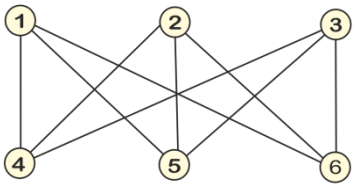
Sergiu CORLAT **Andrei CORLAT**

GRAFURI

Noţiuni. Algoritmi. Implementări



Chişinău, 2012

Aprobată pentru editare de Senatul Universităţii Academiei de Ştiinţe a Moldovei

Lucrarea este destinată studenţilor facultăţilor de matematică şi informatică a instituţiilor de învăţământ superior din republică, care studiază cursul de teorie a grafurilor. Este de asemenea utilă elevilor şi profesorilor de informatică pentru pregătirea către concursurile naţionale şi internaţionale de programare.

Autori:

Sergiu Corlat, lector superior, UnAŞM
Andrei Corlat, dr. conferenţiar, UnAŞM

Recenzenţi:

Sergiu Cataranciuc, dr. conferenţiar, USM
Andrei Braicov, dr. conferenţiar, UST

Cuprins

Introducere

Apărute din necesitatea de a modela diverse situații, relații sau fenomene în formă grafică, grafurile şi-au găsit o multitudine de aplicații în cele mai diverse sfere ale activității umane: construcții şi sociologie, electrotehnică şi politologie, chimie şi geografie … acest şir poate fi continuat la nesfârşit.

Teoria grafurilor a luat naştere de la *problema podurilor din Konigsberg*, cercetată de Euler şi s-a dezvoltat ca un compartiment al matematicii clasice până la momentul apariției sistemelor electronice de calcul şi a teoriei algoritmilor. În contextul rezolvării problemelor de calcul automat, grafurile s-au dovedit a fi un instrument universal şi extrem de flexibil, devenind un compartiment al matematicii aplicate.

O adevărată revoluție a cunoscut-o teoria grafurilor în anii 60 – 80 ai secolului trecut, când a fost stabilită posibilitatea de utilizare a lor pentru rezolvarea problemelor de optimizare. Algoritmii pentru determinarea drumului minim, punctelor mediane, centrelor, de maximizare a fluxurilor, dar şi multe altele au devenit componente vitale ale cercetărilor operaționale şi a metodelor de optimizare.

În aspect informatic grafurile apar şi în calitate de structuri eficiente de date, în special *arborii*, care permit realizarea optimă a algoritmilor de sortare şi căutare.

Numărul de lucrări, care studiază aspectele algoritmice ale teoriei grafurilor este unul impunător. Printre primele apariții se numără *Applied graph theory* de Wai-Kai Chen; *Graph Theory. An Algorithmic approach* de Nicos Christofides; urmate de *Algorithmic ghaph theory* de Alan Gibbons, *Algorithms in C* de Thomas Sedgewick şi multe alte ediții. Totuşi, majoritatea edițiilor se axează doar pe descrierea matematică a algoritmilor, fără a le suplini prin implementări într-un limbaj de programare sau altul, or, tocmai implementarea algoritmului este componenta de importanță maximă pentru programatorii practicieni.

În prezenta lucrare se încearcă abordarea „verticală” a algoritmilor clasici ai teoriei grafurilor: de la noțiuni teoretice, definiții

şi teoreme, către descrieri matematice a algoritmilor, urmate de implementări integrale sau parțiale (fără prezentarea subprogramelor auxiliare) în limbajul de programare C, însoțite şi de prezentarea şi analiza rezultatelor.

Un motiv al prezentării parțiale a implementărilor algoritmilor este concepția de *manual* a lucrării: restabilirea părților lipsă a programelor devine un exercițiu practic pentru toți cei care studiază cursul de „Teorie a grafurilor" în instituțiile de învățământ superior din republică.

Ediția este destinată nu doar studenților facultăților de profil, dar şi elevilor pasionați de informatică, precum şi profesorilor, care au în grija lor pregătirea de performanță în domeniul Informaticii – teoria grafurilor este inclusă în calitate de compartiment obligatoriu în curriculumul internațional de performanță pentru Computer Science. Exercițiile, cu care finalizează fiecare capitol, pot fi folosite în calitate de lucrări de laborator – rezolvarea lor presupune prezența cunoştințelor teoretice dar şi a competențelor practice de programare.

Venim şi cu o recomandare pentru instrumentele informatice, care pot fi utilizate pentru rezolvarea exercițiilor propuse la finele fiecărui capitol: cele mai „prietenoase" medii de programare s-au dovedit a fi compilatoarele Dev C++ şi MinGW Developer Studio – ambele produse în distribuție liberă.

Sperăm că ediția va deveni nu doar un suport teoretic eficient, dar şi unul aplicativ pentru toți cei care studiază elementele de programare şi cursurile de teorie a grafurilor.

În final ținem să aducem sincere mulțumiri academicianului Petru Soltan, care cu ani în urmă ne-a dus cursul de teorie a grafurilor la catedra de Cibernetică Matematică a Universității de Stat din Moldova; colaboratorilor catedrelor de profil de la Universitatea de Stat din Moldova, Universitatea de Stat Tiraspol şi Universitatea Academiei de Ştiințe a Republicii Moldova, care au adus un aport considerabil la redactarea şi recenzarea ediției.

30 noiembrie 2011 *Autorii*

Capitolul 1. Noţiuni generale

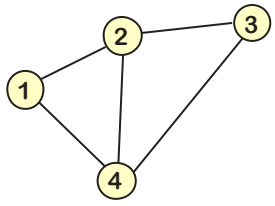
În acest capitol:

- Grafuri. Vârfuri, muchii.
- Grad al vârfului, vecini
- Căi, cicluri
- Subgrafuri
- Grafuri orientate, planare, conexe
- Metode de reprezentare a grafurilor: matricea de adiacenţă, matricea de incidenţă, lista de muchii, lista de vecini

1.1 Definiţii

Def. Graf neorientat (graf) – o pereche arbitrară $G=(V,E)$ $E\subseteq\{\{u,v\}:u,v\in V \,\&\, u\neq v\}$

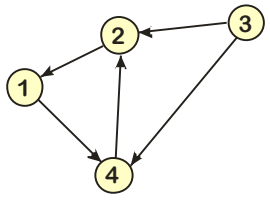
Def. Graf orientat (graf) – o pereche arbitrară $G=(V,E)$, în care $E\subseteq V\times V$



Des 1.1. Graf neorientat

V formează mulțimea vârfurilor grafului, ***E*** – mulțimea muchiilor. De obicei vârfurile grafului sunt reprezentate în plan prin puncte sau cercuri iar muchiile – prin segmente care unesc vârfurile.

Pentru graful din desenul 1.1 $V=\{1,2,3,4\}$, $E=\{\{1,2\},\{2,3\},\{3,4\},\{1,4\},\{2,4\}\}$



Des 1.2 Graf orientat

Într-un graf orientat muchiile[1] se reprezintă prin săgeți, care indică direcția de deplasare pe muchie. Pentru graful orientat din desenul 1.2 $V=\{1,2,3,4\}$, $E=\{(1,4),(2,1),(3,2),(3,4),(4,2)\}$.

Muchia (u,v) este *incidentă* vârfurilor u,v, iar acestea sunt *adiacente* muchiei.

Grade

Def. *Gradul* vârfului v, $d(v)$ este numărul de muchii, incidente acestuia[2]. Un vârf este *izolat*, dacă gradul lui este 0.

Mulțimea de *vecini* ai vârfului v_i, $\Gamma(v_i)$ este formată din vârfurile adiacente la v_i. Într-un graf orientat mulțimea vecinilor este formată din două componente distincte: $\Gamma(v_i)=\Gamma^+(v_i)\bigcup\Gamma^-(v_i)$.

$\Gamma^+(v_i)$ este formată din vârfurile arcelor cu originea în v_i. $\Gamma^-(v_i)$ este formată din vârfurile-origine ale arcelor care se termină în v_i. Pentru vârful 4 din graful reprezentat pe desenul 1.2 $\Gamma^+(4)=\{2\}$, $\Gamma^-(4)=\{1,3\}$.

Căi şi cicluri

Def. *Cale* în graf este o consecutivitate de vârfuri $v_1,v_2,...v_k$ astfel încât $\forall i=1,...,k-1$ vârfurile v_i,v_{i+1} sunt adiacente[3]. Dacă toate vârfurile $v_1,v_2,...v_k$ sunt distincte, calea se numeşte *elementară*. Dacă $v_1=v_k$ atunci $v_1,v_2,...v_k$ formează un *ciclu* în G. Ciclul este *elementar*, dacă $v_1,v_2,...v_{k-1}$ sunt distincte.

[1] Muchia în graful orientat se numeşte şi *arc*.
[2] Pentru vârfurile din grafuri orientate se definesc *semigrade de intrare* (numărul de muchii, care intră în vârf) şi *semigrade de ieşire* (numărul de muchii cu originea în vârful dat)
[3] Există muchia (v_i,v_{i+1})

Pe desenul 1.1 secvența de vârfuri 1,2,4 formează o cale; secvența 1,2,4,1 – un ciclu elementar.

În grafurile orientate noțiunea de cale este substituită prin *lanț*.

Def. Lanțul este o consecutivitate de muchii (arce) luate astfel, încât vârful arcului (*i*) coincide cu originea arcului (*i*+1).

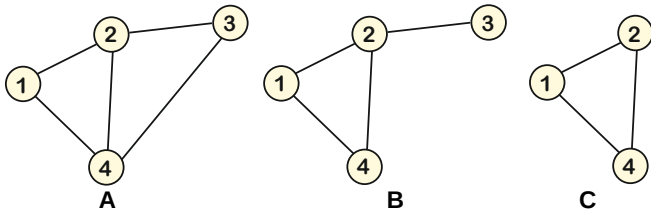
Pe desenul 1.2 secvența de arce (3,4)(4,2)(2,1) formează un lanț; secvența (1,4)(4,2)(2,1) – un ciclu elementar.

Subgrafuri

Def. *Subgraf* al grafului $G=(V,E)$ se numeşte orice graf $G'=(V',E')$ astfel încât $V'\subseteq V, E'\subseteq E$.

Subgrafurile se obțin din graful inițial fie prin excluderea muchiilor, fie prin excluderea muchiilor şi vârfurilor din graful inițial. În cazul când din graful inițial se exclude vârful v_i, odată cu el se exclud şi toate muchiile incidente acestuia. Excluderea muchiei (u,v) nu presupune şi excluderea din graf a vârfurilor u,v.

Dacă în subgraful $G'=(V',E')$ are loc relația $V'=V$, subgraful este unul *bazic* (desenul 1.3 B). Subgraful $G_S^{'}(V_S,E_S)$ în care $V_S\subset V, E_S\subset E$ se numeşte *subgraf generat* dacă pentru orice $v_i\in V_S, \Gamma(v_i)=\Gamma(v_i)\bigcap X_S$ (desenul 1.3 C). Cu alte cuvinte, $G_S^{'}$ este format dintr-o submulțime V_S a vârfurilor din G împreună cu muchiile, care au ambele extremități în V_S.



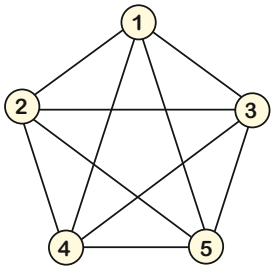
Des 1.3 Graful inițial (A), subgraf bazic (B), subgraf generat (C).

Tipologia grafurilor

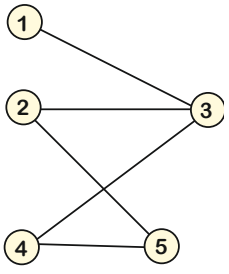
Def. Graful $G=(V,E)$ se numeşte *complet* dacă între oricare pereche de vârfuri $v_i, v_j \in V$ există cel puţin o muchie, care le uneşte. Un graf complet cu n vârfuri se notează K_n. (desenul 1.4 A)

Def. Graful orientat $G=(V,E)$ este *simetric*, dacă pentru orice arc $(v_i, v_j) \in E$ există şi arcul $(v_j, v_i) \in E$. Graful orientat $G=(V,E)$ este *antisimetric*, dacă pentru orice arc $(v_i, v_j) \in E$ arcul (v_j, v_i) nu există.

Def. Graful neorientat $G=(V,E)$ este *bipartit*, dacă mulţimea V a vârfurilor lui poate fi divizată în două submulţimi distincte V^A şi V^B, astfel încât orice muchie din E are începutul în V^A, iar sfârşitul în V^B. (desenul 1.4 B) Graful orientat $G=(V,E)$ este *bipartit*, dacă mulţimea V a vârfurilor lui poate fi divizată în două submulţimi distincte V^A şi V^B, astfel încât orice arc din E are originea în V^A, iar sfârşitul în V^B.



A



B

Des 1.4
Graf complet K_5 (A), graf bipartit (B).

Teorema 1. Pentru ca graful neorientat $G=(V,E)$ să fie *bipartit*, este necesar şi suficient ca el să nu conţină cicluri de lungime impară.

□

Necesitate. $V = V^A \bigcup V^B$, $V^A \bigcap V^B = \varnothing$. Fie în G există un ciclu de lungime impară $v_{i_1}, v_{i_2}, ...v_{i_k}, v_{i_1}$ şi $v_{i_1} \in V^A$. Deoarece lungimea ciclului este impară, numărul de vârfuri, care descriu ciclul este par. G este bipartit, prin urmare, vârfurile cu indici pari din ciclul $v_{i_1}, v_{i_2}, ...v_{i_k}, v_{i_1}$ aparţin componentei V^B. Prin urmare şi $v_{i_1} \in V^B$, ceea ce contrazice presupunerea iniţială.

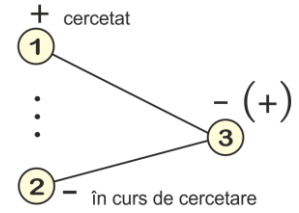
Suficienţă. Fie că în G nu există nici un ciclu de lungime impară. În acest caz se va construi o divizare corectă a V în două submulţimi distincte V^A şi V^B, astfel încât orice arc din E va avea originea în V^A, iar sfârşitul în V^B.

Se alege un vârf arbitrar v_i şi se etichetează cu „+”. Toate vârfurile din $\Gamma(v_i)$ se etichetează cu semnul opus celui atribuit v_i. Vârful v_i se consideră cercetat.

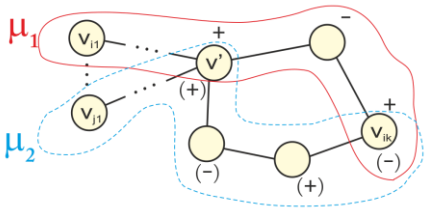
Se alege un vârf etichetat, necercetat. Operaţia de etichetare a vecinilor se repetă până nu apare una din următoarele situaţii:

(a) Toate vârfurile sunt etichetate, şi etichetele atribuite lor corelează (oricare două vârfuri unite prin o muchie au semne diferite)

(b) Există cel puţin un vârf v_{i_k} etichetat deja, care poate fi etichetat repetat cu semnul opus din partea altui vârf (desenul 1.5)



Des. 1.5 Etichetarea nodurilor. Cazul (b)



Des. 1.6 Fragmentul $v', ..., v_{i_k}$ al căii μ_1 are lungime pară. Fragmentul $v', ..., v_{i_k}$ al căii μ_2, are lungime impară.

(c) Toate vârfurile etichetate sunt cercetate, dar există vârfuri fără etichete.

În cazul (a) toate vârfurile etichetate cu „+” se includ în V^A, iar cele etichetate cu”-” în V^B. Deoarece toate muchiile conectează vârfuri cu etichete diferite, graful este bipartit.

Cazul (b). Ar putea să apară doar dacă există o cale $\mu_1 = v_{i_1}, v_{i_2}, ..., v_{i_k}$ în care semnele alternează şi o cale $\mu_2 = v_{j_1}, v_{j_2}, ..., v_{j_l}, v_{i_k}$ cu aceeaşi proprietate.

Fie v' ultimul vârf comun al căilor μ_1, μ_2 diferit de v_{i_k}. Dacă în μ_1 semnele $v' v_{i_k}$ coincid, în μ_2 ele vor fi opuse (şi invers: dacă coincid în μ_2 sunt opuse în μ_1). De aici rezultă că unul din fragmentele $v', ..., v_{i_k}$ al căii μ_1 şi $v', ..., v_{i_k}$ al căii μ_2, are un număr par de muchii, iar celălalt – un număr impar (des. 1.6). Respectiv, ciclul determinat de reuniunea acestor două fragmente va avea o lungime impară, ceea ce contrazice condițiile inițiale. În consecință, cazul (b) este unul imposibil în grafurile bipartite.

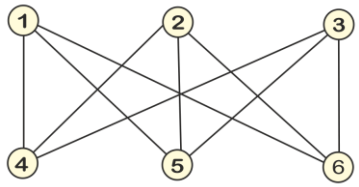
Cazul (c) indică divizarea grafului în subgrafuri izolate, prin urmare fiecare dintre acestea urmează să fie cercetat separat. Prin urmare cazul se reduce la (a).

■

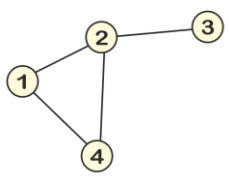
Def. Graful bipartit este numit *complet* dacă
$\forall v' \in V^A, \forall v'' \in V^B \; \exists (v', v'') \in E$

Graful bipartit complet cu părți formate din n şi m vârfuri se notează $K_{n,m}$ (desenul 1.7)

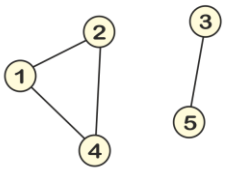
Def. Graful $G = (V, E)$ este *conex*, dacă pentru orice două vârfuri $v_i, v_j \in V$ în G există cel puțin o cale, care le uneşte (desenul 1.8 A). În cazul în care graful este format din câteva subgrafuri conexe separate el este *neconex* (desenul 1.8 B).



Des. 1.7
Graful bipartit complet $K_{3,3}$



A



B

Des. 1.8 Graf conex (A), graf neconex (B)

Def. Graful $G=(V,E)$ este numit *arbore*, dacă este conex şi nu conține cicluri.

Def. Graful $G=(V,E)$ este numit *graf planar*, dacă poate fi reprezentat în plan fără intersecții ale muchiilor.

Def. O *față* a grafului este o regiune a planului, delimitată de muchiile acestuia, care nu conține în interior muchii sau vârfuri ale grafului.

Pentru un graf amplasat pe o suprafață există relație între numărul de vârfuri, muchii şi fețe. Relația se numeşte *caracteristică Euler a suprafeței*.

Teorema 2. (formula Euler) Într-un graf planar conex are loc următoarea relație:

$$n-m+r=2$$

unde: n – numărul de vârfuri ale grafului, m – numărul de muchii, r – numărul de fețe.

□

Demonstrație. Prin inducție după numărul de muchii m.

$m=0. \Rightarrow n=1 \& r=1. \quad 1-0+1=2.$

Fie teorema este adevărată pentru $m=k. \qquad n-k+r=2.$

Se adaugă încă o muchie $m=k+1.$

Dacă muchia uneşte două vârfuri existente, atunci $r'=r+1.$

$n-(k+1)+(r+1)=n-k+r-1+1=n-k+r=2.$

Dacă muchia uneşte un vârf existent cu unul nou, atunci $n' = n+1$. $(n+1)-(k+1)+r = n-k+r+1-1 = n-k+r = 2.$

■

Corolar. Într-un graf planar ($n > 3$), conex, $m \le 3n-6$.

□

Fiecare față este delimitată de cel puțin 3 muchii, iar fiecare muchie delimitează cel mult două fețe. Prin urmare $3r \le 2m$. Înlocuind în formula precedentă, se obține:

$$2 = n-m+r \le n-m+\frac{2m}{3} \quad \Rightarrow \quad 6 \le 3n-3m+2m \quad \Rightarrow \quad m \le 3n-6$$

■

Teorema 3. Un graf este planar atunci şi numai atunci când nu conține subgrafurile K_5 şi $K_{3,3}$

□ (fără demonstrație) ■

Ponderi

În unele cazuri muchiile grafului posedă caracteristici numerice suplimentare, numite ponderi. Muchiei (arcului) (v_i, v_j) i se pune în corespondență valoarea $c_{i,j}$ - *ponderea* (costul, lungimea etc.). Graful, muchiilor căruia î-i sunt asociate ponderi se numeşte *graf cu muchii ponderate*. Pentru rezolvarea unor probleme este necesară şi aplicarea ponderilor la vârfurile grafului. Vârfului v_i i se pune în corespondență caracteristica numerică c_i - *ponderea* (costul). Graful, vârfurilor căruia î-i sunt asociate ponderi se numeşte *graf cu vârfuri ponderate*.

Dacă graful $G = (V, E)$ este unul ponderat, atunci pentru căile din graf se introduce caracteristica numerică l - *cost* (lungime) egală cu suma ponderilor muchiilor din care este formată o cale C.

$$l(C) = \sum_{(v_i, v_j) \in C} c_{i,j}$$

1.2 Structuri de date pentru reprezentarea unui graf

Pentru rezolvarea problemelor pe grafuri cu ajutorul calculatorului, reprezentarea lor „naturală" în formă de puncte (pentru noduri) şi linii care le unesc (muchii) nu este încă cea mai optimă. Selectarea şi utilizarea corectă a structurilor de date care modelează un graf poate influența ordinul complexității algoritmului, prin urmare şi eficiența lui.

Structurile de date, prezentate în paragraful dat sunt cel mai des utilizate în rezolvarea problemelor standard pe grafuri. Ele pot fi folosite atât pentru grafurile neorientate, cât şi pentru cele orientate.

Structura de date clasică pentru reprezentarea unui graf $G=(V,E)$ este considerată **matricea de incidență**. Este realizată prin un tablou bidimensional cu N linii (N – numărul de vârfuri în graf, $N=|V|$) şi M coloane (M – numărul de muchii, $M=|E|$). Fiecare muchie (u,v) este descrisă într-o coloană a tabloului. Elementele coloanei sunt egale cu 1 pentru liniile care corespund vârfurilor u şi v, 0 – pentru celelalte. În cazul grafului orientat vârful din care începe arcul este marcat cu -1, vârful final – cu +1.

Pentru grafurile din des. 1.1,1.2 matricele de incidență vor fi

Des.1.1

	(1,2)	(1,4)	(□,3)	(2,4)	(3,4)
1	1	1	0	0	0
2	1	0	1	1	0
3	0	0	1	0	1
4	0	1		1	1

Des. 1.2

	(2,1)	(1,4)	(3,2)	(2,4)	(3,4)
1	+1	-1	0	0	0
	-1	0	+1	+1	0
3	0	0	-1	0	-1
4	0	+1	0	-	+1

Matricea de incidență pentru un graf cu N noduri va avea N linii, numărul de coloane fiind proporțional cu N^2. Numărul total de elemente în structura de date este proporțional cu N^3. Utilizarea memoriei este în

acest caz ineficientă, deoarece doar câte două elemente din fiecare linie vor conține date real utilizabile.

O altă reprezentare matriceală a grafului $G=(V,E)$ este **matricea de adiacență**. Matricea de adiacență este şi ea realizată prin un tablou bidimensional, cu N linii şi N coloane, în care elementul cu indicii (i, j) este egal cu 1 dacă există muchia care uneşte vârful v_i cu vârful v_j şi 0 – în caz contrar. Datele despre muchia (v_i, v_j) se dublează în elementele tabloului cu indicii (i, j) şi (j, i) În grafurile orientate, pentru arcul (v_i, v_j) primeşte valoarea 1 doar elementul (i, j) al tabloului.

Pentru grafurile din des. 1.1,1.2 matricele de adiacență vor fi

Des. 1.1				
	1	2	3	4
1		**1**	**0**	**1**
2	1	**0**	**1**	**1**
3	**0**	**1**	**0**	**1**
4	**1**	**1**	**1**	**0**

Des. 1.2				
	1	2	3	4
1	**0**	**0**	**0**	**1**
2	**1**	**0**	**0**	**1**
3	**0**	**1**	**0**	**1**
4	**0**	**0**	**0**	**0**

Matricea de adiacență pentru un graf cu N vârfuri are N linii şi N coloane. Numărul total de elemente în structura de date este N^2.

O altă categorie de reprezentări ale grafului o formează reprezentările prin liste. Cea mai simplă pentru implementare listă este **lista de muchii**. Lista de muchii conține M perechi de forma (v_i, v_j), fiecare pereche reprezentând o muchie din graf, descrisă prin vârfurile care o formează. Într-un graf orientat perechea descrie un arc, începutul lui fiind determinat de primul indice din pereche.

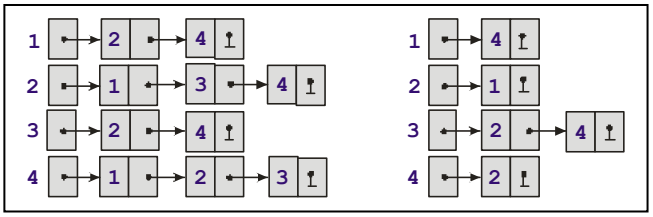
Pentru grafurile din des. 1.1,1.2 listele de muchii vor fi

Des.1.1	Des. 1.2
(1,2)(1,4)(2,3)(2,4)(3,4)	(1,4)(2,1)(2,4)(3,2)(3,4)

Lista de muchii este formată din M perechi de elemente. M – numărul de muchii. Cu toate că structura este mai compactă decât matricea de incidență sau matricea de adiacență, majoritatea operațiilor standard pe graful reprezentat în acest mod necesită parcurgerea întregii liste, ceea ce scade din eficiența structurii. În grafurile orientate, primul element al perechii care descrie arcul va fi vârful sursă, al doilea – vârful destinație. În cazul în care muchiile (arcele) au ponderi asociate, fiecare muchie (arc) va fi descrisă de un triplet: indicii vârfurilor care o formează şi ponderea acesteia.

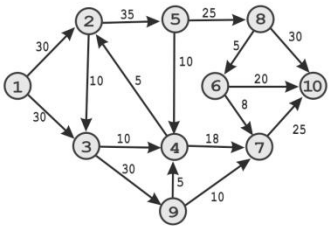
Încă o structură eficientă este **lista de incidență.** Pentru fiecare nod $v \in V$ ea va conține o listă unidirecțională alocată dinamic, cu toate vârfurile $u : \exists (v,u) \in E$. Indicatorii către începutul fiecărei liste pot fi păstrați într-un tablou unidimensional. Elementul cu indicele i din tablou va conține indicatorul către lista de vârfuri incidente vârfului v_i din graf. Pentru grafurile neorientate descrierea fiecărei muchii se dublează, iar operațiile de adăugare (lichidare) a muchiilor presupun prelucrarea a două liste.

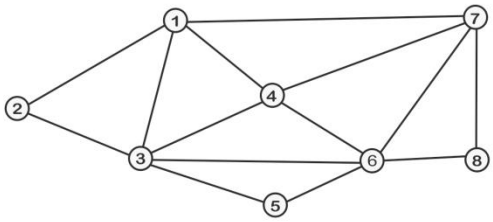
Pentru grafurile din des. 1.1,1.2 listele de vârfuri vor fi:



Exerciţii:

1. Pentru grafurile din imagini construiți:





   a) Matricea de incidență
   b) Matricea de adiacență
   c) Lista de muchii
   d) Lista de vecini

2. Elaborați un program pentru citirea dintr-un fişier text a matricei de adiacență a grafului ($|V| \le 20$) şi afişarea ei pe ecran. Prima linie a fişierului de intrare va conține un număr întreg n – dimensiunea matricei. Următoarele n linii vor conține câte n numere întregi, separate prin spațiu – elementele matricei de adiacență a grafului.

Capitolul 2. Parcurgeri. Conexitate

În acest capitol:

- Parcurgerea grafului în lăţime
- Parcurgerea grafului în adâncime
- Grafuri tare conexe
- Determinarea componentelor tare conexe
- Baze în graf

2.1 Parcurgerea grafului

Cele mai multe din problemele formulate pe grafuri necesită o cercetare a legăturii între vârfurile acestora. Evident, un algoritm eficient va accesa muchia (vârful) de o singură dată, sau de un număr constant de ori. De aici rezultă necesitatea unor metode eficiente pentru parcurgerea vârfurilor unui graf.

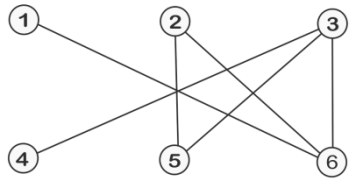
În caz general problema parcurgerii se formulează în felul următor: Fie dat graful $G=(V,E)$. Pentru un vârf dat $v \in V$ să se determine mulțimea $U \subseteq V$: $\forall u \in U$ există cel puțin o cale între v şi u.

Una dintre cele mai eficiente metode de parcurgere în graf este **parcurgerea în adâncime**[4]. La baza metodei stă principiul de selectare recursivă a vârfurilor şi etichetare a lor. Inițial toate vârfurile se consideră *neatinse*. Fie v_0 vârful de la care începe parcurgerea. v_0 se etichetează ca fiind *atins*. Se alege un vârf u, adiacent v_0 şi se repetă procesul, pornind de la u. În general, fie v vârful curent. Dacă există un vârf u, nou (neatins), adiacent v ($\exists (v,u) \in E$), atunci procesul se repetă pornind de la u. Dacă pentru vârful curent v nu mai există vârfuri vecine neatinse, el se etichetează ca fiind *cercetat*, iar procesul de parcurgere revine în vârful precedent (din care s-a ajuns în v). Dacă

[4] Depth first search (eng.)

$v = v_0$ parcurgerea a luat sfârşit. Pentru realizarea algoritmului se vor folosi marcaje aplicate vârfurilor grafului (0 – nod *nou*, 1 – *atins*, 2 – *cercetat*).

Exemplu: pentru graful din imaginea alăturată se va simula parcurgerea în adâncime din vârful 1, în conformitate cu algoritmul descris anterior:



Pas 1	vârf	**1** 2 3 4 5 6
	stare	1 0 0 0 0 0
Pas 2	vârf	1 2 3 4 5 **6**
	stare	1 0 0 0 0 1
Pas 3	vârf	1 2 **3** 4 5 6
	stare	1 0 1 0 0 1
Pas 4	vârf	1 2 3 **4** 5 6
	stare	1 0 1 1 0 1
Pas 5	vârf	1 2 **3** 4 5 6
	stare	1 0 1 2 0 1
Pas 6	vârf	1 2 3 4 **5** 6
	stare	1 0 1 2 1 1

Pas 7	vârf	1 **2** 3 4 5 6
	stare	1 1 1 2 1 1
Pas 8	vârf	1 2 3 4 **5** 6
	stare	1 2 1 2 1 1
Pas 9	vârf	1 2 **3** 4 5 6
	stare	1 2 1 2 2 1
Pas 10	vârf	1 2 3 4 5 **6**
	stare	1 2 2 2 2 1
Pas 11	vârf	**1** 2 3 4 5 6
	stare	1 2 2 2 2 2
Pas 12	vârf	1 2 3 4 5 6
	stare	2 2 2 2 2 2

Un exemplu simplu de realizare a procedurii de parcurgere în adâncime pentru un graf cu *N* vârfuri, descris prin matricea de adiacență, este prezentat în funcția DFS. Matricea de adiacență a grafului este stocată în tabloul **A**. Marcajele vârfurilor se păstrează în tabloul liniar **B** (B[i] – starea vârfului *i*) Inițial toate marcajele vârfurilor sunt nule. Vârful din care este lansată parcurgerea – s.

```
int DFS (int s)
{
 int i;
  b[s]=1;
   for(i=1;i<=n;i++)
        if(a[s][i] !=0 && b[i]==0) DFS(i);
   printf("%d ", s);
```

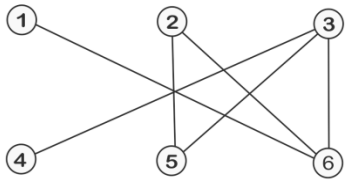
```
    return 0;
}
```

Complexitatea funcției DFS în acest caz este $O(N^2)$. Odată marcat, nodul v nu mai permite relansarea parcurgerii DFS(v), iar numărul maxim de apeluri ale funcției este N. Numărul de operații în corpul funcției este de asemenea proporțional cu N.

Funcția propusă lucrează corect atât pe grafuri neorientate, cât şi pe grafuri orientate.

În procesul de parcurgere, cu cât mai târziu este atins un vârf, cu atât mai repede el va fi cercetat (modelarea prin structuri tip LIFO). Există probleme în care este important ca vârfurile să fie cercetate în ordinea atingerii (modelarea procesului prin structuri de date FIFO). Pentru rezolvarea lor se poate utiliza o altă metodă de cercetare a grafului – **parcurgerea în lățime**[5].

La fel ca metoda precedentă, parcurgerea în lățime începe de la un nod dat v_0, plasat în într-o structură tip coada (inițial vidă). Se foloseşte un principiu de etichetare a vârfurilor identic celui folosit în parcurgerea în adâncime. În caz general, se extrage din coadă nodul v (la prima iterație $v = v_0$), se determină *toate* vârfurile u noi (care încă nu au fost plasate în coadă, *neatinse*), adiacente v ($\exists (v,u) \in E$), şi se adaugă consecutiv în coadă. La adăugarea în coadă vârfurile se etichetează ca fiind *atinse*. După adăugarea vârfurilor adiacente în coadă, nodul v este marcat *cercetat*. Dacă coada devine vidă - parcurgerea a luat sfârşit. Pentru realizarea algoritmului se vor folosi aceleaşi marcaje aplicate vârfurilor ca şi în cazul parcurgerii în adâncime.



Exemplu: pentru graful din imaginea alăturată se va simula parcurgerea în lățime din vârful 1, în conformitate cu algoritmul descris anterior:

[5] Breadth first search (eng.)

Pas 1	atins	**1**
	cercetat	
Pas 2	atins	6
	cercetat	1
Pas 3	atins	2 3
	cercetat	1 6
Pas 4	atins	3 5
	cercetat	1 6 2

Pas 5	atins	5 4
	cercetat	1 6 2 3
Pas 6	atins	4
	cercetat	1 6 2 3 5
Pas 7	atins	
	cercetat	1 6 2 3 5 4

În următorul exemplu coada este implementată prin un tablou unidimensional **B**, începutul ei fiind elementul cu indicele **st**, iar sfârşitul – elementul cu indicele **dr**. Elementele cu indicii **1,...,st-1** formează mulțimea nodurilor cercetate la moment. Structurile de date **A, B, n** au aceleaşi semnificație ca şi în funcția **DFS**. Tabloul C modelează stările curente ale vârfurilor. Funcția BFS realizează parcurgerea în lățime, începând de la vârful cu indicele s.

```
int BFS (int s)
{
  int i, st, dr;
  c[s]=1; b[1]=s; st=1;dr=1;
  while (st<=dr)
  {  for(i=1;i<=n;i++)
      if(a[b[st]][i] !=0 && c[i]==0)
      { dr++; b[dr]=i; c[i]=1; }
   printf("%d ", b[st]);
   st++;
  }
  return 0;
}
```

Funcția BFS este implementată nerecursiv cu o complexitate $O(N^2)$. Numărul de operații în corpul funcției este determinat de două instrucțiuni ciclice incluse, ambele având maxim N repetări.

2.2 Grafuri tare conexe

Def. Un graf orientat $G=(V,E)$ se numeşte *tare conex* dacă pentru orice două vârfuri $v_i, v_j \in V$ există cel puțin câte un lanț care uneşte v_i cu v_j ($v_i \mapsto v_j$) şi v_j cu v_i ($v_j \mapsto v_i$). Într-un graf tare conex orice două vârfuri sunt reciproc accesibile.

Def. Graful $G=(V,E)$ se numeşte *unidirecțional conex* dacă pentru orice două vârfuri $v_i, v_j \in V$ există cel puțin unul din lanțurile $v_i \mapsto v_j$ sau $v_j \mapsto v_i$. Într-un graf unidirecțional conex orice două vârfuri sunt conectate prin cel puțin o cale.

Def. *Componentă tare conexă* a unui graf orientat $G=(V,E)$ se numeşte mulțimea maximală de vârfuri $V' \subseteq V:\ \forall v_i, v_j \in V'$ există cel puțin câte un lanț $v_i \mapsto v_j$ şi $v_j \mapsto v_i$.

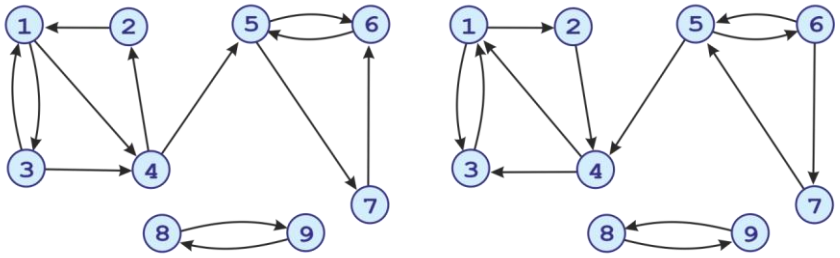
Determinarea componentelor tare conexe

Pentru determinarea componentelor tare conexe va fi folosit *graful transpus* lui *G*. Pentru un graf orientat $G=(V,E)$, graful transpus este $G^T=(V,E^T)$ unde $E^T=\{(v_i,v_j):(v_j,v_i)\in E\}$. Graful transpus are aceleaşi componente tare conexe ca şi graful inițial.

Obținerea grafului transpus $G^T=(V,E^T)$ cere un timp liniar după numărul de arce în graf (sau pătratic față de numărul de vârfuri). Procesul se realizează în mod diferit în dependență de modul de reprezentare a grafului. Fie graful $G=(V,E)$ reprezentat prin matricea de adiacență $A(n\times n)$. Matricea de adiacență a grafului $G^T=(V,E^T)$ se obține conform următoarei formule:

$$A^T_{i,j}=\begin{cases}1 & dacă \quad A_{j,i}=1\\ 0 & în\ caz\ contrar\end{cases} \quad i,j=1,...,n$$

Exemplu: desenul 2.1: Graful inițial $G=(V,E)$ şi graful transpus $G^T=(V,E^T)$, reprezentate grafic.



A **B**

Des. 2.1 Graful iniţial (A) şi graful transpus (B).

Algoritmul pentru determinarea componentelor tare conexe are la bază observația, că componentele tare conexe rămân aceleaşi atât în graful inițial cât şi în cel transpus. Se vor folosi va folosi două parcurgeri în adâncime pe grafurile G^T şi G.

Algoritmul Kosaraju

Pseudocod

Pas 1. Se construieşte graful $G^T=(V,E^T)$

Pas 2. Se lansează căutarea în adâncime pornind de la fiecare vârf necercetat din G^T. Pentru fiecare parcurgere se memorează cumulativ ordinea de cercetare a vârfurilor în vectorul f.

Pas 3. Se lansează căutarea în adâncime pe graful inițial G, consecutiv, pornind de la ultimul vârf inclus în f către primul, după vârfurile necercetate.

Pas 4. La fiecare căutare în adâncime realizată în pasul 3, se afişează vârfurile cercetate – acestea formează o componentă tare conexă.

Exemplu implementare:

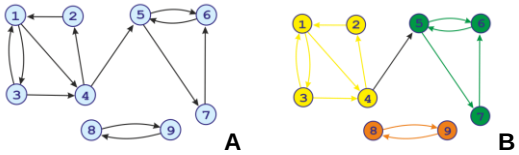
Se foloseşte matricea de adiacenţă **a** pentru reprezentarea grafului iniţial, **at** pentru reprezentarea grafului transpus, vectorul **b** – pentru descrierea stării vârfurilor, vectorul **black** – pentru ordinea de cercetare.

Input: Graful $G=(V,E)$

Output: componentele tare conexe ale grafului, separate pe linii.

```
int DFS_DIR (int s) { // . . . descrisa anterior - DFS }

int DFS_TRANS (int s)
{  int i;
   b[s]=1;
   for(i=1;i<=n;i++)
      if( at[s][i] !=0 && b[i]==0) DFS_TRANS(i);
   //amplasarea in stiva ordinii de cercetare
   k++; black[k]=s;
  return 0;
}

int main()
{ // . . .citirea datelor initiale
  // transpunerea grafului
 for (i=1;i<=n;i++)
      for (j=1; j<=n; j++)
            if (a[i][j]==1) at[j][i]=1;
 // Cautarea in adancime pe graful transpus
 for(i=1;i<=n; i++) if (b[i]==0) DFS_TRANS(i);
 // resetarea starii varfurilor
 for(i=1;i<=n;i++) b[i]=0; printf("\n");
 // parcurgerea in adancime pe graful initial
 for(i=n;i>=1;i--) //Afisarea componentelor tare conexe
    if (b[i]==0) {DFS_DIR(black[i]); printf("\n");}
 return 0;
}
```

Reprezentarea grafică a rezultatelor:



Des. 2.2. Graful iniţial (A), Componentele tare conexe ale grafului (fiecare componentă e colorată aparte) (B)

Algoritmul are o complexitate pătratică față de numărul de vârfuri în graf. Pasul 1 al algoritmului necesită un număr de operații proporțional cu $n \times n$. Paşii 2 şi 3 repetă căutări în adâncime, fiecare cu o complexitate $O(N^2)$. Prin urmare, complexitatea totală a algoritmului este $O(N^2)$. Demonstrația corectitudinii algoritmului poate fi găsită în [7, p.420]

2.3 Baze în graf

O *bază* B a grafului $G = (V, E)$ este formată din o mulțime de vârfuri ale grafului, care posedă următoarele două proprietăți:

(i) Din B poate fi accesat orice vârf al grafului

(ii) Nu există nici o submulțime $B' \subset B$ care să păstreze proprietatea (i)

O bază este determinată elementar în situația în care au fost identificate componentele tare conexe ale grafului. Deoarece în interiorul componentei tare conexe toate vârfurile sunt reciproc accesibile, rezultă:

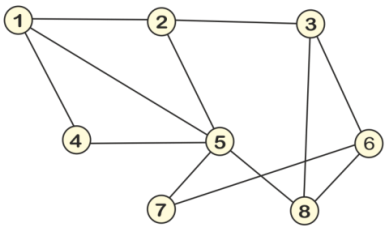
(a) În bază se include exact câte un vârf din fiecare componentă tare conexă

(b) Pentru construcția bazei vor fi folosite toate componentele tare conexe.

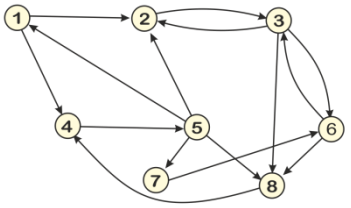
Utilizarea bazelor permite reducerea dimensiunii problemelor pe grafuri, în special în cazurile de cercetare a legăturilor structurale în organizații.

Exemplu: pentru graful din desenul 2.1, în calitate de baze pot servi mulțimile {1,5,8}, {3,6,9}, {2,5,8}, …

Exerciţii:



1. Simulați, pe paşi, pentru graful din imagine, parcurgerea în lățime, de la vârful 1



2. Simulați, pe paşi, pentru graful din imagine, parcurgerea în adâncime, de la vârful 1

3. Elaborați un program pentru parcurgerea în adâncime a grafului ($|V| \le 20$) (abordare recursivă)

4. Elaborați un program pentru parcurgerea în adâncime a grafului ($|V| \le 20$) (abordare iterativă)

5. Elaborați un program pentru parcurgerea în lățime a grafului ($|V| \le 20$)

6. Elaborați un program pentru determinarea componentelor tare conexe ale grafului ($|V| \le 20$)

7. Elaborați un program pentru generarea tuturor bazelor unui graf cu cel mult 20 de vârfuri.

Capitolul 3. Mulţimi independente şi dominante

În acest capitol

- Noţiunea de mulţime independentă
- Mulţimi maximal independente.
- Generarea mulţimilor maximal independente
- Mulţimi dominante

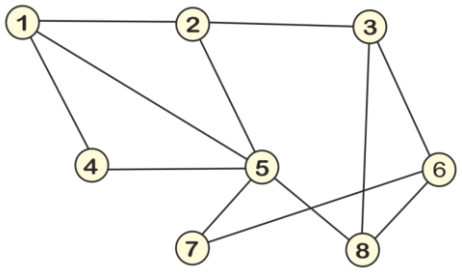
3.1 Mulţimi independente

Fie dat un graf neorientat $G = (V, E)$.

Def. Mulțime *independentă* sau *mulțime interior stabilă* se numeşte o mulțime de vârfuri ale grafului, astfel încât oricare două vârfuri din ea nu sunt unite direct prin muchie.

Formulată matematic, mulțimea independentă S este o mulțime, care satisface relația: $S \subseteq V : S \bigcap \Gamma(S) = \varnothing$.

Def. Mulțimea S se numeşte maximal independentă, dacă nu există o altă mulțime independentă S', pentru care se îndeplineşte condiția: $S \subseteq S'$.



Des. 3.1. {1, 3,7} {2,8,7,4} {4,6} – mulţimi independente. Mulţimile {2,8,7,4} , {2, 4, 6} sunt maximal independente, iar {1,3}, {2,4} – nu.

Nu toate mulțimile maximal independente au acelaşi număr de vârfuri. Prin urmare modul de selecție a celei mai bune mulțimi maximal independente va depinde de condițiile inițiale ale problemei concrete.

Def. Fie Q mulțimea tuturor mulțimilor independente a grafului $G=(V,E)$. Numărul $\alpha[G]=\max_{S\in Q}|Q|$ se va numi *număr de independență* a grafului G, iar mulțimea S^*, pentru care el se obține – *mulțime maximă independentă*.

Exemplu: pentru graful din desenul 3.1 setul de mulțimi maximale independente este {1, 3, 7}, {1, 6}, {1, 7, 8}, {2, 4, 6}, {2, 4, 7, 8}, {3, 4, 7}, {3, 5}, {5, 6}. Cea mai mare putere a mulțimilor este 4, deci α [G] = 4. Mulțimea maximă independentă este {2, 4, 7, 8}

3.2 Generarea tuturor mulţimilor maximal independente

Fie $G=(V,E)$. Prin *graf complementar* se înțelege graful $\tilde{G}=(V,\tilde{E})$ unde $\tilde{E}=\{(u,v):u,v\in V;(u,v)\notin E\}$

Problema mulțimilor maximal independente se reduce direct la problema generării subgrafurilor complete, rezolvată pe graful complementar $\tilde{G}=(V,\tilde{E})$. Aceasta din urmă este o problemă de complexitate exponențială. De aici rezultă şi complexitatea exponențială a algoritmului pentru determinarea tuturor mulțimilor maximal independente. Tehnica generală a abordării este tehnica reluării, care poate fi parțial optimizată prin ordonarea vârfurilor după micşorarea puterii acestora. Algoritmul de parcurgere sistematică a fost propus de Bron şi Carbosh. El evită generarea repetată a mulțimilor, creând noi mulțimi prin îmbunătățirea celor existente.

Motivarea algoritmului

Algoritmul se bazează pe arborele de căutare. Prin urmare, este eficientă realizarea lui recursivă.

În general la pasul k mulțimea independentă S_k se extinde prin adăugarea unui vârf nou, pentru a obține mulțimea S_{k+1} la pasul $k+1$. Procesul se repetă atât timp, cât este posibil. La momentul, în care nu mai putem adăuga vârfuri avem obținută o mulțime maximal independentă.

Fie Q_k va fi la pasul k - mulțimea maximală de vârfuri, pentru care $\Gamma(S_k) \bigcap Q_k = \varnothing$. Prin adăugarea unui vârf din Q_k în S_k se obține S_{k+1}. Q_k e formată în general din 2 componente: Q_k^- a vârfurilor deja folosite în procesul de căutare pentru extinderea S_k şi Q_k^+ a vârfurilor care încă nu s-au folosit în căutare. Pentru adăugarea în S_k se vor folosi doar vârfurile din Q_k^+. Astfel, procedura de adăugare a vârfului nou e următoarea:

a) selectarea unui nod $x_{i_k} \in Q_k^+$ (*)

b) construirea mulțimii $S_{k+1} = S_k \bigcup \{x_{i_k}\}$

c) formarea $\begin{aligned} Q_{k+1}^- &= Q_k^- - \Gamma(x_{i_k}) \\ Q_{k+1}^+ &= Q_k^+ - \{\Gamma(x_{i_k}) \bigcup (x_{i_k})\} \end{aligned}$

Pasul de întoarcere presupune lichidarea vârfului x_{i_k} din S_{k+1} pentru revenirea la mulțimea S_k cu deplasarea x_{i_k} din Q_k^+ în Q_k^-.

Soluția (mulțimea maximal independentă) se va obține în cazul când $Q_k^+ = \varnothing$ şi $Q_k^- = \varnothing$. Dacă c rezultă că mulțimea S_k a fost extinsă la o etapă precedentă din contul adăugării unui vârf din Q_k^-, de aceea nu este maximal independentă.

Prezența în Q_k^- a unui vârf $x \in Q_k^-$: $\Gamma(x) \bigcap Q_k^+ = \varnothing$ (**) este un indicator pentru a efectua pasul de întoarcere, deoarece în acest caz Q_k^- nu va deveni vidă, indiferent de modul de selecție a vârfurilor.

Optimizarea algoritmului

Optimizarea poate fi obținută din contul apariției cât mai rapide a pasului de întoarcere. Prin urmare se va încerca îndeplinirea cât mai grabnică a condiției (**). O metodă sigură (în special pentru grafuri mari) este de a alege mai întâi în Q_k^- un vârf x', pentru care mărimea $\Delta(x') = \left|\Gamma(x') \bigcap Q_k^+\right|$ va fi minimală, apoi, la fiecare pas următor în alegerea unui x_{i_k} din Q_k^+ : $x_{i_k} \in \Gamma(x')$. Aceasta asigură apropierea cu o unitate de situația care generează pasul de întoarcere.

Pseudocod

Pas 1. $S_0 \leftarrow Q_0^- \leftarrow \varnothing, \quad Q_0^+ \leftarrow V, \quad k \leftarrow 0.$

Adăugarea vârfurilor

Pas 2. Este selectat un vârf $x_{i_k} \in Q_k^+$, după principiul formulat anterior. Se formează $S_{k+1}, Q_{k+1}^+, Q_{k+1}^-$ fără a modifica Q_k^+, Q_k^-. $k \uparrow$

Verificarea posibilității de continuare

Pas 3. Dacă există $x \in Q_k^-$: $\Gamma(x) \bigcap Q_k^+ = \varnothing$ se trece la pasul 5, altfel - la pasul 4.

Pas 4.

a) Dacă $Q_k^+ = \varnothing$ şi $Q_k^- = \varnothing$ se afişează (memorează) mulțimea maximal independentă S_k şi se trece la pasul 5.

b) Dacă $Q_k^+ = \varnothing$, dar $Q_k^- \neq \varnothing$ se trece direct la pasul 5

c) Dacă $Q_k^+ \neq \varnothing$, şi $Q_k^- \neq \varnothing$ se revine la pasul 2

Mişcarea înapoi

Pas 5. $k \downarrow$

a) x_{i_k} se exclude din S_{k+1}, pentru a reveni la S_k.

b) Se reconstruiesc Q_k^+, Q_k^-: $Q_k^+ \leftarrow Q_k^+ - \{x_{i_k}\}$ $Q_k^- \leftarrow Q_k^- + \{x_{i_k}\}$

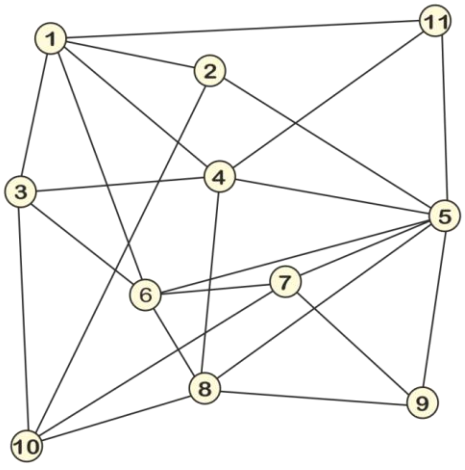
c) Dacă $k = 0$ şi $Q_k^+ = \varnothing$, atunci SFÂRŞIT (au fost afişate [memorate] toate mulțimile independente maximale). În caz contrar se revine la pasul 3.

Implementare. În următorul exemplu este realizată o implementare simplificată a algoritmului, fără optimizarea prin reordonarea vârfurilor. Generarea repetată a mulțimilor maximal independente este evitată prin selectarea vârfurilor pentru includere în ordine lexicografică. Mulțimile Q_k^+, Q_k^-, S_k se formează şi se gestionează prin intermediul apelurilor recursive. Restricțiile de dimensiune $|V| \leq$ **num_el**. Structurile de date: **a** – matricea de adiacență a grafului, **s** – tabloul etichetelor vârfurilor, incluse în soluția curentă (**s[i]=1**), **q** – tabloul etichetelor vârfurilor care pot fi adăugate la soluție (**q[i]=1**), **rest** – tabloul etichetelor pentru restabilirea Q_k^+ .

```
int fillc(int *x, int z, int num)
{ … // elementele cu tabloului x primesc valoarea z}

int readdata()
{ … // citirea matricei de adiacenţă a grafului A}

int print()
{ … // funcţia pentru afişarea mulţimii maximal independente
curente}

int mind(int *s, int *q)
{ int i,j,k=0,r, rest[num_el];
   for (i=1;i<=n;i++) if(q[i]!=0) k=1;
   if (k==0) {print();} // afişarea soluţiei
```

```
   else { j=n;
          while (s[j]==0 && j>=1) j--;
          r=j+1;
          for (i=r;i<=n;i++)
           if (q[i]==1)
              { fillc(rest,0,n);
                s[i]=1; q[i]=0;
                for (j=1;j<=n;j++)
                 if (a[i][j] != 0 && q[j]==1)
                     {q[j]=0; rest[j]=1;}
                mind (s,q);    // mişcarea inainte
                s[i]=0;q[i]=1; //mişcarea înapoi
                for (j=1;j<=n;j++)
                    if(rest[j]==1) q[j]=1;
        }    }
    return 0;
}
int main()
{   readdata();
    fillc(s,0,n); fillc(q,1,n);
    mind(s,q); return 0;
}
```

Pentru graful reprezentat pe desenul 3.2, programul determină următoarele mulțimi maximal independente




```
1 5 10
1 7 8
1 8 10
1 9 10
2 3 7 8 11
2 3 9 11
2 4 6 9
2 4 7
2 6 9 11
3 5
4 6 9 10
6 9 10 11
8 10 11
```

Des. 3.2 Graful iniţial şi mulţimile independente identificate.

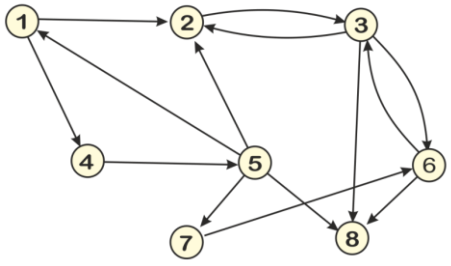
3.3 Mulţimi dominante

Fie dat un graf orientat $G=(V,E)$.

Def. Mulțime *dominantă* sau *mulțime exterior stabilă* se numeşte o mulțime $S \subseteq V$: $\forall x_j \notin S \quad \exists (x_i, x_j), x_i \in S$.

Formulată matematic, mulțimea S este una dominantă, dacă: $S \bigcup \Gamma^{+}(S) = V$.

Def. Mulțimea dominantă S se numeşte minimă, dacă nu există o altă mulțime dominantă S', pentru care se îndeplineşte condiția: $S' \subseteq S$.



Des. 3.3. {4,5,6} {3,8,7,5,4} {3,5,1} – mulţimi dominante. Mulţimile {3,5,1} , {4,5,6} sunt minime dominante, iar {1,2,4,6,7} – nu.

Nu toate mulțimile minime dominante au acelaşi număr de vârfuri. Prin urmare modul de selecție a celei mai bune mulțimi minime dominante va depinde de condițiile inițiale ale problemei cercetate.

Def. Fie Q mulțimea tuturor mulțimilor dominante ale grafului $G=(V,E)$. Numărul $\beta[G] = \min_{S \in Q} |Q|$ se va numi *indice de dominanță* a grafului G, iar mulțimea S^{*}, pentru care el se obține – *mulțime dominantă de putere minimă.*

Legătura dintre mulțimile maxim independente şi mulțimile dominante este evidentă. Algoritmul folosit pentru determinarea

mulțimilor dominante va fi organizat după o tehnică similară celui descris în 3.2.

Teoremă: O mulțime de vârfuri a grafului este maxim independentă atunci şi numai atunci când este una dominantă.

□

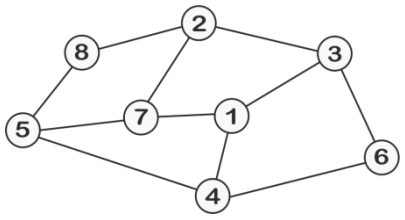
Necesitate. Fie S $(S \subset V)$ maxim independentă. Fie că S nu e dominantă. Atunci $\exists v' \in V : v' \notin \{S \bigcup \Gamma(S)\}$. Rezultă că $S \bigcup \{v'\}$ este independentă, ceea ce contrazice presupunerea inițială.

Suficiență. Fie S $(S \subset V)$ dominantă. Presupunem că S nu e maxim independentă. Atunci $\exists v' \in V$ astfel încât nu există nici o $(u, v') : u \in S$. Rezultă că S nu este dominantă, ceea ce contrazice presupunerea inițială.

■

Exerciţii

1. Pentru graful din imagine, determinați:
   a) Mulțimea maxim independentă de putere minimă
   b) Mulțimea maxim independentă de putere maximă
   c) Una dintre mulțimile dominante



2. Completați implementarea prezentată cu subprogramele lipsă şi structurile de date pentru a obține un program funcțional, care va realiza algoritmul pentru grafuri cu $|V| < 30$.

3. Realizați o modificare a programului, care va verifica toate optimizările indicate în descrierea algoritmului (3.2)

4. Implementați un algoritm similar celui pentru determinarea mulțimii maxim independente, care va construi mulțimile dominante pe un graf orientat.

5. Elaborați o funcție pentru generarea aleatorie a grafurilor şi estimați statistic timpul mediu de lucru al algoritmilor realizați în dependență de numărul de vârfuri ale grafului

Capitolul 4. Colorări

În acest capitol:

- Numărul cromatic al grafului
- Teoreme cu privire la colorarea grafurilor
- Algoritmi exacţi de colorare a grafurilor
- Algoritmi euristici de colorare

4.1 Numărul cromatic

Colorarea grafului – procesul de atribuire unor caracteristici coloristice vârfurilor acestuia.

Graful se numeşte *r*-cromatic, dacă vârfurile lui pot fi colorate, folosind *r* culori diferite, astfel încât să nu existe două vârfuri adiacente, colorate la fel. Valoarea minimă a lui *r*, pentru care o asemenea colorare este posibilă se numeşte *număr cromatic* al grafului $G=(V,E)$ şi se notează $\gamma(G)$.

Nu există o formulă generală pentru determinarea $\gamma(G)$ reieşind din numărul de vârfuri (n) şi muchii (m) ale grafului. Este evidentă posibilitatea de a colora graful în k culori ($\forall\ \gamma(G)\le k\le n$). Totuşi, reieşind din faptul că vârfurile colorate formează mulțimi independente, pot fi stabilite anumite numere cromatice:

$\gamma(\bar{K}_n)=1,\ \gamma(K_n)=n,\ \gamma(K_{n_1,n_2})=2,\ \gamma(T)=2$, etc.

Într-un graf colorat, mulțimea vârfurilor, cărora le este atribuită una şi aceeaşi culoare se numeşte *clasă monocromă*.

Teorema 1. $\gamma(G)\le 1+\Delta(G)$.[6]

□ Inducție după numărul de vârfuri n.

[6] $\Delta(G)$ - puterea maximă a vârfurilor din G.

a) Dacă n=1, $\gamma(G)=1,\ \Delta(G)=0$. 1=1.

b) Fie $\forall G=(V,E):|V|<k\quad \gamma(G)\le 1+\Delta(G).$

c) $|V|=k$. Pentru un vârf arbitrar $v\in V$ are loc relația: $\gamma(G-v)\le\Delta(G-v)+1\le\Delta(G)+1.$ Puterea v nu depăşeşte $\Delta(G)$. Prin urmare cel puțin una din $\Delta(G)+1$ culori este liberă pentru v. Folosind anume o culoare liberă se obține colorarea grafului cu $\Delta(G)+1$ culori. ■

Teorema 2. Fie $\gamma=\gamma(G),\quad \bar{\gamma}=\gamma(\bar{G})$. Atunci sunt adevărate relațiile:

$$2\sqrt{n}\le\gamma+\bar{\gamma}\le n+1$$

$$n\le\gamma\bar{\gamma}\le\left(\frac{n+1}{2}\right)^2$$

□

a) Fie $\gamma(G)=k$ şi $V_1,V_2,...,V_k$ - clase monocrome. $|V_i|=p_i$. $\sum_{i=1}^{k}p_i=n$. Prin urmare, $\max\limits_{i=1}^{k}p_i>\frac{n}{k}$. Deoarece V_i sunt mulțimi independente, în $\bar{G}$ ele vor forma subgrafuri complete. Rezultă că $\bar{\gamma}\ge\max\limits_{i=1}^{k}p_i\ge\frac{n}{k}$. Astfel, $\gamma\bar{\gamma}\ge k\times\frac{n}{k}=n$.

b) Se ştie că media geometrică nu depăşeşte media aritmetică: $\frac{a+b}{2}\ge\sqrt{ab}$. Prin urmare, $\gamma+\bar{\gamma}\ge 2\sqrt{\gamma\bar{\gamma}}\ge 2\sqrt{n}$

c) Prin inducție după n se va demonstra că $\gamma+\bar{\gamma}\le p+1$.

Pasul 1. $n=1,\ \Rightarrow\ \gamma=1\&\bar{\gamma}=1.$

Pasul k-1. Fie $\gamma+\bar{\gamma}\le k$ pentru toate grafurile cu $k-1$vârfuri.

Pasul k. Fie G cu k vârfuri şi un vârf $v\in V$. Atunci $\gamma(G)\le\gamma(G-v)+1.$ Respectiv $\gamma(\bar{G})\le\gamma(\bar{G}-v)+1.$ Dacă $\gamma(G)<\gamma(G-v)+1$ sau $\gamma(\bar{G})<\gamma(\bar{G}-v)+1$ atunci

$\gamma+\bar{\gamma}=\gamma(G)+\gamma(\bar{G})<\gamma(G-v)+1+\gamma(\bar{G}-v)+1\le k+2$. Deoarece există cel puțin o relație „<” rezultă $\gamma+\bar{\gamma}\le k+1$.

Fie $\gamma(G)=\gamma(G-v)+1$ şi $\gamma(\bar{G})=\gamma(\bar{G}-v)+1$. Se consideră $d=d(v)$ în G. Respectiv în $\bar{G}$ gradul vârfului va fi $\bar{d}=k-1-d$.

În continuare, $d\ge\gamma(G-v)$,[7] şi $\bar{d}=k-1-d\ge\gamma(\bar{G}-v)$.

Atunci

$$\gamma+\bar{\gamma}=\gamma(G)+\gamma(\bar{G})=\gamma(G-v)+1+\gamma(\bar{G}-v)+1\ge d+1+k-d-1+1=k+1$$

d) Din a) – c) $2\sqrt{\gamma\bar{\gamma}}\le\gamma+\bar{\gamma}\le n+1$. Prin urmare $\gamma\bar{\gamma}\le\left(\frac{n+1}{2}\right)^2$. ■

Teorema 3 În orice graf planar există un vârf de grad nu mai mare decât cinci. (corolar din formula Euler)

□ Fie $\forall v,\ \ d(v)\ge 6$. Atunci $6n\le\sum_{v\in V}d(v)=2m$. Rezultă $3n\le m$. Dar $m\le 3n-6$. Prin urmare $3n\le 3n-6$ - contrazicere. ■

Teorema (celor 5 culori) Pentru orice graf planar $\gamma(G)\le 5$.

□

Este suficient să se cerceteze grafurile planare conexe, deoarece între componentele izolate nu există legături.

Demonstrația va fi realizată prin inducție, după numărul de vârfuri.

Pasul 0. dacă $n\le 5\ \ \Rightarrow\gamma(G)\le 5$.

Pasul k. Fie teorema este adevărată pentru toate grafurile planare cu k vârfuri.

Pasul k+1. Se va cerceta graful G cu $k+1$vârfuri. Conform corolarului din formula Euler, în G există un vârf v : $d(v)\le 5$. Conform pasului precedent al inducției, $\gamma(G-v)\le 5$. Se va colora vârful v.

[7] S-a presupus că $\gamma(G)=\gamma(G-v)+1$. Dacă $d<\gamma(G-v)$ atunci vârful *v* poate fi colorat în oricare din culorile libere: $\gamma(G-v)-d$. Astfel se va obține o colorare $\gamma(G-v)$ a grafului G.

a) Dacă $d(v) < 5$, atunci există cel puțin o culoare liberă pentru colorarea v.

b) Dacă $d(v) = 5$, dar pentru $\Gamma^+(v)$ nu sunt folosite cinci culori distincte, atunci există o culoare liberă pentru colorarea v

c) Dacă $d(v) = 5$, şi pentru $\Gamma^+(v)$ sunt folosite toate cinci culori, se va încerca recolorarea vârfurilor. Fie $v_1,...,v_5$ - vârfurile din $\Gamma^+(v)$ şi v_i are culoarea i. Prin $G_{1,3}$ se va nota subgraful generat de vârfurile colorate în culorile 1 sau 3 pe colorarea de 5 culori a grafului $G - v$. Dacă v_1 şi v_3 se află în componente de conexitate diferite a $G_{1,3}$, în componenta în care se află v_1 recolorăm $1 \leftrightarrow 3$ pe toate vârfurile. $G - v$ va rămâne 5-colorat, dar culoarea 1 va fi liberă pentru v. Dacă v_1 şi v_3 se află în aceeaşi componentă de conexitate a $G_{1,3}$, există o altă pereche de vârfuri care se află în componente diferite de conexitate (G este planar). Fie v_2 şi v_4 se află în componente diferite de conexitate a $G_{2,4}$. În componenta în care se află v_2 recolorăm $2 \leftrightarrow 4$ pe toate vârfurile. $G - v$ va rămâne 5-colorat, dar culoarea 2 va fi liberă pentru v. ■

Ipoteza (celor 4 culori) Orice graf planar poate fi colorat cu patru culori.

4.2. Algoritmul exact de colorare

Abordare recursivă, pseudocod.

Input: Graful G

Output: Toate colorările posibile ale grafului G

Pas 0. $k \leftarrow 0$ (indicele culorii curente)

Pas 1. În graful G se alege o careva mulțime maximal independentă S.

Pas 2. $k \uparrow$. Mulțimea S se colorează în culoarea k.

Pas 3. $G \leftarrow G - S$. Dacă $G \neq \varnothing$ se revine la pasul 1.

Algoritmul generează toate posibilitățile de formare a mulțimilor independente disjuncte pe vârfurile grafului G. Prin urmare va fi generată şi o colorare optimă cu număr minim de culori. Complexitatea algoritmului este una exponențială – pasul 1 are o asemenea complexitate, deoarece generează mulțimi maximal independente. Suplimentar, se formează şi un arbore de soluții, numărul de noduri în care, în general este proporțional cu 2^n.

Cele expuse denotă ineficiența algoritmului exact pentru grafurile cu un număr mare de vârfuri.

4.3. Algoritmi euristici de colorare

Algoritmul exact, descris anterior nu poate fi utilizat pentru a obține o colorare eficientă în timp redus. Problema poate fi rezolvată cu ajutorul unor algoritmi euristici, care vor furniza soluții ce pot diferi de cea optimă, fiind totuşi, destul de apropiate de ea.

Algoritmul de colorare consecutivă

Input: Graful G
Output: O colorare posibilă a grafului G

Pas 1. Se sortează vârfurile din G în ordinea descreşterii gradelor d.

Pas 2. Inițializare $V \leftarrow \{1,...,n\}$ $C[\,] \leftarrow 0$, $k \leftarrow 1$. Vectorul culorilor se inițializează cu 0. Culoarea activă – 1.

Pas 3. Cât $V \neq \varnothing$ se repetă

Pentru fiecare $v \in V$

  Pentru fiecare $u \in \Gamma^+(v)$

    Dacă $C[u] = k$, se trece la următorul v

  $C[v] \leftarrow k$; $V \leftarrow V - \{v\}$

$k \uparrow$

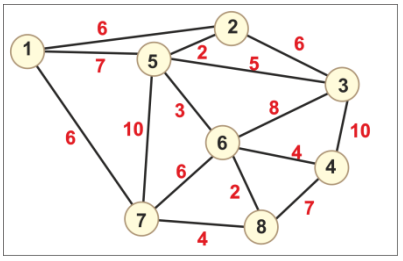
Implementare

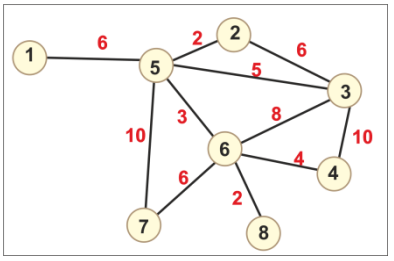
Input: Graful G : matricea de adiacență **a**, structura vârfurilor **v**.
Output: O colorare posibilă a grafului G , în vectorul culorilor **c**.

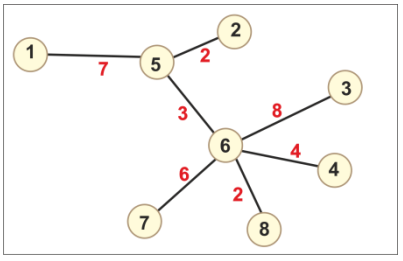
```
int sort ()
{ … // sortează vârfurile în descreşterea gradelor }

int readdata()
{ … // citeşte graful G. – matricea de adiacenţă }

int printcc()
{ … // afişează o colorare a grafului G }

int nocolor()
{ verifică prezenţa vârfurilor necolorate }

int main(int argc, char *argv[])
{ int j,i,r;
  readdata();
// initializare
  for (i=1;i<=n;i++)
   {v[i].ind=i;
    for (j=1; j<=n;j++) v[i].grad+=a[i][j];
   }
// sortare
   sort();
// colorare
k=1;
fillc(c,0,n);
while (nocolor())
{  for( i=1;i<=n;i++)
  { if (c[v[i].ind]==0)
      { r=0;
        for (j=1;j<=n;j++)
            if (a[v[i].ind][j] != 0 && c[j]==k) r++;
        if (r==0)  c[v[i].ind]=k;
      }
   }
   k++;
}
printcc();
return 0;
}
```

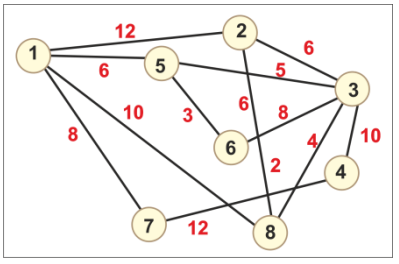
Exerciţii:

1. Pentru grafurile din imagine determinați $\gamma(G)$









2. Elaborați un program care va determina colorarea minimă pentru grafuri cu $|V| \le 10$. Implementați algoritmul exact de colorare.

3. Elaborați un program care va determina o colorarea aproximativă pentru grafuri cu $|V| \le 100$, folosind algoritmul euristic de colorare.

4. Construiți grafuri, pentru care algoritmul euristic, descris anterior va construi o soluție diferită de cea optimă.

Capitolul 5. Drumuri minime în graf

În acest capitol:

- Drumul minim între două vârfuri ale grafului
- Drumul minim de la un vârf la toate vârfurile grafului (algoritmul Dijkstra)
- Drumul minim între toate perechile de vârfuri (algoritmul Floyd)

Preliminarii

Ponderea unei muchii (u,v) în graf este o caracteristică numerică a relației între vârfurile u şi v. Ea poate specifica distanța dintre vârfuri, sau capacitatea unui canal de transmitere a datelor, a unei conducte, magistrale auto, etc. Atribuirea ponderilor la muchiile unui graf permite formularea unei serii noi de probleme, inclusiv probleme de optimizare. Una dintre aceste probleme este problema drumurilor minime.

Problema drumurilor minime într-un graf arbitrar $G=(V,E)$, muchiile căruia au ponderi nenegative descrise de matricea costurilor $C=C[i,j]$ este de a determina un drum de lungime minimă între două vârfuri date s şi t ale grafului, în condiția că un asemenea drum există. Pentru problema dată există mai multe formulări. Iată doar câteva din ele:

- Să se determine distanța minimă între două vârfuri ale grafului (dacă între ele există un drum);
- Să se determine distanța minimă de la un vârf al grafului la toate celelalte vârfuri;
- Să se determine distanțele minimale între toate perechile de vârfuri.

5.1 Distanţa minimă între două vârfuri. Algoritmul Dijkstra

Algoritmul Dijkstra se bazează pe aplicarea marcajelor pentru vârfurile în care se poate ajunge dintr-un vârf dat. Inițial marcajele au valori temporare suficient de mari, care pe parcursul repetării iterațiilor se micşorează, până la atingerea valorilor minime. La fiecare iterație a algoritmului, exact un dintre marcaje devine permanent, adică indică drumul minim până la unul dintre vârfuri. Prin urmare algoritmul va conține cel mult n iterații.

Fie s vârful de la care se calculează distanțele minime, t – vârful până la care se calculează distanța. Pentru vârful v_i marcajul său va fi notat prin $l(v_i)$. Se va folosi un marcaj cu două componente (spre deosebire de algoritmul clasic Dijkstra). Componentele vor conține a) valoarea distanței curente de la s la v_i şi b) indicele vârfului v_j din care se ajunge în v_i.

Pseudocod

Pas 1. Se consideră $l(s) \leftarrow (0, s)$ – marcaj temporar pentru vârful s. Pentru toate vârfurile $v_i \in V, v_i \neq s$ se consideră marcajele nedefinite. Vârful activ $p \leftarrow s$.

Pas 2. (*Recalcularea marcajelor*)
Pentru toate vârfurile $v_i \in \Gamma(p)$ care nu au marcaje permanente, se recalculează componentele distanță ale marcajelor în corespundere cu formula:
$l(v_i).dist \leftarrow \min\{l(v_i).dist, l(p).dist + c[p][v_i]\}$.

Dacă $l(v_i).dist > l(p).dist + c[p][v_i]$ atunci se modifică şi componenta marcajului, care indică sursa $l(v_i).sursa \leftarrow p$.

Vârfului p i se atribuie marcaj permanent.

Pas 3. (*Determinarea vârfului pentru cercetare*)
Între toate vârfurile cu marcaje temporare se determină cel cu marcaj minim pentru componenta distanță:
$l(v_i^*).dist = \min l(v_i).dist$
Se consideră $p \leftarrow v_i^*$

Pas 4. (*Repetarea iterației*)
Dacă $p = t$ atunci marcajul $l(t).dist$ indică costul minim a drumului di *s* în *t*. Pentru restabilirea drumului minim se trece la pasul 5. În caz contrar $p \neq t$ se revine la pasul 2.

Pas 5. Pentru restabilirea traiectoriei se foloseşte a doua componentă a marcajului: se consideră $x = t$.
Se include în traiectorie muchia $(x, l(x).sursa)$. Se consideră $x \leftarrow l(x).sursa$ Procesul se repetă cât timp $x \neq s$.

Demonstrarea corectitudinii

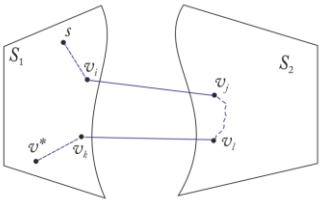
Teoremă. Algoritmul descris determină corect distanța minimă între orice pereche de vârfuri *s* şi *t*.

□

Fie la un careva pas marcajele permanente indică distanțele minime pentru vârfurile dintr-o submulțime $S_1 \in V$. $S_2 = V / S_1$

Se presupune contrariul: pentru un careva v^* din S drumul minim din s către v^* trece prin S_2.

Fie v_j primul punct din S_2, care aparține drumului minim (s,v^*), v_l ultimul punct din S_2, care aparține drumului minim (s,v^*) . De asemenea, fie v_i ultimul punct din S_1 până la părăsire, iar v_k primul punct din S_1 după revenirea drumului minim către vârfurile din această mulțime. Drumul minim $l(v_i,v_k)$ aparține integral S_1. La fel drumurile minime $l(s,v_i)$ şi $l(v_k,v^*)$ aparțin integral S_1. Deoarece drumul minim din v_i în v_k trece doar prin S_1 rezultă că $l(v_i,v_k)<c[i][j]+l(v_j,v_l)+c[l][k]$.



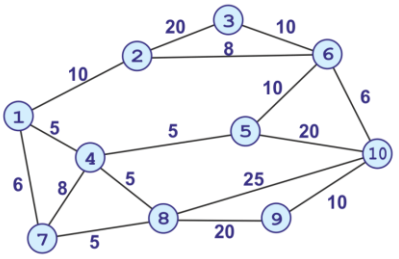
Atunci

$$l(s,v^*)=l(s,v_i)+c[i][j]+l(v_j,v_l)+c[l][k]+l(v_k,v^*)>l(s,v_i)+l(v_i,v_k)+l(v_k,v^*)$$

Prin urmare $l(s,v^*)$ nu este drum minim, ceea ce contrazice presupunerea inițială. ■

Exemplu:

Fie dat graful $G=(V,E)$ (desenul 5.2)



Des. 5.2

Se cere să se determine distanța minimă de la vârful 1 la vârful 10.

Inițializarea

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
sursa	1									
marcaj	p									

Iterația 1

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	∞	5	∞	∞	6	∞	∞	∞
sursa	1	1		1			1			
marcaj	p	t		t*			t			

Iterația 2

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	∞	5	10	∞	6	10	∞	∞
sursa	1	1		1	4		1	4		
marcaj	p	t		p	t		t*	t		

Iterația 3

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	∞	5	10	∞	6	10	∞	∞
sursa	1	1		1	4		1	4		
marcaj	p	t*		p	t		p	t		

Iterația 4

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	30	5	10	18	6	10	∞	∞
sursa	1	1	2	1	4	2	1	4		
marcaj	p	p	t	p	t*	t	p	t		

Iterația 5

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	30	5	10	18	6	10	∞	30
sursa	1	1	2	1	4	2	1	4		5
marcaj	p	p	t	p	p	t	p	t*		t

Iterația 6

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	30	5	10	18	6	10	30	30
sursa	1	1	2	1	4	2	1	4	8	5
marcaj	p	p	t	p	p	t*	p	p	t	t

Iterația 7

Vârf	1	2	3	4	5	6	7	8	9	10
distanţa	0	10	30	5	10	18	6	10	30	24
sursa	1	1	2	1	4	2	1	4	8	6
marcaj	p	p	t	p	p	p	p	p	t	t*

Lungimea drumului minim din vârful 1 în 10 este 24. Traseul va fi determinat de calea: 10, 6, 2, 1

Pentru a determina distanțele minime de la un vârf dat până la toate celelalte vârfuri ale grafului (a componentei conexe) algoritmul va fi oprit în cazul când toate vârfurile au marcaje permanente.

Implementare

Graful este descris prin matricea de adiacență **a**. Marcajele se păstrează în tabloul liniar **vert** cu elemente tip articol, având componentele *distanța, sursa, stare*.

```
int find()
{ … // fuctia returneaza indicele varfului cu marcaj temporar
    //de valoare minima. Daca nu exista, returneaza 0.
}

int tipar()
{ … // functia afiseaza tabloul de marcaje
}

int main()
{… // citire date
 … // formare marcaje
 for (i=1;i<=n;i++)
        for (j=i+1; j<=n; j++) k+=a[i][j];
 k++;
 for (i=1;i<=n;i++)
      {vert[i].dist=k; vert[i].sursa=i; vert[i].stare=0;}
 vert[s].stare=1; vert[s].dist=0;
// repetare iteratii
 while (find ())
 { p=find();
   for(i=1;i<=n;i++)
    if (a[p][i] !=0 && vert[i].stare !=2 )
     { dc= vert[p].dist+a[p][i];
       if(dc<vert[i].dist){vert[i].dist=dc; vert[i].sursa=p;}
       vert[i].stare=1;
```

```
      }
    vert[p].stare=2;
 }
// afisare rezultate
tipar();
return 0;
}
```

5.2 Distanţa minimă între toate vârfurile. Algoritmul Floyd

Din paragraful precedent rezultă o soluție evidentă a problemei determinării tuturor drumurilor minime între vârfurile grafului: algoritmul Dijkstra se lansează având în calitate de sursă consecutiv, toate vârfurile grafului. Eistă şi un algoritm mai eficient, propus de Floyd [7, p. 480]. La baza algoritmului este ideea unei secvențe din n transformări consecutive ale matricei distanțelor C. La transformarea cu indicele k matricea conține lungimea drumului minim între orice pereche de vârfuri, cu restricția că pentru orice două vârfuri v_i, v_j drumul minim dintre ele trece doar prin vârfurile mulțimii $\{v_1, v_2, ..., v_k\}$

Pseudocod

Pas 0. (Preprocesare). Fie dată matricea distanțelor C, în care

$$C[i][j] = \begin{cases} 0, & i = j \\ d_{i,j}, & \exists (i,j) \in E \\ \infty, & \nexists (i,j) \in E \end{cases}$$

Pas 1. (Inițializare) $k \leftarrow 0$

Pas 2 $k \uparrow$

Pas 3 Pentru toți $i \neq k : C[i][k] \neq \infty$, şi pentru toți $j \neq k : C[k][j] \neq \infty$ se calculează $C[i][j] = \min\left[C[i][j], C[i][k] + C[k][j]\right]$

Pas 4 dacă $k = n$ – sfârşit, în caz contrar se revene la pasul 2.

Dacă problema rezolvată necesită şi restabilirea traseelor care formează drumurile minime, este formată o matrice auxiliară T, care iniţial are elementele liniei i egale cu i.

Mai apoi, la transformarea elementului $C[i][j]$ din matricea distanţelor se transformă şi elementul respectiv din matricea T:

$T[i][j]=T[k][j]$ dacă $C[i][j]>C[i][k]+C[k][j]$.

Traseul ce corespunde drumului minim între vârfurile v_i, v_j va fi format din secvenţa $v_i, v_1, v_2, ..., v_r, v_{r+1}, v_j$, unde

$v_{r+1}=T[i][j], \quad v_r=T[i][v_{r+1}], \quad v_{r-1}=T[i][v_r], ...$

Implementare

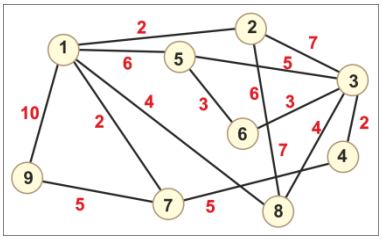
Graful este descris prin matricea de adiacenţă **a**, care ulterior este transformată în matricea distanţelor.

```
int main()
{
… // citire date iniţiale
// modelare infinit
 for (i=1;i<=n;i++)
        for (j=1; j<=n; j++)
              p+=a[i][j];
 p++;
// creare matrice costuri
 for (i=1;i<=n;i++)
        for (j=1; j<=n; j++)
         if (a[i][j]==0 && i != j) a[i][j]=p;
// calculare drumuri minime
for (k=1;k<=n;k++)
 for (i=1;i<=n;i++)
  if (i!=k && a[i][k]!=p)
     for (j=1;j<=n;j++)
         if (j!=k && a[k][j] !=p)
            if (a[i][j]>a[i][k]+a[k][j])
                a[i][j]=a[i][k]+a[k][j];
… // afisare rezultate
return 0;
}
```

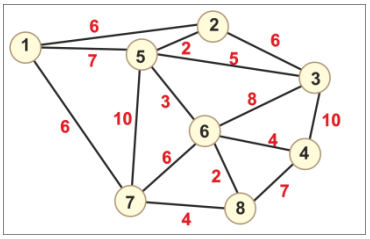
Exerciţii

1. Determinați distanța minimă între vârfurile indicate pe grafurile de mai jos:

3 – 9



1 – 4



2. Pentru implementarea algoritmului Dijkstra, propusă în 5.1 elaborați o funcție pentru restabilirea lanțului care corespunde drumului minim de la vârful dat *s* către destinația *t*.

3. Realizați o implementare a algoritmului Floyd (5.2) fără matricea auxiliară pentru restabilirea lanțurilor care corespund drumurilor minime.

4. Extindeți implementarea din exercițiul 2, adăugând opțiunea de restabilire a lanțurilor care corespund drumurilor minime.

5. Estimați complexitatea algoritmului Dijkstra.

6. Estimați complexitatea algoritmului Floyd.

Capitolul 6. Centre în graf

În acest capitol:

- Centre în graf
- Centre interioare şi exterioare
- Raza grafului
- Algoritmi exacţi pentru determinarea centrului
- Centrul absolut
- P – centru
- Algoritmi euristici pentru determinarea p-centrelor

În activitatea practică deseori apar probleme de amplasare optimă a unor puncte de deservire (staţii, puncte de control, utilaj) într-o reţea de localităţi, încăperi etc. Punctele pot fi unul sau câteva, în dependenţă de condiţiile problemei. Formulată în termeni uzuali, problema este de a găsi un punct, care ar minimiza suma distanţelor de la oricare alt punct al reţelei până la el, sau în termenii teoriei grafurilor - de a determina vârful grafului sau punctul geometric, care aparţine unei muchii, astfel încât acesta va minimiza suma distanţelor până la toate celelalte vârfuri a grafului. Se va considera suplimentar, că drumurile pentru localităţile (vârfurile) situate pe acelaşi lanţ sunt distincte.

6.1 Divizări

Pentru orice vârf v_i al grafului $G=(V,E)$ se va nota prin $R_\lambda^0(v_i)$ mulţimea de vârfuri v_j ale grafului G care pot fi atinse din v_i prin căi care nu depăşesc mărimea λ. Prin $R_\lambda^t(v_i)$ se va nota mulţimea de vârfuri v_j ale grafului G din care v_i poate fi atins prin căi care nu

depăşesc mărimea λ. Astfel, pentru orice vârf v_i al grafului G se notează:

$$R_\lambda^0(v_i) = \left\{ v_j : d\left(v_i, v_j\right) \le \lambda,\ v_j \in V \right\}$$

$$R_\lambda^t(v_i) = \left\{ v_j : d\left(v_j, v_i\right) \le \lambda,\ v_j \in V \right\}$$

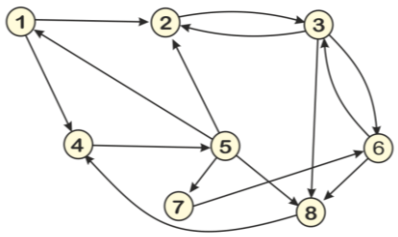
Pentru fiecare din vârfurile v_i vor fi definite 2 mărimi

$$s_0(v_i) = \max_{v_j \in V} \left[d\left(v_i, v_j\right) \right]$$

$$s_t(v_i) = \max_{v_j \in V} \left[d\left(v_j, v_i\right) \right]$$

valorile $s_0(v_i)$ şi $s_t(v_i)$ se numesc *indici de separare exterior* şi respectiv *interior* ai vârfului v_i.

Exemplu:



Des. 6.1 Graf tare conex şi matricea distanţelor lui.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	s_0
v_1	0	1	2	1	2	3	3	3	3
v_2	5	0	1	3	4	2	5	2	5
v_3	4	1	0	2	3	1	4	1	4
v_4	2	2	3	0	1	3	2	2	3
v_5	1	1	2	2	0	2	1	1	2 *
v_6	4	4	1	2	3	0	3	1	4
v_7	6	3	2	3	4	1	0	2	6
v_8	3	3	4	1	2	4	3	0	4
s_t	6	4	4	3 *	4	4	5	3 *	

Este evident, că pentru vârful v_i, $s_0(v_i)$ va fi determinată de valoarea maximă din rândul i al matricei distanţelor, iar $s_t(v_i)$ - de valoarea maximă din coloana i. Tot de aici rezultă că valorile $s_0(v_i)$ şi $s_t(v_i)$ au valori finite pentru orice i numai dacă graful este tare conex.

6.2 Centrul şi raza grafului

Def. Vârful v_0^* pentru care are loc relația $s_0(v_0^*) = \min_{v_i \in V}\left[s_0(v_i)\right]$ se numeşte *centru exterior* al grafului *G*. Vârful v_t^* pentru care are loc relația $s_t(v_t^*) = \min_{v_i \in V}\left[s_t(v_i)\right]$ se numeşte *centru interior* al grafului *G*.

Centrul exterior este vârful (sau vârfurile) care minimizează cea mai lungă distanță de la el spre oricare alt vârf al grafului. Din matricea distanțelor el poate fi determinat ca minimul valorilor maxime de pe fiecare linie. Pentru graful de pe desenul 6.1 centrul exterior este vârful 5. Centrul interior este vârful (sau vârfurile) care minimizează cea mai lungă distanță spre el de la oricare alt vârf al grafului. Din matricea distanțelor el poate fi determinat ca minimul valorilor maxime de pe fiecare coloană. Pentru graful de pe desenul 6.1 centre exterioare sunt vârfurile 4 şi 8. Într-un graf neorientat centrul interior şi cel exterior coincid.

Def. Valoarea $s_0(v_0^*) = \min_{v_i \in V}\left[s_0(v_i)\right]$ se numeşte *raza exterioară* a grafului *G*. Valoarea $s_t(v_t^*) = \min_{v_i \in V}\left[s_t(v_i)\right]$ se numeşte *raza interioară* a grafului *G*.

Pentru graful de pe desenul 6.1 raza exterioară are valoarea 2, în timp ce raza interioară are valoarea 3.

Pentru un graf neorientat raza interioară şi raza exterioară sunt egale, iar centrele exterioare coincid cu cele interioare.

Def. Valoarea $s(v_0^*) = \min_{v_i \in V}\left[s(v_i)\right]$ se numeşte *raza* grafului neorientat *G*. Valoarea $s_t(v_t^*) = \min_{v_i \in V}\left[s_t(v_i)\right]$ se numeşte *raza interioară* a grafului *G*.

Def. Vârful v_0^* pentru care are loc relația $s(v_0^*) = \min_{v_i \in V}\left[s(v_i)\right]$ se numeşte *centru* al grafului neorientat *G*.

6.3 P-centre

Fie în municipiu sunt amplasate mai multe (P) centre de asistență medicală urgentă, cu echipe mobile de medici. În cazul recepționării unei cereri de asistență la centrul comun de apel, către solicitant se deplasează echipajul de medici de la cel mai apropiat centru de asistență.

În acest caz amplasarea inițială a celor P centre de asistență medicală urgentă este organizată într-un mod, care asigură minimul de aşteptare a pacientului, indiferent de locația acestuia.

Fie V_p o submulțime din V. $|V_p| = p$. Prin $d(V_p, v_i)$ se va nota distanța de la cel mai apropiat vârf din V_p până la vârful v_i:

$$d(V_p, v_i) = \min_{v_j \in V_p}\left[d(v_j, v_i)\right]$$

La fel, prin $d(v_i, V_p)$ se va nota distanța de la vârful v_i până la cel mai apropiat vârf din V_p: $d(v_i, V_p) = \min_{v_j \in V_p}\left[d(v_i, v_j)\right]$

Indicii de separare pentru mulțimea V_p se calculează la fel ca şi pentru vârfurile solitare:

$$s_0(V_p) = \max_{v_j \in V}\left[d(V_p, v_j)\right]$$

$$s_t(V_p) = \max_{v_j \in V}\left[d(v_j, V_p)\right]$$

Def. Mulțimea de vârfuri $V_{p,0}^*$ pentru care are loc relația $s_0(V_{p,0}^*) = \min_{V_p \in G}\left[s_0(V_p)\right]$ se numeşte *p-centru exterior* al grafului *G*.

Mulțimea de vârfuri $V_{p,t}^*$ pentru care are loc relația $s_t(V_{p,t}^*) = \min_{V_p \in G}\left[s_t(V_p)\right]$ se numeşte *p-centru interior* al grafului *G*.

Def. Valoarea $s_0(V_{p,0}^*) = \min_{V_p \in G}\left[s_0(V_p) \right]$ se numeşte *p-raza exterioară* a grafului *G*. Valoarea $s_t(V_{p,t}^*) = \min_{V_p \in G}\left[s_t(V_p) \right]$ se numeşte *p-raza interioară* a grafului *G*.

Algoritmul euristic pentru determinarea p-centrelor

Algoritmul face parte din clasa algoritmilor de optimizare locală, care se bazează pe ideea λ optimizării pe grafuri.

Fie graful neorientat $G = (V, E)$. O mulțime de vârfuri $S \subseteq V$, $|S| = p$ se numeşte λ optimă $(\lambda \le p)$ pentru problema Q, dacă înlocuirea oricăror λ vârfuri din S cu λ vârfuri din $V - S$ nu va îmbunătăți soluția problemei Q, obținută pe mulțimea S. Algoritmul descris este unul general şi poate fi folosit pentru diverse probleme formulate pe grafuri.

Inițial în calitate de soluție este selectată o mulțime arbitrară de vârfuri C, care aproximează *p* centrul. Apoi se verifică, dacă un careva vârf $v_j \in V - C$ poate înlocui vârful $v_i \in C$. Pentru aceasta se formează mulțimea $C' = \left\{ C \cup \{v_j\} - \{v_i\} \right\}$ după care se compară $s(C)$ şi $s(C')$. Ulterior C' este cercetată în acelaşi mod , pentru a obține C''. Procesul se repetă până la obținerea unei mulțimi $\overline{C}$, pentru care nu mai pot fi efectuate îmbunătățiri prin procedeul descris. Mulțimea de vârfuri $\overline{C}$ este considerată *p*-centrul căutat.

Pseudocod

Pas 0. Se formează matricea distanțelor minime (algoritmul Floyd)

Pas 1. Se alege în calitate de aproximare inițială a p-centrului o mulțime arbitrară de vârfuri C. Vârfurile $v_j \in V - C$ vor fi considerate având marcajul stării egal cu 0 (neverificate).

Pas 2. Se alege un vârf arbitrar de stare 0 $v_j \in V - C$ şi pentru fiecare vârf $v_i \in C$ se calculează valorile $\Delta_{i,j} = s(C) - s(C - \{v_i\} \cup \{v_j\})$

Pas 3. Se determină $\Delta_{i_0,j} = \max_{v_i \in C} \left[\Delta_{i,j} \right]$.

i. Dacă $\Delta_{i_0,j} < 0$ vârful v_j este marcat „verificat" (i se aplică marcajul de stare - 1) şi se revine la pasul 2.

ii. Dacă $\Delta_{i_0,j} > 0$, $C \leftarrow \left(C - \{v_i\} \cup \{v_j\}\right)$ vârful v_j este marcat „verificat" (i se aplică marcajul de stare - 1) şi se revine la pasul 2.

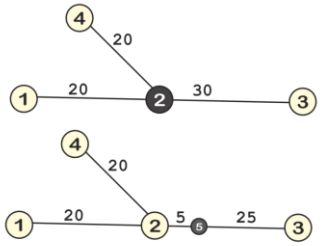
Pas 4. Paşii 2 – 3 se repetă atât timp cât există vârfuri cu marcaj de stare 0. Dacă la ultima repetare a paşilor 2 – 3 nu au fost efectuate înlocuiri (3.ii) se trece la pasul 5. În caz contrar tuturor vârfurilor din $V - C$ li se atribuie marcajul de stare 0, apoi se revine la pasul 2.

Pas 5. STOP. Mulțimea de vârfuri curentă C este o aproximare a *p*-centrului căutat.

6.4 Centrul absolut

În cazul modificării restricțiilor de amplasare a centrului, problema determinării acestuia poate să devină mult mai complicată.

Fie că urmează să fie construit un centru regional al serviciului de ajutor tehnic, care va deservi *n* localități. Acesta poate fi amplasat atât în una din localități, cât şi pe unul din traseele care le unesc. În calitate de criteriu principal de selecție a locației pentru centru se consideră minimizarea distanței până la cel mai îndepărtat punct (localitate) deservit. Problema poate fi modelată pe un graf, vârfurile căruia corespund localităților, iar muchiile – traseelor între localități. Ponderea fiecărei muchii este distanța dintre localitățile adiacente ei. Locația optimă poate fi atât în unul din vârfurile sau un vârf nou, amplasat pe una din muchiile grafului inițial (desenul 6.2).



Des. 6.2
Amplasarea optimă a centrului de deservire:
a. Numai pe vârfurile existente
b. Pe vârfurile sau muchiile existente

Fie (v_i, v_j) muchia cu ponderea $c_{i,j}$. Un punct interior u al acestei muchii poate fi descris prin intermediul lungimilor $l(v_i, u)$ şi $l(u, v_j)$ a segmentelor $[v_i, u]$, respectiv $[u, v_j]$ ținând cont de relația: $l(v_i, u) + l(u, v_j) = c_{i,j}$

În punctul u se va defini un nou vârf al grafului, cu puterea 2, adiacent vârfurilor v_i şi v_j. Pentru u se vor calcula aceleaşi două valori:

$$s_0(u) = \max_{v_j \in V}\left[d\left(u, v_j\right)\right], \qquad s_t(u) = \max_{v_j \in V}\left[d\left(v_j, u\right)\right]$$

Def. Punctul u_0^* pentru care are loc relația $s_0(u_0^*) = \min_{u \in G}\left[s_0(u)\right]$ se numeşte *centru exterior absolut* al grafului *G*. Vârful u_t^* pentru care are loc relația $s_t(u_t^*) = \min_{u \in G}\left[s_t(u)\right]$ se numeşte *centru interior absolut* al grafului *G*.

Def. Valoarea $s_0(u_0^*) = \min_{u \in G}\left[s_0(u)\right]$ se numeşte *raza exterioară absolută* a grafului *G*. Valoarea $s_t(u_t^*) = \min_{u \in G}\left[s_t(u)\right]$ se numeşte *raza interioară absolută* a grafului *G*.

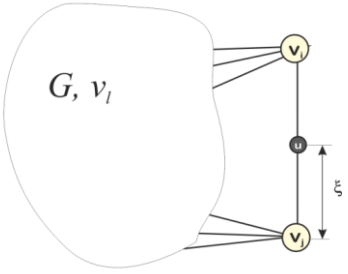
Pentru graful reprezentat în figura 6.2 Centrul absolut va fi un vârf suplimentar (5) poziționat pe muchia (2, 3) la distanța 5 de vârful 2 şi 25 de vârful 3. Pentru acest punct raza absolută va avea valoarea 25. Deplasarea punctului în stânga sau în dreapta va duce la creşterea razei.

6.5 Metoda Hakimi pentru determinarea centrului absolut

În cazul când problema inițială cere amplasarea centrelor de deservire numai în anumite puncte (vârfuri ale grafului) problema se rezolvă nemijlocit, folosind matricea distanțelor din graf, prin metode descrise anterior. Dacă însă se admite şi amplasarea în puncte interioare ale muchiilor grafului, atunci urmează să fie determinat centrul absolut al grafului. În acest scop poate fi folosită metoda propusă de Hakimi [6, p. 104]:

1. Pentru fiecare muchie e_k a grafului se va determina punctul (sau punctele) $u(e_k)$ cu indicele de separare minim
2. Dintre toate punctele $u(e_k)$ se va alege $u^*(e_k) = \min\limits_{e \in E}\left[u(e_k)\right]$.

Realizarea pasului doi este elementară. În continuare se va cerceta pasul 1. Fie muchia e_k între vârfurile v_i şi v_j (desenul 6.3).



Des 6.3 Determinarea centrului absolut pe muchie

Indicii de separare a punctului u se calculează după formula

$$s_0(u) = \max_{v_j \in V}\left[d(u, v_j)\right], \quad s_t(u) = \max_{v_j \in V}\left[d(v_j, u)\right]$$

Pentru punctul u_k de pe muchia e_k se obține

$$s(u_k) = \max_{v_l \in V}\left[d(u_k, v_l)\right] = \max_{v_l \in V}\left[\min\{l(u_k, v_j) + d(x_j, x_l), l(u_k, v_i) + d(x_i, x_l)\}\right]$$

deoarece drumurile minimale până la celelalte vârfuri vor trece în mod obligatoriu prin punctele v_i sau v_j. $l(u_k, v_j)$ şi $l(u_k, v_i)$ sunt lungimile componentelor muchiei e_k, în care o divide u_k.

Fie $l(u_k, v_j) = \xi$. Atunci $l(u_k, v_i) = c_{i,j} - \xi$ iar formula precedentă capătă forma:

$$s(u_k) = \max_{v_l \in V} \left[d(u_k, v_l) \right] = \max_{v_l \in V} \left[\min\{\xi + d(x_j, x_l), c_{i,j} - \xi + d(x_i, x_l)\} \right]$$

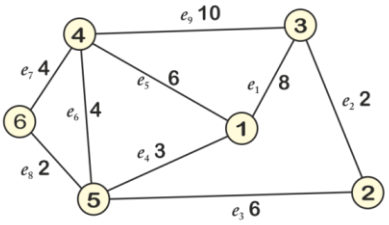
Pentru orice elemente fixate v_l şi e_k valoarea $s(u_k)$ poate fi calculată ca o funcție liniară de variabila ξ. Pentru aceasta sunt separate expresiile:

$$T_l = \xi + d(v_j, v_l)$$

$$T_l' = c_{i,j} - \xi + d(v_i, v_l)$$

şi cercetate ca funcții de ξ. Pentru graficele funcțiilor se determină punctul de intersecție şi semidreptele inferioare ce pornesc din el. Aceste semidrepte se vor numi *semidrepte de minimizare* inferioare. Semidreptele de minimizare se construiesc pentru toate $v_l \in V$, apoi pe baza lor se construieşte linia de maximizare. Aceasta prezintă o linie frântă, cu mai multe puncte de minimum. Din toate punctele de minimum este ales cel maximal (conform formulei.). Acesta va fi centrul absolut situat pe muchia e_k. Pentru a determina centrul absolut al întregului graf se va selecta minimul dintre centrele absolute pe toate muchiile din *G*.

Exemplu



	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	9	8	6	3	5
v_2	9	0	2	10	6	8
v_3	8	2	0	10	8	10
v_4	6	10	10	0	4	4
v_5	3	6	8	4	0	2
v_6	5	8	10	4	2	0

Des. 6.4.

Graful pentru care se calculează centrul absolut şi matricea distanţelor lui.

Pe muchia 1

$T_1 = \xi + d(v_3, v_1) = \xi + 8;$

$T_1' = c_{3,1} + d(v_1, v_1) - \xi = 8 - \xi.$

$T_2 = \xi + d(v_3, v_2) = \xi + 2;$

$T_2' = c_{3,1} + d(v_1, v_2) - \xi = 17 - \xi.$

$T_3 = \xi + d(v_3, v_3) = \xi;$

$T_3' = c_{3,1} + d(v_1, v_3) - \xi = 16 - \xi.$

$T_4 = \xi + d(v_3, v_4) = \xi + 10;$

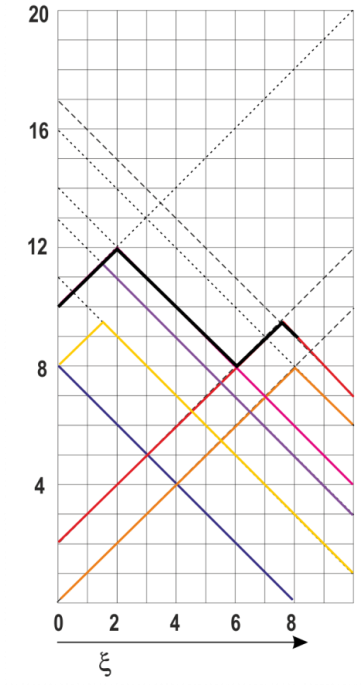
$T_4' = c_{3,1} + d(v_1, v_4) - \xi = 14 - \xi.$

$T_5 = \xi + d(v_3, v_5) = \xi + 8;$

$T_5' = c_{3,1} + d(v_1, v_5) - \xi = 11 - \xi.$

$T_6 = \xi + d(v_3, v_6) = 10 + \xi;$

$T_6' = c_{3,1} + d(v_1, v_6) - \xi = 13 - \xi.$



Des. 6.5.a

Pe muchia 2

$T_1 = \xi + d(v_3, v_1) = \xi + 8;$

$T_1' = c_{3,2} + d(v_1, v_2) - \xi = 11 - \xi.$

$T_2 = \xi + d(v_3, v_2) = \xi + 2;$

$T_2' = c_{3,2} + d(v_2, v_2) - \xi = 2 - \xi.$

$T_3 = \xi + d(v_3, v_3) = \xi;$

$T_3' = c_{3,2} + d(v_3, v_2) - \xi = 4 - \xi.$

$T_4 = \xi + d(v_3, v_4) = \xi + 10;$

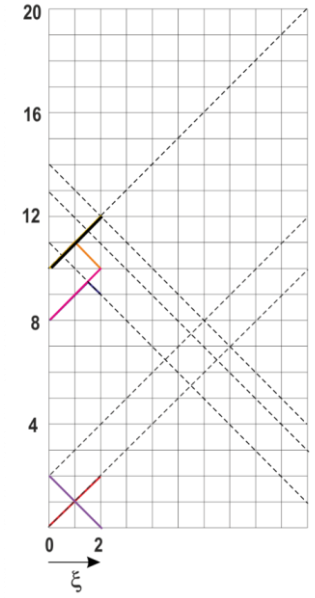
$T_4' = c_{3,2} + d(v_2, v_4) - \xi = 12 - \xi.$

$T_5 = \xi + d(v_3, v_5) = \xi + 8;$

$T_5' = c_{3,2} + d(v_2, v_5) - \xi = 14 - \xi.$

$T_6 = \xi + d(v_3, v_6) = 10 + \xi;$

$T_6' = c_{3,2} + d(v_2, v_6) - \xi = 16 - \xi.$



Des. 6.5.b

Pe muchia 3

$T_1 = \xi + d(v_2, v_1) = \xi + 9;$

$T_1' = c_{5,2} + d(v_2, v_1) - \xi = 9 - \xi.$

$T_2 = \xi + d(v_2, v_2) = \xi;$

$T_2' = c_{5,2} + d(v_5, v_2) - \xi = 12 - \xi.$

$T_3 = \xi + d(v_2, v_3) = \xi + 2;$

$T_3' = c_{5,2} + d(v_5, v_3) - \xi = 14 - \xi.$

$T_4 = \xi + d(v_2, v_4) = \xi + 10;$

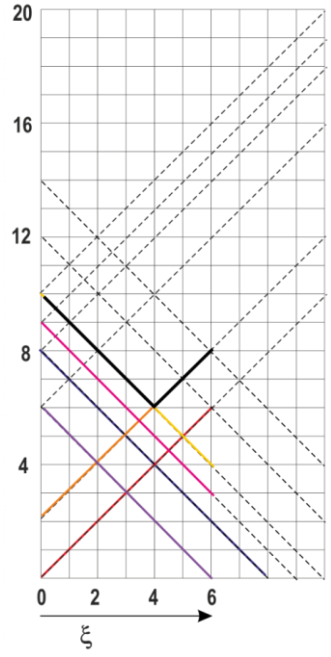
$T_4' = c_{5,2} + d(v_5, v_4) - \xi = 10 - \xi.$

$T_5 = \xi + d(v_2, v_5) = \xi + 6;$

$T_5' = c_{5,2} + d(v_5, v_5) - \xi = 6 - \xi.$

$T_6 = \xi + d(v_2, v_6) = 8 + \xi;$

$T_6' = c_{5,2} + d(v_5, v_6) - \xi = 8 - \xi.$



Des. 6.5.c

Pe muchia 4

$T_1 = \xi + d(v_1, v_1) = \xi;$

$T_1' = c_{5,1} + d(v_5, v_1) - \xi = 6 - \xi.$

$T_2 = \xi + d(v_1, v_2) = \xi + 9;$

$T_2' = c_{5,1} + d(v_5, v_2) - \xi = 9 - \xi.$

$T_3 = \xi + d(v_1, v_3) = \xi + 8;$

$T_3' = c_{5,1} + d(v_5, v_3) - \xi = 11 - \xi.$

$T_4 = \xi + d(v_1, v_4) = \xi + 6;$

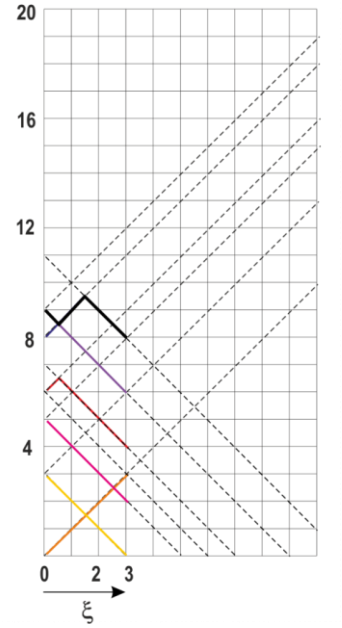
$T_4' = c_{5,1} + d(v_5, v_4) - \xi = 7 - \xi.$

$T_5 = \xi + d(v_1, v_5) = \xi + 3;$

$T_5' = c_{5,1} + d(v_5, v_5) - \xi = 3 - \xi.$

$T_6 = \xi + d(v_1, v_6) = \xi + 5;$

$T_6' = c_{5,1} + d(v_5, v_6) - \xi = 5 - \xi.$



Des. 6.5.d

Pe muchia 5

$T_1 = \xi + d(v_1, v_1) = \xi;$

$T_1' = c_{4,1} + d(v_4, v_1) - \xi = 12 - \xi.$

$T_2 = \xi + d(v_1, v_2) = \xi + 9;$

$T_2' = c_{4,1} + d(v_4, v_2) - \xi = 16 - \xi.$

$T_3 = \xi + d(v_1, v_3) = \xi + 8;$

$T_3' = c_{4,1} + d(v_4, v_3) - \xi = 16 - \xi.$

$T_4 = \xi + d(v_1, v_4) = \xi + 6;$

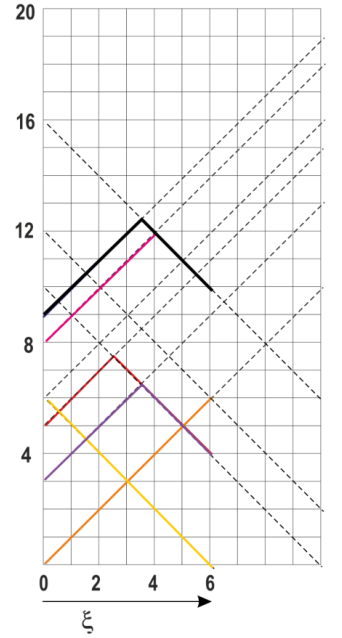
$T_4' = c_{4,1} + d(v_4, v_4) - \xi = 6 - \xi.$

$T_5 = \xi + d(v_1, v_5) = \xi + 3;$

$T_5' = c_{4,1} + d(v_4, v_5) - \xi = 10 - \xi.$

$T_6 = \xi + d(v_1, v_6) = \xi + 5;$

$T_6' = c_{4,1} + d(v_4, v_6) - \xi = 10 - \xi.$



Des. 6.5.e

Pe muchia 6

$T_1 = \xi + d(v_5, v_1) = \xi + 3;$

$T_1' = c_{4,5} + d(v_4, v_1) - \xi = 10 - \xi.$

$T_2 = \xi + d(v_5, v_2) = \xi + 6;$

$T_2' = c_{4,5} + d(v_4, v_2) - \xi = 14 - \xi.$

$T_3 = \xi + d(v_5, v_3) = \xi + 8;$

$T_3' = c_{4,5} + d(v_4, v_3) - \xi = 14 - \xi.$

$T_4 = \xi + d(v_5, v_4) = \xi + 4;$

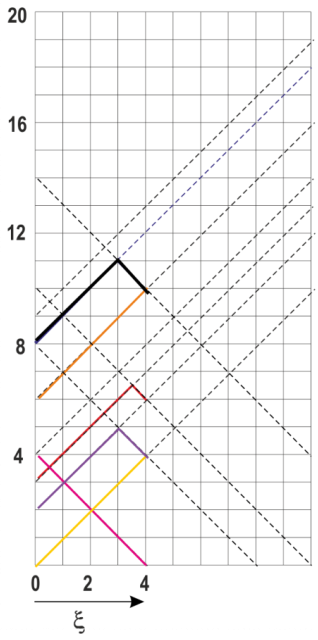
$T_4' = c_{4,5} + d(v_4, v_4) - \xi = 4 - \xi.$

$T_5 = \xi + d(v_5, v_5) = \xi;$

$T_5' = c_{4,5} + d(v_4, v_5) - \xi = 8 - \xi.$

$T_6 = \xi + d(v_5, v_6) = \xi + 2;$

$T_6' = c_{4,5} + d(v_4, v_6) - \xi = 8 - \xi.$



Des. 6.5.f

Pe muchia 7

$T_1 = \xi + d(v_6, v_1) = \xi + 5;$

$T_1' = c_{4,6} + d(v_4, v_1) - \xi = 10 - \xi.$

$T_2 = \xi + d(v_6, v_2) = \xi + 8;$

$T_2' = c_{4,6} + d(v_4, v_2) - \xi = 14 - \xi.$

$T_3 = \xi + d(v_6, v_3) = \xi + 10;$

$T_3' = c_{4,6} + d(v_4, v_3) - \xi = 14 - \xi.$

$T_4 = \xi + d(v_6, v_4) = \xi + 4;$

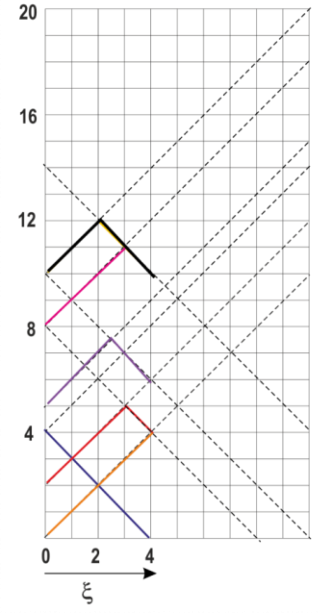
$T_4' = c_{4,6} + d(v_4, v_4) - \xi = 4 - \xi.$

$T_5 = \xi + d(v_6, v_5) = \xi + 2;$

$T_5' = c_{4,6} + d(v_4, v_5) - \xi = 8 - \xi.$

$T_6 = \xi + d(v_6, v_6) = \xi;$

$T_6' = c_{4,6} + d(v_4, v_6) - \xi = 8 - \xi.$



Des. 6.5.g

Pe muchia 8

$T_1 = \xi + d(v_6, v_1) = \xi + 5;$

$T_1' = c_{5,6} + d(v_5, v_1) - \xi = 5 - \xi.$

$T_2 = \xi + d(v_6, v_2) = \xi + 8;$

$T_2' = c_{5,6} + d(v_5, v_2) - \xi = 8 - \xi.$

$T_3 = \xi + d(v_6, v_3) = \xi + 10;$

$T_3' = c_{5,6} + d(v_5, v_3) - \xi = 10 - \xi.$

$T_4 = \xi + d(v_6, v_4) = \xi + 4;$

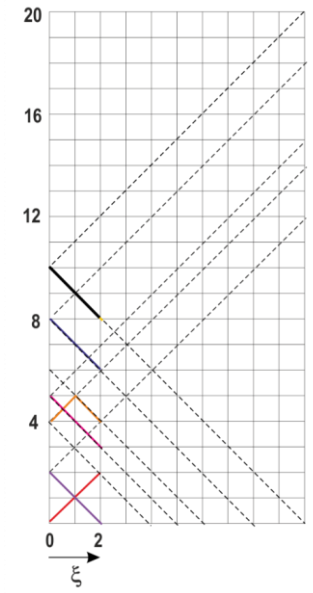
$T_4' = c_{5,6} + d(v_5, v_4) - \xi = 6 - \xi.$

$T_5 = \xi + d(v_6, v_5) = \xi + 2;$

$T_5' = c_{5,6} + d(v_5, v_5) - \xi = 2 - \xi.$

$T_6 = \xi + d(v_6, v_6) = \xi;$

$T_6' = c_{5,6} + d(v_5, v_6) - \xi = 4 - \xi.$



Des. 6.5.h

Pe muchia 9

$T_1 = \xi + d(v_4, v_1) = \xi + 4;$

$T_1' = c_{4,3} + d(v_3, v_1) - \xi = 18 - \xi.$

$T_2 = \xi + d(v_4, v_2) = \xi + 10;$

$T_2' = c_{4,3} + d(v_3, v_2) - \xi = 12 - \xi.$

$T_3 = \xi + d(v_4, v_3) = \xi + 10;$

$T_3' = c_{4,3} + d(v_3, v_3) - \xi = 10 - \xi.$

$T_4 = \xi + d(v_4, v_4) = \xi;$

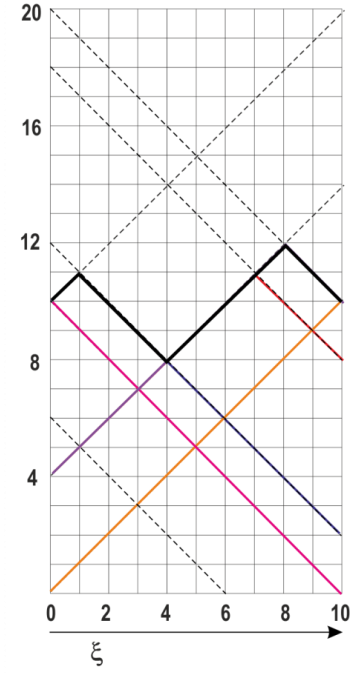
$T_4' = c_{4,3} + d(v_3, v_4) - \xi = 20 - \xi.$

$T_5 = \xi + d(v_4, v_5) = \xi + 4;$

$T_5' = c_{4,3} + d(v_3, v_5) - \xi = 18 - \xi.$

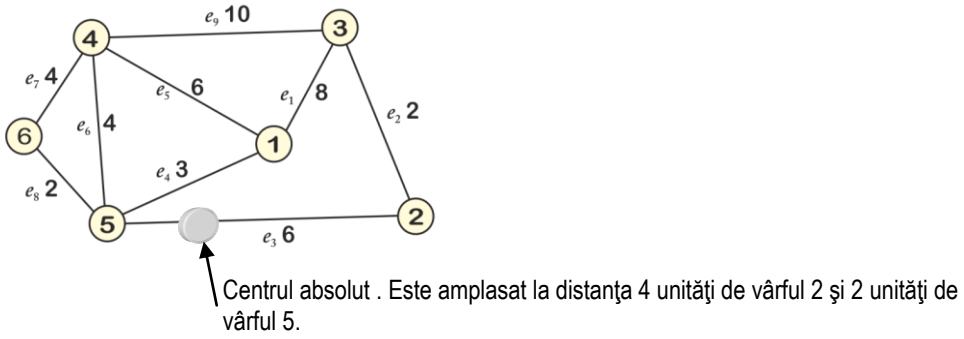
$T_6 = \xi + d(v_4, v_6) = \xi + 4;$

$T_6' = c_{4,3} + d(v_3, v_6) - \xi = 20 - \xi.$



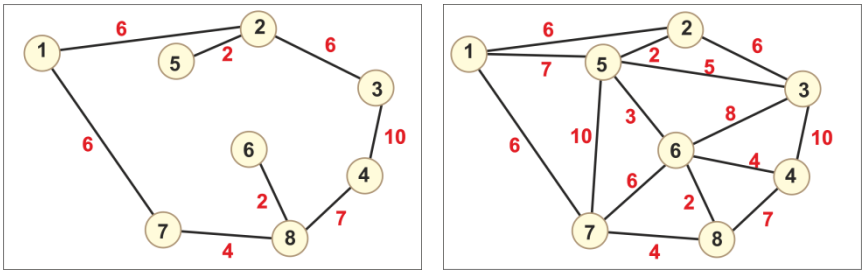
Des. 6.5.i

Indicele de separare cu valoare minimă se obține pe muchia e_3. Valoarea minimă este 6 (desenul 6.5.c) şi se obține în punctul aflat la distanța de 4 unități de la vârful v_2 (desenul 6.6).



Des. 6.6 Amplasarea centrului absolut

Exerciţii:



Des. 6.7. A B

1. Pentru grafurile de pe desenul 6.7 A, 6.7 B, determinați vârfurile centru.

2. Elaborați un program pentru determinarea centrelor relative exterioare şi interioare ale unui graf orientat, descris prin matricea sa de adiacență. $|V| < 50$.

3. Elaborați un program pentru determinarea centrelor relative ale unui graf neorientat. $|V| < 50$.

4. Determinați 2-centrul pentru grafurile din imaginile 6.7 A, 6.7 B.

5. Determinați, folosind metoda Hakimi, centrele absolute ale grafurilor din imaginile 6.7 A, 6.7 B.

6. Elaborați un program, care va determina cu o eroare ce nu depăşeşte 1% centrul absolut al grafurilor cu $|V| < 30$, muchiile cărora au ponderi întregi, mai mici decât 100. (Folosiți metoda trierii)

Capitolul 7. Mediane

În acest capitol:

- Mediane în graf
- Mediane interioare şi exterioare
- Algoritmi exacţi pentru determinarea medianei
- Mediana absolută
- P – mediana
- Algoritmi euristici pentru determinarea p-medianei

7.1 Mediane

Mai multe probleme de amplasare a punctelor de deservire presupun minimizarea sumei distanțelor de la o serie de puncte terminale până la un punct central (de colectare, comutare, depozite, etc.)

Punctele care corespund soluției optime ale problemei se numesc puncte *mediane* ale grafului.

Fie graful $G=(V,E)$. Pentru fiecare vârf v_i se definesc două valori, care se numesc *indici de transmitere*:

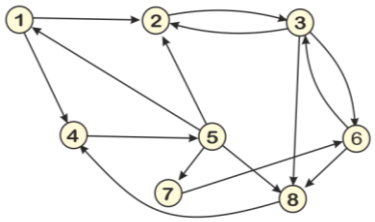
$$\sigma_0(v_i)=\sum_{v_j\in V}\left[d\left(v_i,v_j\right)\right]$$

$$\sigma_t(v_i)=\sum_{v_j\in V}\left[d\left(v_j,v_i\right)\right]$$

Aici $d(v_i,v_j)$ – distanța minimă dintre vârfurile v_j şi v_i. Valorile $\sigma_0\left(v_i\right)$ şi $\sigma_t\left(v_i\right)$ se numesc *indice interior şi exterior de transmitere* a vârfului v_i. Indicii de transmitere pot fi calculați din matricea distanțelor $D(G)$: $\sigma_0\left(v_i\right)$ ca suma elementelor din linia i a matricei D, iar $\sigma_t\left(v_i\right)$ ca suma elementelor din coloana i.

Def. Vârful $\overline{v_0}$ pentru care $\sigma_0\left(\overline{v_0}\right)=\min_{v_i\in V}\left[\sigma_0\left(v_i\right)\right]$ se numeşte *mediană exterioară* a grafului G. Vârful $\overline{v_t}$ pentru care $\sigma_t\left(\overline{v_t}\right)=\min_{v_i\in V}\left[\sigma_t\left(v_i\right)\right]$ se numeşte *mediană interioară*.

Exemplu: Fie graful din desenul 7.1:



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	σ_0
v_1	0	1	2	1	2	3	3	3	15
v_2	5	0	1	3	4	2	5	2	22
v_3	4	1	0	2	3	1	4	1	16
v_4	2	2	3	0	1	3	2	2	15
v_5	1	1	2	2	0	2	1	1	10
v_6	4	4	1	2	3	0	3	1	18
v_7	6	3	2	3	4	1	0	2	21
v_8	3	3	4	1	2	4	3	0	20
σ_t	25	15	15	14	19	16	21	12	

Pentru exemplul prezentat mediana exterioară este vârful v_5 cu indicele de transmitere 10, mediana interioară – vârful v_8 cu indicele de transmitere 12.

Algoritmul general pentru determinarea medianelor presupune calculul distanțelor minime între vârfurile grafului, ulterior calcularea sumelor elementelor matricei pe linii (coloane) şi selectarea minimului din sumele calculate (minimul sumelor pe linii pentru mediana exterioară, pe coloane – pentru mediana interioară).

7.2. Mediana absolută

La determinarea centrelor absolute în graf s-a observat că amplasarea optimă a acestora poate genera apariția unui vârf nou pe una din muchiile grafului. În cazul medianei absolute situația este diferită:

Teorema 1. Pentru orice punct *y* al grafului $G=(V,E)$, va exista cel puțin un vârf *v* al grafului, pentru care $\sigma(v)\le\sigma(y)$

□

Dacă *y* este vârf al grafului, teorema e demonstrată. În caz contrar: Fie *y*– un punct interior al muchiei (v_α,v_β) situat la distanța ξ de v_α. În acest caz $d(y,v_j)=\min\left[\xi+d(v_\alpha,v_j),c_{\alpha,\beta}-\xi+d(v_\beta,v_j)\right]$

Fie V_α mulțimea vârfurilor pentru care $\xi+d(v_\alpha,v_j)\le c_{\alpha,\beta}-\xi+d(v_\beta,v_j)$ iar V_β mulțimea vârfurilor pentru care $\xi+d(v_\alpha,v_j)>c_{\alpha,\beta}-\xi+d(v_\beta,v_j)$.

Atunci: $\sigma(y)=\sum_{v_j\in V}d(y,v_j)=\sum_{v_j\in V_\alpha}\left[\xi+d(v_\alpha,v_j)\right]+\sum_{v_j\in V_\beta}\left[c_{\alpha,\beta}-\xi+d(v_\beta,v_j)\right]$

Tot odată, $d(v_\alpha,v_j)\le c_{\alpha,\beta}+d(v_\beta,v_j)$ (altfel nu ar fi distanța minimă)

Prin înlocuire în formula precedentă se obține:

$\sigma(y)\ge\sum_{v_j\in V_\alpha}\left[\xi+d(v_\alpha,v_j)\right]+\sum_{v_j\in V_\beta}\left[d(v_\beta,v_j)-\xi\right]$.

Deoarece $V_\alpha\cup V_\beta=V$, poate fi efectuată regruparea termenilor:

$\sigma(y)\ge\sum_{v_j\in V}\left[d(v_\alpha,v_j)\right]+\xi\left(|V_\alpha|-|V_\beta|\right)$. Deoarece vârfurile v_α,v_β se aleg arbitrar, v_α va fi vârful, pentru care $|V_\alpha|\ge|V_\beta|$. În final $\sigma(y)\ge\sigma(v_\alpha)$ ■

Pentru un graf neorientat mediana exterioară şi cea interioară coincid, prin urmare se cercetează doar noțiunea de mediană, ca vârf, care minimizează suma distanțelor până la celelalte vârfuri ale grafului.

7.3 P-mediane

Noțiunea de *p*-mediană se introduce prin analogie cu noțiunea de *p*-centru.

Fie V_p o submulțime din V. $|V_p| = p$. Prin $d(V_p, v_i)$ se va nota distanța de la cel mai apropiat vârf din V_p până la vârful v_i:

$$d(V_p, v_i) = \min_{v_j \in V_p} \left[d(v_j, v_i) \right] \qquad (*)$$

La fel, prin $d(v_i, V_p)$ se va nota distanța de la vârful v_i până la cel mai apropiat vârf din V_p: $d(v_i, V_p) = \min_{v_j \in V_p} \left[d(v_i, v_j) \right]$ (**)

Dacă v'_j este vârful din V_p, pentru care se obține valoarea minimă a (*) sau (**), se spune că v_i este arondat la v'_j.

Indicii de transmitere pentru mulțimea V_p se calculează la fel ca şi pentru vârfurile solitare:

$$\sigma_0(V_p) = \sum_{v_j \in V} \left[d(V_p, v_j) \right], \qquad \sigma_t(V_p) = \sum_{v_j \in V} \left[d(v_j, V_p) \right]$$

Def. Mulțimea de vârfuri $\overline{V}_{p,0}$ pentru care are loc relația $\sigma_0(\overline{V}_{p,0}) = \min_{V_p \subseteq V} \left[\sigma_0(V_p) \right]$ se numeşte *p-mediană exterioară* a grafului *G*. Mulțimea de vârfuri $\overline{V}_{p,t}$ pentru care are loc relația $\sigma_t(\overline{V}_{p,t}) = \min_{V_p \subseteq V} \left[\sigma_t(V_p) \right]$ se numeşte *p-mediană interioară* a grafului *G*.

Teorema 2. Pentru orice mulțime Y din p puncte ale grafului $G = (V, E)$, va exista cel puțin o mulțime $V_p \subset V$ din p vârfuri ale grafului, pentru care $\sigma(V_p) \le \sigma(Y)$

□ Se demonstrează similar Teoremei 1 din acest capitol.■

Pentru un graf neorientat *p*-mediana exterioară şi cea interioară coincid, prin urmare se cercetează doar noțiunea de *p*-medianã, ca mulțime de vârfuri, care minimizează suma distanțelor până la toate celelalte vârfuri ale grafului.

7.3 Algoritmi pentru determinarea *p*-medianei

Algoritmul direct

Algoritmul direct presupune generarea tuturor submulțimilor din p vârfuri ale grafului G şi selectarea mulțimii $\overline{V}_p$ pentru care $\sigma(\overline{V}_p) = \min_{V_p \subseteq V} \left[\sigma(V_p) \right]$. O asemenea abordare presupune cercetarea a C_n^p mulțimi, ceea ce necesită resurse exponențiale de timp pentru valori mari ale n sau valori ale p apropiate de $n/2$. În particular, pentru valori ale lui n,p, care nu depăşesc 20 poate fi folosit un algoritm clasic de generare a submulțimilor unei mulțimi, descris în [9, p.32]. O modificare elementară va permite generarea doar a mulțimilor din p elemente, ceea ca va reduce considerabil timpul de lucru al algoritmului. O altă componentă a soluției este calcularea matricei D a distanțelor minime, care va servi în calitate de sursă de date pentru determinarea p-medianei.

Algoritmul euristic

Pentru valori mari ale lui n şi p poate fi folosit un algoritm euristic, similar algoritmului pentru determinarea p-centrului.

Pas 0. Se formează matricea distanțelor minime (algoritmul Floyd)

Pas 1. Se alege în calitate de aproximare inițială a p-medianei o mulțime arbitrară de vârfuri C. Vârfurile $v_j \in V - C$ vor fi considerate având marcajul stării egal cu 0 (neverificate).

Pas 2. Se alege un vârf arbitrar de stare 0 $v_j \in V - C$ şi pentru fiecare vârf $v_i \in C$ se calculează valorile $\Delta_{i,j} = \sigma(C) - \sigma(C - \{v_i\} \cup \{v_j\})$

Pas 3. Se determină $\Delta_{i_0,j} = \max_{v_i \in C}\left[\Delta_{i,j}\right]$.

iii. Dacă $\Delta_{i_0,j} < 0$ vârful v_j este marcat „verificat" (i se aplică marcajul de stare - 1) şi se revine la pasul 2.

iv. Dacă $\Delta_{i_0,j} > 0$, $C \leftarrow \left(C - \{v_i\} \cup \{v_j\}\right)$ vârful v_j este marcat „verificat" (i se aplică marcajul de stare - 1) şi se revine la pasul 2.

Pas 4. Paşii 2 – 3 se repetă atât timp cât există vârfuri cu marcaj de stare 0. Dacă la ultima repetare a paşilor 2 – 3 nu au fost efectuate înlocuiri (3.ii) se trece la pasul 5. În caz contrar tuturor vârfurilor din $V - C$ li se atribuie marcajul de stare 0, apoi se revine la pasul 2.

Pas 5. STOP. Mulțimea de vârfuri curentă C este o aproximare a *p*-medianei căutate.

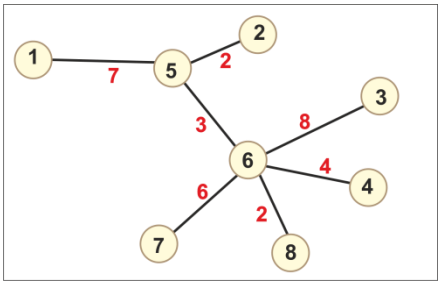
Implementare

Input: Graful G: matricea de adiacență **d**, ulterior matricea distanțelor; vectorul p-medianei **pmed**, vectorul stare **st**.
Output: O p-mediană posibilă a G, în vectorul **pmed.**

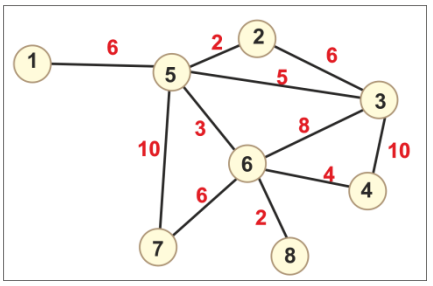
```
int tipar()
{…// afişează mediana curentă }

int calcmed()
{ … // calculează valoarea p-medianei curente }

int main()
{ …// citire date
  …// modelare valoare „infinit" - p
  …// algoritmul Floyd. D – matricea distanţelor
// initializare p-mediana
   for(i=1;i<=n; i++) st[i]=0;
   for(i=1;i<=pmed; i++) {st[i]=2; med[i]=i;}
do {
   for (i=1;i<=n; i++) if (st[i]==1) st[i]=0;
```

```
  // reinitializare stare varfuri
  for (i=1;i<=n;i++)
      if (st[i]==0)
       {delta=0; sumtot=calcmed();
         for (j=1;j<=pmed; j++)
          { tmp=med[j]; med[j]=i;
            st[i]=2; st[tmp]=0; // i se introduce in poz j
            medcurent=calcmed();
            if (delta<sumtot - medcurent)
               {delta=sumtot-medcurent; sw1=j;sw2=i;}
            // micsorare indice de transmtere
            med[j]=tmp; st[tmp]=2; // restabilire j
          }
           if (delta>0 ) // a fost detectata o îmbunătăţire
           { tmp=med[sw1]; med[sw1]=sw2;
             st[tmp]=1; st[sw2]=2;}       // inlocuire
           else st[i]=1;
        }
} while (delta>0);
tipar(); printf(" %d ", calcmed());
return 0; }
```

Exerciţii:



A



B

Des 7.2

1. Determinați medianele pentru grafurile din imaginile 7.2 A, 7.2 B.
2. Elaborați un program pentru determinarea medianelor interioare şi exterioare ale unui graf $G=(V,E),\ |V|\le 200$
3. Determinați 2-medianele pentru grafurile din imaginile 7.2 A, 7.2 B.
4. Elaborați un program pentru determinarea exactă a 2-medianelor interioare şi exterioare ale unui graf $G=(V,E),\ |V|\le 20$
5. Elaborați un program pentru determinarea exactă a p-medianelor ale unui graf $G=(V,E),\ |V|\le 20$
6. Elaborați un program pentru determinarea aproximativă, folosind algoritmul euristic $(\lambda=1)$ a p-medianelor ale unui graf $G=(V,E),\ |V|\le 100$
7. Elaborați un program pentru determinarea aproximativă, folosind algoritmul euristic $(\lambda=2)$ a p-medianelor ale unui graf $G=(V,E),\ |V|\le 100$
8. Demonstrați Teorema 2.

Capitolul 8. Arbori

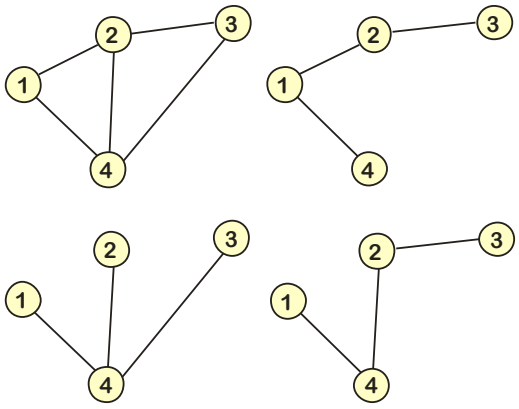
În acest capitol:

- Definiţii ale arborilor
- Generarea tuturor arborilor parţiali ai unui graf
- Arbori parţiali de cost minim
- Algoritmul Kruskal pentru determinarea arborelui de cost minim.
- Algoritmul Prim pentru determinarea arborelui de cost minim

8.1 Arbori de acoperire

Carcase

Un graf neorientat este numit *arbore* dacă este conex şi nu conține cicluri. Pentru graful $G=(V,E)$ fiecare arbore $G^*=(V,T)$ $T \subseteq E$ este numit *carcasă* sau *arbore parțial de acoperire* a grafului G. Muchiile din G, care apar în G^* sunt numite *ramuri*, cele din E/T - *corzi*. Numărul carcaselor unui graf este determinat de structura acestuia.



Des. 8.1. Graful (stânga sus) şi câteva dintre carcasele sale

Algoritmii de parcurgere a unui graf conex permit şi construirea concomitentă a unei carcase. Într-adevăr, proprietatea de conexitate asigură atingerea fiecărui nod u a grafului pe parcursul lucrului algoritmului. Pentru nodul u, atins din v, algoritmii de parcurgere nu

mai permit atingerea repetată din alt nod. Prin urmare, apariția ciclurilor în procesul parcurgerii este imposibilă.

Următoarea modificare a funcției DFS afişează lista muchiilor pentru carcasa ce corespunde parcurgerii în adâncime a grafului:

```
int DFS (int s)
{ int i;
  b[s]=1;
  for(i=1;i<=n;i++)
       if(a[s][i] !=0 && b[i]==0)
          {printf("\n %d - %d", s, i); DFS(i);}
 return 0;
}
```

Generarea tuturor carcaselor unui graf

Mai multe probleme impun necesitatea de a genera nu o carcasă a grafului dat, ci toate carcasele posibile. De exemplu, trebuie să fie selectată „cea mai bună carcasă" după un criteriu complex, care nu permite aplicarea unor optimizări directe. În alte situații, generarea tuturor subarborilor în calitate de subproblemă permite reducerea complexității la rezolvarea unor probleme de calcul economic.

Numărul posibil de carcase ale unui graf variază în dependență de numărul de muchii în graful inițial dar şi de structura conexiunilor. Pentru un graf complet cu n vârfuri, de exemplu, numărul calculat al carcaselor este egal cu n^{n-2}. Prin urmare, problema generării tuturor carcaselor este una de complexitate exponențială (în caz general) . Respectiv, algoritmul pentru generarea acestora va avea la origine tehnica reluării. O posibilă structură a algoritmului a fost dată de Kristofides. Totuşi, algoritmul propus în [6, p.154] este unul iterativ, ceea ce face mai complicată implementarea lui. În cele ce urmează se propune un algoritm recursiv, care formează carcasele în baza listei de muchii a grafului *G*.

Preprocesare: fie $G=(V,E)$. Se formează lista de muchii $e_1, e_2, ..., e_m$ grafului *G*.

Funcția recursivă **CARCASE(k,c)**

Pas A.

(i) $i \leftarrow k$.

(ii) Muchia curentă $e_i = (v', v'')$. Pe graful T se lansează funcția de căutare în adâncime modificată **DFS**(v', v''), pentru a determina existența unei căi între v' şi v''.

(iii) Dacă funcția returnează 1 (în T există o cale între v' şi v'') STOP

În caz contrar funcția returnează 0 (nu există o cale între v' şi v'')

$T = T \bigcup e_i$, $c \leftarrow c+1$

Dacă $c = n-1$ se afişează carcasa construită *T*. apoi se trece la pas 4.

în caz contrar pentru toți i de la $k+1$ la m **CARCASE(i,c)**

Pas B.

Excludere muchie $T = T - e_i$ $c \leftarrow c-1$

Sfârşit funcție **carcase**.

În funcție se generează toate carcasele, care încep de la muchia e_k

Algoritm

Pas 1. $k \leftarrow 0$

Pas 2 Indicele muchiei de la care continuă construcția carcasei $k \leftarrow k+1$.

Se formează un graf *T* din vârfurile izolate ale lui *G:* $T = (V, \varnothing)$

Numărul de muchii în carcasă $c \leftarrow 0$

Pas 3. **CARCASE(k,0)**

Pas 4 dacă $k < m-n+1$ se revine la pasul 2, altfel STOP – toate carcasele au fost generate

Implementare

Input: Graful G : matricea de adiacență **a**; lista de muchii **v**;
Output: Arborii parțiali a grafului G , generați consecutiv în tabloul **arb.**

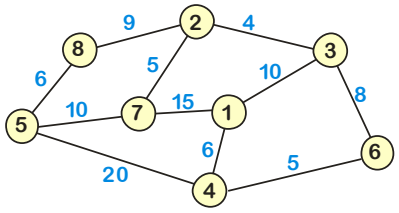
```
int tipar()
{ .. //afişează soluţia curentă – matricea de adiacenţă a
arborelui parţial}

int scoatemuchie(int ind)
{ …// exclude muchia cu indicele ind din arborele în
construcţie}

int punemuchie(int ind)
{ …// include muchia cu indicele ind în arborele în
construcţie}

int DFS (int s,int tt)
{ …// verifică prezenţa unui drum între vârfurile s şi tt}

int ciclu(int im)
{ …// verifică apariţia unui ciclu la adăugarea muchiei cu
indicele im }

int back(int indicemuchie, int numarmuchii)
{ int i;
      if (ciclu(indicemuchie)) return 0;
      else
      { punemuchie(indicemuchie);// se adaugă muchia
        numarmuchii++;
        if (numarmuchii==n-1) // e detectată o soluţie
         {tipar(); scoatemuchie(indicemuchie); numarmuchii--;
          return 0;}
        else
        {for (i=indicemuchie+1;i<=k;i++) back(i,numarmuchii);
          // se adaugă următoarele muchii
          scoatemuchie(indicemuchie);numarmuchii--; return 0;
          // se exclude muchia la pasul de întoarcere
        }
      }
}
int main()
```

```
{ …// citire date
   // formarea lista muchii
   for (i=1;i<=n;i++)
        for (j=i+1; j<=n; j++)
              if (a[i][j]==1) {k++; ed[k].vi=i; ed[k].vj=j;}
   // căutare carcase
   for (i=1;i<=k-n+1;i++)
        back(i,0);
   return 0;  }
```

8.2 Arbori de acoperire de cost minim

Mai multe dintre problemele formulate pe grafuri presupun existența unei caracteristici suplimentare, atribuite muchiilor grafului – *ponderea*. Pentru muchia (v,u) ponderea se notează $c_{v,u}$. De obicei ponderea are o expresie numerică, care indică distanța dintre noduri, capacitatea sau timpul de transfer de-a lungul muchiei etc.

Pentru stocarea ponderilor muchiilor nu sunt necesare modificări esențiale în structurile de date utilizate pentru descrierea grafurilor: în matricele de incidență sau adiacență valorile unitare ce corespund muchiilor sunt înlocuite prin valorile ponderilor (desenul 8.2); în listele de muchii şi de incidență fiecare pereche (element) este suplinită de o componentă ce conține valoarea ponderii.



$$\begin{bmatrix} 0 & 0 & 10 & 6 & 0 & 0 & 15 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 5 & 9 \\ 10 & 4 & 0 & 0 & 0 & 8 & 0 & 0 \\ 6 & 0 & 0 & 0 & 20 & 5 & 0 & 0 \\ 0 & 0 & 0 & 20 & 0 & 0 & 10 & 6 \\ 0 & 0 & 8 & 5 & 0 & 0 & 0 & 0 \\ 15 & 5 & 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 6 & 0 & 0 & 0 \end{bmatrix}$$

Des. 8.2. Graf ponderat şi matricea lui de adiacenţă

Prin *cost* al arborelui $G^* = (V,T)$ se va înțelege suma ponderilor muchiilor, care îl formează.

$$S(G^*) = \sum_{(v,u)\in T} c_{v,u}$$

Pentru grafurile neponderate toți arborii de acoperire au acelaşi cost. În cazul prezenței ponderilor în graf apare

şi problema selectării unui arbore de acoperire de cost minim. Există mai mulți algoritmi eficienți pentru determi-narea arborelui de acoperire de cost minim, cei mai cunoscuți fiind:

Algoritmul Kruskal

Fie $G=(V,E)$ un graf ponderat cu *n* vârfuri. Algoritmul Kruskal foloseşte tehnica greedy şi presupune crearea unui graf $G^*=(V,T)$, în care inițial $T=\varnothing$. Ulterior, muchiile din *G* se sortează după creşterea ponderilor. La următoarea etapă se efectuează adăugarea consecutivă în G^* a muchiilor din şirul sortat, cu condiția că la adăugarea muchiei în G^* nu se formează un ciclu. Lucrul algoritmului se sfârşeşte când numărul de muchii adăugate devine *n*-1.

Pseudocod:

Pas 1. Se construieşte $G^*=(V,T)$, $T=\varnothing$.

Pas 2. Muchiile din $G=(V,E)$ se sortează în ordinea creşterii ponderilor. Se obține şirul $m_1,...,m_{|E|}$

Pas 3. $k\leftarrow 0,\quad i\leftarrow 1$

Pas 4. While $k\neq|V|-1$ do

a) if $T\cup m_i$ *nu formează un ciclu în* G^*, then

$T\leftarrow T\cup m_i, k\uparrow$

b) $i\uparrow$

Implementare:

Input: Graful G: matricea de adiacență **a**; liata de muchii **v**; arborele parțial de cost minim **arb**, tabloul componentelor conexe **c**.
Output: Un arbore parțial de cost minim a G, în tabloul **arb.**

```
int sort ()
{ ...// functia de sortare a listei de muchii }
```

```
int readdata()
{ …// functa de citire a datelor initiale }

int printv()
{ … // functia de tipar a rezultatelor}

int makeedgelist()
{ …// functia pentru crearea listei de muchii }

int main(int argc, char *argv[])
{ int j,i,r;
  readdata();
  makeedgelist();
  sort();
for (i=1;i<=n;i++) c[i]=i; // initializare componente conexe
i=0;
while(k<n-1)
{ i++;
if (c[v[i].vi] != c[v[i].vj])
// verificare posibilitate adaugare muchie
  {k++; arb[k]=v[i];
   for(j=1;j<=n;j++)
        if (c[j] == c[v[i].vj] ) c[j]=c[v[i].vi];
  }
 }
printv(); // afisare rezultate
return 0;
}
```

Algoritmul Prim

Spre deosebire de algoritmul Kruskal algoritmul Prim generează arborele parțial de cost minim prin extinderea unui singur subgraf T_s, care inițial este format dintr-un vârf. Subarborele T_s se extinde prin adăugarea consecutivă a muchiilor (v_i, v_j), astfel încât $v_i \in T_s, \quad v_j \notin T_s)$, iar ponderea muchiei adăugate să fie minim posibilă. Procesul continuă până la obținerea unui număr de $n-1$ muchii în T_s.

Pseudocod

Pas 1. Se construieşte $G^* = (V, T)$, $T = \varnothing$.

Pas 2. Muchiile din $G = (V, E)$ se sortează în ordinea creşterii ponderilor. Se obține şirul $m_1, ..., m_{|E|}$. Toate muchiile se consideră *nefolosite*.

Pas 3. $k \leftarrow 0$

Pas 4. Cât timp $k < |V| - 1$:

a) $i \leftarrow 1$

b) Dacă muchia $m_i = (v', v'')$ este *nefolosită* şi $v' \in T_s \ \& \ v'' \notin T_s)$ sau $v' \notin T_s \ \& \ v'' \in T_s)$, atunci m_i se adaugă la T_s, m_i se consideră *folosită* şi $k \uparrow$, după care se revine la începutul pasului 4. În caz contrar $i \uparrow$ şi se repetă pas 4.b.

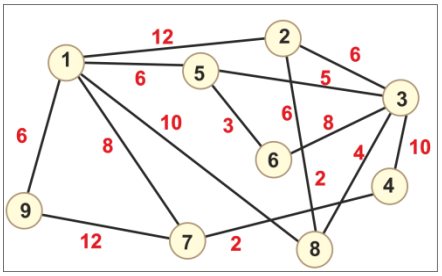
Implementare

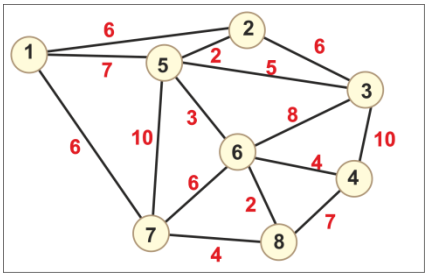
Input: Graful G : matricea de adiacență **a**; lista de muchii **v**; arborele parțial de cost minim **arb**, tabloul **c** de stare a nodurilor (se foloseşte pentru a modela starea muchiilor).
Output: Un arbore parțial de cost minim a G , în tabloul **arb.**

```
int sort ()
{ …// functia de sortare a listei de muchii }

int readdata()
{ …// functa de citire a datelor initiale }

int printv()
{ … // functia de tipar a reyultatelor}

int makeedgelist()
{ …// functia pentru crearea listei de muchii }

int main(int argc, char *argv[])
```

```
{ int j,i;
  readdata();
  makeedgelist();
  sort();
  for (i=1;i<=n;i++) c[i]=0;
  c[v[1].vi]=c[v[1].vj]=1; k=1;
  arb[1]=v[1];
while(k<n-1) // cat arboreal nu e construit
{ i=1;
 while (c[v[i].vi]+ c[v[i].vj] != 1 ) i++;
 // se gaseste cea mai scurta muchie care poate fi adaugata
 k++; arb[k]=v[i]; // se adauga muchia
 c[v[i].vj]=c[v[i].vi]=1; // se modifica starea nodurilor
muchiei adaugate
}
printv();
return 0;
}
```

Exerciţii





Des. 8.3

1. Determinați arborii parțiali de cost minim pentru grafurile din imaginile 8.3 A, 8.3 B.
2. Elaborați un program pentru determinarea tuturor carcaselor unui graf $G=(V,E),\quad |V|\le 20$
3. Elaborați un program pentru determinarea arborelui parțial de cost minim a unui graf $G=(V,E),\quad |V|\le 20$. (folosiți algoritmul Kruskal).
4. Elaborați un program pentru determinarea arborelui parțial de cost minim a unui graf $G=(V,E),\quad |V|\le 20$. (folosiți algoritmul Prim).
5. Estimați complexitatea temporală a algoritmului Kruskal.
6. Estimați complexitatea temporală a algoritmului Prim.
7. Estimați complexitatea temporală a algoritmului de generare a tuturor carcaselor unui graf $G=(V,E)$.

Capitolul 9. Cicluri

În acest capitol:

- Numărul ciclomatic al grafului
- Mulţimea fundamentală de cicluri
- Determinarea mulţimii fundamentale de cicluri
- Tăieturi în graf
- Problema Euler
- Ciclul eulerian
- Teorema de existenţă a ciclului (lanţului) eulerian
- Algoritmi pentru construirea ciclului (lanţului) eulerian

9.1 Numărul ciclomatic şi mulţimea fundamentală de cicluri

Fie dat graful $G=(V,E)$ cu n vârfuri, m muchii şi p componente conexe.

Valoarea $\rho(G)=n-p$ stabileşte numărul total de muchii, în fiecare din arborii parțiali ai grafului G pe toate componentele de conexitate ale acestuia. În particular, dacă graful este conex, numărul de muchii în oricare arbore parțial va fi egal cu $n-1$.

Valoarea $v(G)=m-n+p=m-\rho(G)$ se numeşte *numărul ciclomatic* al grafului G. Valoarea $\rho(G)$ se numeşte număr *cociclomatic*. Caracteristica ciclomatică a grafului stabileşte numărul maximal de cicluri independente[8], care pot fi construite concomitent pe graf.

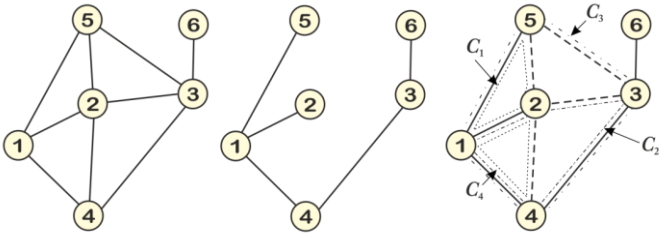
Astfel, dacă se construieşte un arbore parțial T al grafului, apoi se formează cicluri prin adăugarea a câte o muchie a grafului, care nu aparține arborelui T, în final se va obține o mulțime de cicluri $C_1, C_2, ..., C_{v(G)}$, independente între ele. De remarcat că ciclul C_i se

[8] Cicluri independente – dacă conțin cel puțin câte o muchie, care aparține doar unuia din ele

formează prin adăugarea unei muchii (v_j, v_k) la lanțul care uneşte vârfurile v_j, v_k în arborele T.

Fie dat un arbore parțial T. Mulțimea de cicluri ale grafului G, fiecare dintre care este format prin adăugarea la T a unei muchii din G/T formează mulțimea ciclurilor fundamentale, asociate arborelui T. Oricare două dintre ciclurile fundamentale sunt independente între ele şi orice alt ciclu, care nu face parte din mulțimea ciclurilor fundamentale poate fi reprezentat ca o combinație liniară a acestora.

Exemplu. Fie dat graful $G = (V, E)$ şi unul din arborii lui parțiali – T. (desenul 9.1).

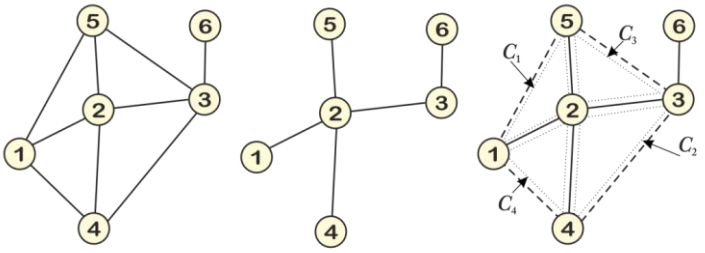


a **b** **c**

Desenul 9.1 Graful *G* (a), un arbore parţial *T* (b), ciclurile fundamentale (c)

Pentru graful G: $n = 6, m = 9, p = 1$. Numărul ciclomatic $\nu(G) = 9 - 6 + 1 = 4$. Ciclurile fundamentale sunt: C_1, C_2, C_3, C_4

De menționat că ciclurile fundamentale se modifică în dependență de arborele parțial construit (desenul 9.2)



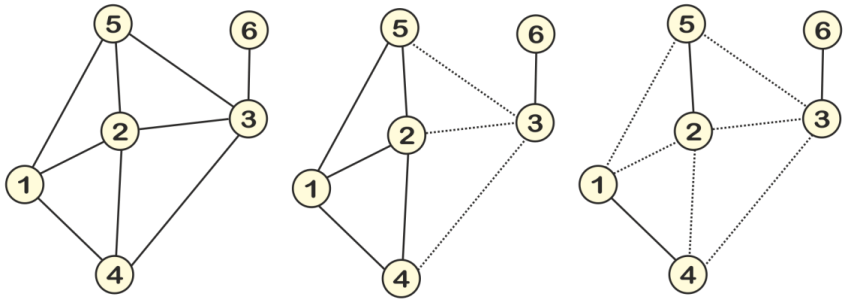
Desenul 9.2 modificarea mulţimii de cicluri fundamentale a grafului din exemplul precedent la selecţia unui alt arbore parţial.

9.2 Tăieturi

Def. Dacă vârfurile unui graf neorientat $G=(V,E)$ sunt separate în două mulțimi V_0 şi V_0 ($V_0 \subset V$, $V_0 = V/V_0$), atunci mulțimea de muchii, cu proprietatea că o extremitate aparține V_0 şi cealaltă aparține V_0 se numeşte *tăietură* $C(V_0)$ a grafului G $C(V_0)=\{(v',v''):"v' \in V_0, v'' \in V_0\}$

Arborele parțial $G_p=(V,E/C(V_0))$ va fi format din cel puțin două componente conexe.

Exemplu: dacă din graful cercetat în exemplul precedent vor fi excluse muchiile (2,3) (3,4) (3,5) se vor obține două componente conexe, generate de mulțimile de vârfuri {1,2,4,5} şi {3,6}. Excluderea muchiilor (1,2) (1,5) (2,4) (3,4) (2,3) (3,5) va genera trei componente conexe {1,4} {2,5} {3,6} (figura 9.3).



Des. 9.3 . Tăieturi în graf.

Fie S o mulțime de muchii din G, lichidarea cărora generează un subgraf bazic $G_p=(V,E-S)$ neconvex.

Def. Mulțimea S se numeşte o *tăietură regulată* a grafului G dacă nu există nici o submulțime $S' \subseteq S$ astfel încât $G'_p = (V, E - S')$ să fie de asemenea neconvex.

Este evident că o tăietură regulată divizează graful în exact două componente convexe, una dintre care are în calitate de vârfuri elementele mulțimii V_0 şi cealaltă – elementele mulțimii V_0.

În exemplul precedent, tăietura determinată de muchiile (2,3) (3,4) (3,5) în figura 9.3 (b) este o tăietură regulată, iar cea determinată de muchiile (1,2) (1,5) (2,4) (3,4) (2,3) (3,5) – nu.

Pe un graf orientat tăietura este definită în mod similar:

Def. Dacă vârfurile unui graf orientat $G = (V, E)$ sunt separate în două mulțimi V_0 şi V_0 ($V_0 \subset V$, $V_0 = V / V_0$), atunci mulțimea de arce (u, v) , cu proprietatea că $u \in V_0$ şi $v \in V_0$ sau $v \in V_0$ şi $u \in V_0$ se numeşte *tăietură* $C(V_0)$ a grafului G

$$C(V_0) = \{(u,v) : u \in V_0, v \in V_0\} \cup \{(u,v) : u \in V_0, v \in V_0\}$$

La fel ca şi pentru cicluri, pentru tăieturi se defineşte mulțimea fundamentală asociată unui arbore parțial T.

Tăieturi fundamentale, asociate arborelui parțial T se va numi o mulțime din $n-1$ tăieturi, fiecare conținând exact o muchie distinctă a arborelui T.

Următoarea teoremă stabileşte legătura între mulțimea fundamentală de cicluri şi mulțimea fundamentală de tăieturi asociate unui arbore T.

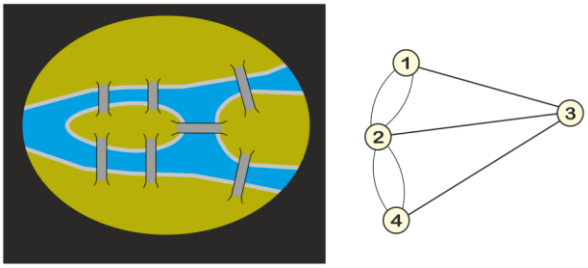
Teoremă. Dacă T este un arbore parțial al grafului $_{\text{c}}$ atunci tăietura fundamentală determinată de muchia e_i din T este formată din muchia e_i şi din acele muchii ale grafului G, care nu aparțin T, dar prin adăugare la T generează cicluri fundamentale, care conțin e_i .

□■

9.3 Cicluri Euler

Problema podurilor din Königsberg

Oraşul german Konigsberg (acum Kaliningrad) este traversat de râul Pregel. Zonele oraşului, situate pe ambele maluri ale râului dar şi pe două insule erau unite prin şapte poduri, după schema din desenul 9.4. În anul 1736 locuitorii oraşului i-au trimis celebrului matematician Euler o scrisoare, rugându-l să rezolve următoarea problemă: poate fi găsit un traseu, care pornind dintr-un punct dat, să parcurgă toate cele şapte poduri câte o singură dată şi să revină în punctul de pornire?

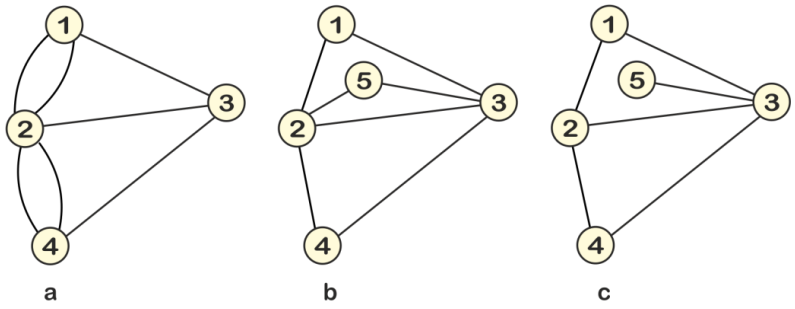


Des. 9.4 Schema amplasării podurilor în Konigsberg şi graful asociat.

Euler a demonstrat imposibilitatea existenței unui asemenea traseu, iar demonstrația a pus începutul teoriei grafurilor.

Cicluri Euler

Def. Fie $G=(V,E)$ un graf neorientat. *Ciclu eulerian* se numeşte ciclul care trece pe fiecare muchie a grafului G o singură dată. *Lanț eulerian* se numeşte lanțul, care trece exact câte o dată pe fiecare muchie a grafului.



Des. 9.5 Graf fără ciclu şi lanţ eulerian (a), cu ciclu eulerian (b), cu lanţ eulerian (c).

În procesul cercetării problemei despre podurile din Konigsberg a fost formulată următoarea teoremă:

Teorema 1. Un graf conex, neorientat G conține un ciclu (lanț) eulerian atunci şi numai atunci când numărul vârfurilor de grad impar este 0 (0 sau 2 – pentru lanț).

□ (ciclu)

Necesitate. Orice ciclu eulerian intră în vârf pe o muchie şi părăseşte vârful pe alta. Rezultă că puterea tuturor vârfurilor trebuie să fie pară, dacă graful G conține un ciclu eulerian.

Suficiență. Fie G un graf conex neorientat, toate vârfurile lui având puteri pare. Demonstrația va fi realizată prin construcție. Ciclul începe din un vârf arbitrar v_0 şi continuă pe muchii neparcurse, consecutive, până la revenirea în vârful inițial. Dacă au fost folosite toate muchiile – STOP – ciclul eulerian căutat a fost construit. În caz contrar - fie F – ciclul recent construit (după construcția lui au rămas muchii nefolosite). G este conex, prin urmare, F trece prin un vârf v_i, care aparține unei muchii neincluse în F. Dacă din G va fi exclus ciclul F, puterile tuturor vârfurilor vor rămâne pare. Începând cu v_i se construieşte un alt ciclu F', pe muchiile rămase în G. Dacă la construcția ciclului F' au fost consumate toate muchiile din G, procesul se opreşte. Ciclul eulerian începe în v_0, continuă pe F până la v_i, urmează integral F', revine în v_i şi continuă pe F până la v_0. Dacă însă au mai rămas vârfuri

nefolosite în G, se consideră $F \leftarrow F \cup F'$, apoi se identifică un vârf v_j , care aparține ciclului F , dar şi unei muchii, care rămâne în G după excluderea F. Urmează identificarea unui alt ciclu F', care începe în v_j şi parcurge doar muchii încă neutilizate. Procesul se repetă până la epuizarea setului de muchii din G, ceea ce echivalează cu construcția unui ciclu eulerian.

■

Teorema 2. Un graf conex, orientat G conține un ciclu (lanț cu începutul în p şi sfârşitul în q) eulerian atunci şi numai atunci când:

(i) $\forall v_i \in V, \left|\Gamma^+(v_i)\right| = \left|\Gamma^-(v_i)\right|$ - pentru ciclul eulerian

(ii) $\forall v_i \in V, v_i \neq p, q \left|\Gamma^+(v_i)\right| = \left|\Gamma^-(v_i)\right|; \left|\Gamma^+(p)\right| = \left|\Gamma^-(p)\right| + 1; \left|\Gamma^+(q)\right| = \left|\Gamma^-(q)\right| - 1$

- pentru lanțul eulerian.

□■

9.4 Algoritmul de construcţie a ciclului eulerian

Input. Graful $G = (V, E)$ dat prin matricea de adiacență **a**.

Output: Lista ordonată de parcurgere a vârfurilor L pentru obținerea lanțului eulerian

Pas 0. $L = \varnothing$

Pas 1. Este selectat un vârf arbitrar v_0. $L \leftarrow v_0$. vârful curent $v \leftarrow v_0$.

Pas 2. Din vârful v se parcurge o muchie, care nu divizează graful în componente conexe separate. Fie aceasta (v, v'). Muchia (v, v') se exclude din G. $L \leftarrow v' \quad v \leftarrow v'$

Pas 3. Cât timp în G mai există muchii se revine la pasul 2.

Pas 4. Se afişează lista L.

Implementare (cazul lanţului Euler)

Input. Graful $G=(V,E)$ dat prin matricea de adiacență **a**.

Output: Lista ordonată de parcurgere a vârfurilor S pentru obținerea lanțului eulerian

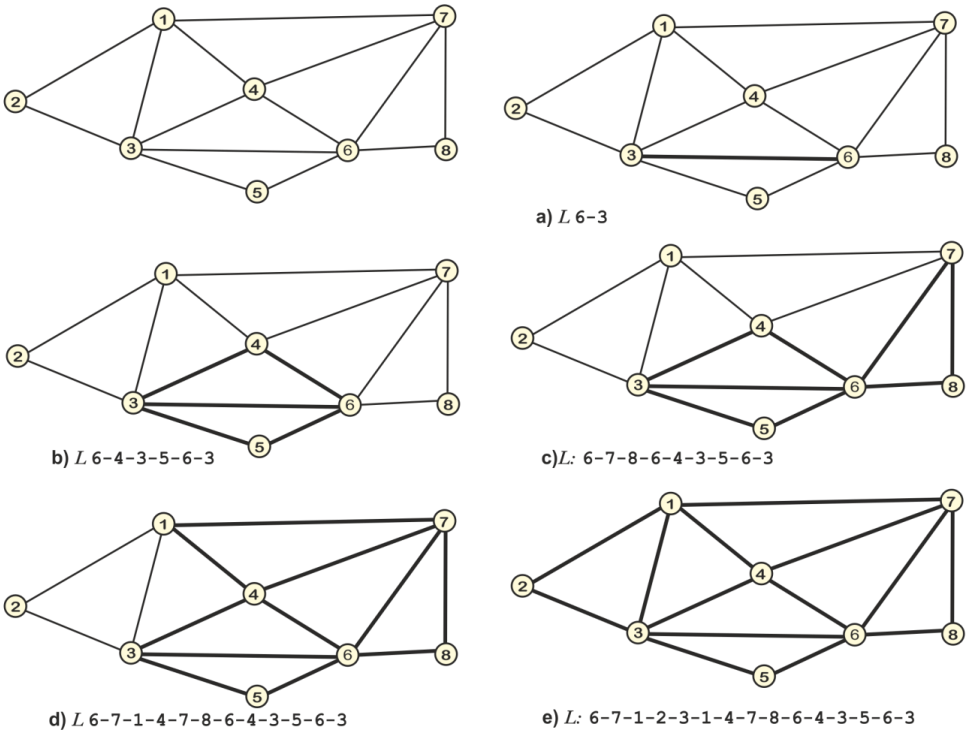
Structuri de date auxiliare: `transa` - matricea de adiacență a grafului curent `lant` - ciclul de creştere curent, `d` - marcajele vârfurilor la crearea ciiclului

```
int readdata()
{ … // subprogramul de citire a datelor initiale din fisier }

int print( int m)
{ … // subprogramul de afisare a lantului curent }

int verif()
// subprogramul de verificare dacă graful este unul eulerian
// se stabilesc şi vârfurile între care se va construi lanţul
{ int i,j, k;
for (i=1;i<=n;i++)
{   k=0;
    for (j=1;j<=n; j++)
      if (a[i][j] !=0 ) k++;
      if (k%2==1 && s1==0) s1=i; else
      if (k%2==1 && s1 !=0 && t==0) t=i; else
      if (k%2==1 && s1 !=0 && t!=0) return 0;
}
      if (s1==0 && t==0 ) {s1=1; t=2; return 1;}
      if (s1 != 0 && t==0 ) {return 0;}
      return 1;
}

int lan () // subprogramul de construcţie a primului lanţ
{ int  i, x;
  for (i=1;i<=n;i++) d[i].st=0;
  d[s1].st=1; x=s1;
do
{ for(i=1;i<=n;i++)
      if(a[x][i] !=0 && d[i].st==0)
        {   d[i].st=1; d[i].sursa=x;}
```

```
 d[x].st=2;
 k=1; while (d[k].st !=1) k++;
 x=k;
} while (x!=t);
 l=1; k=1;
 s[l]=lant[k]=t;
 while (x !=s1)
 {      x=d[x].sursa;
        l++; k++;
        s[l]=lant[k]=x;
 }
return 1;
}

int grad(int x)
// subprogram pentru determinarea gradului vârfului în graf
{ int s=0,i;
for (i=1;i<=n; i++)
       if (transa[x][i]!=0) s++;
return s;
}

int exclude_cic()
// subprogram pentru excluderea ciclului curent din graf
{ int i;
  transa[s[1]][s[l]]=transa[s[l]][s[1]]=0;
  for (i=1;i<l;i++)
      transa[s[i]][s[i+1]]= transa[s[i+1]][s[i]]=0;
  return 1;
}

int exist()
// verificare a existenţei vârfurilor pentru adăugare
{ int i;
  for (i=1;i<=n;i++) if (grad(i)>0) return i;
  return 0;
}

int insert(int x) // subprogram pentru inserarea ciclului
curent în lanţ
{  int i,j, news[???];
   for (i=1;i<=x; i++) news[i]=s[i]; i--;
   for (j=1;j<=k; j++) news[i+j]=lant[j];j--;
   for (i=x+1;i<=l;i++) news[i+j]=s[i];
```

```
   l+=k;
   for (i=1;i<=l;i++) s[i]=news[i];
   return 1;
}

int primul()
// determinarea primului vârf de creştere a lanţului
{ int i;
      for (i=1;i<=l;i++)
            if (grad(s[i])>0) {pr=i; return s[i];}
      return 0;
}

int ciclu () // determinarea ciclului de creştere a lanţului
{ int  i, x;
  s1=primul();
  for (i=1;i<=n;i++) if (transa[s1][i] > 0) {t=i; break;}
  for (i=1;i<=n;i++) d[i].st=0;
  d[s1].st=1; x=s1;
  transa[s1][t]=transa[t][s1]=0;
do
{ for(i=1;i<=n;i++)
      if(transa[x][i] !=0 && d[i].st==0)
        {   d[i].st=1; d[i].sursa=x;}
  d[x].st=2;
  k=1; while (d[k].st !=1) k++;
  x=k;
} while (x!=t);
  k=1; lant[k]=t;
  while (x !=s1)
  {
        x=d[x].sursa;
        k++; lant[k]=x;
  }
 return 1;
}

int main()
{ readdata();
  if (verif()==0) {printf ("/n Graful nu este eulerian /n ");
      return 0;}
  lan();
  do {   exclude_cic();
         if (exist()!=0)
```

```
            {  ciclu();
               insert(pr);
               print(l);
            }
      } while (exist()>0);
  return 0;
}
```



Des. 9.6 Ilustrarea rezultatelor generate de program. Lanţul iniţial **6-3** (**a**). Cicluri de creştere **6-4-3-5-6 (b)**, **6-7-8-6 (c)**, **7-1-4-7 (d)**, **1-2-3-1 (e).**

Exerciţii

1. Elaborați un program pentru determinarea mulțimii fundamentale de cicluri în baza unei tuturor carcase a unui graf $G = (V, E), \quad |V| \leq 20$

2. Elaborați un program pentru verificarea prezenței unui ciclu Euler într-un graf $G = (V, E), \quad |V| \leq 20$

3. Elaborați un program pentru verificarea prezenței unui lanț Euler într-un graf $G = (V, E), \quad |V| \leq 20$

Capitolul 10. Cicluri hamiltoniene

În acest capitol:

- Ciclul şi lanţul hamiltonian
- Algoritmi exacţi pentru determinarea ciclului (lanţului) hamiltonian
- Complexitatea algoritmilor pentru determinarea ciclului hamiltonian
- Problema comisului voiajor

10.1 Cicluri şi lanţuri hamiltoniene

Mai multe procese industriale presupun prelucrarea unui număr de produse cu ajutorul unei singure instalații. Timpul total şi cheltuielile depind doar de numărul operațiilor de calibrare (ajustare, curățare etc.) a instalației între prelucrarea a oricare două produse p_i, p_j. Mai mult, există perechi de produse, prelucrarea consecutivă a cărora permite funcționarea continuă a instalației, fără calibrare. Prin urmare atât timpul total de prelucrare, cât şi cheltuielile pot fi minimizate prin selectarea unei consecutivități de prelucrare a produselor care va necesita un număr minim (sau nul) de operații de calibrare.

Dacă produsele prelucrate $p_1, p_2, ..., p_n$ formează vârfurile unui graf, iar perechile de produse „compatibile" (p_i, p_j) - muchiile lui, atunci determinarea unui ciclu care trece exact câte o singură dată prin fiecare vârf al grafului este echivalentă cu determinarea unei consecutivități de prelucrare a produselor care nu necesită nici o operație de calibrare. Problema rămâne aceeaşi, indiferent de faptul dacă graful este unul orientat sau nu.

Ciclul, care trece exact câte o dată prin fiecare vârf al grafului se numeşte *ciclu hamiltonian*.

Există câteva formulări ale problemei, în dependență de tipurile de grafuri şi restricții. În continuare vor fi cercetate două probleme distincte:

a. Este dat un graf (ne)orientat . Se cere să se determine unul sau toate ciclurile hamiltoniene, dacă acestea există
b. Este dat un graf complet cu muchii ponderate (ponderea muchiei $(i, j) \rightarrow c_{i,j}$). Se cere să se determine un ciclu (lanț) hamiltonian de cost minim.

Spre deosebire de cazul ciclului eulerian, pentru ciclul hamiltonian nu există un criteriu exact de determinare a existenței acestuia într-un graf arbitrar. Totuşi, există câteva teoreme, care pot stabili prezența ciclului hamiltonian în anumite grafuri:

Teorema Dirac

Fie $G=(V,E)$ un graf neorientat cu $n>2$ vârfuri. Dacă $d(v)\geq \frac{n}{2}$ pentru orice vârf v din graf, atunci graful este hamiltonian.

□■

Teorema Ore

Fie $G=(V,E)$ un graf neorientat cu $n>2$ vârfuri. Dacă suma gradelor oricăror două vârfuri neadiacente din graf $d(u)+d(v)\geq n, \quad \forall u,v\in V:\bar{\exists}(u,v)\in E$, atunci graful este hamiltonian.

□■

Teorema Bondy-Chvátal

Notații: Fie $G=(V,E)$ un graf neorientat cu n vârfuri. Se numeşte închidere a grafului G graful G', obținut prin adăugarea a câte o muchie (u,v) pentru fiecare pereche de vârfuri u,v neadiacente în G, cu proprietatea că $d(u)+d(v)\geq n$. Închiderea grafului G se notează şi $cl(G)$.

Teorema

Fie $G=(V,E)$ un graf neorientat. G este hamiltonian dacă şi numai dacă $cl(G)$ este un graf hamiltonian

□■

10.2 Algoritmul pentru determinarea ciclurilor (lanţurilor) hamiltoniene

Pentru problema determinării ciclurilor (lanțurilor) hamiltoniene nu există algoritmi eficienți de complexitate polinomială. Problema este una din clasa NP [11, p. 378]. Prin urmare, o abordare admisibilă devine un algoritm recursiv, bazat pe tehnica reluării. Inițial, algoritmul propus de Roberts şi Flores [6, p. 249] presupunea abordarea iterativă:

Pseudocod (Roberts-Flores)

Pas 1. Se formează tabloul $M=[m_{i,j}]$ de dimensiuni $(k\times n)$, în care elementul $m_{i,j}$ este cel de al i–ulea vârf (fie v_q), pentru care în $G=(V,E)$ există arcul (v_j,v_q). Numărul de linii în matrice va fi determinat de cel mai mare semigrad de ieşire a vârfurilor din G. S – mulțimea vârfurilor incluse în soluție, în ordinea parcurgerii lor. $S\leftarrow\{\varnothing\}$.

Pas 2. Se alege un vârf arbitrar (fie v_i), de la care începe construcția ciclului (lanțului). Se include în $S\leftarrow S\bigcup v_i \quad v\leftarrow v_i$.

Pas 3. Se alege primul vârf nefolosit din coloana v . Fie acesta v_j Se încearcă adăugarea acestuia la S . Există două cazuri, când un vârf nu poate fi adăugat la mul-țimea S:

a. În coloana v nu există vârfuri nefolosite
b. Numărul de elemente în S este n. Secvența de vârfuri din S formează un lanț hamiltonian. În acest caz, dacă există un arc

(muchie) de la ultimul element din S la primul – a fost identificat un ciclu hamiltonian. În caz contrar există doar lanțul hamiltonian.

În cazurile (a),(b) se trece la pasul 5, altfel $S \leftarrow S \bigcup v_j$ şi $v \leftarrow v_j$

Pas 4. Se repetă pasul 3 până la apariția uneia din situațiile descrise în 3(a) sau 3(b)

Pas 5. Fie $S = (v_1, v_2, v_{k-1}, v_k)$. $S \leftarrow S - \{v_k\}$. Dacă $S \neq \{\varnothing\}$, $v \leftarrow v_{k-1}$ apoi se trece la pasul 3, altfel STOP – toate secvențele posibile au fost cercetate.

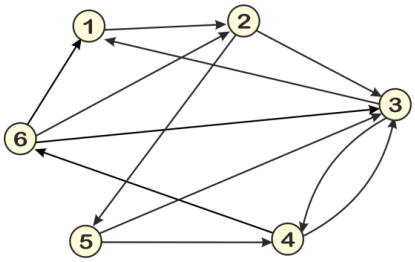
Implementare (recursiv)

Următorul program permite determinarea lanțurilor hamiltoniene. Matricea de adiacență este stocată în tabloul a, soluțiile se formează în tabloul s. Programul permite urmărirea dinamicii construcției lanțurilor hamiltoniene. Tehnica de implementare – reluare.

```
#include <conio.h>
#include <stdio.h>

int a[30][30], m[30][30], s[30], n,i,j,k,ne;
FILE *f;

int readdata()
{ int i,j;
  f=fopen("dataham.in", "r");
  fscanf(f, "%d", &n);
  for (i=1;i<=n;i++)
   for (j=1; j<=n; j++)
    fscanf(f, "%d", &a[i][j] );
  fclose(f);
  return 0;
}

int make_m()
{ int i,j,k;
for (i=1;i<=n;i++)
{ k=0;
  for (j=1;j<=n;j++)
        if ( a[i][j] != 0)
```

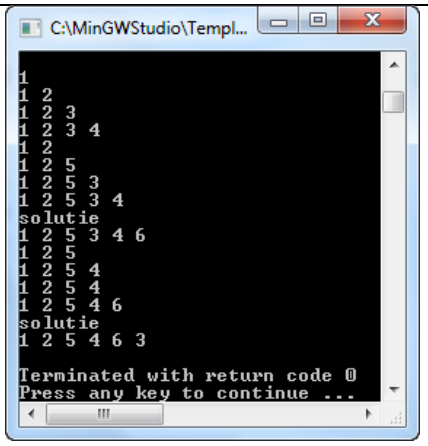
```
        { k++;
          m[k][i]=j;
        }
}
}

int print_s(int q)
{ int i;
  printf("\n");
  for (i=1; i<=q; i++) printf("%d ", s[i]);
  getch();
  return 0;
}
int exist (int a,int pos)
{ int i,k=0;
for (i=1;i<=pos;i++)
      if (s[i] == a) k=1;
return k;
}
int hamilton(int element, int index)
{ int i,j;
  if (exist (element, index-1)) return 0;
  s[index]=element;
  if  (index==n)  {  printf("\n  solutie  ");  print_s(index);
index--; return 0;}
  else { index++;
         j=1;
         do
         {  hamilton(m[j][element], index); j++; }
        while (m[j][element] !=0);
  index--;
  }
  return 0;
}

int main()
{ readdata();
  make_m();
  hamilton(1,1);
  return 0;
}
```

Exemplu



Matricea M pentru graful din imagine este

2	3	1	3	3	1
0	5	4	6	4	2
0	0	0	0	0	3



10.3 Problema comisului voiajor

Problema determinării ciclului (lanțului) hamiltonian de cost minim pe un graf complet, ponderat, mai este cunoscută şi ca *problema comisului voiajor*.

O abordare directă a problemei comisului voiajor prin construcția arborelui de soluții şi selectarea celui mai puțin costisitor ciclu hamiltonian este evidentă. Totuşi, se observă o legătură între problema comisului voiajor şi problema arborelui parțial de cost minim, cercetată anterior. Dacă arborele de cost minim formează un lanț, atunci aceasta este şi un lanț hamiltonian, iar dacă suplimentar, în graful inițial mai există şi muchia care uneşte vârfurile terminale ale lanțului – acesta se transformă în ciclu hamiltonian. Prin urmare, pentru a rezolva problema comisului voiajor este suficient să se determine arborele parțial de cost minim cu proprietatea că $n-2$ vârfuri ale arborelui au gradul doi, iar două – gradul 1.

Se va încerca abordarea euristică a rezolvării problemei comisului voiajor

Există două probleme pentru detrminarea celui mai scurt lanț hamiltonian, formulate separat:

- ***Problema* (*a*)** *Cel mai scurt lanț hamiltonian*. Să se determine o carcasă $T=(V,E_T)$ a grafului K_n astfel încât gradele tuturor vârfurilor din T să nu depăşească 2.
- ***Problema* (*b*)** *Cel mai scurt lanț hamiltonian cu vârfuri terminale fixe*. Fiind date două vârfuri v_1 şi v_2 ale grafului K_n să se determine o carcasă $T=(V,E_T)$ a grafului astfel încât gradele vârfurilor v_1 şi v_2 în T să fie 1, iar gradele tuturor celorlalte vârfuri din T - 2.

Teorema 1 Fie $C=[c_{i,j}]$ matricea costurilor grafului inițial K_n şi w - o valoare suficient de mare (care depăşeşte lungimea oricărui lanț hamiltonian). Atunci rezolvarea problemei (a) cu matricea costurilor $C'=[c'_{i,j}]$ unde

$$\left.\begin{array}{l} c'_{j,1}=c_{j,1}+w \\ c'_{1,j}=c_{1,j}+w \\ c'_{2,j}=c_{2,j}+w \\ c'_{j,2}=c_{j,2}+w \end{array}\right\} \forall v_j \neq v_1, v_2$$

$$c'_{i,j}=c_{i,j}+2w \quad \forall v_i, v_j = v_1, v_2$$

$$c'_{i,j}=c_{i,j} \qquad \forall v_i, v_j \neq v_1, v_2$$

este în acelaşi timp şi rezolvarea problemei (b) pentru matricea inițială a costurilor pentru vârfurile terminale v_1 şi v_2.

□■

Teorema 2 Dacă matricea costurilor $C=[c_{i,j}]$ unui graf se transformă în matricea $C'=[c'_{i,j}]$ astfel încât $c'_{i,j}=c_{i,j}+p(i)+p(j) \quad i,j=1,...,n$ unde $p(k)$ este un vector de valori numerice constante, atunci lungimile relative ale tuturor lanțurilor hamiltoniene nu se modifică.

□■

Algoritmul de aplicare a amenzilor

Algoritmul presupune transformarea ponderilor muchiilor grafului inițial astfel încât arborii parțiali în formă de lanț să devină prioritari în procesul de construire a arborelui de cost minim.

Va fi cercetată problema construcției lanțului hamiltonian de cost minim între două vârfuri date v_1 şi v_2.

Fie dat graful complet $K_n = (V, E)$, cu matricea costurilor $C = [c_{i,j}]$, şi vârfurile $v_1, v_2 \in V$ între care urmează să fie construit lanțul hamiltonian de cost minim.

Pas 1. Matricea costurilor se modifică conform formulelor date de teorema menționată anterior.

Pas 2. Se determină un arbore parțial de cost minim $T = (V, E_T)$. Dacă acesta este un lanț hamiltonian – STOP - problema a fost rezolvată. în caz contrar se trece la pasul 3

Pas 3. Tuturor vârfurilor v_i din T cu grade ce depăşesc 2 li se aplică amenzi, conform formulei: $p(v_i) = z(d_{v_i}^T - 2)$.

Pas 4. Se recalculează matricea costurilor conform formulei: $c'_{i,j} = c_{i,j} + p(i) + p(j) \quad i, j = 1, ..., n$, apoi se revine la pasul 2.

Fiind un algoritm euristic, soluția obținută nu întotdeauna este una optimă.

Exerciţii

1. Elaborați un program pentru determinarea ciclului hamiltonian (sau a lipsei acestuia) într-un graf $G = (V, E)$, $|V| \leq 20$

2. Elaborați un program pentru rezolvarea problemei comisului voiajor pe un graf complet K_N, $N \leq 20$. (folosiți tehnica reluării)

3. Elaborați un program pentru rezolvarea problemei comisului voiajor pe un graf complet K_N, $N \leq 20$. (folosiți algoritmul de aplicare a amenzilor)

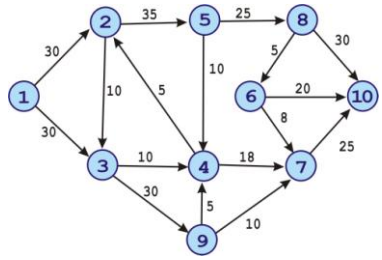
Capitolul 11. Fluxuri în reţea

În acest capitol

- Noţiune de reţea pe graf. Elemente ale fluxului
- Teorema despre tăietura minimă.
- Algoritmul Ford-Fulkerson
- Fluxul maxim pentru surse şi destinaţii multiple
- Fluxul maxim pe grafuri bipartite
- Aplicaţii ale problemei de flux maxim

11.1 Preliminarii

Problema fluxului maxim, la fel ca multe alte probleme formulate pe grafuri, îşi are rădăcinile în economia modernă, mai exact în optimizarea economică. Mai recente sunt aplicațiile în domeniul rețelelor şi fluxurilor informaționale. Dacă este cercetată o rețea de transportare a unui material din un punct în care acesta este produs (sursă) într-un punct de depozitare sau prelucrare (stoc) prin canale de transport cu anumite capacități, inevitabil apare problema determinării capacității de transportare a materialului de la sursă către stoc pentru toată rețeaua. Materialul şi canalele de transport pot avea cea mai diversă natură: produse petroliere şi conducte; piese şi transportoare; pachete de date şi canale informaționale, etc.



Des. 11.1.
Reţea de transport cu sursa în vârful 1 şi stocul – în vârful 10.

Pe grafuri problema se formulează astfel: Este dat un graf orientat $G=(V,E)$, cu ponderi anexate arcelor: pentru arcul (i,j) ponderea este $c_{i,j}$. Pentru două vârfuri date $s,t\in V$ se cere să se

determine fluxul maxim care poate fi deplasat din s (sursă) în t (stoc).(Des. 11.1)

11.2.Algoritm

Descriere generală

Algoritmul se bazează pe determinarea iterativă a unor drumuri de creştere a fluxului şi acumularea acestora într-un flux total, până la apariția în rețea a unei tăieturi[9], care separă sursa de stoc.

Se cercetează graful $G=(V,E)$ cu capacitățile arcurilor $c_{i,j}$, sursa s şi stocul t ($s,t\in V$). Pe arcuri se definesc marcajele $e_{i,j}$, care vor indica valoarea curentă a fluxului de-a lungul arcului (i,j). Pentru marcajele arcurilor se va respecta următoarea condiție:

$$\sum_{v_j\in\Gamma(v_i)} e_{i,j} - \sum_{v_k\in\Gamma^{-1}(v_i)} e_{k,i} = \begin{cases} w & v_i = s \\ -w & v_i = t \\ 0 & v_i \neq s,t \end{cases}$$

Cu alte cuvinte: sursa s produce un flux de mărime w. Pentru orice vârf intermediar fluxul de intrare este egal cu fluxul de ieşire. În stocul t intră un flux de mărime w.

Problema fluxului maxim se reduce la determinarea unui w:

$$w = \sum_{v_j\in\Gamma^{+}(s)} e_{i,s} = \sum_{v_k\in\Gamma^{-}(t)} e_{k,t} \to \max$$

Legătura între fluxul maxim şi tăietura minimă în graf este dată de următoarea teoremă:

Teoremă: Într-un graf orientat $G=(V,E)$ valoarea w a fluxului maxim din s în t este egală cu mărimea tăieturii minime $(V_m \to V_m)$, care separă s de t.

[9] Tăietură în graf – un set de muchii (arcuri), lichidarea cărora divide graful în două componente conexe.

Tăietura $(V_0 \to V_0)$ separă s de t dacă $s \in V_0, t \notin V_0$. Mărimea tăieturii este suma ponderilor arcurilor din $G = (V, E)$, cu începutul în V_0 şi sfârşitul în V_0, sau $w\left(V_0 \to V_0\right) = \sum_{(v_i, v_j) \in (V_0 \to V_0)} c_{i,j}$.

Tăietura minimă $(V_m \to V_m)$ este tăietura pentru care $w\left(V_m \to V_m\right) = \min_{(V_0 \to V_0)} \left(\sum_{(v_i, v_j) \in (V_0 \to V_0)} c_{i,j} \right)$ □ ■

Notații:

Pentru implementarea algoritmului vor fi folosite atât *marcaje pentru arcuri* cât şi *marcaje pentru vârfurile grafului*. Marcajul unui vârf v este format din trei componente: *precedent*, *flux*, *stare*, care au semnificația:

precedent – vârful care îl precede pe v în drumul de creştere curent

flux – mărimea fluxului, care ajunge în vârful v pe drumul de creştere curent

stare – starea curentă a vârfului v (vârful poate fi în una din trei stări: *necercetat* [marcaj nul, valoarea - 0], *atins*[vecin al unui vârf cercetat, valoarea - 1], *cercetat*[toți vecinii – atinşi, valoarea - 2]).

Vecinii vârfului x

- $\Gamma^+(x)$: $V^+ \subset V : \forall z \in V^+, \exists (x, z) \in E$ (mulțimea vârfurilor în care se poate ajunge direct din vârful x).
- $\Gamma^-(x)$: $V^- \subset V : \forall z \in V^-, \exists (z, x) \in E$ (mulțimea vârfurilor din care se poate ajunge direct în vârful x).

Pseudocod

Pas 0 Marcajele arcurilor se inițializează cu 0.

Pas 1 Marcajele vârfurilor se inițializează: *precedent* ← 0, *stare* ← 0, *flux* ← ∞. Marcajele muchiilor ***se păstrează***.

Pas 2 Inițializarea sursei *s*.

Marcajul sursei *s* : $(+s, +\infty, 1)$[10] Se consideră $v_i \leftarrow s$.

Pas 3 Cercetarea vârfului atins v_i.

a. Pentru toate vârfurile necercetate $v_j \in \Gamma^+(v_i) : e_{i,j} < c_{i,j}$ se aplică marcajul $(+v_i, \Delta, 1)$, unde $\Delta = \min(\Delta_{v_i}, c_{i,j} - e_{i,j})$;

b. Pentru toate vârfurile necercetate $v_j \in \Gamma^-(v_i) : e_{j,i} > 0$ se aplică marcajul $(-v_i, \Delta, 1)$, unde $\Delta = \min(\Delta_{v_i}, e_{j,i})$.

c. Vârful v_i este marcat cercetat. (Componenta *stare* primeşte valoarea 2)

Pas 4

a. Dacă vârful *t* este *atins*, se trece la pasul 5;(drumul curent de creştere a fost construit), altfel:

b. dacă există vârfuri *atinse*, dar *t* încă nu este *atins*, este selectat un vârf *atins* v_i şi se revine la pasul 3, altfel:

c. fluxul maxim a fost obținut. SFÂRŞIT[11].

Pas 5 *Creşterea fluxului. Modificarea marcajelor pe arcuri.*

a. Se consideră $v \leftarrow t$; $\Delta_c = \Delta_t$

b. Dacă vârful v are marcajul de forma $(+z, \Delta, *)$ valoarea fluxului de-a lungul arcului (*z, v*) este majorată cu Δ_c :

$e_{z,v} \leftarrow e_{z,v} + \Delta_c$,

altfel, dacă vârful v are marcajul de tip $(-z, \Delta, *)$ valoarea fluxului de-a lungul arcului (*v, z*) este micşorată cu Δ_c :

$e_{v,z} \leftarrow e_{v,z} - \Delta_c$;

c. $v \leftarrow z$. Dacă $v \neq s$, se revine la pasul 5.b, altfel la pas 1.

---

[10] *Precedent* – se consideră tot nodul sursă *s*, valoarea *fluxului* în *s* se consideră infinită, *s* se consideră *atins*.

[11] Nu mai există o creştere a fluxului, care ar ajunge la destinaţia (stocul) *t*. Creşterea precedentă a fluxului a determinat tăietura minimă, care separă *s* de *t*.

Exemplu

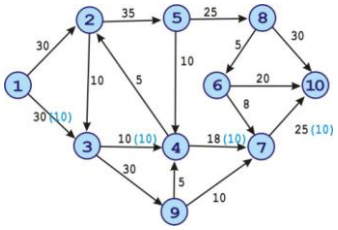
Graful 11.1. Sursa - vârful **1**, stocul – vârful **10**.

ITERAŢIA 1

iniţializare sursa 1: 1-(1, ∞,1)

```
cercetare 1 : 2-(1,30,1); 3-(1,30,1);   1-(1,∞,2 )
cercetare 2 : 5-(2,30,1);               2-(1,30,2)
cercetare 3 : 4-(3,10,1); 9(3,10,1)     3-(1,30,2)
cercetare 4 : 7-(4,10,1);               4-(3,10,2)
cercetare 5 : 8-(5,25,1);               5-(2,30,2)
cercetare 7 : 10 -(7,10,1);             7-(4,10,2)
```

nod	1	2	3	4	5	6	7	8	9	10
precedent	1	1	1	3	2	0	4	5	3	7
flux	∞	30	30	10	30	0	10	25	10	10
stare	2	2	2	2	2	0	2	1	1	1



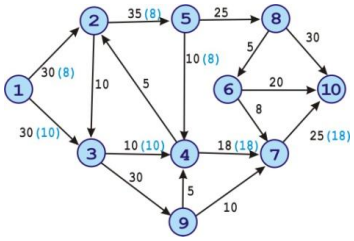
În stoc se ajunge cu un flux de valoare 10. Marcajele arcurilor, care formează drumul de creştere a fluxului (7,10) (4,7) (3,4) (1,3) se modifică.

ITERAŢIA 2

iniţializare sursa 1: 1-(1, ∞,1)

```
cercetare 1 : 2-(1,30,1); 3-(1,20,1);   1-(1,∞,2 )
cercetare 2 : 5-(2,30,1);               2-(1,30,2)
cercetare 3 : 9(3,10,1)                 3-(1,20,2)
cercetare 5 : 4-(5,10,1); 8-(5,25,1);   5-(2,30,2)
cercetare 4 : 7-(4,8,1);                4-(5,10,2)
cercetare 7 : 10 -(7,8,1);              7-(4,8,2)
```

nod	1	2	3	4	5	6	7	8	9	10
precedent	1	1	1	5	2	0	4	5	3	7
flux	∞	30	20	10	30	0	8	25	10	8
stare	2	2	2	2	2	0	2	1	1	1



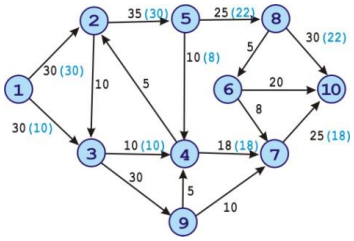
În stoc se ajunge cu o creştere a fluxului de valoare 8. Marcajele arcurilor, care formează drumul de creştere (7,10) (4,7) (5,4) (2,5) (1,2) se modifică.

ITERAŢIA 3

iniţializare sursa 1: 1-(1, ∞,1)

```
cercetare 1 : 2-(1,22,1); 3-(1,20,1);   1-(1,∞,2 )
cercetare 2 : 5-(2,22,1);               2-(1,22,2)
cercetare 3 : 9(3,10,1)                 3-(1,20,2)
cercetare 5 : 4-(5,2,1); 8-(5,22,1);    5-(2,22,2)
cercetare 4 :                           4-(5,2,2)
cercetare 8 : 6-(8,5,1);10 -(8,22,1);   8-(5,22,2)
```

nod	1	2	3	4	5	6	7	8	9	10
precedent	1	1	1	5	2	8	0	5	3	8
flux	∞	22	20	2	22	5	0	22	10	22
stare	2	2	2	2	2	1	0	2	1	1



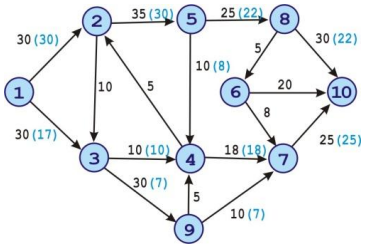
Creşterea fluxului: 22. Marcajele arcurilor, care formează drumul de creştere (8,10) (5,8) (2,5) (1,2) se modifică.

ITERAŢIA 4

iniţializare sursa 1: 1-(1, ∞,1)

```
cercetare 1 : 3-(1,20,1);                    1-(1,∞,2 )
cercetare 3 : 9(3,10,1)                      3-(1,20,2)
cercetare 9 : 4-(9,5,1); 7-(9,10,1);         9-(3,10,2)
cercetare 4          : 2-(4,5,1); 5-(-4,5,1)      4-
(9,5,2)
cercetare 2          :                            2-
(4,5,2)
cercetare 5          : 8-(5,3,1)                  5-(-
4,5,2)
cercetare 7          : 10-(7,7,1)                 7-
(9,10,2)
```

nod	1	2	3	4	5	6	7	8	9	10
precedent	1	4	1	9	-4	0	9	5	3	7
flux	∞	5	20	5	5	0	10	3	10	7
stare	2	2	2	2	2	0	2	1	2	1



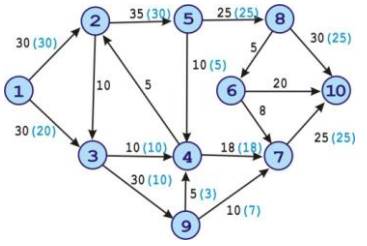
Creşterea fluxului: 7. Marcajele arcurilor, care formează drumul de creştere (7,10) (9,7) (3,9) (1,3) se modifică.

ITERAŢIA 5

iniţializare sursa 1: 1-(1, ∞,1)

```
cercetare 1 : 3-(1,13,1);                    1-(1,∞,2 )
cercetare 3 : 9(3,3,1)                       3-(1,13,2)
cercetare 9 : 4-(9,3,1); 7-(9,3,1);          9-(3,3,2)
cercetare 4 : 2-(4,3,1); 5-(-4,3,1)          4-(9,3,2)
cercetare 2 :                                2-(4,3,2)
cercetare 5 : 8-(5,3,1)                      5-(-4,3,2)
cercetare 7 :                                7-(9,3,2)
cercetare 8 : 10-(8,3,1)                     8-(5,3,2)
```

nod	1	2	3	4	5	6	7	8	9	10
precedent	1	4	1	9	-4	8	9	5	3	8
flux	∞	3	13	3	3	3	3	3	3	3
stare	2	2	2	2	2	1	2	2	2	1



Creşterea fluxului: 3. Marcajele arcurilor, care formează drumul de creştere (8,10) (5,8) (5,4) (9,4) (3,9) (1,3) se modifică. Se observă micşorarea fluxului pe arcul (5,4) cu compensarea pe arcurile (3,9)(9,4).
Tot odată poate fi observată şi tăietura, formată de flux(5,8)(7,10). Prin urmare fluxul maxim între nodurile 1 şi 10 are valoarea 50. Următoarea iteraţie nu va mai realiza o creştere a fluxului.

Implementare algoritm

Următorul fragment de cod realizează o variantă demonstrațională a algoritmului pentru determinarea fluxului maxim în rețea. Rețeaua din **N** vârfuri este descrisă prin matricea de adiacență **A[N×N]**, marcajele arcurilor (fluxul generat) se păstrează în tabloul **E[N×N]**, marcajele vârfurilor – în tabloul **VERT[N]**.

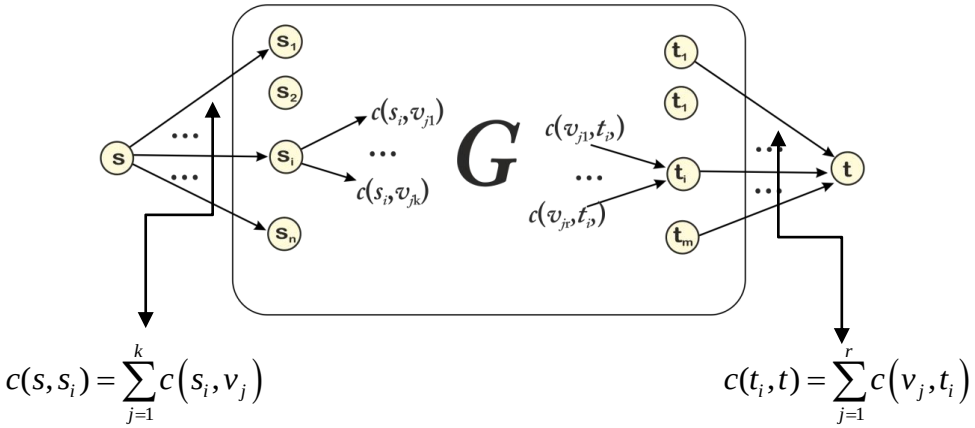
```
#include <conio.h>
#include <stdio.h>

int a[30][30], e[30][30],i,j,n,s,t, delta, big=0;
struct vertex{int flux; int sursa; int stare;} vert[30];
FILE *f;

int readdata()
 { int i,j;
   f=fopen("flux.in", "r");
   fscanf(f, "%d", &n);
   fscanf(f, "%d %d", &s, &t);
    for(i=1;i<=n;i++)
          for(j=1;j<=n;j++)
                  {fscanf(f, "%d", &a[i][j]); big+=a[i][j];}
  fclose(f);
  return 0;
 }

int init_vert()
 { int i;
   for (i=1;i<=n;i++)
     {vert[i].flux=big; vert[i].sursa=vert[i].stare=0;}
   vert[s].stare=1; vert[s].sursa=+s;
   return 0;
 }

int activ()
 {    int i;
      for  (i=1;i<=n;i++)
       if (vert[i].stare ==1) return i;
  return 0;
 }
```

```
int fluxtotal()
 { int i,ft=0;
   for (i=1;i<=n;i++) ft+=e[s][i];
  return ft;
 }

int abs(int x)
{ if (x<0) return x*-1; else return x;
}

int flux()
{ int x,i,d,q;
  // miscarea inainte, constructie lant
  do
    { x=activ();
      //dupa G+
       for (i=1;i<=n;i++)
        if (vert[i].stare==0 && a[x][i]>0 && e[x][i]<a[x][i])
                { d=a[x][i]-e[x][i];
                   if ( d<vert[x].flux) vert[i].flux=d;
                   else vert[i].flux=vert[x].flux;
                   vert[i].stare=1; vert[i].sursa=+x;
                };
      // dupa G-
       for (i=1;i<=n;i++)
        if (vert[i].stare==0 && e[i][x]>0)
                { d=e[i][x];
                  if (d<vert[x].flux) vert[i].flux=d;
                  else vert[i].flux=vert[x].flux;
                  vert[i].stare=1; vert[i].sursa=-x;
                }
     vert[x].stare=2;
    }
 while (vert[t].stare !=1 && activ() !=0 );
   // miscarea inapoi, extinderea fluxului
   delta=0;
 if (vert[t].stare==1 )
      { x=t;
        delta=vert[t].flux;
        do
          { q=abs(vert[x].sursa);
            if (vert[x].sursa>0) e[q][x]=e[q][x]+delta;
            if (vert[x].sursa<0) e[x][q]=e[x][q]-delta;
            x=q;
```

```
          }
        while (x!=s);
 }

int main()
{ readdata();
  do
  { init_vert();
    flux();
  } while (delta !=0);
 printf("\n%d", fluxtotal());
 return 0;
}
```

11.3 Flux maxim cu surse şi stocuri multiple

Fie dat un graf $G=(V,E)$ cu multiple surse (n_s) şi stocuri (n_t). Se presupune că fluxul poate fi direcționat de la orice sursă la orice stoc. Problema determinării unui flux de mărime maximă de la toate sursele la toate stocurile se reduce la problema clasică a fluxului maxim prin adăugarea unei surse virtuale şi a unui stoc virtual, conectate prin arce la celelalte surse (respectiv stocuri)(desenul 11.2).



Des. 11.2 Adăugarea sursei şi stocului virtual

Pentru fiecare arc (s,s_i) ponderea acestuia, $c(s,s_i)=\sum_{j=1}^{k} c(s_i,v_j)$ se calculează ca fiind suma ponderilor arcelor cu originea în s_i.

Pentru fiecare arc (t_i,t) ponderea acestuia, $c(t_i,t)=\sum_{j=1}^{r}c\left(v_j,t_i\right)$ se calculează ca fiind suma ponderilor arcelor cu vârful în t_i.

11.4 Flux maxim pe grafuri bipartite

Algoritmul pentru determinarea fluxului maxim poate fi utilizat eficient şi pentru rezolvarea altor probleme pe anumite clase de grafuri. Astfel, problema *cuplajului maxim* pe grafuri bipartite se rezolvă eficient prin adăugarea unei surse virtuale la o parte a grafului şi a unui stoc virtual la cealaltă parte.

Fie $G=(V,E)$ bipartit: $V=V'\bigcup V''$, $V'\bigcap V''=\varnothing$ şi $\forall e=(u,v)\in E \quad u\in V',v\in V''$ sau $v\in V',u\in V''$, neorientat, neponderat.

Transformare graf

Pas 1. Se adaugă două vârfuri *s, t* – inițial izolate. $V=V\bigcup\{s,t\}$.

Pas 2. Toate muchiile se transform în arce, orientate de la V' către V''

Pas 3.

a) Pentru fiecare vârf $v\in V'$ se formează arcul $_{\text{C}}$, ponderea căruia este $c_{s,v}=\left|\Gamma^{+}(v)\right|$

b) Pentru fiecare vârf $u\in V''$ se formează arcul (u,t), ponderea căruia este $c_{u,t}=\left|\Gamma^{-}(u)\right|$

Pas 4. Pe graful astfel format se lansează algoritmul determinării fluxului maxim din s în t.

Pas 5 Rezultat:

Arcele de forma şi $e=(u,v)\in E \quad u\in V',v\in V''$ sau $v\in V',u\in V''$ incluse în fluxul maxim vor forma cuplajul maxim pe muchiile grafului inițial G.

11.5 Flux maxim pe grafuri cu capacităţi restricţionate ale vârfurilor şi muchiilor

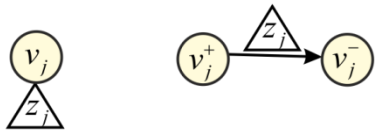
Mai multe probleme, rezolvarea cărora implică utilizarea algoritmilor de flux maxim conțin restricții (caracteristici) asociate nu doar arcurilor grafului ($c_{i,j}$), dar şi vârfurilor acestuia (z_j).

Fie graful $G=(V,E)$ cu ponderile arcelor $c_{i,j}$ $(i=1,...,n;\ j=1,...,n)$ şi capacitățile vârfurilor z_j $(j=1,...,n)$. Restricția impusă de capacitățile vârfurilor determină valoarea fluxului total $e_{i,j}$ $(i=1,...,n)$, la intrare în vârful v_j ca $\sum_{i=1}^{n} e_{i,j} \le z_j \quad (j=1,...,n)$

Pe graful $G=(V,E)$ se cere determinarea unui flux maxim de la vârful *s* la vârful *t*.

Rezolvarea presupune transformarea grafului inițial $G=(V,E)$, într-un alt graf $G_0=(V_0,E_0)$, care va conține doar restricții la capacitățile arcelor. Pe graful $G_0=(V_0,E_0)$ se va aplica un algoritm clasic pentru determinarea fluxului maxim, care va genera şi soluția problemei inițiale.

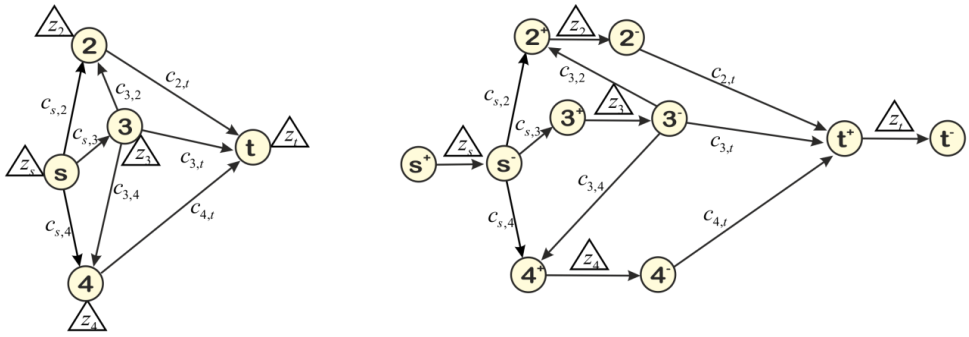
Modelul de transformare este următorul: fiecare vârf v_j al grafului inițial este înlocuit cu o pereche de vârfuri v_j^+, v_j^- şi un arc $\left(v_j^+, v_j^-\right)$ care le uneşte. Capacitatea arcului este egală cu capacitatea vârfului $v_j : c\left(v_j^+, v_j^-\right) = z_j$. (desen 11.3)



Des. 11.3 Transformarea vârfurilor grafului *G*.

Arcele $\left(v_i, v_j\right)$ din graful inițial se transformă în G_0 în arce $\left(v_i, v_j^+\right)$ cu aceeaşi capacitate. La fel se transformă arcele care îşi au

originea în vârful v_j: (v_j, v_i) trec în (v_j^-, v_i). Exemplu transformare, desenul 11.4:



Des. 11.4 Transformarea grafului *G*

După transformările efectuate fluxul maxim pe graful G_0 va corespunde fluxului maxim pe graful G . Este important de menționat, că în cazul când tăietura determinată de fluxul maxim pe G_0 nu conține nici unul din arcele formate la transformarea grafului G, restricțiile de capacitate ale vârfurilor nu sunt semnificative.

Exerciţii

1. Elaborați un program pentru separarea mulțimii de vârfuri a unei rețele de transport în două componente distincte: mulțimea de surse şi mulțimea de stocuri.

2. Elaborați un program pentru determinarea fluxului maxim pe grafuri cu surse şi destinații multiple.

3. Elaborați un program pentru transformarea grafului cu restricții aplicate pe vârfuri într-un graf cu restricții aplicate pe muchii.

4. Elaborați un program pentru determinarea fluxului maxim pe grafuri cu restricții aplicate pe vârfuri.

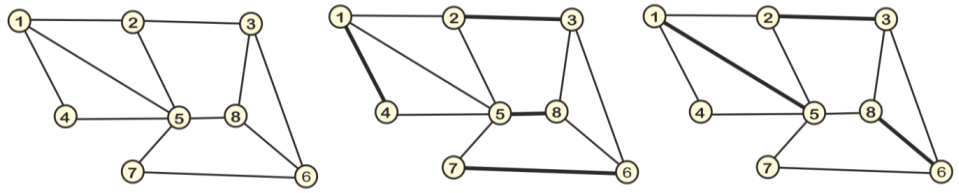
Capitolul 12. Cuplaje

În acest capitol

- Noţiune de cuplaj, cuplaj maxim
- Grafuri asociate
- Legătura între problema cuplajului maxim şi a mulţimii maximal independente
- Algoritmi pentru determinarea tuturor cuplajelor maxime
- Complexitatea algoritmului pentru determinarea tuturor cuplajelor maxime

12.1 Cuplaje

Într-un graf neorientat $G=(V,E)$ mulțimea de muchii M se numeşte *cuplaj* dacă oricare două muchii din M nu an vârfuri comune. Cuplajul M se numeşte *maxim*, dacă $\forall e' \in E-M$, în mulțimea $M \cup \{e'\}$ există muchia $e'': e' \cap e'' = v^*; v^* \in V$. La fel ca şi mulțimile independente maxime, cuplajele maxime pot fi formate din mulțimi cu număr diferit de elemente (vârfuri – pentru mulțimile maximal independente, muchii – pentru cuplaje), desen 12.1.



a **b** **c**

Des. 12.1 Cuplaje maxime în graful (a). Cuplaj maxim pe patru muchii (1,4) (2,3) (5,8) (6,7) **(b);** cuplaj maxim pe trei muchii (1,5) (2,3) (6,8) **(c)**

Pentru un graf arbitrar pot fi formulate mai multe probleme, pentru rezolvarea cărora este necesară determinarea unui cuplaj cu

proprietăți prestabilite. De exemplu problema cuplajului maxim pe anumite categorii de grafuri se rezolvă eficient (de exemplu pe grafurile bipartite, unde poate fi redusă la problema fluxului maxim). De asemenea este cunoscut algoritmul maghiar pentru determinarea cuplajului maxim într-un graf arbitrar [6, p.396].

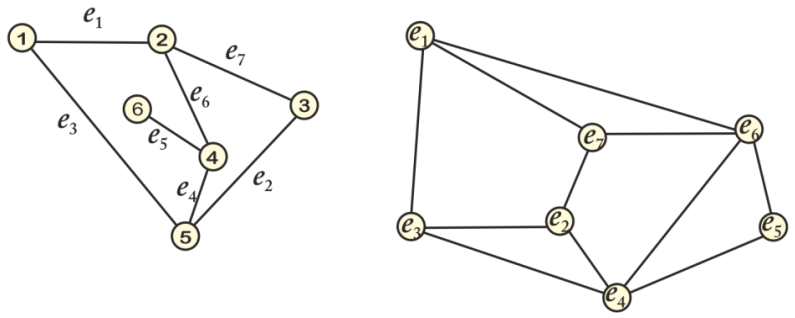
În continuare va fi cercetată problema generală de generare a tuturor cuplajelor unui graf neorientat arbitrar $G=(V,E)$

Problema va fi rezolvată prin reducerea în timp pătratic la o altă problemă pe grafuri, rezolvarea căreia este deja cunoscută – problema mulțimii maximal independente, mai exact – *problema generării tuturor mulțimilor maximal independente.*

12.2 Graf asociat

Fie graful neorientat $G=(V,E)$ $E=\{e_1,...,e_M\}$. Se va construi graful $G_A=(V_A,E_A)$, unde $V_A=\{e_1,...,e_M\}$ iar $E_A=\{(e_i,e_j):\exists u\in V, u\in e_i \,\&\, u\in e_j\}$.

Exemplu: fie graful $G=(V,E)$ din figura 12.2.a. Fiecare muchie din se transformă într-un vârf al grafului $G_A=(V_A,E_A)$ din desenul 12.2.b



Des. 12.2 Graful inițial (a) şi asociat (b).

Oricare două vârfuri $\left(e_i, e_j\right)$ din $G_A = (V_A, E_A)$ sunt conectate prin muchie, dacă în graful inițial există un vârf comun u al muchiilor e_i, e_j.

Teoremă: fiecărui cuplaj maxim în graful $G = (V, E)$ îi corespunde o mulțime maximal independentă în $G_A = (V_A, E_A)$ şi invers: fiecărei mulțimi maximal independente în $G_A = (V_A, E_A)$ îi corespunde un cuplaj maxim în $G = (V, E)$.

□

Direct. Rezultă direct din proprietățile $G_A = (V_A, E_A)$ Oricare două muchii $G = (V, E)$ care nu au vârf comun nu sunt adiacente în $G_A = (V_A, E_A)$. Prin urmare, muchiilor din cuplajul maxim în $G = (V, E)$ le corespunde o mulțime independentă în $G_A = (V_A, E_A)$. Aceasta este şi maxim independentă, altfel în $G = (V, E)$ ar exista cel puțin încă o muchie, care ar putea fi adăugată în cuplaj, păstrând proprietățile acestuia.

Invers. Fie $M_A = \left\{e_A^1, ...e_A^k\right\}$ o mulțime maxim independentă în $G_A = (V_A, E_A)$. Atunci în $G = (V, E)$ există cuplajul $\left\{e_A^1, ...e_A^k\right\}$. Dacă el nu este unul maxim, $\exists e^* : \left\{e_A^1, ...e_A^k\right\} \bigcup \left\{e^*\right\}$ - cuplaj. În acest caz vârfurile u, v, care formează muchia e^*, nu sunt adiacente vârfurilor din mulțimea independentă M_A. În consecință $\left\{e_A^1, ...e_A^k\right\} \bigcup e^*$ este o mulțime maxim independentă, ceea ce contrazice presupunerile inițiale. ■

12.3 Funcţia de generare a grafului asociat

Din cele expuse anterior rezultă că problema determinării tuturor cuplajelor maxime pe $G = (V, E)$ este echivalentă cu problema determinării tuturor mulțimilor maxim independente pe $G_A = (V_A, E_A)$.

Pentru a rezolva problema este necesar, mai întâi să se construiască graful asociat $G_A = (V_A, E_A)$.

Algoritm:

Pas 1. Se creează lista muchiilor din $G = (V, E)$. $L = \{e_1, ...e_m\}$.

Pas 2. Se creează tabloul bidimensional E' – matricea de adiacență a grafului G_A .

Pas 3. Pentru toți *i* de la 1 la *m*-1.

Pentru toți *j* de la *i*+1 la *m*

Dacă în $G = (V, E)$ $e_i \bigcap e_j \neq \varnothing; \left(e_i, e_j \in E\right)$,

atunci $E'_{i,j} \leftarrow E'_{j,i} \leftarrow 1$ altfel $E'_{i,j} \leftarrow E'_{j,i} \leftarrow 0$

Implementare

Intrare: graful $G = (V, E)$, descris în tabloul bidimensional **a**.

Ieşire: matricea de adiacență a grafului $G_A = (V_A, E_A)$. Matricea de adiacență este localizată în tabloul bidimensional b.

```
int asociat ()
{ int i,j;
// modelare lista muchii
 for (i=1;i<=n;i++)
        for (j=1+i; j<=n; j++)
            if (a[i][j]!=0 ){m++; list[m].v1=i;list[m].v2=j;}
// modelare matrice adiacenta
 for (i=1;i<=m;i++)
        for (j=1+i; j<=m; j++)
            if (list[i].v1==list[j].v1 ||
            list[i].v1==list[j].v2 || list[i].v2==list[j].v1
            || list[i].v2==list[j].v2  ) b[i][j]=b[j][i]=1;
}
```

12.4 Generarea tuturor cuplajelor maxime

Problema este rezolvată direct prin aplicarea consecutivă a doi algoritmi descrişi anterior. Mai întâi se formează graful asociat $G_A = (V_A, E_A)$, apoi pentru acesta este aplicat algoritmul de generare a mulțimilor maxim independente. La generarea fiecărei soluții vârfurile care o formează în $G_A = (V_A, E_A)$ sunt transformate în muchiile corespunzătoare din $G = (V, E)$.

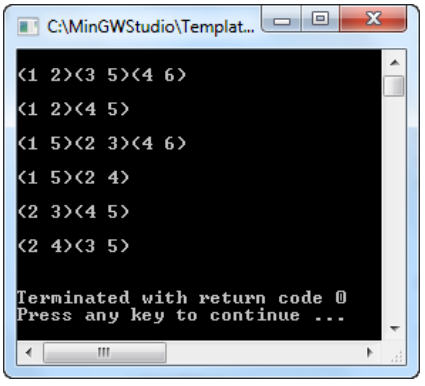
Exemplu: prototip program pentru generarea tuturor cuplajelor maxime

Intrare: graful $G = (V, E)$.

Ieşire: toate cuplajele maxime ale grafului $G = (V, E)$.

```
#include <conio.h>
#include <stdio.h>
struct edge{int v1; int v2;} list[400];
int a[20][20],b[400][400], q[20], s[20], m=0,i,j,n,k;
FILE *f;

int asociat ()
{ int i,j;
  for (i=1;i<=n;i++)
        for (j=1+i; j<=n; j++)
        if (a[i][j]!=0 ){m++; list[m].v1=i;list[m].v2=j;}
  for (i=1;i<=m;i++)
        for (j=1+i; j<=m; j++)
         if (list[i].v1==list[j].v1 || list[i].v1==list[j].v2
         || list[i].v2==list[j].v1 || list[i].v2==list[j].v2
         ) b[i][j]=b[j][i]=1;
}

int fillc(int *x, int z, int num)
{ int i;
  for (i=1;i<=num;i++) x[i]=z;
  return 0;
}

int print()
{ int i;
printf("\n");
```

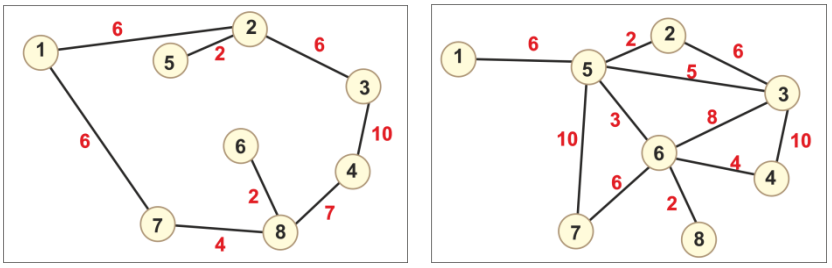
```
   for (i=1;i<=m;i++)
   if (s[i]==1) printf("(%d %d)", list[i].v1, list[i].v2);
  printf("\n");
}

int mind(int *s, int *q)
{ int i,j,k=0,r, rest[20];
   for (i=1;i<=m;i++) if(q[i]!=0) k=1;
   if (k==0) {print();}
   else { j=m;
          while (s[j]==0 && j>=1) j--;
          r=j+1;
          for (i=r;i<=m;i++)
           if (q[i]==1){ fillc(rest,0,m);
                         s[i]=1; q[i]=0;
                         for (j=1;j<=m;j++)
                            if (b[i][j] != 0 && q[j]==1)
                              {q[j]=0; rest[j]=1;}
                         mind (s,q);
                         s[i]=0;q[i]=1;
                         for (j=1;j<=m;j++)
                           if(rest[j]==1) q[j]=1;
                        }
        }
    return 0;
   }

int main()
{
f=fopen("data.in", "r");
fscanf(f, "%d", &n);
 for (i=1;i<=n;i++)
       for (j=1;j<=n;j++)
              fscanf(f,     "%d",
&a[i][j]);
 fclose(f);
 asociat();
 fillc(s,0,m);
 fillc(q,1,m);
 mind(s,q);
return 0;
}
```



Rezultate:

Pentru graful din desenul 12.2 (a) se obțin cuplajele:

Exerciţii



A B

Des. 12.3

1. Determinați cuplajele maxime de putere maximală în grafurile de pe desenul 12.3 A, 12.3 B.

2. Determinați cuplajele maxime de putere minimală în grafurile de pe desenul 12.3 A, 12.3 B

3. Elaborați un program pentru generarea tuturor cuplajelor fără construcția grafului asociat. $G = (V, E), |V| \leq 20$

4. Elaborați un program pentru determinarea cuplajului cu un număr minim de muchii. $G = (V, E), |V| \leq 20$

5. Elaborați un program pentru determinarea cuplajului cu un număr maxim de muchii. $G = (V, E), |V| \leq 20$

Bibliografie

1. Sedgewick Th, *Algorithms in C*, 2001, Addison Wesley

2. Gibbons Alan, *Algorithmic ghaph theory*, 1999, Addison Wesley

3. Липский В., *Комбинаторика для программистов*, 1988, Мир, Москва

4. Новиков Ф.А., *Дискретная математика для программистов*, 2001, Питер, Санкт Петербург

5. Майника Э., *Алгоритмы оптимизации на сетях и графах*,1981, Мир,Москва

6. Кристофидес П., *Теория графов. Алгоритмический подход*, 1978, Мир, Москва

7. Cormen Th., Leiserson Ch., Rivest R., *Introducere în algoritmi.* Agora, Cluj, 2001.

8. Cristian A.Giumale, *Introducere în analiza algoritmilor. Teorie şi aplicaţie*. Polirom, Iaşi, 2004

9. Pătruţ Bogdan. *Programarea calculatoarelor*. Teora, Bucureşti, 1998.

10. Cerchez Em., Şerban M. *Programarea în limbajul C/C++ pentru liceu. Vol III. Teoria Grafurilor*. Polirom, Iaşi, 2006.

11. Пападимитриу Х., Стайглц К., *Комбинаторная оптимизация. Алгоритмы и сложность*. Москва, Мир, 1985

Abrevieri şi notaţii

$\vee$	- disjuncția (operația logică SAU [OR])
$\wedge$	- conjuncția (operația logică ŞI [AND])
$\neg$	- negația (operația logică NU [NOT])
$A \Rightarrow B$	- din A rezultă B.
$A \Leftrightarrow B$	- A este echivalent cu B
$x \in X, x \notin X$	- x aparține X (x nu aparține X)
$\{x \in X : Q\}$	- submulțimea elementelor x din X, care satisfac condiția Q
$A \subseteq B$	- A se conține în B (A este submulțime a mulțimii B)
$\wp(X)$	- mulțimea tuturor submulțimilor mulțimii X
$\lvert X \rvert$	- cardinalul mulțimii X
$a_1, ..., a_n$	- secvență din n elemente
(a, b)	- pereche ordonată
$A \times B$	- produs cartezian al mulțimilor A şi B; mulțimea tuturor perechilor posibile $(a,b) : a \in A, b \in B$
$a \leftarrow b$	- valoarea elementului b este atribuită elementului a .
$i \uparrow$	- valoarea elementului i este incrementată cu 1.
$i \downarrow$	- valoarea elementului i este decrementată cu 1.
$a \rightleftarrows b$	- valorile elementelor a şi b sunt interschimbate $(a \leftarrow b) \wedge (b \leftarrow a)$