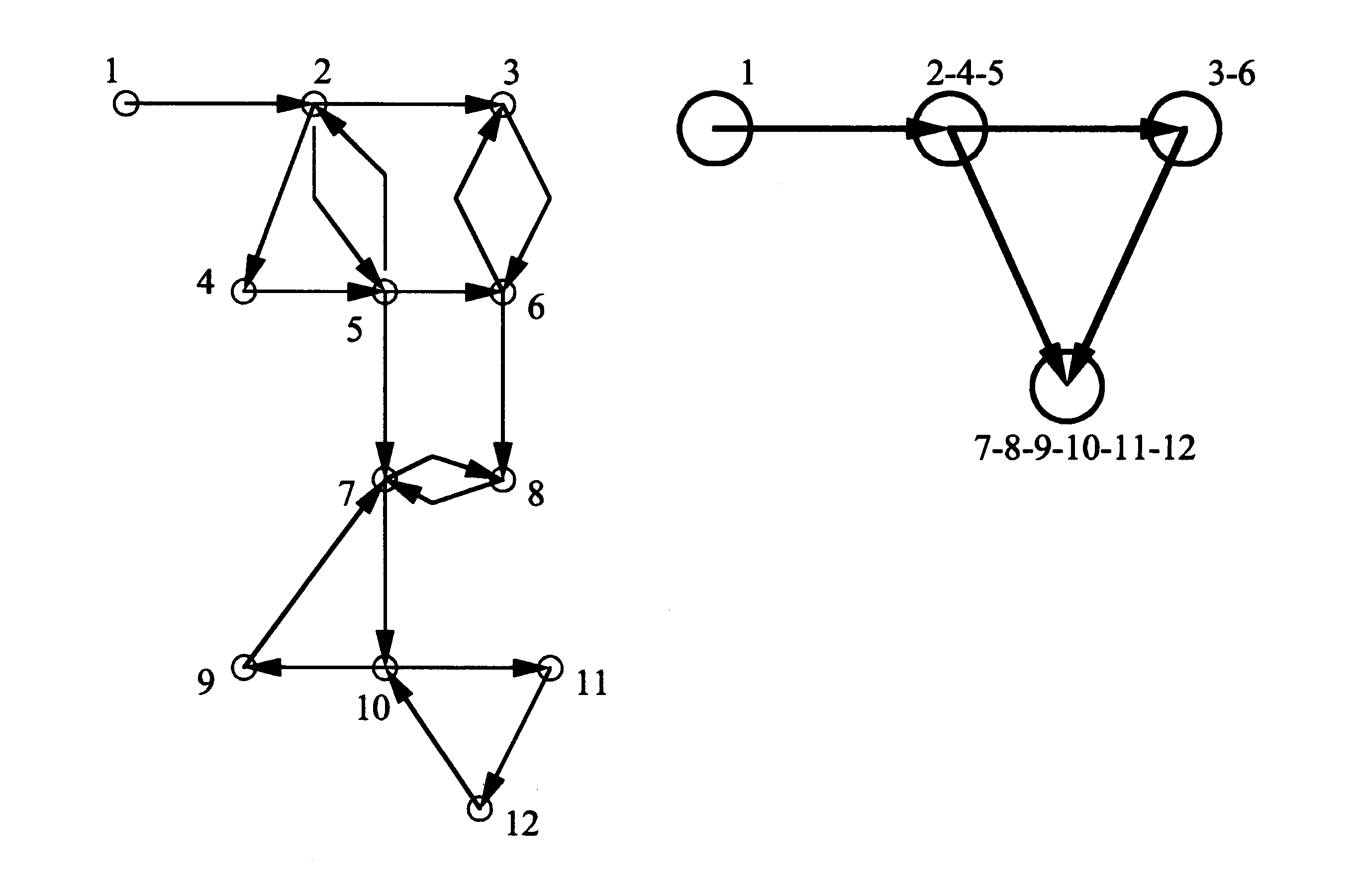
**ГРАФИ. ДВУСВЪРЗАНОСТ. СИЛНА СВЪРЗАНОСТ. ОЙЛЕРОВИ И ХАМИЛТОНОВИ ЦИКЛИ. МИНИМАЛНИ ПОКРИВАЩИ ДЪРВЕТА. МЕТОД НА КРИТИЧНИЯ ПЪТ.**

Програмиране +=Алгоритми, Увод в дискретната математика – Красимир Манев, Руско Шиковр judge.openfmi.net, referati.com, <http://www.cyberforum.ru/pascal/thread542095.html>, и др.

**1. СИЛНО СВЪРЗАНИ КОМПОНЕНТИ В ОРИЕНТИРАН ГРАФ**

Ориентираният граф G(V,E) се нарича силно свързан, ако от всеки връх има път до всички останали върхове в графа. Ако графът не е силно свързан, се интересуваме от всички компоненти на силна свързаност. Компоненти на силна свързаност наричаме всички максимални силно свързани подграфи на G, т.е. множеството от всички върхове, свързани с даден връх u е една компонента на силна свързаност.

Всеки граф може да се разбие на компоненти на силна свързаност. Ако графът е силно свързан, той има единствена компонента на силна свързаност, която съдържа всичките му върхове.

На схемата вдясно е даден ориентиран граф и неговите силно свързани компоненти.

Ще разгледаме два алгоритъма, за намиране на компонети на силна свързаност. Те се базират на обхождане в дълбочина и на факта, че ако променим посоките на всички ребра в графа G, то компонентите на свързаност остават същите в новият транспониран граф GT.

**Алгоритъм 1**

1) Избираме произволен връх u, изпълняваме dfs(u) и намираме множеството R от всички върхове, достижими от u. Тук освен върховете на силно свързаната компонента, ще са и върховете, до които има път от u в граф G, но няма обратно към u.

2) Създаваме обърнат граф G’(V,E’), в който посоките на ребрата са обърнати – на всяко ребро в граф G сме съпоставили ребро в обратната посока в граф G’.

3) Изпълняваме dfs(u) и във втория граф и намираме множеството Q от всички върхове, достижими от u в граф G’. Тук ще са върховете от същата силно свързана компонента и такива до които има път от u в графа G’, но не и обратно.

4) Сечението на R и Q дава търсената силно свързана компонента.

5) Изключваме тази компонента и ако има още непосетени върхове, повтаряме стъпка 1.

Сложността на този алгоритъм е О(n.(n+m)).

**Алгоритъм 2**

Ще модифицираме горния алгоритъм, като в dfs добавим номерация на върховете – след рекурсивното извикване. Всеки връх ще получи номер, който трябва да бъде по-голям от този на наследниците му при обхождането му. Тази стратегия за номерирана на английски се нарича postnum и се реализира с добавянето на реда postnum[i] = count++; където count е броячът на номерацията (брой обходени върхове до момента).

До тази стратегия достигаме след следните разсъжения:

Ако търсенето в дълбочина започва от връх *u*, така ще посетим всички всички върхове, както от силно свързаната компонента на *u*, така и от всички други силно свързани компоненти, които са достижими от връх *u*. Ако върхът *u* няма достъп до други компоненти, то процедурата ще завърши с посещаването на върховете само от една силно свързана компонента.

Върхът с най-голяма стойност в postnum принадлежи на такава силно свързана компонента, която не е достъпна от други силно свързани компоненти.

Ето и самия алгоритъм:

1) Изпълняваме dfs в граф G и запазваме реда на обхождане – по този начин извършваме номерацията на върховете.

2) Създаваме обърнат граф G’(V,E’), както в предходния алгоритъм

3) Обхождаме в дълбочина графа G’, като започваме търсенето от върха с най-голяма стойност. Всички достигнати върхове при обхождането принадлежат на една силно свързана компонента. Ако още има непосетени върхове, значи графът не е бил силно свързан и продължаваме обхождането от друг връх с максимална стойност в postnum.

Сложността на този алгоритъм е О(n+m).

Ще направим програмна реализация на този алгоритъм.

**Програмна реализация /Наков, преработена/**

Тъй като използването на втора матрица при този подход е неефективно, се използва втора функция за обхождане в дълбочина (backDFS), която гледа на ребрата на графа G в „обърнат” вид.

#include <stdio.h>

#define MAXN 150/\* Максимален брой върхове в графа \*/

const unsigned n = 10; /\* Брой върхове в графа \*/

/\* Матрица на съседство на графа \*/

const char A[MAXN][MAXN] = {

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

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

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

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

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

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

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

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

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

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

};

bool used[MAXN];

unsigned postnum[MAXN], count = 0;

void DFS(unsigned i)/\* Обхождане в дълбочина със запазване на номерацията \*/

{ unsigned j;

used[i] = 1;

for (j = 0; j < n; j++)

if (!used[j] && A[i][j]) DFS(j);

postnum[i] = count++;

}

void backDFS(unsigned i)/\* Обхождане в дълбочина на графа G’ \*/

{ unsigned j;

printf("%u ", i + 1);

count++; used[i] = 1;

for (j = 0; j < n; j++)

if (!used[j] && A[j][i]) backDFS(j);

}

void strongComponents(void) /\*Намира силно свързаните компоненти на графа\*/

{ unsigned i;

for (i = 0; i < n; i++) used[i] = 0;

while (count < n - 1) {

for (i = 0; i < n; i++)

if (!used[i]) DFS(i);

}

for (i = 0; i < n; i++) used[i] = 0;

count = 0;

while (count < n - 1)

{ unsigned max = 0, maxv = 0;

for (i = 0; i < n; i++)

if (!used[i] && postnum[i] > max) {

max = postnum[i];

maxv = i;

}

printf("{ ");

backDFS(maxv);

printf("}\n");

}

}

int main(void) {

printf("Silno svarzani komponenti w grafa sa:\n");

strongComponents();

return 0;

}

**ИЗХОД**:

1 3 2 6 5 4

7 10 9 8

**Програмна реализация /http://judge.openfmi.net, допълнена/**

Представяме двата графа чрез vector-и. При това обхождане добавяме посетения връх във вектор с компонентите. След като изведем, изчистваме елементите във този вектор, за да е свободен за новата компонента.

#include <iostream>

#include <vector>

using namespace std;

vector <vector <int> > g, gr;

vector <bool> used;

vector <int> order, component;

int n, m;

void dfs1 (int v)

{ used[v] = true;

for (int i = 0; i < g[v].size(); i++)

if (!used[ g[v][i] ])

dfs1 (g[v][i]);

order.push\_back(v);

}

void dfs2 (int v)

{ used[v] = true;

component.push\_back(v);

for (int i = 0; i < gr[v].size(); i++)

if (!used[ gr[v][i] ])

dfs2(gr[v][i]);

}

int main()

{ cin >> n>>m;

g.resize(n); gr.resize(n);

int x, y;

for (int i=1; i<=m;i++)

{ cin >> x >> y;

x--; y--;

g[x].push\_back(y);

gr[y].push\_back(x);

}

used = vector < bool > (n, false);

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

if (!used[i]) dfs1(i);

used = vector <bool> (n, false);

for (int i = n - 1; i >= 0; i--)

if (!used[order[i]])

{ dfs2(order[i]);

for (int i = 0; i < component.size(); i++)

cout << component [i] + 1<<" "; //т.к. x--; y--;

cout << endl;

component.clear();

}

return 0;

}

**Тест 1: ВХОД: /графа от схемата в началото/** 12 19

1 2 2 3 2 4 2 5 3 6 4 5 5 2 5 6 5 7 6 3 6 8 7 8 7 10 8 7 9 7 10 9 10 11 11 12 12 10

**ИЗХОД**:

1

2 5 4

3 6

8 7 9 10 12 11

**Тест 2: ВХОД: /графа от реализацията на Наков/** 10 12

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

**ИЗХОД**:

1 3 2 6 5 4

7 10 9 8

**2. РАЗДЕЛЯЩИ ТОЧКИ В НЕОРИЕНТИРАН ГРАФ. ДВУСВЪРЗАНОСТ.**

Даден е неориентиран граф. Разделяща точка в графа се нарича връх, след чието премахване (както и на всички инцидентни с него ребра), графът престава да бъде свързан.

Силно свързан граф се нарича двусвързан, ако няма разделяща точка. Това означава още, че той остава свързан и след премахването на произволен единствен негов връх.

За свързания граф на фигурата вдясно разделящи върхове са 2, 4 и 6. Ако премахнем връх 6, то графът се разделя на две свързани компоненти: {1, 2, 3, 4, 5} и {7}.

Ще предложим два алгоритъма за намиране на разделящи точки в неориентиран граф.

**Алгоритъм 1**

Обхождаме графа и за всеки връх проверяваме дали след неговото премахване, графът остава свързан. Използваме алгоритъма за проверка дали даден неориентиран граф е свързан. Ако графът престава да бъде свързан, то това е разделяща точка.

Този алгоритъм е лесен за реализация, но не е най-ефективния. Сложността му е О(n.(n+m)).

Тъй като се използва алгоритъма за проверка дали даден неориентиран граф е свързан, тук ще дадем реализацията му.

**Програмна реализация /Наков – strcon1.c/:**

#include <stdio.h>

#define MAXN 200/\* Максимален брой върхове в графа \*/

const unsigned n = 6;/\* Брой върхове в графа \*/

const char A[MAXN][MAXN] = /\* Матрица на съседство на графа \*/

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

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

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

};

char used[MAXN];

void DFS(unsigned i)/\* модифицирано DFS \*/

{ unsigned k;

used[i] = 1;

printf("%u ", i + 1);

for (k = 0; k < n; k++)

if (A[i][k] && !used[k]) DFS(k);

}

int main(void)

{ unsigned i, comp;

for (i = 0; i < n; i++) used[i] = 0;/\* инициализация \*/

printf("\nVsichki komponenti na swyrzanost: \n");

comp = 0;

for (i = 0; i < n; i++)

if (!used[i]) {

comp++;

printf("{ ");

DFS(i);

printf("}\n");

}

if (1 == comp)

printf("Grafyt e swyrzan.\n");

else

printf("Broj swyrzani komponenti: %d \n", comp);

return 0;

}

В сила са следните твърдения:

**Твърдение 1:** Върхът k е разделяща точка, тогава и само тогава, когато съществуват други два върха i и j в графа, такива, че всеки път между тях минава през k.

**Твърдение 2:** Нека G(V, E) е свързан неориентиран граф, а T(V, D) е покриващо дърво на G, построено при обхождане в дълбочина. Върхът k ще бъде разделяща точка, тогава и само тогава, когато съществуват върхове i, j ∈V такива, че да са изпълнени едновременно условията:

1/ (k, i) ∈ D

2/ j ≠ k

3/ j не е наследник на i в дървото

4/ lowest[i] ≥ prenum[k], където prenum[k] е номерът, който е получил върхът k при обхождане, като номерацията е преди рекурсивното извикване, а lowest[i] е минималната стойност от трите: prenum[i], prenum[w], /за всеки връх w за който (i, w)∈E и (i, w)∉D/ и lowest[w] /за всеки наследник.

Следващият алгоритъм е построен на базата на второто твърдение:

**Алгоритъм 2**

1. Обхождаме в дълбочина с начало произволен връх, получаваме покриващо дърво Т и пресмятаме номера prenum[i] на всеки връх i.
2. Обхождаме дървото в ЛДК и за всеки връх i изчисляваме lowest[i].
3. Определяме разделящите точки по следния начин:
   1. Коренът на дървото е разделяща точка ⇔ има повече от един наследник
   2. Връх i, различен от корена е разделяща точка ⇔ има пряк наследник х, за който е изпълнено lowest[x] ≥ prenum[i]

Сложността на този алгоритъм при представяне със списък на съседи е Θ(n+m), а при представяне с матрица на съседство е Θ(n2).

**Програмна реализация /Наков – artic.c /:**

#include <stdio.h>

#define MAXN 150/\* Максимален брой върхове в графа \*/

const unsigned n = 7;/\* Брой върхове в графа \*/

char A[MAXN][MAXN] = {/\* Матрица на съседство на графа\*/

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

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

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

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

unsigned prenum[MAXN], lowest[MAXN], cN;

unsigned min(unsigned a, unsigned b) { return (a < b) ? a : b; }

void DFS(unsigned i)

{ unsigned j;

prenum[i] = ++cN;

for (j = 0; j < n; j++)

if (A[i][j] && !prenum[j]) {

A[i][j] = 2; /\* строим покриващо дърво T \*/

DFS(j);

}

}

void postOrder(unsigned i)/\* Обхождане на дървото в ЛДК\*/

{ unsigned j;

for (j = 0; j < n; j++)

if (2 == A[i][j]) postOrder(j);

lowest[i] = prenum[i];

for (j = 0; j < n; j++)

if (1 == A[i][j]) lowest[i] = min(lowest[i], prenum[j]);

for (j = 0; j < n; j++)

if (2 == A[i][j]) lowest[i] = min(lowest[i], lowest[j]);

}

void findArticPoints()

{ unsigned artPoints[MAXN], i, j, count;

for (i = 0; i < n; i++) {

prenum[i] = 0; lowest[i] = 0; artPoints[i] = 0;

}

cN = 0;

DFS(0);

for (i = 0; i < n; i++)

if (0 == prenum[i]) {

printf("Графът не е свързан - \n");

return;

}

postOrder(0);

count = 0; /\* проверяваме 3.1) \*/

for (i = 0; i < n; i++)

if (2 == A[0][i]) count++;

if (count > 1) artPoints[0] = 1;

for (i = 1; i < n; i++) {/\* прилагаме стъпка 2) \*/

for (j = 0; j < n; j++)

if (2 == A[i][j] && lowest[j] >= prenum[i]) break;

if (j < n) artPoints[i] = 1;

}

printf("Razdelqshti tochki sa:\n");

for (i = 0; i < n; i++)

if (artPoints[i]) printf("%u ", i + 1);

printf("\n");

}

int main(void) {

findArticPoints();

return 0;

}

**3. ХАМИЛТОНОВ ЦИКЪЛ. ЗАДАЧА ЗА ТЪРГОВСКИЯ ПЪТНИК**

Хамилтонов цикъл в граф се нарича цикъл, който съдържа всички върхове от графа еднократно. Графът, съдържащ такъв цикъл, се нарича хамилтонов граф. Той може да бъде претеглен или непретеглен. В него може да се построи хамилтонов цикъл, като се започне от произволен връх.

Този вид графи са наречени на името на ирландския математик Уилям Хамилтон, който през 1859 година предлага занимателната задача „Околосветско пътешествие”, в която се изисква да се обходят всички върхове на додекаедър по такъв начин, че през всеки връх да се мине точно един път.

Съществуват две класически задачи, свързани с хамилтонови графи:

* проверката дали даден граф е хамилтонов;
* задача за търговския пътник /Traveling salesman problem/: Търговски пътник трябва да посети всеки от дадени n града по един път, тръгвайки от един начален град и връщайки се в него. Ако са известни разстоянията между градовете, да се намери най-краткия път).

И двете задачи се определят като NP-пълни, което означава, че сложността им в най-лошия случай е експоненциална. Първата задача може да бъде решена чрез проверка на циклите в графа (пълно изчерпване на вариантите).

Ще се спрем на втората задача, която ще формулираме като задача за намиране на хамилтонов цикъл с най-малка дължина в претеглен хамилтонов граф. В зависимост от смисъла на теглата, съпоставени на ребрата и формулировката, можем да търсим път с минимална дължина, с минимален разход на гориво, с най-ниска цена, или проверка за съществуване на път, започващ от даден град и завършващ в него, или съществуване на път с дължина ≤ L, където L е предварително зададено положително число и др.

Ще решим задачата чрез пълно изчерпване, използвайки подхода за намиране на всички пътища между два зададени върха в граф, като допълнително ще изискваме графът да е претеглен и началният и крайният връх на търсените пътища да е един и същ. За всеки намерен нов хамилтонов цикъл ше се изчислява дължината и ще се сравнява с дължината на най-краткия от тях до момента.

**Алгоритъм:**

- избираме произволен връх за начало и го обявяваме за посетен

- обхождаме върховете и търсим непосетен наследник, ако намерим такъв, продължаваме по него. На тази стъпка, ако графът е с неотрицателни ребра, можем да оптимизираме решението като прекъснем рекурсията, ако текущо генерираният път е по-голям от запомнения минимален до момента.

- ако не съществува непосетен наследник, проверяваме дали всички върхове са обходени:

- ако всички върхове са посетени и текущия съвпада с началния – намерен е нов Хамилтонов цикъл, проверяваме дали дължината е по-малка от текущата минимална.

- ако все още има непосетени върхове, но не можем да продължим нататък, прекъсваме рекурсията, обявяваме текущия връх за непосетен и се връщаме назад в неговия родител, за да потърсим други непосетени наследници.

Алгоритъмът е неприложим за големи стойности на n.

Друг подход за решаване на задачата е, да се генерират всичките възможни пермутации, за всяка от тях да се пресметне дължината и да се избере пермутацията със най-малката дължина.

**Програмна реализация** на Задачата за търговския пътник: Намиране на хамилтънов цикъл с минимална дължина в тегловен граф, предстаен чрез матрица на съседство. /Наков/

#include <stdio.h>

#define MAXN 150 /\* Максимален брой върхове в графа \*/

const unsigned n = 6;/\* Брой върхове в графа \*/

const int A[MAXN][MAXN] = {

{ 0, 5, 0, 0, 7, 7 },

{ 5, 0, 5, 0, 0, 0 },

{ 0, 5, 0, 6, 5, 0 },

7

3

6

5

5

7

5

5

5

3

{ 0, 0, 6, 0, 3, 3 },

{ 7, 0, 5, 3, 0, 5 },

{ 7, 0, 0, 3, 5, 0 }

};

const int MAX\_VALUE = 10000;

char used[MAXN];

unsigned minCycle[MAXN], cycle[MAXN];

int curSum, minSum;

void printCycle()

{ unsigned i;

printf("Minimalen hamiltonov cikyl: 1");

for (i = 0; i < n - 1; i++) printf(" %u", minCycle[i] + 1);

printf(" 1, dyljina %d\n", minSum);

}

void hamilton(unsigned i, unsigned level) /\* Намира минимален Хамилтонов цикъл \*/

{ unsigned k;

if ((0 == i) && (level > 0)) {

if (level == n) {

minSum = curSum;

for (k = 0; k < n; k++) minCycle[k] = cycle[k];

}

return;

}

if (used[i]) return;

used[i] = 1;

for (k = 0; k < n; k++)

if (A[i][k] && k != i)

{ cycle[level] = k;

curSum += A[i][k];

if (curSum < minSum) /\* прекъсване на генерирането \*/

hamilton(k, level + 1);

curSum -= A[i][k];

}

used[i] = 0;

}

int main()

{ unsigned k;

for (k = 0; k < n; k++) used[k] = 0;

minSum = MAX\_VALUE;

curSum = 0;

cycle[0] = 1;

hamilton(0, 0);

printCycle();

return 0;

}

**Зад. 1.**Да се провери дали граф е хамилтонов, т.е. да се намери един хамилтонов цикъл .

**Зад. 2.** Да се намери един хамилтонов път и да се определи дължината му.

**Зад. 2.** Да се намерят всички хамилтонови цикли и да се определят дължините им.

## 4. ОЙЛЕРОВ ЦИКЪЛ. ЗАДАЧА ЗА КИТАЙСКИЯ ПОЩАЛЬОН

Ойлер преподавал в университета в Калининград, наричан тогава Кьонигсберг. Градът е разположен на двата бряга и на два острова на река Прегел. Четирите части на града се свързвали помежду си чрез седем моста. Гражданите на този град много обичали разходките из града и докато се разхождали се запитали, дали е възможно човек да излезе от дома си, да мине по един път през всеки мост и да се прибере отново у дома си. Тази, на пръв поглед проста задача, се оказала такава главоблъсканица, че привлякла вниманието дори на именития Ойлер.

Ойлер представил четирите части от града с кръгчета, а мостовете - чрез линии и веднага заключил, че описаната разходка не е възможна, тъй като има върхове с нечетна степен. За да е възможна, всека част на града (A, B, C и D) трябва да е свързана с останалите с четен брой мостове. Това е така, защото, когато разхождащият се навлезе в терирорията на някоя част по един мост, трябва да има още един мост, по който да излезе от нея.

Ойлеров път е път, съдържащ всички ребра на графа точно по веднъж.

Ойлеров цикъл в граф се нарича цикъл, който започва от даден връх, минава точно по веднъж през всички негови ребра и се връща в началния връх. При това обхождане всеки връх може да бъде посетен многократно.

Ойлеров граф е свързан граф, съдържащ Ойлеров цикъл.

Теорема на Ойлер (L.Euler, 1736). Непразен свързан граф е съдържа Ойлеров цикъл тогава и само тогава, когато не съдържа върхове от нечетна степен.

Следствие: Ойлеров път в неориентиран граф съществува тогава и само тогава, когато графа има точно два върха с нечетни степени.

На следващата схема са дадени Ойлеров цикъл /(1,2), (2,3), (3,4), (4,2), (2,5), (5,3), (3,5), (5,6), (6,3), (3,7), (7,8), (8,1)/ и Ойлеров път.

Задача: За даден граф да се провери може ли да има Ойлеров граф /цикъл/.

Упътване: Бройм степените на върховете

Задача за китайския пощальон (Chinese postman's problem) – в претеглен граф да се намери цикъл, минаващ през всяко ребро поне по един път, за който сумата от теглата на ребрата да бъде минимална. Дължината на всяко ребро се включва в сумата толкова пъти, колкото реброто се среща в цикъла. /без решение/

Задача: Да се намери Ойлоров цикъл /път/.

**Алгоритъм за намиране на Ойлеров цикъл**

Започваме обхождане на графа от произволен връх i. Намираме инцидентно с него ребро (i, j) и го маркираме като посетено. Продължаваме с върха j: за него намираме непосетено ребро (j,k), маркираме го като посетено и преминаваме в k. Продължавайки по този начин, в даден момент ще се озовем в началния връх i и ще затворим цикъла. Ако всички ребра на графа вече са маркирани, то този цикъл е Ойлеров. Ако са останали непосетени ребра, то намираме връх x, принадлежащ на току-що намерения цикъл и инцидентен с поне едно непосетено ребро. От това, че всеки връх трябва да бъде от четна степен, следва, че x е инцидентен с четен брой (т.е. поне две) непосетени ребра . От х започваме да изграждаме цикъл по вече описания начин, докато отново се върнем в него. Получаваме втори цикъл, който обединяваме с първия (общата им точка ще бъде върхът х). Така, след краен брой стъпки, всички ребра ще се включат в един общ цикъл - търсеният Ойлеров цикъл.

Описаният алгоритъм може да се използва и за намиране на Ойлеров път, като в този случай можем да съединим върховете от нечетна степен с ребро (съществуват точно два такива върха) и да намерим Ойлеров цикъл по описания алгоритъм. След отстраняване на добавеното ребро получаваме търсения Ойлеров път. Пътя трябва да започва от единия връх с нечетна степен.

Алгоритъм за намиране на Ойлеров цикъл, описан с псевдокод  
 Нека графът е представен с матрица на съседство *А[][].* За реализацията ще се използват два стека: *stack[]* - за текущо построявания цикъл, и *cstack[]* - за обединението на всички построени до момента цикли. И двата стека първоначално ще бъдат празни.

Добавяме произволен връх i в stack;  
While (stack не е празен)  
 { Вземаме върха i от върха на stack без да го изключваме;  
 if (i има наследници)  
 { Вземаме произволен наследник j на i;  
 Слагаме j в stack;  
 A[j][i]=0; A[j][i]=0; //изключваме реброто от графа  
 }  
 else { Изключваме i от stack;  
 Включваме i в cstack;  
 }  
 }

**Програмна реализация /Наков – ориентиран граф, описан с матрица на съседство/**

#include <stdio.h>

#define MAXN 100/\* Максимален брой върхове в графа \*/

const unsigned n = 8;/\* Брой върхове в графа \*/

char A[MAXN][MAXN] = {

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

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

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

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

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

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

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

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

};

/\*Проверява дали е възможно да има Ойлеров цикъл (теорема на Ойлер)\*/

char isEulerGraph(void)

{ unsigned i, j;

for (i = 0; i < n; i++) {

int din = 0, dout = 0;

for (j = 0; j < n; j++) {

if (A[i][j]) din++;

if (A[j][i]) dout++;

}

if (din != dout) return 0;

}

return 1;

}

/\* Намира Ойлеров цикъл \*/

void findEuler(int i)

{ unsigned cStack[MAXN \* MAXN], stack[MAXN \* MAXN];

unsigned k, j, cTop = 0, sTop = 1;

stack[sTop] = i;

while (sTop > 0) {

i = stack[sTop];

for (j = 0; j < n; j++)

if (A[i][j]) {

A[i][j] = 0; i = j;

break;

}

if (j < n)

stack[++sTop] = i;

else

cStack[++cTop] = stack[sTop--];

}

printf("Euler's cicle is: ");

for (k = cTop; k > 0; k--) {

printf("%u ", cStack[k] + 1);

}

printf("\n");

}

int main(void)

{ if (isEulerGraph()) findEuler(0);

else printf("Not Euler's graph!");

return 0;

}

**Алгоритъм за намиране на Ойлеров път /Р. Шиков/**

Ще предложим алгоритъм за намиране на Ойлеров път в неориентиран граф, описан със статичен списък на съседство – двумерен масив. Идеята на алгоритъма е същата, разликата е в избора на начален връх и реализацията на вмъкването на подцикъла.

Ще пазим последователността от върхове в масив. Тъй като два съседни върха определят едно ребро, то масивът ще е с m+1 елемента, където m е броя на ребрата на графа.

Разделяме масива на 3 области – преден сегмент, среда – празна, заден сегмент и пазим индексите на края на първия и началото на третия сегмент. При обхождане попълваме първия сегмент, когато се затвори цикъл, последния елемент на първия сегмент съдържа елемента, от който трябва да продължим по друго необходено ребро.

Ако има такова ребро, допълваме в първия сегмент и продължаваме да търсим следващо необходено ребро.

Ако не е намерено необходено ребро, този връх се мести в трети сегмент. Третият сегмент се попълва отзад-напред, затова първият елемент, който няма необходени ребра става последен в масива, следващият става предпоследен и т.н.

За графа на предходната фигура започваме пътя от връх 2. Попълването на масива става по следния начин:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 6 | 8 | 5 | 6 2 | 1 | 3 | ~~4~~ 7 |  |  |  |  |  |  | 6 |

Ако търсим лексикографски най-големия /?малкия?/ път – като търсим ребро, инцидентно с i, тръгваме отзад-напред, за да го разгледаме първи и след това като няма вече непосетени съседи да го пратим отзад.

**Програмна реализация /неориентиран граф, описан със статични списъци на съседство/**

За реализацията на графа се използва двумерен масив, всеки ред съдържа съседите на съответния връх, първата колона съдържа броя на съседите на всеки връх.

#include <iostream>

using namespace std;

int g[101][101], p[101];

int m, n, a, b, br=0, i, j, U1, U2;

int main ()

{

cin >> n >> m;

for (i = 1; i <=m; i++)

{ cin >> a >> b;

g[a][0]++;

g[b][0]++;

g[a][g[a][0]]=b;

g[b][g[b][0]]=a;

}

U1 = 1; U2 = m+1;

for (i = 1; i <=n; i++)

if (g[i][0]%2==1)

{ br++;

if (br==1)p[U1]=i;//първият нечетен става начало на пътя

}

if (br==0)p[U1]=1; //ако всички са четни, избираме произволен връх за начало

if (br>2) {cout << "No way" << endl; return 0;}

while (U1!=U2)

{

if (g[p[U1]][0]== 0)

{ p[U2]=p[U1];

U2--;

U1--;

}

else

{ p[U1+1] = g [p[U1]][g[p[U1]][0]]; //последния съсед

g [p[U1]][g[p[U1]][0]] = 0;

g[p[U1]][0]--;

for (i = 1; i <=g[p[U1+1]][0]; i++)

if (g[p[U1+1]][i]==p[U1])

{

//for (j = i+1; j<=g[p[U1+1]][0]; j++) g[p[U1+1]][j-1] = g[p[U1+1]][j]; //преместване вляво

g[p[U1+1]][i]=g[p[U1+1]][g[p[U1+1]][0]];//размяна с последния

g [p[U1+1]][g[p[U1+1]][0]]=0;

g[p[U1+1]][0]--;

}

U1++;

}

}

for (i = 1; i <= m ; i++) cout << p[i]<<" ";

cout << p[m+1] << endl;

return 0;

}

/\*

8 12

1 2 1 3 2 5 2 6 3 4 3 7 3 8 4 5 5 6 5 8 6 8 7 8

\*/

**5. МИНИМАЛНИ ПОКРИВАЩИ ДЪРВЕТА** (MST – minimum-weight spanning tree)

**Определение**: Покриващо дърво за неориентирания граф G(V,E) наричаме всеки подграф T(V,E'), който е дърво. Ако графът е претеглен, тогава можем да определим и теглото на покриващото дърво, като сума от теглата на отделните ребра. Минимално е всяко покриващо дърво, което има най-малкото възможно тегло измежду всички покриващи дървета на G.

**Задача**: Да си представим група острови, които искаме да свържем с мостове по такъв начин, че да може да се стигне от всеки остров до всеки друг в групата. За всяка двойка острови знаем дали можем да се свърже с мост и колко ще струва той. Целта, която ще си поставим, ще бъде множеството от мостове, което построим да струва минимално.

3

2

13

1

1

1

13

4

3

12

16

14

1

Изказана с термините от теорията на графите, горната задача ще изглежда така:

Даден е неориентиран граф G(V, E), в който върховете от множеството V са островите, а ребро (i, j) съществува тогава, когато е възможно да се построи мост между два върха i и j. Теглото f(i, j) определя евентуалната цена на този мост. Търси се покриващо дърво T(V,D) на G с минимална сума от теглата на участващите в него ребра.

На фигурата горе с удебелени линии са посочени ребрата на минималното покриващо дърво на дадения граф.

Ще разгледаме два евристични алгоритъма, които решават дадената задача.

Алгоритъм на Крускал

Идея: Алгоритъмът на всяка стъпка избира ребро с минимално тегло и такова, че след прибавянето му към T да не се затвори цикъл (т.е. T да остане дърво).

1. Създаваме *n* множества, като във *i*-тото множество поставяме *i*-тия връх от графа.
2. Сортираме ребрата на графа във възходящ ред.
3. Създаваме празно дърво *T*(*V*, ∅). След приключване на алгоритъма *T* ще бъде търсеното покриващо дърво. Последователно (*n*-1 пъти) изпълняваме стъпка 4):
4. Добавяме ребро(*i*, *j*)∈*Е* към *T*, така че да бъде изпълнено:

* теглото *f*(*i*, *j*) да бъде възможно най-малко (разполагаме със сортиран по теглата списък на ребрата от графа)
* върховете *i*и*j* да се намират в различни множества

След всяко добавено ребро (*i*, *j*) обединяваме множествата, в които се намират *i* и *j*.

Сложността на алгоритъма е О(m.log(m) + n.log(n)), която се получава от сложността на сортирането O(m.log(m)) и сложността на проверката дали два върха принадлежат на едно и също множество чрез компоненти на свързаност и обединяването на множества – О(n.log(n)).

Обяснение на алгоритъма за дадения граф:

* първоначално деветте върха на графа се намират в девет различни множества: {1} {2} {3} {4} {5} {6} {7} {8} {9}
* сортираме ребрата и получаваме следния списък: (1, 2)=1; (5, 8)=1; (7, 6)=1; (6, 9)=1; (1, 4)=2; (2, 3)=3; (3, 6)=3; (3, 4)=4; (5, 6)=12; (5, 9)=13; (2, 5)=13; (4, 7)=14; (4, 6)=14;
* Избираме първото ребро (1, 2) и свързваме множествата в които се намират върховете с номера 1 и 2. Получаваме {1, 2} {3} {4} {5} {6} {7} {8} {9}.
* Избираме следващото по големина ребро (5, 8). Множествата са {1, 2} {3} {4} {5, 8} {6} {7} {9}.
* По-нататък към покриващото дърво последователно включваме (7, 6), (6, 9), (1, 4), (2, 3), (3, 6) и множествата стават {1, 2, 3, 4, 6, 7, 9}{5, 8}.
* Пропускаме реброто (3, 4), тъй като върховете 3 и 4 са вече в една и съща компонента.
* Вземаме следващото ребро (5, 6) и всички върхове на графа попадат са в една компонента. Минималното покриващо дърво е построено и то се състои от ребрата (1, 2), (1, 4), (2, 3), (3, 6), (6, 7), (6, 9), (6, 5), (5, 8).

**Програмна реализация /Наков, преработена/:**

#include <stdio.h>

#include <algorithm>

using namespace std;

#define MAXN 200/\* Максимален брой върхове в графа \*/

#define MAXM 2000/\* Максимален брой ребра в графа \*/

int n = 9, m = 14; /\* Брой върхове и брой ребра в графа \*/

struct arc { int i, j, f;};

bool comp(arc x, arc y) { return (x.f<y.f);}

arc S[MAXM] = { /\* Списък от ребрата на графа и техните тегла \*/

{ 1, 2, 1 }, { 1, 4, 2 }, { 2, 3, 3 }, { 2, 5, 13 }, { 3, 4, 4 },

{ 3, 6, 3 }, { 4, 6, 16 }, { 4, 7, 14 }, { 5, 6, 12 }, { 5, 8, 1 },

{ 5, 9, 13 }, { 6, 7, 1 }, { 6, 9, 1 } };

int prev[MAXN + 1]; //родител на връх i, родителят на корена е -1

int getRoot(int i)/\* намиране на корена на дървото \*/

{ int root = i, savei;

while (prev[root] != -1) root = prev[root];

while (i != root) /\* свиване на пътя \*/

{ savei = i;

i = prev[i];

prev[savei] = root;

}

return root;

}

void kruskal()

{ int MST = 0;

unsigned i, j;

sort(S, S+m, comp);/\* сортира списъка с ребрата в нарастващ ред\*/

printf("Rebrata v MST:\n");

for (i = 0; i < m; i++)

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

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

if (r1 != r2)

{ printf("(%u,%u) ", S[i].i, S[i].j);

MST += S[i].f;

prev[r2] = r1;

}

}

printf("\nCenata na MST e %d.\n", MST);

}

int main()

{ unsigned i;

for (i = 0; i < n + 1; i++) prev[i] = -1;

kruskal();

return 0;

}

Задача: Да се реализира алгоритъма на Крускал с използване на друга /?/ подходяща структура.

Алгоритъм на Прим

Идея: На всяка стъпка се търси минималното тегло не от всички не обходени клони, а само от тези клони, които са съседни на възли от намерената до момента част от дървото т.е. непрекъснато се поддържа свързаността на намереното дърво.

1. Започваме строенето на MST от произволен връх i: в началото дървото Т ще съдържа само върха i.
2. Повтаряме n–1 пъти:

2.1) Избираме реброто (*i*, *j*)∈*Е*, такова че:

* *f*(*i*, *j*) е минимално.
* върхът *i* се намира в множеството от върхове на *Т*, а върхът *j* — не.

2.2) Добавяме върха j и реброто (i, j) в дървото Т.

Сложността на алгоритъма на Прим е квадратична. При по-внимателно подбиране на структурите от данни (например, ако се използва пирамидална структура), сложността може да достигне до O(m + n.log(n)).

Обяснение на алгоритъм за дадения граф:

* избираме произволен начален връх, например 1. Минималното ребро, което го свързва с връх, неучастващ в Т, е (1, 2). Добавяме върха 2 към Т.
* следващото минимално ребро, което свързва връх от Т с връх непринадлежащ на Т е (1, 4).
* По нататък избраните ребра ще бъдат (2, 3), (3, 6), (6, 7), (6, 9), (6, 5) и (5, 8), с което минималното покриващо дърво е построено.

**Програмна реализация /Наков, преработена/:**

#include <stdio.h>

#define MAXN 150/\* Максимален брой върхове в графа \*/

int n = 9, MAX\_VALUE = 10000;/\* Брой върхове, макс. ст-ст на ребро\*/

int A[MAXN][MAXN] = /\* Матрица на теглата на графа \*/

{ { 0, 1, 0, 2, 0, 0, 0, 0, 0 },

{ 1, 0, 3, 0, 13, 0, 0, 0, 0 },

{ 0, 3, 0, 4, 0, 3, 0, 0, 0 },

{ 2, 0, 4, 0, 0, 16, 14, 0, 0 },

{ 0, 13, 0, 0, 0, 12, 0, 1, 13 },

{ 0, 0, 3, 16, 12, 0, 1, 0, 1 },

{ 0, 0, 0, 14, 0, 1, 0, 0, 0 },

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

{ 0, 0, 0, 0, 13, 1, 0, 0, 0 }

};

bool used[MAXN];

int prev[MAXN];

int T[MAXN]; //най-кратките разстояния до непринадлежащите на дървото върхове

void prim()

{ int MST = 0; /\* цената на минималното покриващо дърво \*/

int i, k;

for (i = 0; i < n; i++) { used[i] = 0; prev[i] = 0; }

used[0] = 1; /\* избираме произволен начален връх \*/

for (i = 0; i < n; i++)

T[i] = (A[0][i]) ? A[0][i] : MAX\_VALUE;

for (k = 0; k < n - 1; k++)

{ /\* търсене на следващо минимално ребро \*/

int minDist = MAX\_VALUE, j = -1;

for (i = 0; i < n; i++)

if (!used[i])

if (T[i] < minDist) {

minDist = T[i];

j = i;

}

used[j] = 1;

printf("(%d,%d) ", prev[j] + 1, j + 1);

MST += minDist;

for (i = 0; i < n; i++)

if (!used[i] && A[j][i])

{ if (T[i] > A[j][i])

{ T[i] = A[j][i];

prev[i] = j;/\* запазване на предшественика, за евентуално отпечатване на следващо минимално ребро \*/

}

}

}

printf("\nCenata na MST e %d.\n", MST);

}

int main()

{ prim();

return 0;

}

Програмна реализация – Прим /Бойко Банчев/

// Намиране на миним. скелетно дърво на претеглен граф-алгоритъм на Прим

// Вход:един ред с <брой-върхове><брой-ребра>, следван от съответния брой

// редове с по едно ребро (двойка номера на върхове и тегло) на всеки.

// Изход: ребрата на дървото – двойки номера на върхове, по една на ред.

// Забележка. Графът се представя като множество от ребра. Реализацията

// използва това,че множествата(set)в STL се изброяват в нарастващ ред

// на членовете им, от гледна точка на някаква подредба между тях– с това

// избирането на „най-леко“ ребро става удобно,без изрично подреждане или

// търсене

#include <iostream>

#include <vector>

#include <set>

using namespace std;

// ребро на граф: номера на върховете-краища и тегло

struct edge {int u,v; int wt;};

// сравняваща функция за подреждане на ребрата като елементи на множество

struct setcmp {

bool operator()(const edge a, const edge b) const {

return a.wt<b.wt;

}

};

// граф: множество от ребра, подреждани със setcmp от по-горе

typedef multiset<edge,setcmp> edges;

//намиране на минималното дърво:tree[i] е номерът на родителя на връх i

void findmintree(edges & es, vector<int> & tree) {

set<int> ns; // възли: множество от номера

int nv = tree.size();

// добавяме върхове и ребра докато върховете станат nv на брой

for (ns.insert(1),tree[0]=0; ns.size()<nv;)

//намираме първото ребро, чиито краища са включен и невключен в дървото върхове

for (edges::const\_iterator p=es.begin();; ++p) {

int a = p->u, b = p->v;

#define chosen(n) (ns.find(n)!=ns.end())

if (chosen(b)) swap(a,b);

if (chosen(a) && !chosen(b)) {

ns.insert(b);

tree[b-1] = a-1;

es.erase(p);

break;

}

}

}

int main() {

edges es;

edge e;

int nv, ne;

cin >> nv >> ne; // четене на брой върхове и ребра

vector<int> tree(nv);

while (ne--) { // четене на ребра и поставяне в графа

cin >> e.u >> e.v >> e.wt;

es.insert(e);

}

findmintree(es,tree); // намиране на дървото

for (int i=1; i<nv; ++i)

cout << 1+tree[i] << ' ' << 1+i << endl;

return 0;

}

Вход: 9 13

1 2 1

1 4 2

2 3 3

2 5 13

3 4 4

3 6 3

4 6 16

4 7 14

5 6 12

5 8 1

5 9 13

6 7 1

6 9 1

Изход: 1 2

2 3

1 4

6 5

3 6

6 7

5 8

6 9

Задача: Да се реализира алгоритъма на Прим с използване на структура опашка

Упътване: Нека множеството от ребра X е дърво, а множеството от върхове S съдържа върховете на X. Отначало X съдържа един произволен начален връх от графа. За намирането на най-лекото ребро между S и V-S алгоритъмът поддържа приоритетна опашка, в която се намират всички върхове от V-S, които са съседни на някой връх от S. Приоритета на даден връх, според който се построява пирамидата е теглото на най-лекото ребро от този връх до някой връх от S. Това много напомня на Алгоритъма на Дейкстра (където за ключове се използват дължини на пътища вместо тегла на ребра). Както и при Алгоритъма на Дейкстра всеки връх има предшественик prev[v], който е другия край на най-лекото ребро от v до връх от S. Псевдокодът на алгоритъма на Прим е почти идентичен с псевдокода на Алгоритъма на Дейкстра.

## 6. МРЕЖА НА ДЕЙНОСТИТЕ. МЕТОД НА КРИТИЧНИЯ ПЪТ

Едно важно приложение на графите е планирането изпълнението на проект. Някои добре известни техники в тази област са CPМ (Critical Path Мethod – метод на критичния път) и PERT (Perfonmance Evaluation and Review Technique – оценка на изпълнението и рецензия). Критичен път е последователността от дейности от началото до края на проекта. Някои от дейностите са свързани с останалите. Например, когато строим къща, изграждането на покрива не може да започне, преди да са завършени стените. Забавянето на някои от дейностите може да доведе до забавяне на проекта.

**Определяне на критичния път**

За всяка дейност в мрежата трябва да се определят четири параметъра:

Най-ранен начален час (ES) - Една дейност може да започне най-рано след като предишните зависими дейности са приключили.

Най-ранен краен час (EF) - ES + продължителност на дейността.

Най-късен краен час (LF) – последния възможен край на дейността без да се забави проекта.

Най-късен начален час (LS) - LF - продължителност на дейност.

Плаващо време за дадена дейност е времето между най-ранния и последния начален час или между най-ранния и най-късния краен час. По това време дейността може да се забави, без това да се отрази на крайния срок за финал на проекта.

Методът на критичния път пресмята най-дългия път от планираните дейности до края на проекта. Дейностите в критичния път имат ефект върху крайния срок на проекта. Ако дейността на този път се забави, проектът ще се забави.

**Представяне на дейности чрез граф**

Има два начина за представяне на дейности чрез граф: всяка дейност е връх или всяка дейност е ребро. Ще изберем да представяме дейностите чрез ребра. Завършването на някаква дейност ще разглеждаме като събитие и в графа то ще присъства като връх. Накратко графът, представящ изпълнението на проект, ще се състои от ребра (дейности) и върхове (събития) и ще бъде ориентиран и претеглен. Тегло на дадено ребро ще бъде времетраенето за реализацията на дейността, която то представя. На следващата фигура е представен графът на дейности по създаването на някакво устройство, което се състои от два възела: възел А и възел Б.

Доставка на възел А

Изработване на възел Б

Изпитване на възел Б

50 дни

Корекция на възел Б

20 дни

25 дни

15 дни

Написване на ръководство за експлоатация

60 дни

Възел А се доставя от друг производител и доставката отнема 50 дни. Възел Б се изработва на място и това отнема 20 дни. След това възел Б се изпитва, което отнема 25 дни. Накрая, въз основа на резултатите от изпитването, се правят корекции, които отнемат още 15 дни. Докато възелът се изпитва и коригира, трябва да се напише и ръководство за експлоатация. Писането може да започне едва след като възел Б е изработен и отнема 60 дни.

Задачата е да определим кога най-рано и кога най-късно можем да започнем изпълнението на всяка дейност, съответно кога най-рано и кога най-късно ще завършим с изпълнението на всяка дейност, плаващо време, цялото резервирано време и критичния път.

**Програмна реализация /** [**http://www.cyberforum.ru/pascal/thread542095.html /**](http://www.cyberforum.ru/pascal/thread542095.html%20/)

#include <iostream>

using namespace std;

void Critical\_Path (int n, int i[], int j[], int dij[],

int \*s1, int \*s2, int \*f1, int \*f2, int \*tf, int \*ff)

{

int k,index,max,min;

int ti[20],te[20];

index = 0;

for (k=0;k<n;k++)

{

if ( i[k]==index+1 ) index = i[k];

ti[k] = 0; te[k] = 9999;

}

for (k=0;k<n;k++)

{

max = ti[i[k]] + dij[k];

if ( ti[j[k]]<max ) ti[j[k]] = max;

}

te[j[n-1]] = ti[j[n-1]];

for (k=n-1;k>=0;k--)

{

min = te[j[k]] - dij[k];

if ( te[i[k]]>min ) te[i[k]] = min;

}

for (k=0;k<n;k++)

{

s1[k] = ti[i[k]]; f1[k] = s1[k] + dij[k];

f2[k] = te[j[k]]; s2[k] = f2[k] - dij[k];

tf[k] = f2[k] - f1[k]; ff[k] = ti[j[k]] - f1[k];

}

}

int main()

{ int n; // Общее количество работ по проекту

// (количество ребер ориентированного графа).

int i[20]; // Вектор-пара, представляющая k-ю работу,

int j[20]; // которая понимается как стрелка, связыва-

// ющая событие i[k] с событием j[k]

// Граф задан массивом ребер:

// (i[0],j[0]),(i[1],j[1]),...,(i[n-1],j[n-1])

// Должно быть выполнено:

// i[0]=1, i[k]<i[k+1], i[k]<j[k].

int dij[20];// dij[k] - продолжительность k-й операции.

int s1[20]; // s1[k] - самый ранний срок начала k-й операции.

int s2[20]; // s2[k] - самый поздний срок начала k-й.

int f1[20]; // f1[k] - самый ранний срок завершения k-й.

int f2[20]; // f2[k] - самый поздний срок завершения k-й операции.

int tf[20]; // tf[k] - полный резерв времени k-й операции.

int ff[20]; // ff[k] - свободный резерв времени k-й операции.

int k; // Параметр цикла.

cout << "Total project activities : ";//брой дейности

cin >> n;

for (k=0;k<n;k++)

{

cout << "Beginning and end of edge and duration: "; //Начало и край на ребро и продължителност

cin >> i[k] >> j[k] >> dij[k];

}

Critical\_Path (n,&i[0],&j[0],&dij[0],&s1[0],&s2[0],&f1[0],&f2[0],&tf[0],&ff[0]);

//Вывод результатов.

cout << "The earliest start date: ";

for (k=0;k<n;k++) cout << s1[k] << " "; cout << endl;

cout << "The deadline for the start: ";

for (k=0;k<n;k++) cout << s2[k] << " "; cout << endl;

cout << "The earliest date for completion: ";

for (k=0;k<n;k++) cout << f1[k] << " "; cout << endl;

cout << "The deadline for completion: ";

for (k=0;k<n;k++) cout << f2[k] << " "; cout << endl;

cout << "Floating time: ";

for (k=0;k<n;k++) cout << ff[k] << " "; cout << endl;

cout << "Full-time reserve: ";

for (k=0;k<n;k++) cout << tf[k] << " "; cout << endl;

// Определение критического пути. Критический путь задается

// стрелками, соединяющими события, для которых полный резерв

// времени равен нулю.

cout << "The critical path: " << 1 << " ";

for (k=0;k<n;k++)

if ( tf[k]==0 ) cout << j[k] << " ";

return 0;

}

/\*

5

1 2 20

1 4 50

2 3 25

2 4 60

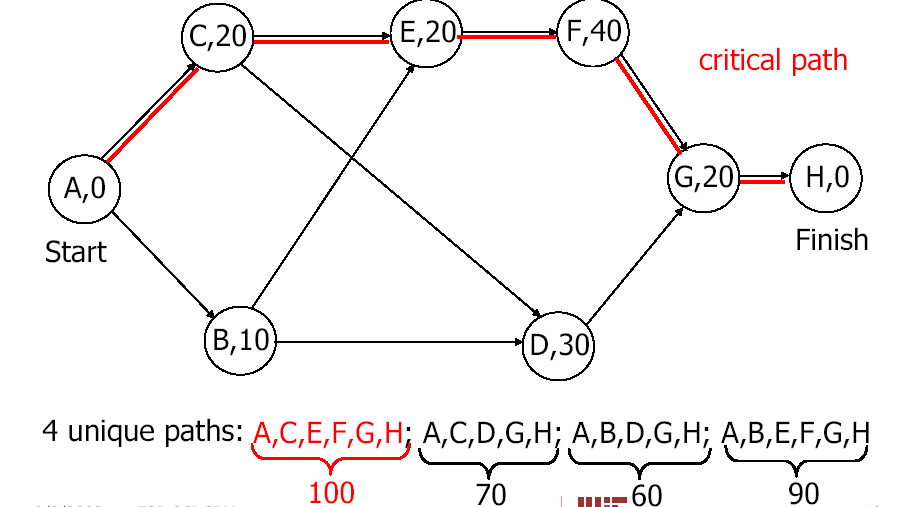
3 4 15

\*/

**II реализация – Амереал /леко променена/ – в папката**

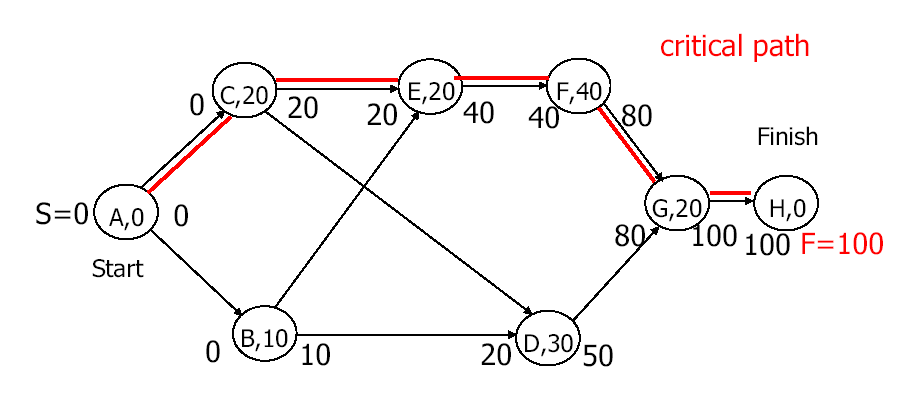
**Допълнение 1 за Метод на критичния път - От презентацията – нередактирано**

* Задачите представляват възли от графа
* Задачите, които нямат предшестващи задачи се свързват с началния възел
* Задачите, от които не следват други задачи, се свързват с крайния възел
* Резултат – краен брой пътища между началния и крайния възел
* Сумата от времената на отделните задачи в даден път – общото време за този път
* Пътят с най-дълго общо време – критичен път – може да бъде повече от един



Алгоритъм

* Критичният път представлява пътят при безизходица
* Скъсяването или увеличаването на времетраенето на задачи от критичния път пряко влияе върху времето за завършване на целия проект
* Времетраенето на “некритичните” задачи не е от такава важност, макар прекаленото увеличаване на времетраенето на някоя от тях да може да измени критичния път
* Скъсяване на времетраенето на поставените задачи – икономическо и техническо предизвикателство
* Некритични задачи могат да се превърнат в критични
* За големи проекти съществуват много пътища
* Необходим е алгоритъм за определяне на ефикасността на критичния път
* Необходима е информация за отделните задачи в контекста на целия проект
* Времетраене
  + Начално време (S)
  + За всяка задача – най-ранно време за започване – (ES)
  + Времетраене на задачата – t
  + Най-ранно време за приключване – (EF)=(ES)+t
* Време на приключване – най-ранното време за приключване на целия проект – (F)
* Маркирайте стойността на S вляво и вдясно от началния възел
* Разгледайте всички нови немаркирани задачи, чиито предшественици са маркирани. Маркирайте вляво от новата задача най-голямото число, стоящо вдясно на непосредствено предходната задача.
* Прибавете към ES времето t и маркирайте резултата вдясно

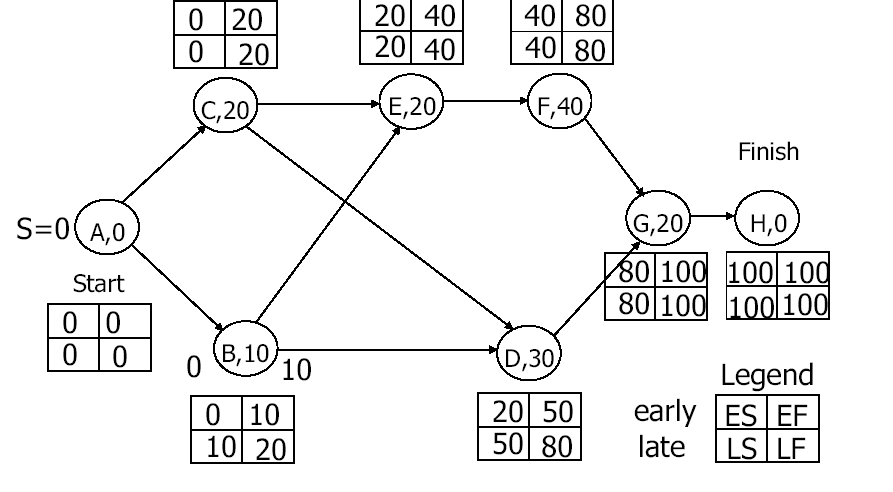


**Най-късно начално и крайно време**

* Определете целевото време за приключване на проекта T>=F
* Определете най-късното време за започване на проекта
* Определете най-късното време за приключване на дадена задача (LF), без забавяне на проекта след целевото време на приключване (S)
* LS=LF-t

**Определяне на LF и LS**

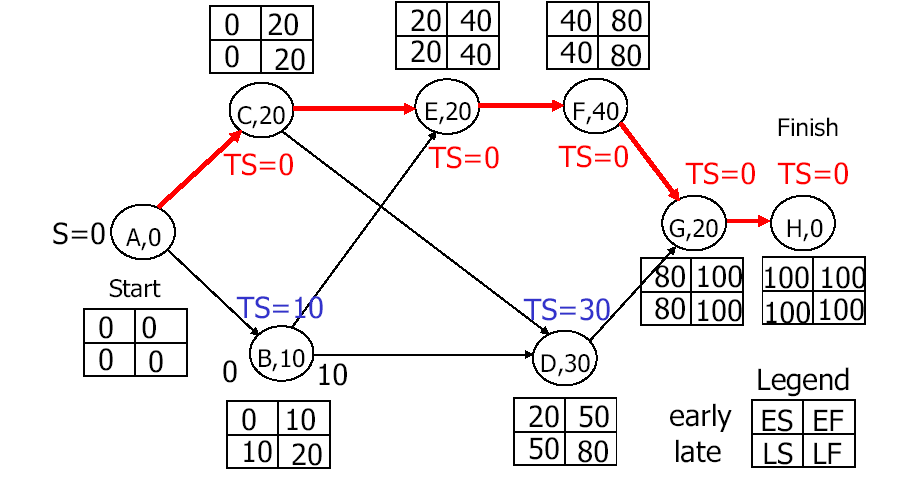
* Вземаме за начална точка целевото време за завършване на проекта – Т
* Маркираме стойността на Т вляво и вдясно от крайния възел
* Разглеждаме всички нови, немаркирани задачи, чиито наследници са маркирани. Маркираме вдясно от най-малкото LS време, маркирано вляво от който и да е от пряко следващите възли
* От числото LF изваждаме времето за изпълнение на задачата – t, и маркираме резултата вляво от задачата – LS
* Продължаваме до достигане на началния възел



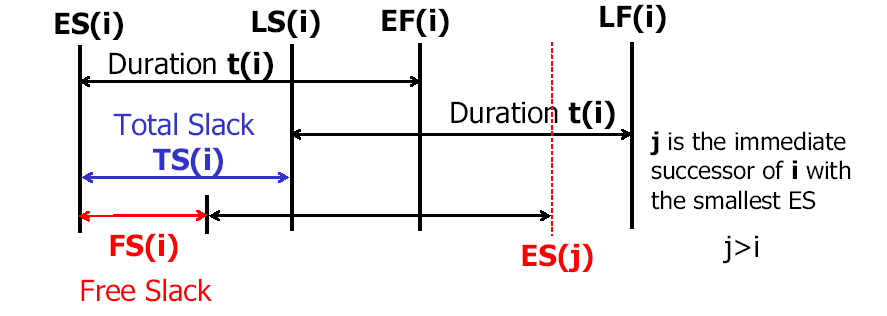
**Резервно време**

* При някои задачи ES=LS и няма slack
* Общият slack за дадена задача TS=LS-ES
* Максималното време, което може да бъде забавена дадена задача, след ранния старт, без това да повлияе на цялостното време за изпълнение на проекта
* Slack времето е ценно и представлява свобода на мениджъра, която не трябва да се пропилява
* Когато T=F, всички критични задачи имат TS=0
* Съществува поне един път от началния до крайния възел, съдържащ само критични задачи
* Когато T>F, всички критични задачи имат TS=T-F

**Граф на проекта – резервно време**



**Времетраене на задачи – задача i**



**Допълнение 2 за Метод на критичния път – от referati.com – нередактирано**

Задачата е да определим кога най-рано и кога най-късно можем да започнем изпълнението на всяка дейност и съответно кога най-рано и кога най-късно ще завършим с изпълнението на всяка дейност.

Тази задача се решава чрез топологично сортиране. Затова и програмата за нейното решаване ще получим чрез целесъобразни изменения в програмата за топологично сортиране. Тук ще се ограничим само с програма за граф, представен с матрица на съседство.

Програмата ще се състои от три части.

I-ва част. Задачата на тази част е намирането на най-ранните моменти на настъпване на събитията в графа. Тази част получаваме от програмата за топологично сортиране, като направим следните промени:

* добавяме нов масив за регистриране най-ранно настъпване на събитието nrns[]. Елемент nrns[i] съответства на i-ия връх и стойността му трябва да показва момента, в който най-рано може да настъпи събитие i, например nrns[1]=45 означава, че събитие 1 ще настъпи най-рано 45 дни след началото;
* след като вземем връх (събитие) от опашката, не го извеждаме в топологичния ред, а изчисляване най-дългия път до него;
* въведена е допълнителна променлива maxT за максималното време за реализация на всички дейности.

В края на I-та част от програмата елементите на масива nrns[] и променливата maxT вече имат стойности. За разглеждания пример те са:

nrns[0]=0,

nrns[1]=45,

nrns[2]=80,

nrns[3]=20,

maxT=80.

II-pа част. Задачата на тази част е намирането на най-късните моменти на настъпване на събитията в графа. Това става пак чрез топологично сортиране, но в обратен ред, т.е. със следните особености:

* вместо масива с броя на предшествениците br\_pred ще ползваме масив с броя на наследниците br\_nasl. Броят на наследниците на всеки връх намираме като преброим ненулевите елементи в съответния ред в матрицата на съседство;
* вместо насива nrns[] сега ще ползваме масив за регистрация на най-късното настъпване на събитията nkns[]. Елемент nkns[i] съответства на i-ия връх и стойността трябва да показва момента, в който най-късно настъпва събитие i, например nkns[1]=65 означава, че събитие 1 ще настъпи най-късно 65 дни след началото;
* най-напред в опашката ще изпратим събитията (върховете) без наследници. След това ще ги вземем от опашката, ще отбележим, че моментът на тяхното настъпване е maxT, ще намaлим броя на наследниците на техните предшественици с 1 и ще изчислим времето до тяхното настъпване и т.н.
* за тази втора част можем да копираме първата част, но трябва да отчетем, че сега наследниците и предшествениците са си разменили местата, а това налага да разменим i-тата и j-тата.

III-та част. В тази част от масивите nrns[] и nkns[] получаваме за всяка дейност (ребро) от графа следните времена:

nrzap – най-ранно започване на дадена дейност

nrzav – най-ранно завършване на дадена дейност

nkzap - най-късно започване на дадена дейност

nkzav – най-късно завършване на дадена дейност

Резултатите се извеждат във вид на таблица в текстов файл. За нашия пример програмата извежда следната таблица:

i j d nrzap nrzav nkzap nkzav

0 2 50 0 50 30 80

0 3 20 0 20 0 20

1 2 15 45 60 65 80

3 1 25 20 45 40 65

3 2 60 20 80 20 80

В таблицата i e началото на реброто-дейност, j е краят на реброто-дейност, а d е теглото на реброто или броя дни за извършване на тази дейност. Останалите означения обяснихме по-горе.

**Програмна реализация** Организация на изпълнението на проект

# include <iostream.h>

# include <conio.h>

#include "graf\_mat.h"

void clasGraph::OrgProekt()

{unsigned i,j,k,n,t,maxT=0,

br\_visit=0, // Брой на посетените върхове

\*visit, // За масива, отчитащ посетеността на върховете

\*br\_pred, // За масива, отчитащ броя на предшествениците на всеки връх

\*br\_nasl, // За масива, отчитащ броя на наследниците на всеки връх

\*nrns, // За масива най-ранно настъпване на събитие

\*nkns; // За масива най-късно настъпване на събитие

if (!(visit=new unsigned[BrV])) exit(0);

if (!(br\_pred=new unsigned[BrV])) exit(0);

if (!(br\_nasl=new unsigned[BrV])) exit(0);

if (!(nrns=new unsigned[BrV])) exit(0);

if (!(nkns=new unsigned[BrV])) exit(0);

for (i=0;i<BrV;i++) {visit[i]=0;br\_pred[i]=0; nrns[i]=0;}

// 1. Търсене най-ранните моменти на настъпване на събитията от графа

for (j=0;j<BrV;j++){

for (i=0;i<BrV;i++) if (A[i][j]) br\_pred[j]++;//Преброяване на предшествениците

if (br\_pred[j]==0) Opashka.DobEl(j);//Връх без предшественици отива в опашката

}

while (Opashka.Br) { //Докато опашката не е празна,...

n=Opashka.Br;

for (i=0;i<n;i++){

Opashka.VzeEl(k);//... вземаме връх от опашката ...

visit[k]=1; //... маркираме го като посетен и...

br\_visit++; //... го преброяваме, като посетен.

for (j=0;j<BrV;j++)

if (A[k][j]!=0 && visit[j]==0){ //На непосетените наследници на к-ия връх ...

br\_pred[j]--; //... намаляваме входната степен и...

t=nrns[k]+A[k][j];

if (t>nrns[j]) nrns[j]=t;

if (t>maxT) maxT=t;

if (br\_pred[j]==0) Opashka.DobEl(j); //ако стане 0, пращаме го в опашката

}

}

}

if (br\_visit<BrV) {cout<<"Графът е цикличен.\n"; getch();return;}

// 2. Търсене най-късните моменти на настъпване на събитията от графа

for (i=0;i<BrV;i++) {visit[i]=0;br\_nasl[i]=0; nkns[i]=maxT;}

for (j=0;j<BrV;j++){

for (i=0;i<BrV;i++) if (A[j][i]) br\_nasl[j]++;//Преброяване на наследниците

if (br\_nasl[j]==0) Opashka.DobEl(j);//Връх без наследници отива в опашката

}

while (Opashka.Br) { //Докато опашката не е празна,...

n=Opashka.Br;

for (i=0;i<n;i++){

Opashka.VzeEl(k);//... вземаме връх от опашката ...

visit[k]=1; //... маркираме го като посетен и...

br\_visit++; //... го преброяваме, като посетен.

for (j=0;j<BrV;j++)

if (A[j][k]!=0 && visit[j]==0){ //На непосетените предшественици на j-ия връх ...

br\_nasl[j]--; //... намаляваме входната степен и...

t=nkns[k]-A[j][k];

if (t<nkns[j]) nkns[j]=t;

if (br\_nasl[j]==0) Opashka.DobEl(j); //ако стане 0, пращаме го в опашката

}

}

}

// 3. Отпечатване на графика на операциите

int nrzap,nkzap,nrzav, nkzav,d;

fout.open("c:\\sdp\\graphs\\DopInf",ios::out);

fout<<"\n i j d nrzap nrzav nkzap nkzav\n";

for (i=0;i<BrV;i++)

for (j=0;j<BrV;j++)

if (A[i][j]!=0) {

d=A[i][j];

nrzap=nrns[i];

nrzav=nrzap+d;

nkzav=nkns[j];

nkzap=nkzav-d;

fout<<setw(4)<<i<<" " <<setw(3)<<j<<" "<<setw(3)<<d<<" "

<<setw(5)<<nrzap<<" "<<setw(5)<<nrzav <<" <<setw(5)<<nkzap<<" "

<<setw(5)<<nkzav<<" \n";

} fout.close();

}

void main()

{

char ime\_vh\_fl[21];

cout<<"\nИме на входния файл: "; cin>>ime\_vh\_fl;

fin.open(ime\_vh\_fl,ios::in);

if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}

fin>>BrV; //Въвеждаме броя на върховете от текстовия файл

clasGraph Graph;

Graph.make\_graf();

Graph.display\_graf();

//Graph.display\_grafTF();

Graph.OrgProekt();

cout<<"\nДопълнителна информация можете да намерите във файл \"DopInf.\"\n";

21

11

13

16

18

14

15

12

17

19

}

Интересен пример е даденият вдясно граф на дейностите. Той има два върха (0 и 2) без предшественици т.е. възможно е изпълнението на дейностите от графа да започне едновременно от връх 0 и от връх 2. Той има и два върха без наследници.

Резултатите от изпълнението на програмата са следните:

i j d nrzap nrzav nkzap nkzav

2

3

1

3

6

5

5

1

4

2

3

2

0 1 11 0 11 0 11

0 4 19 0 19 4 23

1 3 13 11 24 11 24

2 5 15 0 15 4 19

2 6 12 0 12 28 40

2 7 14 0 14 26 40

3 7 16 24 40 24 40

4 6 17 19 36 23 40

5 6 18 15 33 22 40

