }
      }

      cout << "lista vf. parcurse cu metoda BF pornind de la varful "
            << i + 1 << ": " << endl;

      for(j = 2; j <= u; j++)
            cout << c[j] + 1 << endl;
}

void graf_matrice::parcurgere_df(int i)
{
      int vizitat[MaxN], urmator[MaxN], s[MaxN], k, v, j, ps;

      for (int i=0; i < nr_noduri; urmator[i] = 0, vizitat[i] = 0, i++);
```

```
	s[1] = i; ps = 1; vizitat[i] = 1;

	cout << "ordinea in DF este " << i+1 << " ";

	while(ps >= 1)
	{
		j = s[ps];
		k = urmator[j] + 1;
		while ((k <= nr_noduri) &&
			((matm[j][k]==0) || (matm[j][k]==1)&&(vizitat[k]==1)))
		{
		    k++;
		}

		urmator[j] = k;

		if(k == nr_noduri + 1)
		{
			ps--;
		}
		else
		{
			cout << k + 1 << " ";
			vizitat[k] = 1;
			ps++;
			s[ps] = k;
		}
	}
}

void graf_matrice::prim()
{
	int s[MaxN],t[MaxN],p[MaxN];
	int min,k,l,c,i,j,v;

	cout << "Nodul de pornire: ";
	cin >> v;

	for (int i = 0; i < nr_noduri; s[i] = t[i] = p[i] = 0, i++);
	s[v - 1] = 1;

	for(k = 1; k <= nr_noduri - 1; k++)
	{
		min = Infinit;

		for(i = 0; i < nr_noduri; i++)
		{
			for(j = 0; j < nr_noduri; j++)
			{
				if ((s[i]==1)&&(s[j]==0) && (min > matc[i][j]))
				{
					min = matc[i][j];
					l = i; c = j;
				}
			}
		}

	 s[c] = 1;
	 t[c] = l + 1;
	 p[c] = matc[l][c];
	}
```

```
	for(i = 0; i < nr_noduri; i++)
	{
		cout << t[i] << " ";
	}
	cout << endl;

	for(i = 0; i < nr_noduri; i++)
	{
		cout << p[i] << " ";
	}
}

void graf_matrice::hamiltonian()
{
	int x[MaxN], v, k, j, sol, i;
	sol = 0;
	x[1] = 1; x[2] = 1; k = 2;

	while(k > 1)
	{
		v = 0;
		while((x[k] + 1 <= nr_noduri) && (v == 0))
		{
			x[k]++;
			v = 1;
			for(i = 1;i <= k - 1; i++)
			{
				if(x[k] == x[i])
				{
					v = 0;
					break;
				}
			}

			if(matm[x[k - 1] - 1][x[k] - 1] == 0)
			{
				v = 0;
			}
		}

		if(v == 0)
		{
			k--;
		}
		else
		{
			if ((k==nr_noduri)&&(matm[x[nr_noduri]-1][x[1]-1]==1))
			{
				sol++;

				if(sol==1)
				{
					cout << "Cicluri hamiltoniene:" << endl;
				}

				for (int j = 1; j <= nr_noduri; j++)
					cout << x[j] << " ";
				cout << endl;
			}
			else
```

```
                  {
                        if(k < nr_noduri)
                        {
                              k++;
                              x[k] = 1;
                        }
                  }
            }
      }

      if (sol == 0)
            cout << "Graful nu este hamiltonian.";
}

void graf_matrice::componente_conexe()
{
      int b[MaxN],k,j,i;

      for(i = 0; i < nr_noduri; i++)
      {
            for(j = 0; j < nr_noduri; j++)
            {
                  matr[i][j] = matm[i][j];
            }
      }

      cout << "Componentele conexe sunt: ";
      for(k = 0; k < nr_noduri; k++)
      {
            for(j = 0; j < nr_noduri; j++)
            {
                  b[j] = -2;
            }

            b[k] = k;

            if(matr[k][k] > -1)
                  cout << endl << "   ";

            for(i = 0; i < nr_noduri; i++)
            {
                  for(j = 0;j < nr_noduri; j++)
                  {
                        if((matr[i][j] == 1) && (b[i] == k))
                        {
                              b[j] = k;
                              matr[i][j] = 0;
                              matr[j][i] = 0;
                        }
                  }

                  for(j = 0; j < nr_noduri; j++)
                  {
                        if((b[j] == k) && (matr[j][j] > -1))
                        {
                              matr[j][j] = -1;
                              cout << " " << j + 1;
                        }
                  }
            }
      }
```

```
}

void graf_matrice::toate_drm(int a, int b)
{
      int h;

      for (int f = 0; f < MaxN; f++)
            vect_rez[f] = -1;

      vect_rez[0] = a;
      h = 1;
      while(h >= 1)
      {
            while(vect_rez[h] < nr_noduri)
            {
                  vect_rez[h]++;
                  if(valid(h))
                  {
                        if(vect_rez[h] == b)
                        {
                              cout << endl;
                              for (int e = 0; e <= h; e++)
                                    cout << vect_rez[e] + 1 << " ";
                        }
                        else
                        {
                              h++;
                              vect_rez[h] = -1;
                        }
                  }
            }
            h--;
      }
}

#endif //GRAF_MATRICE_H
```

De cele mai multe ori nu se cunoaşte numărul de noduri ale grafului, apelându-se la construirea dinamică a grafului pe parcursul rezolvării problemei, deci nu se cunoaşte dimensiunea matricei de adiacenţă. În aceste situaţii graful este reprezentat printr-o reţea de liste înlănţuite, *liste de adiacenţă*. Asemenea matricei de adiacenţă descrierea grafului cuprinde mulţimea de noduri şi pe cea de arce, precizând orientarea arcului şi după caz greutatea lui.

Se defineşte structura *arc* care este asociată elementelor din mulţimea arcelor:

```
struct arc
{
 struct nodgraf * destinație; /*adresa nodului către care
                                există arc;*/
 struct arc* next_arc; /*referință către elementul următor
                        din lista de arce;*/
 int greutate; };         //greutatea arcului;
```

Este vorba de o listă a arcelor ce este construită dinamic. Structura este cea a unei liste oarecare, cuprinzând informaţia propriu-zisă, greutatea arcului, precum şi cea necesară înlănţuirii în listă, adresa elementului

următor. Cu toate că *nodgraf * destinaţie* este un pointer, el face parte din informaţia de bază şi nu din cea destinată regăsirii în listă. În listă există mai multe liste, organizate pe principiul descendenţei dintr-un nod. Cum fiecare nod din graf este unic, se elimină astfel posibilitatea ca un arc să fie în mai multe liste.

Tipul de structura *nodgraf* este tot o structură de tip listă. Pe lângă informaţia nodului şi adresa următorului nod, ea conţine şi adresa de start a listei ce cuprinde arcele descendente din nodul respectiv.

```
struct nodgraf {
      int info;         //informația nodului;
      struct nodgraf* next;  //referință către următorul nod;
      struct arc *capat;      //capătul listei de arce;
}
```

La crearea grafului se introduc iniţial informaţiile nodurilor, creându-se astfel lista lor. După aceasta se vor crea listele de arce introducându-se informaţia nodului sursă, a nodului destinaţie şi greutatea arcului.

Pentru graful cu greutate din figura 16.6 reprezentarea sa în memorie prin intermediul listelor de liste este dată în figura 16.8.

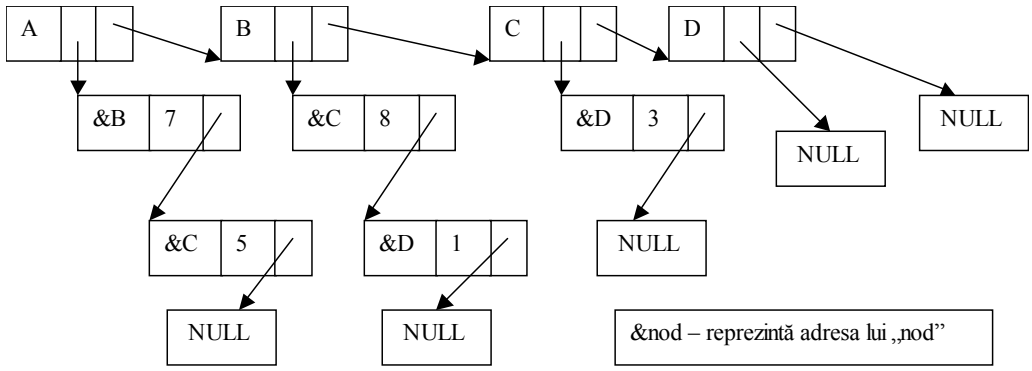


Figura 16.8 *Reprezentarea în memorie a unui graf cu ajutorul listelor*

Clasa *graf_liste* care implementează graful definit cu ajutorul listelor înlănţuite este:

```
#ifndef GRAF_LISTE_H
#define GRAF_LISTE_H

#include <iostream>

using namespace std;

typedef struct arc
{
      struct nodgraf *destinatie;  // adresa nodului catre care e arc;
      int greutate;                 // greutatea arcului;

      // referinta catre elementul urmator din lista de arce
      struct arc *next_arc;
} arc;
```

```
typedef struct nodgraf
 {
      int info;               //informatia nodului;
      struct nodgraf *next;   //referinta catre urmatorul nod;
      struct arc *capat;      //capatul listei de arce;
 } nodgraf;

typedef struct stiva
 {
      nodgraf *n;             //elementul stivei
      struct stiva *next;     //referinta catre urmatorul element;
 } stiva;

struct lista
{
      nodgraf *n;             //elementul listei
      struct lista *next;     //referinta catre urmatorul element;
};

typedef struct coada
{
      struct lista *pred;
      struct lista *succ;
} coada;

class graf_liste
{
      int nr;

public:

      static const int MaxN = 50;         // numarul maxim de noduri
      static const int Infinit = INT_MAX;  /* nu exista drum intre
noduri */

      graf_liste(){}
      graf_liste(int nrnoduri){ nr=nrnoduri;}
      nodgraf *inserare_nod(nodgraf *,int); //insereaza un nod in graf
      nodgraf *gaseste_nod(nodgraf *, int); //gaseste un nod in graf
      void inserare_arc(nodgraf *,int,int,int); //insereaza arc
      int adiacent(nodgraf *,nodgraf *); /* vf dc 2 noduri sunt
adiacente */
      int sterge_arc(nodgraf *,nodgraf *); //sterge arcul

      void vizitare(nodgraf *nd,int vizitat[]) /* march. nodul ca
vizitat */
      {
            vizitat[nd->info]=1;
            cout<<nd->info<<" - ";
      }

      stiva *push(stiva *stk,nodgraf *nd) //pune un element in stiva
      {
            stiva *t = new stiva();
            t->n = nd;
            t->next = stk;
            return t;
      }

      stiva *pop(stiva *stk, nodgraf **nd) /* scoate un element din
stiva */
```

```
{
        if(!stk)
        {
              return NULL;
        }
        else
        {
              stiva *t = stk->next;
              (*nd) = stk->n;
              delete stk;

              return t;
        }
 }

 void depth(nodgraf *); //parcurgere in adancime

 void put(struct coada *q, nodgraf *nd)
 {
       lista *t = new lista();
       t->n=nd;
       t->next=NULL;

       lista *keep = q->succ;
       if (keep != NULL)
       {
             keep->next=t;
       }
       else
       {
             q->pred=t;
       }

       q->succ = t;
 }

 nodgraf *get(struct coada *q) //scoate un element din coada
 {
       if(q->pred == NULL)
       {
             return NULL;
       }
       else
       {
             lista* t = q->pred;
             nodgraf* n=t->n;

             if(q->pred == q->succ)
             {
                   q->succ = NULL;
             }

             q->pred = q->pred->next;

             delete t;
             return n;
       }
 }

 void breadth(nodgraf *); //parcurgere in latime
 int drum_minim(nodgraf *,int,int,int[]); //gaseste drumul minim
```

```
      void stergere_nod(nodgraf *&,int);// sterge un nod din graf
};

nodgraf* graf_liste::inserare_nod(nodgraf *cap,int info)
{

  nodgraf *nou= new nodgraf();
  nou->info=info;
  nou->next=NULL;
  nou->capat=NULL;

  if(cap==NULL)
  {
        return nou;
  }
  else
  {
      nodgraf *temp = cap;
      while(temp->next != NULL)
        {
              temp=temp->next;
        }

      temp->next = nou;
      return nou;
  }
}

nodgraf* graf_liste::gaseste_nod(nodgraf *cap, int info)
{
      while(cap != NULL && cap->info != info)
      {
            cap = cap->next;
      }

      return cap;
}

void graf_liste::inserare_arc(nodgraf *cap,int sursa,int dest,int
greutate)
{
      nodgraf* s = gaseste_nod(cap,sursa);
      if(s == NULL)
      {
            s = inserare_nod(cap,sursa);
      }

      nodgraf* d=gaseste_nod(cap,dest);
      if(d == NULL)
      {
            d = inserare_nod(cap,dest);
      }

      arc *temp = s->capat,*keep=NULL;
      int gasit = 0;

      while(temp != NULL && !gasit)
      {
            if(temp->destinatie == d)
            {
                  temp->greutate = greutate;
```

```
                  gasit = 1;
            }
            else
            {
                  keep = temp;
                  temp = temp->next_arc;
             }
      }

      if(!gasit)
      {
            temp= new arc();
            temp->destinatie=d;
            temp->greutate=greutate;
            temp->next_arc=NULL;

            if(keep == NULL)
            {
                  s->capat=temp;
            }
            else
            {
                  keep->next_arc=temp;
            }
      }
}

int graf_liste::adiacent(nodgraf *s,nodgraf *d)
{
  arc *temp = s->capat;

  while(temp != NULL && temp->destinatie != d)
  {
        temp=temp->next_arc;
  }

  return temp != NULL ? temp->greutate : 0;
}

int graf_liste::sterge_arc(nodgraf *s,nodgraf *d)
{
      arc *keep = NULL;
      arc* temp=s->capat;

      while(temp != NULL)
      {
            if(temp->destinatie == d)
            {
                  if(keep == NULL)
                  {
                        s->capat=temp->next_arc;
                  }
                  else
                  {
                        keep->next_arc=temp->next_arc;
                  }

                  delete temp;
                  return 1; // nodul a fost şters
            }
            else
```

```
            {
                  keep = temp;
                  temp = temp->next_arc;
            }
      }

      return 0; // nodul nu a fost şters
}

void graf_liste::depth(nodgraf *g)
{
      int vizitat[MaxN];
      for(int i=0; i < MaxN; i++)
      {
            vizitat[i] = 0;
      }

      nodgraf *curent,**nd;
      arc *temp;
      stiva *stk=NULL;

      while(g != NULL)
    {
            if(!vizitat[g->info])
            {
                  cout << endl << "Componenta : " <<endl;
                  stk = push(stk,g);

                  do
                  {
                        stk = pop(stk,nd);
                        curent = *nd;
                        vizitare(curent,vizitat);
                        temp=curent->capat;

                        while(temp)
                        {
                              if(!vizitat[temp->destinatie->info])
                                    stk=push(stk,temp->destinatie);

                              temp=temp->next_arc;
                        }
                  } while(stk != NULL);
            }
            g=g->next;
      }
}

void graf_liste::breadth(nodgraf *g)
{
      nodgraf *curent;
      arc *temp;
      coada *q,coada={NULL,NULL};
      q = &coada;

      int vizitat[MaxN];
      for(int i = 0; i < MaxN; i++)
      {
            vizitat[i] = 0;
      }
```

```
      while(g != NULL)
      {
            if(!vizitat[g->info])
            {
                  cout << endl << "Componenta : " << endl;
                  put(q,g);

                  do
                  {
                        curent = get(q);

                        if(!vizitat[curent->info])
                        {
                              vizitare(curent,vizitat);
                              temp = curent->capat;

                              while(temp)
                              {
                                    if(!vizitat[temp->destinatie-
>info])
                                      put(q,temp->destinatie);

                                    temp=temp->next_arc;
                              }
                        }
                  } while(q->pred != NULL);
            }

            g = g->next;
      }
}

int graf_liste::drum_minim(nodgraf *g,int sursa,int dest,int
precede[MaxN])
{
   nodgraf *tmp;
   arc *temp;

      int distanta[MaxN],i,k,min,curent, perm[MaxN], dc, distanta_noua;

      for(i = 0; i < MaxN; i++)
      {
            distanta[i]=precede[i]=graf_liste::Infinit;
            perm[i]=-1;
      }

      distanta[sursa] = 0;
      curent = sursa;
      perm[sursa] = 1;

      while(curent != dest)
      {
            dc = distanta[curent];
            tmp = gaseste_nod(g,curent);
            temp = tmp->capat;

            while(temp)
            {
                  distanta_noua = dc + temp->greutate;
                  tmp = temp->destinatie;
```

```
                  if(distanta_noua < distanta[tmp->info])
            {
                        distanta[tmp->info] = distanta_noua;
                        precede[tmp->info] = curent;
                  }

                  temp = temp->next_arc;
            }

            for(i = 0; i < MaxN && perm[i] > 0; i++);

            k = i;
            min = distanta[k];
            for(i = k; i < MaxN; i++)
            {
                  if(perm[i] < 0 && distanta[i] < min)
                  {
                        min = distanta[i];
                        k = i;
                  }
            }
            curent = k;
            perm[k] = 1;
      }

      return distanta[dest];
}

void graf_liste::stergere_nod(nodgraf *&temp,int sursa)
{
      if(temp->info == sursa)
      {
            nodgraf *a = temp;
            temp = a->next;
            delete a;
      }
      else
      {
            stergere_nod(temp->next, sursa);
      }
}

#endif //GRAF_LISTE_H
```

16.3 Traversarea unui graf

Un graf este în esenţă o reţea, care de cele mai multe ori are corespondenţă în lume reală. Cum principala caracteristică a unei reţele este mobilitatea continuă din interiorul său, se pune problema parcurgerii grafului. În cazul altor structuri de date, vectori, liste şi chiar arbori lucrurile sunt clare: se pornea de la un capăt şi trecându-se de la un element la următorul se parcurgea integral structura fără ca un element să fie vizitat de mai multe ori.

Graful fiind o structură de date mai generală în care nodurile au mai mult de un predecesor se pune deci problema trecerii o singură dată prin fiecare nod. Pentru a complica mai mult problema se ia în considerare şi faptul că oricare nod al grafului este un posibil punct de start al traversării,

lucru care demonstrează că aceasta nu este unică, rezultatele variind de la caza la caz.

Evitarea revenirii într-un nod vizitat se face asociind acestuia o etichetă care să indice acest lucru. Metodele de traversare a grafului sunt bazate pe acest principiu, deosebindu-le doar modul în care stochează şi revin asupra unor direcţii necercetate.

Cele două metode sunt :

- traversarea în adâncime (depth-first traversal);
- traversarea în lăţime (breadth-first traversal).

Pe scurt, cele două traversări sunt asemenea drumului parcurs de exploratori într-o peşteră, numai că în cazul traversării în adâncime avem un singur explorator care se întoarce de fiecare dată când ajunge la un capăt de drum la ultima intersecţie, iar în cazul traversării în lăţime sunt o echipă, fiecare luând-o pe un drum.

Odată traversat graful cu una dintre aceste metode, se creează o pădure acoperitoare pentru el, lucru util pentru punerea în evidenţă a componentelor sale conexe, dar şi un arbore acoperitor. Arborele acoperitor este un subgraf ce conţine nodurile grafului iniţial şi doar atâtea arce încât să fie construit un arbore.

Pentru un graf implementat prin intermediul unei matrice de adiacenţă, ordinul de complexitate al operaţiei de traversare este $O(n^2)$. Graful cu n noduri, are matricea asociată de dimensiune nxn. Rezultă că timpul alocat prelucrării unui nod în vederea găsirii tuturor nodurilor adiacente este $O(n)$; se parcurge linia de n elemente a nodului. Deci pentru n noduri timpul necesar traversării este de $n*O(n) = O(n^2)$. După cum se observă, indicatorul depinde doar de numărul nodurilor, numărul arcelor dintre noduri neavând nici o influenţă.

În cealaltă situaţie, pentru un graf reprezentat cu ajutorul listelor înlănţuite, ordinul de complexitate al operaţiei de traversare este cuprins între $O(n)$ şi $O(n^2)$. Indicatorul depinde aici de numărul de arce al fiecărui nod. Cel mai fericit caz este acela când nu există nici un arc între noduri şi atunci traversarea are ordinul de complexitate minim $O(n)$. Dacă fiecare nod al grafului are arce către toate celelalte n-1 noduri ale grafului, se obţine complexitatea maximă $O(n^2)$.

Procesul de *traversare în adâncime* a unui graf, este unul de tip backtracking, analogic cu traversarea în preordine a unui arbore.

Algoritmul foloseşte în acest scop un vector sau o listă în care pune nodurile vizitate şi o stivă în care sunt puse nodurile adiacente nodului curent. Odată vizitat un nod, traversarea se îndepărtează în adâncime până când ajunge la un capăt de drum.

Fie nodul de start al parcurgerii, nodul notat cu X. Acesta este etichetat ca vizitat şi este trecut în listă. Toate n nodurile adiacente lui X, X_i cu i = 1..n, sunt puse în stiva de aşteptare. Primul nod din stivă, X_1, este verificat dacă nu este în listă, caz în care este vizitat, fiind scos şi pus în listă. În cealaltă situaţie, nodul se afla deja în listă, el este scos din stivă şi este verificat nodul de sub el. Aceşti paşi sunt repetaţi pentru fiecare nod adiacent al lui X_1.

Altfel spus, se pleacă pe drumul dat de primul nod adiacent al nodului de start, primul nod adiacent al nodului deja vizitat şi tot aşa până când se ajunge la un nod al cărui prim nod adiacent a fost vizitat sau nu există fiind un capăt de drum. Atunci se trece la următorul nod adiacent pe care se continuă, dacă este posibil, sau în acest moment, algoritmul se întoarce la

penultimul nod vizitat şi pleacă pe al doilea nod adiacent al acestuia, în condiţiile în care el nu a fost vizitat şi există. Algoritmul se întoarce atâta timp cât există noduri în stivă.

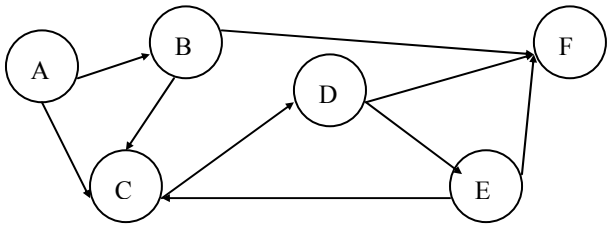


Figura 16.9 *Graf orientat*

Parcurgerea în adâncime a grafului orientat din figura 16.9 presupune parcurgerea etapelor :

1. se alege nodul A ca punct de start al traversării. Nodul A este vizitat şi este trecut în lista nodurilor pe la care s-a trecut. În stivă sunt trecute nodurile către care are arce direcţionate: B şi C;

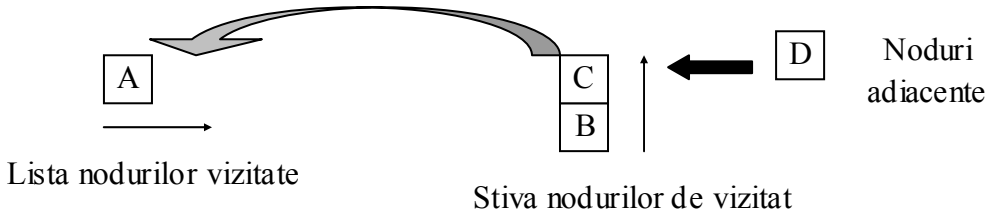


Figura 16.10 *Paşii primei etape*

2. se scoate primul nod din stivă, C, şi cum acesta nu a fost vizitat (nu se afla în lista nodurilor vizitate) este trecut acum în listă. Nodurile sale adiacente, doar D, sunt trecute în stivă;

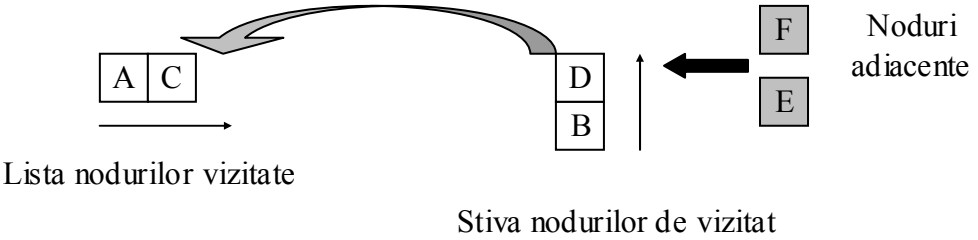


Figura 16.11 *Paşii etapei 2*

3. se scoate nodul D din stivă şi este pus în listă, deoarece nu a fost vizitat. Nodurile sale adiacente, F şi E, se pun în stivă;

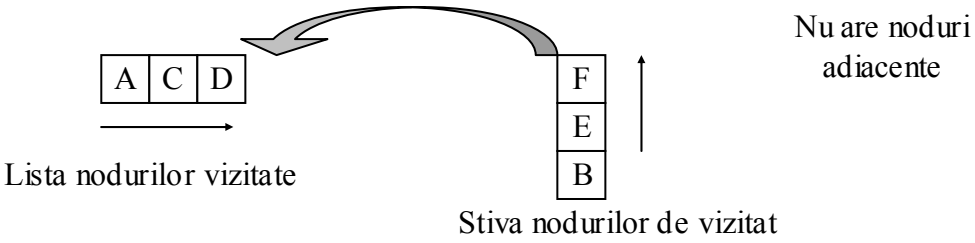


Figura 16.12 *Paşii etapei 3*

4. se scoate nodul din stivă nodul F şi este etichetat ca vizitat. În acest punct se ajunge la un sfârşit de drum şi astfel trece la următorul element din stivă;

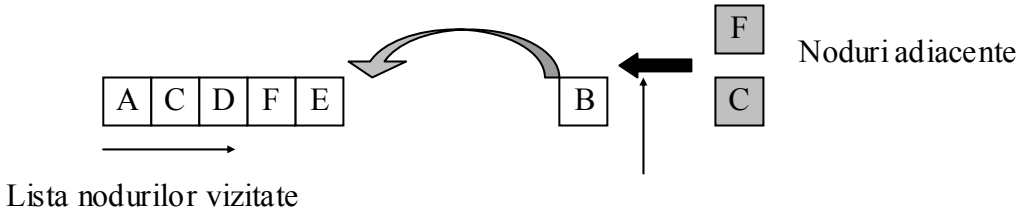


Figura 16.13 *Paşii etapei 4*

5. se vizitează nodul E. Iniţial se trece în stivă nodul adiacent lui E, anume nodul F, dar el este scos apoi din stivă existând, deja în lista nodurilor vizitate. În stivă rămâne doar nodul B;

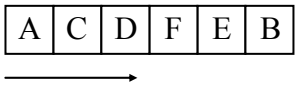


Figura 16.14 *Lista nodurilor vizitate*

6. traversarea este completată prin vizitarea nodului B. Cele două noduri adiacente ale sale, F şi C, sunt trecute în stivă, însă sunt scoase apoi unul câte unul, ele fiind deja însemnate ca vizitate.

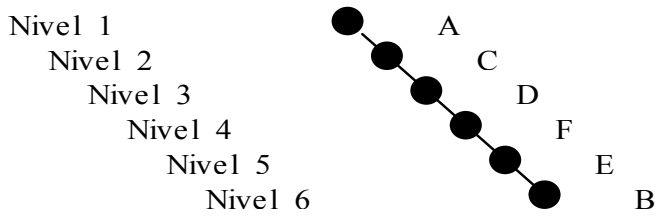


Figura 16.15 *Arbore acoperitor în adâncime pentru graful din figura 16.9*

La scrierea codului sursă, algoritmul se îmbunătăţeşte făcându-se o căutare în listă a nodurilor care se introduc în stivă pentru a se vedea dacă sunt sau nu deja vizitate. Traversarea grafului neorientat nu implică

modificarea în vreun fel a algoritmului, acesta fiind aplicat fără nici o restricţie.

Arborele acoperitor este construit odată cu lista nodurilor vizitate, adăugând un nod la listă se completează şi arborele.

Traversarea grafului, folosind acest procedeu, dă un rezultat diferit pentru un nod de start altul decât A, acest lucru fiind valabil şi pentru arborele acoperitor din figura 16.9.

Procesul de *traversare în lăţime* a unui graf orientat sau neorientat, este analog procesului de traversare în inordine a unui arbore, şi constă în parcurgerea o singură dată a tuturor nodurilor din graf.

Deosebirile de cealaltă metodă constau în folosirea unei cozi de data aceasta pentru a păstra nodurile de verificat, şi în faptul că algoritmul se îndepărtează de nodul vizitat doar după ce a examinat şi vizitat, dacă este posibil, toţi succesorii săi. În schimb şi această metodă este aplicabilă atât grafului orientat cât şi neorienatat dând rezultate ce variază în funcţie de alegerea nodului de pornire.

Parcurgerea în lăţime a grafului din figura 16.9, presupune paşii:

1. se alege nodul A ca punct de start al traversării. Nodul A este vizitat şi este trecut în lista nodurilor pe la care s-a trecut. În coadă sunt trecute nodurile către care are arce (direcţionate sau nedirecţionate, în funcţie de tipul grafului): B şi C;

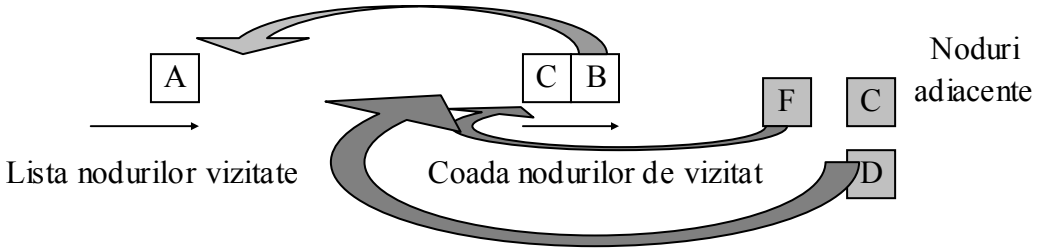


Figura 16.16 *Paşii primei etape*

2. sunt verificate nodurile adiacente şi sunt vizitate dacă nu se află deja în listă. În momentul trecerii în lista nodurilor deja parcurse, nodurile lor adiacente sunt adăugate la coadă, nodul F şi C adiacente lui B şi nodul D adiacent lui C;

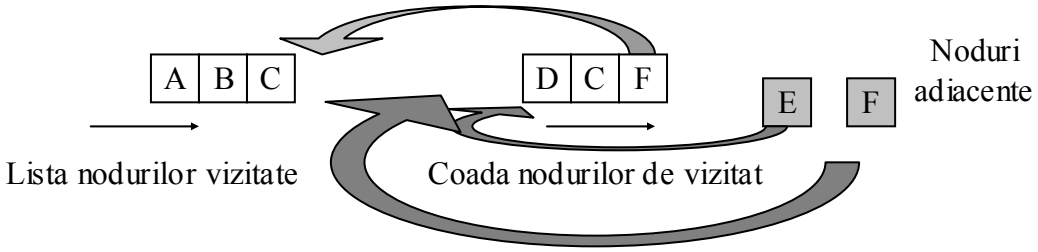


Figura 16.17 *Paşii etapei 2*

3. din coadă trec în lista nodurilor traversate, nodurile F şi D. Nodul C există deja în listă şi doar este scos din coadă. Nodul F nu are adiacenţi şi reprezintă un capăt de drum, nici un nod nefiind adăugat în coadă. În schimb, coada este completată cu nodurile E şi F, care sunt adiacente lui D;

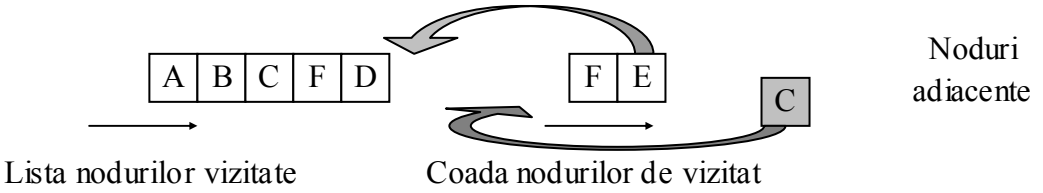


Figura 16.18 *Paşii etapei 3*

4. traversarea este încheiată prin adăugarea nodului E la listă. Nodurile F şi C au fost vizitate şi nu se mai pun iar în listă.

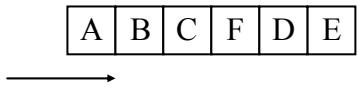


Figura 16.19 *Lista nodurilor vizitate*

Ca şi în cazul metodei precedente, algoritmul se optimizează verificându-se înainte de a fi pus în coada de aşteptare dacă nodul respectiv se află în listă sau nu.

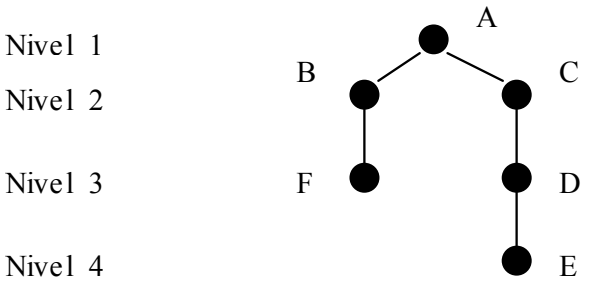


Figura 16.20 *Arbore acoperitor în lăţime pentru graful din figura 16.9*

Arborele acoperitor este construit odată cu lista nodurilor vizitate. Adăugând un nod la listă se completează şi arborele, iar toate nodurile adiacente cu acesta şi care nu au fost vizitate sunt adăugate la arbore ca noduri copii (ulterior devin noduri părinte pentru nodurile adiacente din graf).

În concordanţă cu rezultatul dat de traversarea în lăţime a arborelui, şi forma arborelui acoperitor variază în funcţie de nodul de start.

16.4 Închiderea tranzitivă a grafului

Graful este de cele mai multe ori reprezentarea matematică a problemelor economice şi nu numai, legate de transport, de costul deplasării dintr-un punct în altul, de durata realizării acestuia. Cea mai sugestivă aplicaţie legată de implicarea grafului în rezolvarea acestor probleme este cea a drumului minim, însă de multe ori este necesar să se ştie doar dacă este drum între două noduri. Soluţia acestei probleme este dată de realizarea unei matrice, numită închiderea tranzitivă a matricei de

adiacenţă, care să arate pentru fiecare nod în parte unde se poate ajunge plecând din el.

O modalitate de creare a acesteia este dată de traversarea în adâncime a grafului. Traversând graful din fiecare nod al său, se obţin atâtea liste câte noduri sunt, liste care arată în ce noduri se ajunge din nodul de start. Acestea din urmă se transpun într-o matrice M[n][n], care are elementul m_{ij} =1 dacă există drum de la nodul N_i la nodul N_j şi 0 în rest.

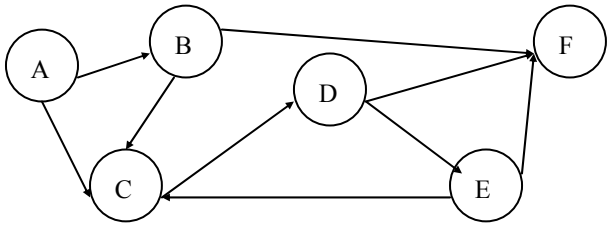


Figura 16.21 *Exemplu de structură de graf*

Pentru graful din figura 16.21 traversarea prin metoda DFS pornind din fiecare nod are ca rezultat următoarele liste :

- pornind din A avem: A, C, D, F, E, B;
- pornind din B avem: B, C, D, F, E;
- pornind din C avem: C, D, F, E;
- pornind din D avem: D, F, E, C;
- pornind din E avem: E, F, C, D;
- pornind din F avem: F.

Matricea închiderii tranzitive obţinute prin intermediul listelor este:

$$MIT = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (16.2)$$

Folosind matricea, creăm graful extins numit închidere tranzitivă. Luăm fiecare nod în parte, iar pe reprezentarea grafului iniţial se desenează săgeţi punctate către nodurile la care se ajunge şi care nu sunt adiacente lui.

Închiderea tranzitivă este prezentată în figura 16.22.

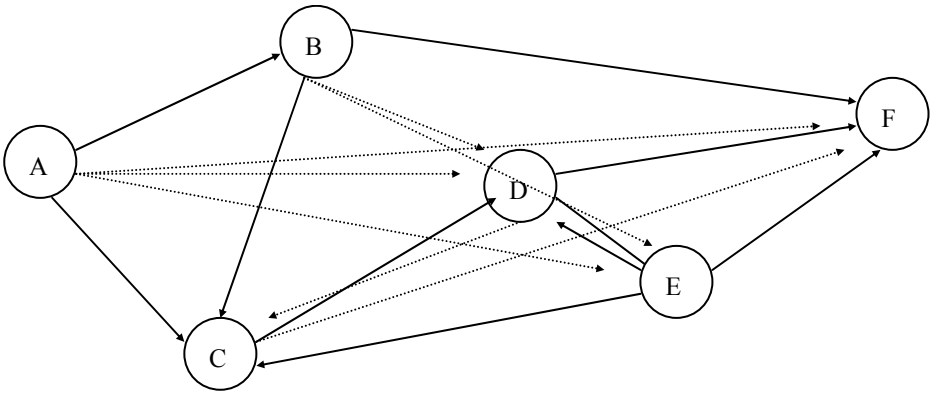


Figura 16.22 *Închiderea tranzitivă a grafului din figura 16.9*

Ordinul de complexitate al acestei operaţii este foarte mare, şi anume $O(n^3)$, pentru că pentru fiecare dintre cele n noduri se aplică traversarea în adâncime, care are complexitatea maximă $O(n^2)$. Deşi este uşor de implementat, pentru un graf foarte mare metoda consumă multe resurse.

O altă soluţie, mai elegantă dar cu acelaşi grad de complexitate, a fost dată de Stephan Warshall. În construirea algoritmului său, el a plecat de la ideea următoare: luând nodurile N_i, N_j, N_k, (cu i, j, k = 1..n şi $i \neq j \neq k$) şi dacă există arcele A_{ik} şi A_{kj} atunci există drum de la nodul N_i la nodul N_j, acest lucru însemnându-se în închiderea tranzitivă a matricei de adiacenţă.

Algoritmul iniţializează matricea închiderii tranzitive cu valoarea matricei de adiacenţă a grafului şi pune valoarea 1 pe diagonala principală (evident există arc de la un nod la el). În urma a trei cicluri după variabilele i, j, k (cu i, j, k = 0 ... n-1), elementele matricei închiderii tranzitive A'[n][n] iau valoarea 1 dacă a'[i][k] = a'[k][j], adică:

$$a'[i][j] = a'[i][k] \;\&\; a'[k][j] \qquad (16.3)$$

În clasa *graf_matrice*, metoda asociată algoritmului este:

```
void graf_matrice::inchidere_tranzitiva()
{
      int i,j,k;
      int a_tranz[MaxLungGraf][MaxLungGraf];

//se creeza matricea închiderii tranzitive inițială plecând de la
//matricea de adiacență a grafului
      for(i = 0; i<nr_noduri; i++)
            for(j = 0; j<nr_noduri; j++)
                  if(i==j)
                        a_tranz[i][j] = 1;
                  else
                        a_tranz[i][j] = matm[i][j];
//se cerceteaza perechi de câte 3 noduri si se formeaza matricea
      for(i = 0; i<nr_noduri; i++)
            for(j = 0; j<nr_noduri; j++)
                  for(k = 0; k<nr_noduri; k++)
                        a_tranz[i][j] = a_tranz[i][k] && a_tranz[k][j];
//se afiseaza matricea de adiacență a grafului
      for(i = 0; i<nr_noduri; i++)
      {
```

```
            for(j = 0; j<nr_noduri; j++)
                  cout << a_tranz[i][j] << " ";
            cout << endl;
      }
}
```

16.5 Problema drumului de lungime minimă în graf

Revenim la problema iniţială, cea a parcurgerii traseului Arad – Bucureşti având cel mai mic cost. Soluţia constă în găsirea acelor oraşe prin care trece traseul astfel încât suma distanţelor parcurse să fie cea mai mică, în raport cu lungimea altor posibile drumuri de parcurs. La nivelul grafului în loc de oraşe, distanţe avem noduri şi arce cu greutate.

Parţial, problema este rezolvată deoarece folosind cele două metode de traversare a unui graf avem capacitatea de a afla ce noduri se află pe trase şi astfel putem forma o serie de drumuri de urmat. Nu mai rămâne decât să vedem în cazul grafului cu greutate care drum are suma valorilor arcelor minimă sau în cazul grafului fără greutate care drum are mai puţine arce. Deşi această soluţia este simplu de implementat, ea este mare consumatoare de resurse în cazul unui graf mare, aşa că ne trebuie un program care să combine cele două etape, reducând traversările repetate ale grafului la una.

Acest lucru este făcut de algoritmul Dijkstra, care examinează toate drumurile ce pornesc din nodul curent, actualizând distanţele dintre el şi celelalte noduri. Pentru a păstra nodurile prin care trece drumul cel mai scurt, programul le reţine într-o listă pe care o notăm cu L. În final lista conţine mulţimea minimă de noduri care să le conţină pe toate cele care vor forma efectiv drumul optim.

Nodurile care se adaugă în această listă sunt acele noduri ale grafului la care se ajunge prin arce directe doar de la nodurile din lista L (ele reprezintă nodurile adiacente celor din L) şi care au lungimea cumulată până în acel moment minimă.

Pentru a nu calcula de fiecare dată distanţa minimă până în acel nod, ea este atribuită ca informaţie nodului, asemenea unei etichete. Ele sunt implementate utilizând un vector de lungime *n*, unde *n* este numărul de noduri al grafului (*vector[i]* reţine eticheta nodului *i*), sau creând o listă cu n elemente de tip etichetă.

Drumul minim este găsit în momentul în care în lista L se află nodul destinaţie.

Algoritmul constă în paşii:

Pasul 1. Se construieşte lista L şi elementele vectorului/listei distanţelor sunt iniţializate cu o valoare foarte mare;

Pasul 2. Se alege nodul sursă, el devenind şi primul nod pus în lista L. Valoarea etichetei corespunzătoare lui ia valoarea 0;

Pasul 3. Se repetă până când nodul destinaţie se află în L. Sunt analizate nodurile grafului care nu sunt în lista L.

Dacă există noduri N_i în care se poate ajunge prin arce directe de la noduri din L, se calculează distanţa de la nodul de start până la ele:

$$distanţa(N_i)=min(val\_etichetă(N_i),val\_etichetă(N_k)+greutate\_arc(N_k,N_i)) \quad (16.4)$$

unde:

- val_etichetă(N_i) – reprezintă valoarea etichetei asociată nodului N_i;
- greutate_arc(N_k,N_i)) – reprezintă valoarea arcului dintre nodurile N_k şi N_i;
- N_k – este un nod din lista L de la care se ajunge prin arc direct la nodul N_i care nu se află în listă.

Se adaugă la lista L acel nod N_i care are distanţa(N_i) obţinută minimă. Pentru el ca şi pentru celelalte noduri pentru care s-au calculat distanţele se reactualizează etichetele:

$$val\_etichetă(N_i) = min(val\_etichetă(N_i), distanţa(N_i)) \qquad (16.5)$$

Dacă nu există nici un nod de acest tip atunci nu există nici un drum până la destinaţie.

Pasul 4. Dacă nodul destinaţie se află în lista L, atunci valoarea etichetei sale reprezintă drumul de lungime minimă. Pentru găsirea acestui drum, se porneşte înapoi de la nodul final şi folosind nodurile din L.

Pentru a nu pierde timp la Pasul 4 reconstituind drumul, aşa cum s-a ataşat fiecărui nod o etichetă, i se asociază o listă în care sunt memorate nodurile precedente care au dat valoarea etichetei în acel moment. Nodul nou introdus în lista L, iniţializează lista drumului deja parcurs cu valorile din lista predecesorului său direct şi apoi îl adaugă şi pe acesta.

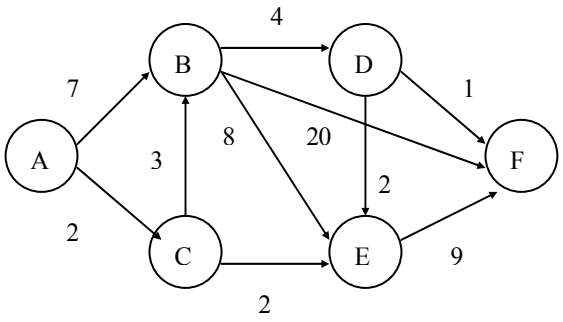


Figura 16.23 *Graf orientat cu greutate*

Pentru a exemplifica metoda, se aplică algoritmul lui Dijkstra pentru a calcula drumul minim de la nodul A la nodul F, noduri ce aparţin grafului din figura 16.23.

Se fac notaţiile ajutătoare:

- $E(N_i)$ – valoarea etichetei nodului N_i;
- $L(N_i)$ – lista nodurilor prin care s-a ajuns la nodul N_i;
- L – lista nodurilor care au fost luate în considerare.

Etapele parcurse sunt :

- se iniţializează eticheta nodului A cu valoarea 0, E(A) = 0, iar pentru celelalte noduri cu o valoare foarte mare, E(B) = E(C) = E(D) = E(E) = E(F) = ∞. Se pune nodul A în lista L;
- se calculează valoarea etichetei vecinilor nodului A, E(B) = 7 şi E(C) = 2. Cum nodul C are valoarea etichetei minimă şi nu se află în lista L, el este adăugat la aceasta. În lista nodurilor precedente lui C, L(C), se pune nodul A, iar L = { A, C };

- se calculează valoarea etichetelor vecinilor nodului C, E(B) = 5 şi E(E) = 4. Vechea valoare al lui E(B), care este 7, este înlocuită de noua valoare calculată, aceasta din urmă fiind mai mică. Cum nodul E are eticheta minimă şi nu se află în lista L, este adăugat la aceasta şi L(E) = { A, C }, iar L = { A, C, E };
- se calculează etichetele pentru nodul F care este vecinul direct al nodului E, E(F)=13. Dintre toate etichetele, cea a nodului are valoarea minimă, 5, şi cum el nu este în L, este pus în această listă, deci L = {A, C, E, B}. În lista predecesorilor săi sunt puşi predecesorii nodului de la care s-a ajuns la B, adică ai nodului C, şi acesta din urmă, L(B) = {A, C};

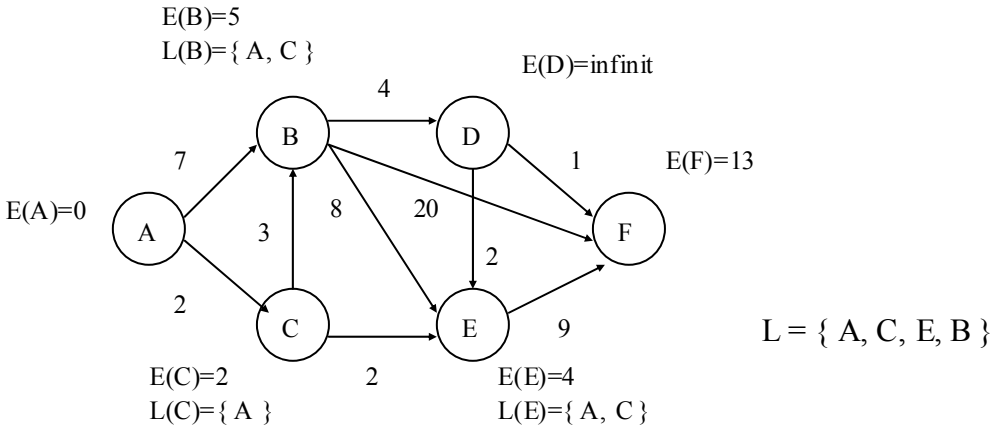


Figura 16.24 *Pas intermediar*

- se calculează valoarea etichetelor vecinilor nodului B, E(D) = 9, E(E) = 13 şi E(F) = = 25. În cazul nodurilor E şi F etichetele îşi păstrează vechile valori, care sunt mai mici. Cu toate că în acest moment nodul E are eticheta cu valoare minimă, el nu este ales ca fiind următorul nod al drumului pentru că se află deja în lista L. Deci nodul care se traversează este D, iar L(D) = {A, C, B} şi L = {A, C, E, B, D};
- se calculează valoarea etichetei pentru nodurile E şi F (sunt noduri adiacente directe pentru nodul D), E(E) = 11 şi E(F) = 10. Pentru nodul E valoarea etichetei nu este înlocuită cu cea nouă. Nodul care nu se află în lista L şi care are valoarea etichetei minimă este F. Este adăugat la listă şi în acest moment căutarea ia sfârşit. Drumul minim este A – C – B – D – F.

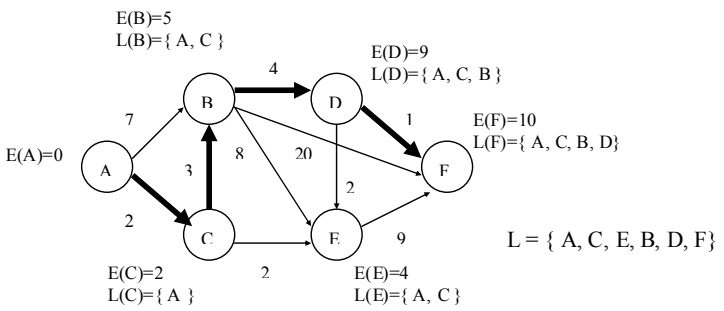


Figura 16.25 *Drumul minim*

Afişarea drumului minim şi a lungimii sale folosind funcţia din clasa *graf_liste* are codul sursă:

```
int lungime = x.drum_minim(g,start,stop,precede);
if(test != MAXINT)
{
      cout<<endl<<"Drum minim "<<stop<<"<-"<<start<<" are lungimea :
"<<lungime<<" si este : ";
     for(int i=stop;i>0&&(i !=MAXINT);i = precede[i])
          cout<<i<<"<-";
}
else
        cout<<"Nu exista drum de la nodul "<<start<<" la nodul
"<<stop;
```

16.6 Operaţii de concatenare şi deconcatenare cu grafuri

Se defineşte concatenarea a doua grafuri ca fiind operaţia de adăugare a altui graf la unul din nodurile grafului iniţial respectând următoarele reguli:

- nodul capăt al grafului al doilea se va lega printr-un arc de un nod al primului graf; arcul este orientat pe direcţia nod graf 1 ->capăt graf 2;
- se introduce de la tastatură informaţia nodului unde se va adăuga graful al doilea;
- dacă nodul capăt al grafului 2 este nod în graful 1 şi concatenarea se face în acel nod nu se va mai face nici un arc, completându-se lista de arce a nodului din graful 1 cu arcele nodului din graful 2;
- dacă nodul final al grafurilor diferă, atunci între nodul final al grafului 2 şi cel al primului graf se va forma un arc a cărui informaţie se va citi de la tastatură;
- dacă al doilea graf se leagă la nodul final al grafului 1 atunci nodul final al grafului 2 devine nod final al noului graf obţinut prin concatenarea celor două grafuri;
- dacă în graful al doilea se află minim două noduri care se află şi în primul graf, o condiţie esenţială de a face concatenarea celor două grafuri este că, dacă există în ambele grafuri arc în acelaşi sens între cele două noduri, acesta să aibă aceeaşi *greutate*.

Funcţia care verifică această ultima condiţie este următoarea:

```
int graf_liste::verificare(nodgraf *cap,nodgraf *cap2)
{
nodgraf *p,*q,*aux1,*aux2;
int k=1;
for(p=cap;p!=NULL;p=p->next)
   for(q=cap2;q!=NULL;q=q->next)
   {
       if (q->info==p->info)
      for(aux1=cap;aux1!=NULL;aux1=aux1->next)
/*nodurile comune le compar 2 câte 2 să văd care este greutatea
arcului dacă există vreunul*/
         for(aux2=q->next;aux2!=NULL;aux2=aux2->next)
              {
```

```
            if(aux2->info==aux1->info)
            if (verif_arc(p,aux1)!=verif_arc(q,aux2))
            {    //dacă au greutate diferită atunci
                  printf("\n Nu se poate face concatenarea !");
                  k=0;
                  return k;
            }
             }
    }
return k;
}
```

Funcţia care realizează concatenarea celor două grafuri primeşte ca date de intrare doi pointeri la capetele celor două grafuri şi returnează *0* dacă nu se poate face concatenarea şi *1* dacă a reuşit. Funcţia respectând condiţiile de mai sus, adaugă la lista nodurilor grafului 1 şi nodurile care nu sunt comune ale grafului 2, iar listele de arce ale acestor noduri sunt şi ele copiate. În cazul nodurilor comune, listele arcelor sunt doar completate cu arce noi. Există şi cazuri când este nevoie să se creeze arce noi (când se leagă de exemplu nodurile finale ale grafurilor) iar atunci *greutatea* lor este citită de la tastatură.

Funcţia este:

```
int graf_liste::concatenaregraf(nodgraf *cap,nodgraf *cap2)
{
int k;
nodgraf *p,*q,*aux,*ultim1,*ultim2;
arc *r;
//verifica dacă se poate face concatenarea
if(verificare(cap,cap2)==0) return 0;
printf("\n La ce nod are loc concatenarea ?");
/*se introduce nodul unde se face concatenarea şi se verifică dacă el
există în primul graf*/
k=citire();

ultim1=cauta_nodgraf_final(cap);
// memorez ultimul nod al grafului 1
ultim2=cauta_nodgraf_final(cap2);
//memorez ultimul nod al grafului 2

/*se inserează în lista nodurilor primului graf nodurile necomune din
al doilea graf*/
for(q=cap2;q!=NULL;q=q->next)
{
      p=cauta_nodgraf(cap,q->info);
      if(p==NULL) cap=ins_nodgraf(cap,q->info);
}
/* se copiază pentru noduri şi lista arcelor, iar pentru acele noduri
care existau se completează această listă*/
for(q=cap2;q!=NULL;q=q->next)
{
      p=cauta_nodgraf(cap,q->info);
      if(p!=NULL) for(r=q->capat;r!=NULL;r=r->next_arc)
/*functia ins_arc(nod cap,int sursă,int destinație,int greutate)
inserează un nou arc catre nodul cu informația destinație de greutate
greutate în lista arcelor nodului cu informația sursă din graful cu
capăt cap */
      ins_arc(cap,p->info,r->destinatie->info,r->weight);
}
```

```
/* dacă nodul unde se face concatenarea nu este capăt al grafului 2
atunci se face arc între cele două cu citirea informației de la
tastatură*/
if(cap2->info!=aux->info)
{
printf("\n Distanta dintre nodul ales si capatul grafului 2 este ?");
k=citire();
ins_arc(cap,aux->info,cap2->info,k);
}
/* dacă al doilea graf nu se leagă la nodul final al primului graf şi
dacă nodurile finale nu sunt aceleaşi, atunci ele se leagă printr-un
arc */
if(aux->info!=ultim1->info)
    if(ultim1->info!=ultim2->info)
        if(cauta_nodgraf(cap,ultim2->info)==NULL)
/* funcția cauta_nodgraf(nod * cap,int k) cauta un nod cu informația k
în graful cu capăt cap, returnând adresa nodului sau NULL */
       {
               printf("\n Distanta dintre nodul final al grafului 2 si
cel al lui 1 este ?");
        k=citire();
//se crează arc intre nod final al grafului 1 si cel al grafului 2
       ins_arc(cap,ultim2->info,ultim1->info,k);
       }
return 1;
}
```

Pentru a exemplifica procesul de concatenare a două grafuri luăm grafurile din figura 16.26.

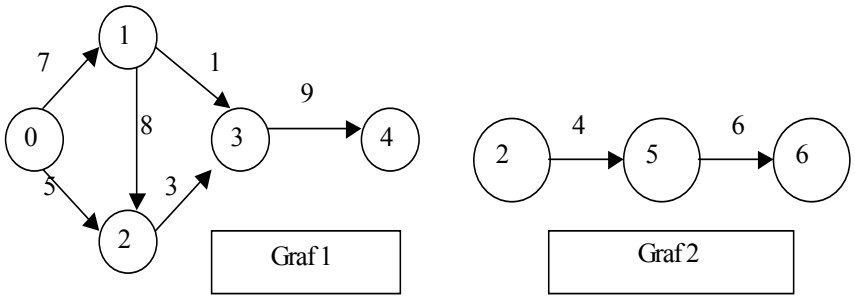


Figura 16.26 *Exemple de grafuri*

Dacă se doreşte concatenarea grafului 2 la graful 1 în nodul cu valoare 2 atunci graful care va rezulta va fi:

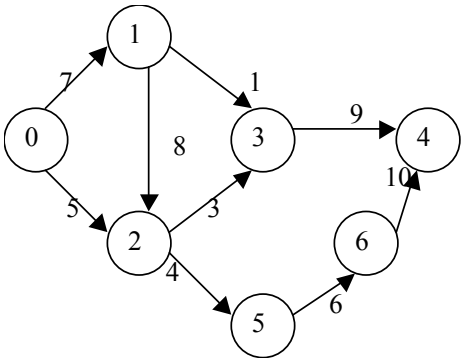


Figura 16.27 *Concatenarea a două grafuri*

Informaţia arcului dintre nodul cu informaţia 6 şi cel cu informaţia 4 a fost introdusă de la tastatură fiind ceruta de funcţia de concatenare.

Deconcatenarea unui graf este operaţia de *rupere* a acestui graf în două grafuri diferite din punct de vedere al nodurilor care le formează şi al arcelor ce le leagă.

Funcţia care realizează deconcatenarea grafului primeşte ca date de intrare pointer la capătul grafului de deconcatenat şi ea va returna în pointer la capătul celui de-al doilea graf.

Pentru a exemplifica deconcatenarea se consideră graful din figura 16.28.

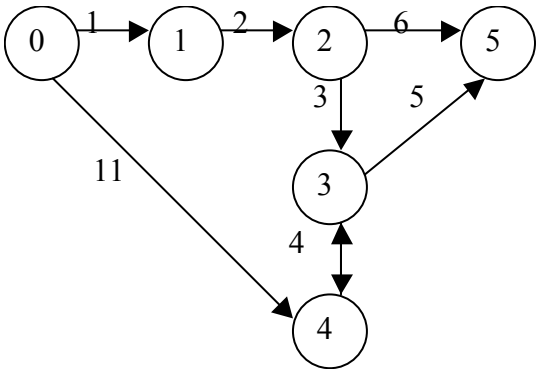


Figura 16.28. *Deconcatenarea unui graf*

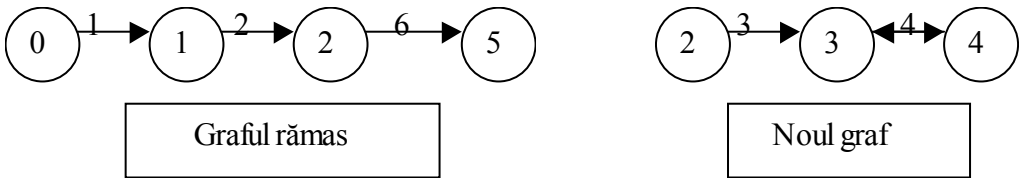


Figura 16.29. *Deconcatenarea unui graf*

În momentul lansării în execuţie se cere să se introducă informaţia nodurilor care formează al doilea graf. Astfel se va crea lista nodurilor noului graf care se va obţine. Primul lucru care se face după aceasta este crearea listelor de arce pentru aceste noduri. Pentru fiecare nod al noului graf se verifică dacă el are arce cu nodurile acestui nou graf cu ajutorul funcţiilor:

```
/*funcția verifică dacă nodul referit prin *s are arc către nodul
referit prin *d, şi întoarce ca rezultat greutatea arcului sau 0 dacă
nu este arc*/

int graf_liste::verif_arc(nodgraf *s,nodgraf *d)
{
arc * p,*aux;
int gasit=0;
for(p=s->capat;p!=NULL;p=p->next_arc)
   if(p->destinatie==d) {
      gasit=1;
      aux=p;
      return aux->weight;
    }

if (gasit==0) {
       return 0;
}
else return aux->weight;
}

/* funcția primeşte ca date de intrare pointer la capătul grafului şi
informația nodului pe care îl caută şi va returna adresa nodului
căutat sau NULL*/

nodgraf *cauta_nodgraf(nodgraf * cap,int info)
{
nodgraf *p,*q;
int gasit=0;
for(p=cap;p!=NULL;p=p->next)
 if(p->info==info) {
      gasit=1;
      q=p;
  }
if(gasit==0){
      return NULL;
}
  else return q;
}
```

Odată creată lista nodurilor noului graf şi a listelor de arce asociate acestora, se va reactualiza lista nodurilor şi a arcelor grafului iniţial. Pentru a realiza acest lucru trebuie respectate următoarele:

- nodurile celui de-al doilea graf care se formează şi care sunt *sursă* sau *destinaţie* în arce numai cu noduri care formează şi ele al doilea graf, sunt şterse dintre nodurile grafului iniţial;
- nodurile grafului care se formează şi care sunt şi *sursă* şi *destinaţie* în arce cu noduri care nu intră în al doilea graf rămân în primul graf, dar se şterg arcele cu nodurile care nu rămân; funcţia care verifică dacă un nod respectă această condiţie sau nu este:

```
/* funcția primeşte ca date de intrare referințe la capătul grafului
inițial, la al celui nou şi la nodul care se verifică ; ea va returna
0 dacă nu are legătură cu noduri care rămân în graful inițial şi o
valoare diferită de 0 în caz contrar*/

int graf_liste::verif_stergere(nodgraf *cap,nodgraf *cap2,nodgraf *q)
{
```

```
nodgraf *p,*z;
int k=0;
int vb=0;
for(p=cap;p!=NULL;p=p->next)
{
     if(cauta_nodgraf(cap2,p->info)==NULL)
    {
        z=cauta_nodgraf(cap,q->info);
        if(z!=NULL) k=verif_arc(p,z);
        if(k!=0) vb=k;
   }
}
return vb;
}
```

Funcţiile care realizează deconcatenarea unui graf sunt:

```
/* functia se apelează dupa funcția deconcatengraf şi are ca scop
reactualizarea listei de noduri a grafului inițial primeste ca date de
intrare referinta la capetele celor doua grafuri*/

void graf_liste::stergerenodgrafuri(nodgraf *&cap,nodgraf *cap2)
{
int t;
nodgraf *p,*q,*z,w;
for(p=cap;p!=NULL;p=p->next)
    for(q=cap2;q!=NULL;q=q->next)
    {
    z=cauta_nodgraf(cap,q->info);
    if(z!=NULL)
      if(z->capat==NULL) sterg_arc(p,z);
        else
        {
             t=verif_stergere(cap,cap2,q);
             if(t==0){
                         z->capat=NULL;
            }
         }
    }
for(q=cap2;q!=NULL;q=q->next)
 {
    p=cauta_nodgraf(cap,q->info);
    if(p!=NULL)
       if(p->capat==NULL) cap=sterg_nodgraf(cap,p->info);
}
}
```

```
/* funcția realizează deconcatenarea grafului inițial creând un nou
graf
primeşte ca date de intrare referinta la capătul primului graf şi
întoarce adresa capătului noului graf*/

nodgraf * graf_liste::deconcatengraf(nodgraf * &cap)
{
nodgraf *cap2,*p,*q,*w,*z;
arc *r;
int k,nd,arc;
cap2=NULL;

/*citeşte de la tastatură informația nodului capăt al noului graf*/
```

```
printf("\n Nodul capat al grafului al 2-lea este :");
k=citire2(&nd,cap,MAX);

/*o dată citită informația este creat şi inserat un nod cu această
informație în lista de noduri */
cap2=ins_nodgraf(cap2,nd);
if(k==0)
{
      printf("\n Urmatoarele nodgrafuri sunt:");
      k=citire2(&nd,cap,MAX);
      cap2=ins_nodgraf(cap2,nd);
 //se creează şi celelalte noduri
      while(k==0)
      {
            k=citire2(&nd,cap,MAX);
            cap2=ins_nodgraf(cap2,nd);
      }
}
/* în secvența următoare se creează listele de arce ale nodurilor
noului graf ; se parcurge lista nodurilor inițiale verificându-se care
se află în noul graf ; dacă se găseşte un astfel de nod se verifică
dacă are arce cu alte noduri ale noului graf ; când se găsesc aceste
arce, ele se scriu în noul graf şi se şterg din graful inițial din
listele acelor noduri*/
for(p=cap2;p!=NULL;p=p->next)
{
      q=cauta_nodgraf(cap,p->info);
           //caută echivalentul lui în graful  inițial*/
      for(z=cap2;z!=NULL;z=z->next)
      {
           w=cauta_nodgraf(cap,z->info);
           arc=verif_arc(q,w);    /* funcția verifică dacă există arc
intre nodurile cu adresele q şi w*/
           if(arc!=0) {    /*dacă se găseşte arc se scrie în graful
nou şi se şterge de aici*/
                ins_arc(cap2,q->info,w->info,arc);
                sterg_arc(q,w);
                }
      }
}
return cap2;//returnează adresa nodului capăt a noului graf }
```