- брой и номериране на обхващащи дървета в граф

Намирането и номериране на комбинаторни обекти (за каквито се считат и графите) е обширна тема и съответно са извести доста специфични алгоритми и математически теории. В тази точка ще разгледаме накратко задачата за отпечатване на всички обхващащи дървета в граф. Едно от приложенията на тази задача е при анализ на електрически схеми, представени като граф.

***Теорема*.** Даден граф *G*(*V*, *E*) *с матрица на инцидентност* *A*. Нека изтрием последния ред и последния стълб на *А* и новополучената матрица да означим отново с *A*. Нека *А*'е резултата от умножението на *A* с нейната транспонирана матрица. Тогава, броят на различните обхващащи дървета в *G* е детерминантата на *А*'.

Тъй-като са известни ефективни алгоритми за:

* намиране на транспонирана матрица на дадената матрица
* умножение на матрици
* изчисляване на детерминанта на матрица

то с помощта на показаната теорема можем с обща сложност *O*(*n*3) да намерим броя на обхващащите дървета в граф.

Доста дълъг период от време (до 1992 г.) единствените решения за определяне, номериране и отпечатване на обхващащите дървета на граф са се базирали на пълно изчерпване. От скоро са известни и алгоритми, които решават задачата по такъв начин, че всяко покриващо дърво се получава от предишното само с размяната на едно единствено ребро [*reference - алгоритми на Kapoor, Ramesh-1992; Matsui-1993; Shioura, Tamura-1993*]. При тях изходът е “компресиран”: вместо всички обхващащи дървета, се извежда само първото, а след това се отпечатва единствено кое ребро с кое разменяме. Сложността на подобен алгоритъм е *O*(*r+n+m*), като *r* е броя на обхващащите дървета, а паметта която се използва е *O*(*m+n*) [*Shioura-Uno*].

За решението на задачата ще използваме пълно изчерпване, което е модификация на алгоритъма на Крускал. На всяка от основните стъпки на алгоритъм на Крускал вместо да избираме минималното ребро (*i*, *j*) ще продължаваме да строим покриващо дърво *T* с всички възможни ребра (*i*, *j*), такива че *i* и *j* принадлежат на различни множества.

Трябва да се внимава при търсенето да не се повтарят обхващащи дървета. Например, ако на първите две стъпки изберем ребрата (*1*, *2*) и (*2*, *3*) и по-нататък намерим някакви обхващащи дървета *T*1, ..., *Tp*, същите ще намерим и ако на първата стъпка изберем (*2*, *3*), а на втората (*1*, *2*).

Следва изходният код на Си. Графът е представен като списък от ребрата S. Във модифицирана функция kruskal освен броя на добавените до момента ребра count (count == N-1 е дъното на рекурсията) има още един параметър — lastarc, който показва кое е последното взето от списъка ребро. Така следващото избрано ребро ще има задължително по-голям номер в списъка с ребра, което ще реши проблема с отпечатване на покриващо дърво повече от един път. Примерните входни данни, използвани в програмата, описват следният граф:



**Фиг.5.7.а.**

*allst.cpp*

#include <stdio.h>

#include <stdlib.h>

const N=5; // брой на върховете на графа

const M=6; // брой на ребрата на графа

typedef struct {int i,j;} arc;

arc S[M]={ // списък от ребрата на графа

{1, 2},

{2, 3},

{2, 4},

{2, 5},

{3, 4},

{4, 5}};

int prev[N+1];

arc tree[N];

int getRoot(int i){

int root, savei;

root=i;

while (prev[root]) {

root=prev[root];

}

// свиване на пътя

while (i!=root) {

savei = i;

i = prev[i];

prev[savei] = root;

}

return root;

}

void kruskal(int count, int lastarc) {

if (count == N-1) {

printf("Едно покриващо дърво е: \n");

for (int i=0; i<N-1; i++)

printf("(%d,%d) ",tree[i].i,tree[i].j);

printf("\n");

return;

}

int saveprev[N+1];

for (int k=0; k<=N; k++) saveprev[k] = prev[k]; // запазва старите множества

for (int i=lastarc; i<M; i++) {

int r1 = getRoot(S[i].i);

int r2 = getRoot(S[i].j);

if (r1!=r2) {

tree[count].i = S[i].i;

tree[count].j = S[i].j;

prev[r2] = r1;

kruskal(count+1, i+1);

}

for (int k=0; k<=N; k++) prev[k] = saveprev[k]; // възстановява множествата

}

}

void main() {

for (int i=0; i<=N; i++) prev[i]=0;

kruskal(0, 0);

}

За упражнение остава следната

*Задача*: Възможно ли е да се модифицира алгоритъма на Прим, така че:

* да отпечатва всички обхващащи дървета на граф
* да не отпечатва едно покриващо дърво повече от един път

*Упътване*: Трудността при адаптация на алгоритъма на Прим се състои в това, че ако въведем наредба на ребрата ще изпуснем част от решенията, от друга страна, ако на всяка стъпка избираме произволно ребро — ще намерим някои решения повече от един път.

5.7.2. Независими подмножества в граф

Ще припомним две дефиниции от началото на главата: Граф *G*(*V*, *E*) се нарича *пълен*, ако съществува ребро (*i*, *j*), за всяко *i*, *j*∈*V*. *Кликово число* се нарича максималния пълен подграф *G*'(*V*', *E*') на *G*. Обратното понятие на пълен граф е граф, несъдържащ нито едно ребро и той се нарича *празен*. Ще разгледаме две задачи и ще дефинираме и понятие, обратно на кликово число.

*Задача 1*: Дадена група от *n* човека, в която двама души или са приятели, или — не. Да се изберат максимален брой хора от дадените, така че всички в избраната група да бъдат приятели.

*Задача 2*: Дадена група от *n* човека, в която двама души или са врагове, или — не. Да се изберат максимален брой хора от дадените, така че никои двама души от избраната група да не бъдат врагове.

В настоящата точка ще решим втората задача. Първата може да се реши с помощта на втората (оставяме като упражнение на читателя да съобрази как, след като прочете материала по-нататък).

***Дефиниция***. Даден е неориентиран граф *G*(*V*, *E*). Множество *H*⊆*V* се нарича *независимо множество от върхове* (или само *независимо множество*), ако в *H* не участват съседни върхове.

***Дефиниция***. Броят на елементите на независимото множество, състоящо се от най-много върхове се нарича *число на независимост* на *G*.

Изказано с други думи, числото на независимост на неориентиран граф е броят върхове на максималния му празен подграф.

Примери за независими множества от върхове са показани на *фиг*.5.*7*.*б*.



**Фиг.5.7.б.** За даденият неориентиран граф независими множества са {*7*, *8*, *2*},{*3*, *1*},{*7*, *8*, *2*, *5*}.Числото на независимост на графа е 4.

- максимални независими множества в граф

***Дефиниция***. Независимо множество *H* в неориентиран граф се нарича *максимално*, ако не съществува друго независимо множество *H*′, което да съдържа *H*.

За примера от *фиг*.*5.7*.*2*, максимално независимо множество е *H*1*=*{*7*, *8*, *2*, *5*}, докато *H*2*=*{*7*, *8*, *2*} не е, тъй като се съдържа във *H*1. Независимите множества {*1*, *3*, *7*} и {*4*, *6*} са също максимални.

*Алгоритъм за намиране на всички максимални независими множества*

Нека разглеждаме граф *G*(*V*, *E*). Първо ще покажем как се намира *едно* максимално независимо множество *Т*:

1. Нека *S* и *Т* в началото са празни множества.
2. Взимаме *произволен* връх от графа *i*∈*V*, непринадлежащ на *S* и *T*, т.е. *i*∉*S*, *i*∉*T*.
3. Добавяме *i* в *T*. В *S* добавяме всички върхове инцидентни с *i*.
4. Повтаряме стъпка 2 и 3, докато достигнем положение, при което всеки връх от графа се намира в някое от множествата *S* или *T*, т.е. *S*∪*T=V*. Тогава, *T* е максимално независимо множество.

За да намерим всички независими множества, ще използваме рекурсия. На стъпка 2) от алгоритъма няма да избираме произволен връх, а ще изследваме какво ще се случи при добавяне на всеки възможен връх от графа (т.е. всеки връх, непринадлежащ на *S* и *T*). Това ще осъществим чрез рекурсивната функция maxSubSet(int last):

maxSubSet(int last) {

if (SuT = V) { Отпечатва се едно максимално независимо множество; return; }

for (всеки връх i | i=/=S, i=/=T, i>last) {

Добавяме\_i\_в\_T;

Добавяме\_съседните\_на\_i\_върхове\_в\_S;

maxSubSet(i);

Изключваме\_i\_от\_T;

Възстановяваме\_S\_от\_преди\_рекурсивното\_извикване;

}

}

В показания фрагмент last е последният добавен връх в максималното независимото множество, което строим. Избираме само тези върхове *i*, за които *i*>last. Това се налага, за да се избегне конструирането на еднакви независими множества (от вида {1, 3, 7} и {1, 7, 3} и т.н.).

Следва изходният код на програмата. Графът е представен с матрица на съседство и входните данни са зададени като константи:

*maxnezav.cpp*

#include <stdio.h>

#include <math.h>

const N=8;

int A[N][N]= {

{0,1,0,0,0,1,0,1},

{1,0,1,0,0,1,0,0},

{0,1,0,1,1,0,0,0},

{0,0,1,0,1,0,1,0},

{0,0,1,1,0,1,0,0},

{1,1,0,0,1,0,1,1},

{0,0,0,1,0,1,0,0},

{1,0,0,0,0,1,0,0}};

int S[N], T[N], sN, tN;

void showSol() {

printf("{ ");

for (int i=0; i<N; i++)

if (T[i]) printf("%d ",i+1);

printf("} \n");

}

void maxSubSet(int last){

int saveS[N], savesN; /\* запазва съдържанието на S преди рекурсията и след

връщането го възстановява \*/

if (sN+tN==N) { showSol(); return; } // SuT=V -> отпечатва се решението

for (int i=last; i<N; i++)

if (!S[i] && !T[i]) {

savesN = sN;

for (int j=0; j<N; j++) {

saveS[j]=S[j];

if (A[i][j] && !S[j]) { S[j]=1; sN++; }

}

T[i]=1; tN++;

maxSubSet(i); // рекурсия

T[i]=0; tN--;

for (j=0; j<N; j++) S[j]=saveS[j];

sN = savesN;

}

}

void main(){

printf("Ето всички максимални независими множества в графа:\n");

for (int i=0; i<N; i++) { S[i]=0; T[i]=0; }

sN=0; tN=0;

maxSubSet(0);

}