[1. Сеть](#_Toc509246494)

[1.1. Сети (общие сведения)](#_Toc509246495)

[1.2. Кратчайшие пути в сети. Алгоритм Форда-Беллмана](#_Toc509246496)

[1.3. Кратчайшие пути в сети. Алгоритм Дейкстры](#_Toc509246497)

[1.4. Пути в бесконтурной сети](#_Toc509246498)

[1.5. Потоки в сетях. Система дорог](#_Toc509246499)

[1.6. Описание алгоритма Дейкстры](#_Toc509246500)

[1.7. Алгоритм Флойда](#_Toc509246501)

[1.8. Задача о максимальном потоке](#_Toc509246502)

[1.9. Алгоритм нахождения максимального потока](#_Toc509246503)

[1.10. Нахождение потока наименьшей стоимости (общие замечания)](#_Toc509246504)

[1.11. Нахождение потока наименьшей стоимости. Сетевая модель](#_Toc509246505)

[1.12. Нахождение потока наименьшей стоимости. Сетевая модель как задача линейного программирования](#_Toc509246506)

[1.13. Нахождение потока наименьшей стоимости. Симплексный алгоритм для сетей с ограниченной пропускной способностью](#_Toc509246507)

[1.14. Методы сетевого планирования (общие сведения)](#_Toc509246508)

[1.15. Методы сетевого планирования. Построение сети проекта](#_Toc509246509)

[1.16. Методы сетевого планирования. Метод критического пути](#_Toc509246510)

[1.17. Методы сетевого планирования. Построение временного графика](#_Toc509246511)

[1.18. Приложение "Сетевые модели"](#_Toc509246512)

1. Сеть

## Сети (общие сведения)

На этом шаге мы приведем ***общие сведения о сетях***.

Наиболее общий вид ***многосвязной структуры*** - многосвязная структура, которая характеризуется следующими свойствами [1, с.95].

1. Каждый элемент структуры содержит произвольное число направленных связей с другими элементами (или ссылок на другие элементы).
2. С каждым элементом может связываться произвольное число других элементов (т.е. каждый элемент может быть объектом ссылки произвольного числа других элементов).
3. Каждая связь в структуре имеет не только направление, но и вес.

Такую многосвязную структуру называют ***сетевой*** структурой или ***сетью*** [2].

Логически сеть эквивалентна взвешенному ориентированному графу общего вида [2, 3], и поэтому вместо термина "сеть" часто употребляются термин "графовая структура", или просто "граф".

Сетевые структуры широко применяются при организации банков данных, систем управления базами данных, в системах программного имитационного моделирования сложных комплексов и т.д. Особое значение сетевые структуры приобрели в системах искусственного интеллекта, в которых они адекватно отражают логику организации данных и сложные отношения, возникающие в таких системах между различными элементами данных. В этих системах сетевые структуры применяются для построения семантических сетей, фреймов и других логических конструкций, необходимых для представления знаний, образования понятий и осуществления логических выводов.

(1) Костин А.Е., Шаньгин В.Ф. Организация и обработка структур данных в вычислительных системах. - М.: Высш.шк., 1987. - 248 с.

(2) Трамбле Ж., Соренсон П. Введение в структуры данных. - М.: Машиностроение, 1982. - 784 с.

(3) Харари Ф. Теория графов. - М.: Мир, 1973. - 300 с.

Со следующего шага мы начнем приводить ***алгоритмы нахождения кратчайшего пути в сетях***.

## Кратчайшие пути в сети. Алгоритм Форда-Беллмана

На этом шаге мы приведем ***реализацию алгоритма Форда-Беллмана***.

Приведем текст программы, реализующей алгоритм ***Форда-Беллмана***.

**//Нахождение расстояния от источника до всех вершин**

**//(метод Форда-Беллмана).**

**//Нахождение кратчайшего пути из S в T.**

**#include** <iostream.h>

**#define** MaxNodes 7

**#define** B 1000 **//Машинный эквивалент бесконечности.**

**//Описание типа узла стека.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

**int** Element;

svqz Sled;

};

**class** Spisok

{

**private**:

**int** A[MaxNodes][MaxNodes]; **//Матрица весов дуг.**

**int** D[MaxNodes]; **//Матрица расстояний от источника до**

**//всех вершин графа.**

svqz Stack; **//Указатель на рабочий стек.**

**void** UDALENIE (svqz \*, **int** \*);

**void** W\_S (svqz \*, **int**);

**public**:

Spisok() {Stack = **NULL**;}

**void** Vvod\_Ves();

**void** Reshenie ();

};

**void** main ()

{

Spisok A;

A.Vvod\_Ves();

A.Reshenie();

}

**void** Spisok::Reshenie ()

{

**int** S; **// Начальная вершина пути (источник).**

**int** T; **// Конечная вершина пути.**

**int** u,v;

**int** i,j,k;

svqz UkZv;

cout << "Введите источник: ";

cin >> S; S--;

**//Инициализация.**

**for** (i=0;i<MaxNodes;i++) D[i] = A[S][i];

D[S] = 0;

**//Вычисление матрицы расстояний от**

**//источника до всех вершин графа.**

**for** (k=0;k<MaxNodes-2;k++)

**for** (i=0;i<MaxNodes;i++)

**if** ( i!=S )

**for** (j=0;j<MaxNodes;j++)

**if** ( D[i] > D[j]+A[j][i] ) D[i] = D[j]+A[j][i];

**//Вывод матрицы расстояний от источника**

**//до всех вершин графа.**

cout << "Матрица расстояний: \n";

**for** (i=0;i<MaxNodes;i++) cout << D[i] << " ";

cout << endl;

**// -----------------------------------------------------**

**// Нахождение кратчайшего пути из S в T с использованием**

**// построенной матрицы расстояний.**

**// -----------------------------------------------------**

cout << "Введите конечную вершину пути: ";

cin >> T; T--;

W\_S (&Stack,T); v = T;

**while** ( v!=S )

{

**for** (i=0;i<MaxNodes;i++)

**if** ( D[v]==D[i]+A[i][v] ) u = i;

W\_S (&Stack,u);

v = u;

}

**//Вывод кратчайшего пути на экран дисплея.**

cout << "Кратчайший путь: ";

UkZv = Stack;

**while** ( UkZv != **NULL** )

{ cout << UkZv->Element << " ";

UkZv = UkZv->Sled; }

cout << endl;

}

**void** Spisok::Vvod\_Ves()

**//Ввод матрицы весов дуг заданного графа.**

{

cout << "Вводите элементы матрицы весов дуг по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout << "Введите A[" << (i+1) << "," << (j+1) << "]: ";

cin >> A[i][j];

**if** ( A[i][j]==0 ) A[i][j] = B;

}

}

**void** Spisok::W\_S (svqz \*stk, **int** Elem)

**//Помещение Elem в стек stk.**

{

svqz q=new (Zveno);

(\*q).Element = Elem;

(\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::UDALENIE (svqz \*stk, **int** \*Klad)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметре Klad.**

{

svqz q;

**if** (\*stk==**NULL**) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*Klad = (\*\*stk).Element;

q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din119_1.zip).

Рассмотрим ***тестовый пример*** [1].

Вычислить кратчайшее растояние между вершинами **s** и **t** в сети:

**(s,a,4), (s,d,6), (a,d,-3), (a,b,8), (b,t,2), (a,e,5),**

**(d,e,2), (e,c,3), (d,c,11), (c,b,-4), (c,t,7), (b,d,-6).**

Перенумеруем вершины:

**(s,1), (a,2), (d,3), (b,4), (c,5), (e,6), (t,7),**

выпишем матрицу весов дуг

0 4 6 0 0 0 0

0 0 -3 8 0 5 0

0 0 0 0 11 2 0

0 0 -6 0 0 0 2

0 0 0 -4 0 0 7

0 0 0 0 3 0 0

0 0 0 0 0 0 0

и применим ***алгоритм Форда-Беллмана*** к этой матрице. В результате получим:

Введите источник: 1

Матрица расстояний 0: 0 4 -4 2 1 -2 4

Введите конечную вершину пути: 7

Кратчайший путь: ...

Компьютер "зависает", ибо ***в графе имеется контур отрицательной длины*** **3-6-5-4** (его длина -6+2+3-4=-5), что приводит к появлению бесконечного множества путей, каждый из которых "короче" предыдущего, например:

путь 1-2-3-6-5-4-7 с длиной 4,

путь 1-2-3-6-5-4-3-6-5-4-7 с длиной -1,

путь 1-2-3-6-5-4-3-6-5-4-3-6-5-4-7 с длиной -6 и т.д.

***Поэтому кратчайший путь алгоритмом Форда-Беллмана определить нельзя.***

***Замечание [2, с.47-48].*** *Приведенный алгоритм отыскания кратчайших путей в графах с отрицательными длинами дуг, принадлежащий* ***Форду, Муру*** *и* ***Беллману****, может служить одним из возможных способов обнаружения контуров отрицательной длины (или циклов в неориентированном графе).*

(1) Евстигнеев В.А., Мельников Л.С. Задачи и упражнения по теории графов и комбинаторике. - Новосибирск: НГУ, 1981. - 88 с.

(2) Евстигнеев В.А., Касьянов В.Н. Деревья: основные понятия, свойства и алгоритмы. - Новосибирск: НГУ, 1992. - 80 с.

На следующем шаге мы рассмотрим ***алгоритм Дейкстры***.

## Кратчайшие пути в сети. Алгоритм Дейкстры

На этом шаге мы рассмотрим ***алгоритм Дейкстры***.

Описание алгоритма ***Дейкстры*** дано на [123 шаге](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0123.html). Здесь мы приведем текст программы, реализующей алгоритм ***Дейкстры***.

**//Нахождение расстояния от источника до всех вершин в графе**

**//с неотрицательными весами (метод Дейкстры).**

**//Нахождение кратчайшего пути из S в T.**

**#include** <iostream.h>

**#define** MaxNodes 7

**#define** B 1000 **//Машинный эквивалент бесконечности.**

**//Описание типа узла стека.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

**int** Element;

svqz Sled;

};

**class** Spisok

{

**private**:

**int** A[MaxNodes][MaxNodes]; **//Матрица весов дуг.**

**int** D[MaxNodes]; **//Матрица расстояний от источника до**

**//всех вершин графа.**

svqz Stack; **//Указатель на рабочий стек.**

**void** UDALENIE (svqz \*, **int** \*);

**void** W\_S (svqz \*, **int**);

**int** Pusto\_Q (**int** \*);

**public**:

Spisok() {Stack = **NULL**;}

**void** Vvod\_Ves();

**void** Reshenie ();

};

**void** main ()

{

Spisok A;

A.Vvod\_Ves();

A.Reshenie();

}

**int** Spisok::Pusto\_Q (**int** \*Q)

{

**for** (**int** i=0;i<MaxNodes;i++)

**if** ( \*(Q+i)!=0 ) **return** 0; **//Ложь - не пусто!**

**return** 1; **//Истина - пусто!**

}

**void** Spisok::Reshenie ()

{

**int** S; **// Начальная вершина пути (источник).**

**int** T; **// Конечная вершина пути.**

**int** u,v,Min;

**int** i,j,k;

svqz UkZv;

**int** Q[MaxNodes];

cout << "Введите источник: ";

cin >> S; S--;

**//Инициализация.**

**for** (i=0;i<MaxNodes;i++) { D[i] = A[S][i]; Q[i] = 0; }

D[S] = 0;

**for** (i=0;i<MaxNodes;i++) Q[i] = 1;

Q[S] = 0;

**//Вычисление матрицы расстояний от**

**//источника до всех вершин графа.**

**while** (!Pusto\_Q(&Q[0])) **//Пока Q не пусто.**

{

Min = B;

**for** (i=0;i<MaxNodes;i++)

**if** (Q[i]==1 && D[i]<Min) { Min = D[i]; u = i; }

Q[u] = 0;

**for** (i=0;i<MaxNodes;i++)

**if** (Q[i] == 1)

**if** ( D[i] > D[u]+A[u][i] ) D[i] = D[u] + A[u][i];

}

**//Вывод матрицы расстояний от источника**

**//до всех вершин графа.**

cout << "Матрица расстояний: \n";

**for** (i=0;i<MaxNodes;i++) cout << D[i] << " ";

cout << endl;

**// -----------------------------------------------------**

**// Нахождение кратчайшего пути из S в T с использованием**

**// построенной матрицы расстояний.**

**// -----------------------------------------------------**

cout << "Введите конечную вершину пути: ";

cin >> T; T--;

W\_S (&Stack,T); v = T;

**while** ( v!=S )

{

**for** (i=0;i<MaxNodes;i++)

**if** ( D[v]==D[i]+A[i][v] ) u = i;

W\_S (&Stack,u);

v = u;

}

**//Вывод кратчайшего пути на экран дисплея.**

cout << "Кратчайший путь: ";

UkZv = Stack;

**while** ( UkZv != **NULL** )

{ cout << (UkZv->Element+1) << " ";

UkZv = UkZv->Sled; }

cout << endl;

}

**void** Spisok::Vvod\_Ves()

**//Ввод матрицы весов дуг заданного графа.**

{

cout << "Вводите элементы матрицы весов дуг по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout << "Введите A[" << (i+1) << "," << (j+1) << "]: ";

cin >> A[i][j];

**if** ( A[i][j]==0 ) A[i][j] = B;

}

}

**void** Spisok::W\_S (svqz \*stk, **int** Elem)

**//Помещение Elem в стек stk.**

{

svqz q=new (Zveno);

(\*q).Element = Elem;

(\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::UDALENIE (svqz \*stk, **int** \*Klad)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметре Klad.**

{

svqz q;

**if** ( \*stk == **NULL** ) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*Klad = (\*\*stk).Element;

q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din120_1.zip).

Приведем ***контрпример*** к алгоритму Дейкстры.

Вычислить кратчайшее растояние между вершинами **s** и **t** в сети[1]:

**(s,a,4), (s,d,6), (a,d,-3), (a,b,8), (b,t,2), (a,e,5),**

**(d,e,2), (e,c,3), (d,c,11), (c,b,-4), (c,t,7), (b,d,-6).**

Заметим, что мы допустили существование отрицательных весов!

Перенумеруем вершины:

**(s,1), (a,2), (d,3), (b,4), (c,5), (e,6), (t,7),**

выпишем матрицу весов дуг

0 4 6 0 0 0 0

0 0 -3 8 0 5 0

0 0 0 0 11 2 0

0 0 -6 0 0 0 2

0 0 0 -4 0 0 7

0 0 0 0 3 0 0

0 0 0 0 0 0 0

и применим ***алгоритм Дейкстры*** к этой матрице. В результате получим:

Введите источник: 1

Матрица расстояний:

0 4 1 2 6 3 4

Введите конечную вершину пути: 7

Кратчайший путь: 1 2 3 6 5 4 7

Однако, как уже было отмесено ранее, ***в графе имеется контур отрицательной длины*** **3-6-5-4** (его длина -6+2+3-4=-5), что приводит к появлению бесконечного множества путей, каждый из которых "короче" предыдущего, например:

путь 1-2-3-6-5-4-7 с длиной 4,

путь 1-2-3-6-5-4-3-6-5-4-7 с длиной -1,

путь 1-2-3-6-5-4-3-6-5-4-3-6-5-4-7 с длиной -6 и т.д.

Таким образом, кратчайший путь алгоритмом Дейкстры определить нельзя, так как в графе имеется контур отрицательной длины.

(1) Евстигнеев В.А., Мельников Л.С. Задачи и упражнения по теории графов и комбинаторике. - Новосибирск: НГУ, 1981. - 88 с.

На следующем шаге мы рассмотрим ***пути в бесконтурной сети***.

## Пути в бесконтурной сети

На этом шаге мы рассмотрим ***вычисление путей в бесконтурных сетях***.

Займемся теперь вторым случаем, для которого известен алгоритм нахождения расстояний от фиксированной вершины за время **O(n2)**, а именно случаем, когда граф является бесконтурным (***веса дуг могут быть произвольными***).

При описании алгоритма нахождения путей в бесконтурном графе мы можем предположить, что каждая дуга идет из вершины с меньшим номером в вершину с большим номером.

Пример. ***Нахождение расстояний от источника до всех вершин в бесконтурном графе***.

**#include** <iostream.h>

**#define** MaxNodes 9

**#define** B 1000 **//Машинный эквивалент бесконечности.**

**class** Spisok

{

**private**:

**int** A[MaxNodes][MaxNodes]; **//Матрица весов дуг.**

**int** D[MaxNodes]; **//Матрица расстояний от источника до**

**//всех вершин графа.**

**public**:

**void** Vvod\_Ves();

**void** Reshenie ();

};

**void** main ()

{

Spisok A;

A.Vvod\_Ves();

A.Reshenie();

}

**void** Spisok::Reshenie ()

{

**int** S; **// Начальная вершина пути (источник).**

**int** i,j;

S = 0;

**//Инициализация.**

**for** (j=1;j<MaxNodes;j++) D[j] = B;

D[S] = 0;

**//Вычисление матрицы расстояний от источника до**

**//всех вершин графа.**

**for** (j=1;j<MaxNodes;j++)

**for** (i=0;i<MaxNodes;i++)

**if** (A[i][j]!=0 && D[j]>D[i]+A[i][j]) D[j] = D[i]+A[i][j];

cout << "Матрица расстояний: \n";

**for** (i=0;i<MaxNodes;i++) cout << D[i] << " ";

cout << endl;

}

**void** Spisok::Vvod\_Ves()

**//Ввод матрицы весов дуг заданного графа.**

{

cout << "Вводите элементы матрицы весов дуг по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout << "Введите A[" << (i+1) << "," << (j+1) << "]: ";

cin >> A[i][j];

**if** ( A[i][j]==0 ) A[i][j] = B;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din121_1.zip).

Проверьте работу программы для следующей путевой матрицы:

0 1 0 2 0 0 1 0 0

0 0 2 0 7 0 0 0 0

0 0 0 0 0 10 0 0 5

0 0 0 0 5 0 0 0 0

0 0 0 0 0 6 0 1 0

0 0 0 0 0 0 0 0 4

0 0 0 0 0 0 0 3 0

0 0 0 0 0 0 0 0 5

0 0 0 0 0 0 0 0 0

При выполнении приложения будет получен следующий результат:

Матрица расстояний:

0 1 3 2 7 13 1 4 8

Приведенный алгоритм может найти применение в методах управления выполнением проекта, называемых **PERT (Project Evaluation and Review Technique)** или **CPM (Critical Path Method)**. Эти методы основываются на построении графа ([***сети* PERT**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0093.html) или ***сети* CPM**), дуги которого соответствуют некоторым элементарным задачам, составляющим проект, а их веса указывают на время, необходимое для решения отдельных задач. Кроме этого, мы предполагаем, что для произвольных дуг этого графа **(u,v)** и **(v,t)** задача, изображаемая дугой **(u,v)**, должна быть закончена перед началом решения задачи, изображаемой дугой **(v,t)**. Легко заметить, что такой граф должен быть бесконтурным.

Нашей задачей является нахождение ***самого длинного пути*** из вершины **s**, соответствующей началу проекта, до вершины **t**, соответствующей его окончанию. Такой путь называется ***критическим***. Его длина определяет время, необходимое для реализации всего проекта.

Эта задача сводится к задаче о кратчайшем пути путем изменения знака каждого веса **A(u,v)**, где **u->v**, на обратный.

Так, например, выполните предыдущую программу для следующей путевой матрицы:

0 -1 0 -2 0 0 1 0 0

0 0 -2 0 -7 0 0 0 0

0 0 0 0 0-10 0 0 -5

0 0 0 0 -5 0 0 0 0

0 0 0 0 0 -6 0 -1 0

0 0 0 0 0 0 0 0 4

0 0 0 0 0 0 0 -3 0

0 0 0 0 0 0 0 0 -5

0 0 0 0 0 0 0 0 0

Результат будет следующим:

Матрица расстояний:

0 -1 -3 -2 -8 -14 1 -9 -14

Пример [1, с.100-104].

**//Планирование критического пути (анализ сети ПЕРТ)**

**//Автор алгоритма: Leavenworth B. (CACM, 1961, 3).**

**#include** <iostream.h>

**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];

}

}

**void** 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 << "Введите общее количество работ по проекту: ";

cin >> n;

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

{

cout << "Вводите начало, конец дуги и продолжительность: ";

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 << "Самый ранний срок начала : ";

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

cout << "Самый поздний срок начала : ";

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

cout << "Самый ранний срок завершения : ";

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

cout << "Самый поздний срок завершения : ";

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

cout << "Свободный резерв времени : ";

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

cout << "Полный резерв времени : ";

**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 << "Критический путь: " << 1 << " ";

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

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

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din121_2.zip).

(1) Библиотека алгоритмов 1б-50б. (Справочное пособие.) - М.: Сов.радио, 1975. - 176 с.

На следующем шаге мы приведем ***программу вычисления потока в сети***.

## Потоки в сетях. Система дорог

На этом шаге мы приведем ***программу вычисления потока в сети*** и дадим несколько определений, связанных с ***системой дорог***.

Пример [1, 2].

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**#define** MaxNodes 8

**#define** MaxInt 1000 **//Машинный эквивалент бесконечности.**

**class** Spisok

{

**private**:

**int** cap[MaxNodes][MaxNodes]; **//Функция пропускной способности.**

**int** flow[MaxNodes][MaxNodes]; **//Максимальная функция потока.**

Boolean Any (Boolean \*);

**public**:

**void** Vvod();

**void** Maxflow (int, int, int \*);

**void** Vyvod (**int**);

};

**void** main()

{

Spisok A;

**int** s; **//Источник.**

**int** t; **//Сток.**

**int** totflow; **//Значение потока от узла s к узлу t.**

**//Задайте пропускную способность.**

A.Vvod();

cout << "Укажите источник... "; cin >> s;

cout << "Укажите сток... "; cin >> t;

**//Вывод результатов.**

A.Maxflow (s,t,&totflow);

A.Vyvod(totflow);

}

**void** Spisok::Vvod()

{

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout << "Введите элемент " << i << " " << j<< " ";

cin >> cap[i][j];

}

}

**void** Spisok::Vyvod(**int** totflow)

{

**for** (**int** i=0;i<MaxNodes;i++)

{

**for** (**int** j=0;i<MaxNodes;i++)

cout << flow[i][j] << " ";

cout << endl;

}

cout << "Значение потока: " << totflow << endl;

}

Boolean Spisok::Any (Boolean \*Q)

{

Boolean A = FALSE;

**for** (**int** i=0;i<MaxNodes;i++) A = (A || Q[i]);

**return** A;

}

**void** Spisok::Maxflow (**int** s, **int** t, **int** \*totflow)

{

**int** nd,i,x,pred;

Boolean endpath[MaxNodes];

Boolean onpath[MaxNodes];

Boolean forward[MaxNodes];

**int** improve[MaxNodes];

**int** precede[MaxNodes];

**for** (nd=0;nd<MaxNodes;nd++)

**for** (i=0;i<MaxNodes;i++)

flow[nd][i] = 0;

\*totflow = 0;

**do** {

**//Попытка найти полупуть из s в t.**

**for** (nd=0;nd<MaxNodes;nd++)

{

endpath[nd] = FALSE;

onpath [nd] = FALSE;

}

endpath[s] = TRUE;

onpath [s] = TRUE;

improve[s] = MaxInt;

**// Мы допускаем, что s обеспечивает неопределенный поток.**

**while** (!onpath[t] && Any (&endpath[0]))

{

**// Попытка расширить существующий путь.**

nd = 0;

**while** (!endpath[nd]) nd++;

endpath[nd] = FALSE;

**for** (i=0;i<MaxNodes;i++)

{

**if** (flow[nd][i]<cap[nd][i] && !onpath[i])

{

endpath[i] = TRUE;

onpath [i] = TRUE;

precede[i] = nd;

forward[nd] = TRUE;

x = cap[nd][i] - flow[nd][i];

**if** ( improve[nd]<x ) improve[i] = improve[nd];

**else** improve[i] = x;

}

**if** (flow[i][nd]>0 && !onpath[i])

{

onpath [i] = TRUE;

endpath[i] = TRUE;

precede[i] = nd;

forward[nd] = FALSE;

**if** ( improve[nd]<flow[i][nd] )

improve[i] = improve[nd];

**else** improve[i] = flow[i][nd];

}

}

}

**if** ( onpath[t] )

**// Поток на полупути к t может быть увеличен.**

{

x = improve[t];

\*totflow += x;

nd = t;

**while** ( nd!=s )

**// Возвращение обратно вдоль пути.**

{

pred = precede [nd];

**if** ( forward[pred] )

**// Увеличение потока от pred.**

flow[pred][nd] += x;

**else** **// Уменьшение потока от pred.**

flow[pred][nd] -= x;

nd = pred;

}

}

} **while** ( onpath[t] );

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din122_1.zip).

***Система дорог*** - это размеченный мультиграф (без петель), который отличается от графа тем, что в нем одна и та же пара (различных) вершин может быть связана более чем одним ребром [3, с.97-98]. При этом вершины соответствуют городам, а ребра - дорогам. Односторонним дорогам соответствуют дуги, а двусторонним дорогам - ребра.

Каждая дорога имеет некоторую ***длину*** - положительное вещественное число. Понятие пути, достижимости и замкнутого пути определяются для системы дорог аналогично подобным понятиям для графа и ориентированного графа.

***Длина пути в системе дорог*** - это сумма длин дорог этого пути.

***Расстояние между двумя городами*** - это длина минимального пути между этими городами.

(1) Tenenbaum A., Augenstein M. Data Structures Using Pascal. Englewood Cliffs. - N.Y.: Prentice-Hall, Inc. 1981.

(2) Лэнгсам Й., Огенстейн М., Тененбаум А. Структуры данных для персональных ЭВМ: Пер. с англ. - М.: Мир, 1989. - 568 с.

(3) Касьянов В.H., Сабельфельд В.К. Сборник заданий по практикуму на ЭВМ. - М.: Hаука, 1986. - 272 с.

На этом шаге мы намеревались закончить изложение материала, связанного со структурами данных. Однако определенная недосказанность последних шагов вынудила нас отыскать дополнительный материал по сетям. Мы не будем менять ранее опубликованные шаги, а в следующих шагах вернемся к ранее изложенному материалу и рассмотрим его более подробно.

Таким образом, на следующем шаге мы подробно рассмотрим алгоритм ***Дейкстры***.

## Описание алгоритма Дейкстры

На этом шаге мы приведем ***описание алгоритма Дейкстры***.

Напомним, что алгоритм Дейкстры разработан для нахождения кратчайшего пути между заданным исходным узлом и любым другим узлом сети.

В процессе выполнения этого алгоритма при переходе от узла **i** к следующему узлу **j** используется специальная процедура пометки ребер. Обозначим через **ui** кратчайшее расстояние от исходного узла 1 до узла **i**, через **dij** - длину ребра **(i, j)**. Тогда для узла **j** определим метку **[uj, i]** следующим образом:

[uj, i] = [ui + dij, i], dij >= 0

Метки узлов в алгоритме Дейкстры могут быть двух типов: ***временные*** и ***постоянные***. Временная метка впоследствии может быть заменена на другую временную, если будет найден ***более короткий путь*** к данному узлу. Когда же станет очевидным, что не существует более короткого пути от исходного узла к данному, статус временной метки изменяется на постоянный.

Вычислительная схема алгоритма состоит из следующих шагов.

***Шаг 0.*** Исходному узлу (узел 1) присваивается метка **[0, -]**. Полагаем **i** = 1.

***Шаг i.*** а) Вычисляются временные метки **[ui + dij, i]** для всех узлов **j**, которые можно достичь непосредственно из узла **i** и которые не имеют постоянных меток. Если узел **j** уже имеет метку **[uj, k]**, полученную от другого узла **k**, и если **ui + dij < uj**, тогда метка **[uj, k]** заменяется на **[ui + dij, i]**.

b) Если все узлы имеют постоянные метки, процесс вычислений заканчивается. В противном случае выбирается метка **[ur, s]** с наименьшим значением расстояния **ur** среди всех временных меток (если таких меток несколько, то выбор произволен). Полагаем **i = r** и повторяем шаг **i**.

Пример. На рисунке 1 показана транспортная сеть, состоящая из пяти городов (расстояния между городами (в километрах) приведены возле соответствующих дуг сети). Необходимо найти кратчайшие расстояния от города 1 (узел 1) до всех остальных четырех городов.

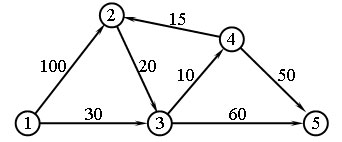


Рис.1. Транспортная сеть

***Шаг 0.*** Назначаем узлу 1 постоянную метку **[0, -]**.

***Шаг 1.*** Из узла 1 можно достичь узлов 2 и 3. Вычисляем метки для этих узлов, в результате получаем следующую таблицу меток:

Узел Метка Статус метки

1 **[0, -]** **Постоянная**

2 [0 + 100, 1] = [100, 1] Временная

3 [0 + 30, 1] = [30, 1] <-Временная

Среди узлов 2 и 3 узел 3 имеет наименьшее значение расстояния **(u3 = 30)**. Поэтому статус метки этого узла изменяется на ***"постоянная"***.

***Шаг 2.*** Из узла 3 (последнего узла с постоянной меткой) можно попасть в узлы 4 и 5. Получаем следующий список узлов:

Узел Метка Статус метки

1 **[0, -]** **Постоянная**

2 [100, 1] Временная

3 **[30, 1]** **Постоянная**

4 [30 + 10, 3] = [40, 3] <-Временная

5 [30 + 60, 3] = [90, 3] Временная

Временный статус метки [40, 3] узла 4 заменяется на постоянный **(u4 = 40)**.

***Шаг 3.*** Из узла 4 можно достичь узлов 2 и 5. После вычисления меток получим следующий их список:

Узел Метка Статус метки

1 **[0, -]** **Постоянная**

2 [40 + 15, 4] = [55, 4] <-Временная

3 **[30, 1]** **Постоянная**

4 **[40, 3]** **Постоянная**

5 [90, 3] или [40 + 50, 4] = [90, 4] Временная

Временная метка [100, 1], полученная узлом 2 на втором шаге, изменена на [55, 4]. Это указывает на то, что найден более короткий путь к этому узлу (проходящий через узел 4). На третьем шаге узел 5 получает две метки с одинаковым значением расстояния **u5 = 90**.

***Шаг 4.*** Из узла 2 можно перейти только в узел 3, но он уже имеет постоянную метку, которую нельзя изменить. Поэтому на данном шаге получаем такой же список меток, как и на предыдущем шаге, но с единственным изменением: метка узла 2 получает статус постоянной. С временной меткой остается только узел 5, но так как из этого узла нельзя попасть ни в какой другой, процесс вычислений заканчивается.

Алгоритм позволяет проводить вычисления непосредственно по схеме сети, как показано на рисунке 2.

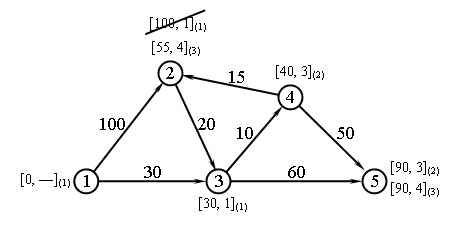


Рис.2. Вычисления по схеме (в скобках указан номер шага)

Кратчайший маршрут между узлом 1 и любым другим узлом определяется начиная с узла назначения путем прохождения их в обратном направлении с помощью информации, представленной в постоянных метках. Например, для определения кратчайшего маршрута между узлами 1 и 2 получаем такую обратную последовательность узлов:

(2) -> [55, 4] -> (4) -> [40, 3] -> (3) -> [30, 1] -> (1).

Таким образом, получаем путь **1->3->4->2** общей длиной 55 километров.

Проиллюстрируем описанный алгоритм программой, приведенной на [120 шаге](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0120.html). Для разобранного примера результат работы этой программы будет следующим:

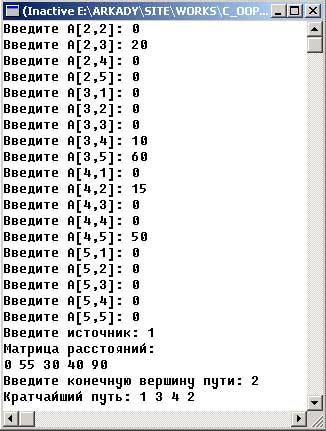


Рис.3. Результат работы приложения

На следующем шаге мы рассмотрим ***алгоритм Флойда***.

## Алгоритм Флойда

На этом шаге мы рассмотрим ***алгоритм Флойда***.

Этот алгоритм более общий по сравнению с алгоритмом Дейкстры, так как он находит кратчайшие пути между любыми двумя узлами сети. В этом алгоритме сеть представлена в виде квадратной матрицы с **n** строками и **n**столбцами. Элемент **(i, j)** равен расстоянию **dij** от узла **i** к узлу **j**, которое имеет конечное значение, если существует дуга **(i, j)**, и равен бесконечности в противном случае.

Покажем сначала основную идею метода Флойда. Пусть есть три узла **i, j** и **k** и заданы расстояния между ними (рис. 1). Если выполняется неравенство **dij + djk < dik**, то целесообразно заменить путь **i -> k** путем **i -> j -> k**. Такая замена (далее ее будем условно называть ***треугольным оператором***) выполняется систематически в процессе выполнения алгоритма Флойда.

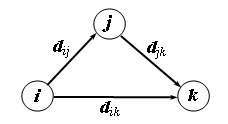


Рис.1. Треугольный оператор

Алгоритм Флойда требует выполнения следующих действий.

***Шаг 0.*** Определяем начальную матрицу расстояния **D0** и матрицу последовательности узлов **S0**. Диагональные элементы обеих матриц помечаются знаком "-", показывающим, что эти элементы в вычислениях не участвуют. Полагаем **k = 1**:

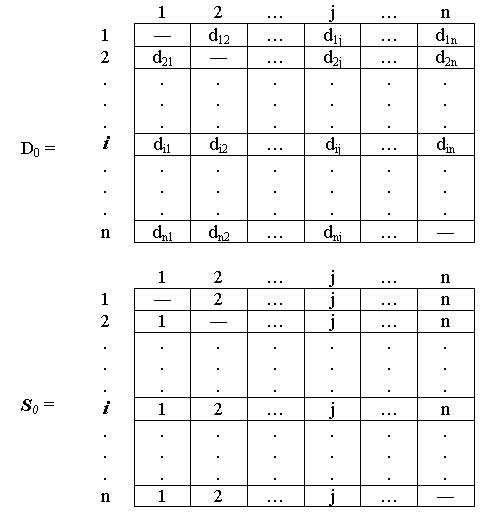


Рис.2. Начальная ситуация

***Основной шаг k.*** Задаем строку **k** и столбец **k** как ***ведущую строку*** и ***ведущий столбец***. Рассматриваем возможность применения треугольного оператора ко всем элементам **dij** матрицы **Dk-1**. Если выполняется неравенство **dik + dkj < dij, (i <> k, j <> k, i <> j)**, тогда выполняем следующие действия:

* создаем матрицу **Dk** путем замены в матрице **Dk-1** элемента **dij** на сумму **dik + dkj**,
* создаем матрицу **Sk** путем замены в матрице **Sk-1** элемента **sij** на **k**. Полагаем **k = k + 1** и повторяем шаг **k**.

Поясним действия, выполняемые на **k**-м шаге алгоритма, представив матрицу **Dk-1** так, как она показана на рисунке 3. На этом рисунке строка **k** и столбец **k** являются ведущими. Строка **i** - любая строка с номером от 1 до **k - 1**, а строка **p** - произвольная строка с номером от **k + 1** до **n**. Аналогично столбец **j** представляет любой столбец с номером от 1 до **k - 1**, столбец **q** - произвольный столбец с номером от **k + 1** до **n**. Треугольный оператор выполняется следующим образом. Если сумма элементов ведущих строки и столбца (показанных в квадратах) меньше элементов, находящихся в пересечении столбца и строки (показанных в кружках), соответствующих рассматриваемым ведущим элементам, то расстояние (элемент в кружке) заменяется на сумму расстояний, представленных ведущими элементами:

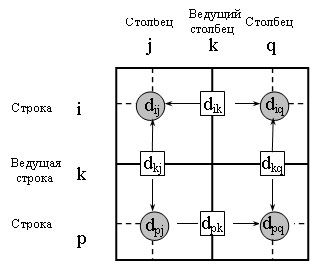


Рис.3. Иллюстрация алгоритма Флойда

После реализации **n** шагов алгоритма определение по матрицам **Dn** и **Sn** кратчайшего пути между узлами **i** и **j**выполняется по следующим правилам.

1. Расстояние между узлами **i** и **j** равно элементу **dij** в матрице **Dn**.
2. Промежуточные узлы пути от узла **i** к узлу **j** определяем по матрице **Sn**. Пусть **sij = k**, тогда имеем путь **i -> k -> j**. Если далее **sik = k** и **skj = j**, тогда считаем, что весь путь определен, так как найдены все промежуточные узлы. В противном случае повторяем описанную процедуру для путей от узла **i** к узлу **k** и от узла **k** к узлу **j**.

Пример. Найдем для сети, показанной на рисунке 4, кратчайшие пути между любыми двумя узлами. Расстояние между узлами этой сети проставлены на рисунке возле соответствующих ребер. Ребро (3, 5) ориентированно, поэтому не допускается движение от узла 5 к узлу 3. Все остальные ребра допускают движение в обе стороны:

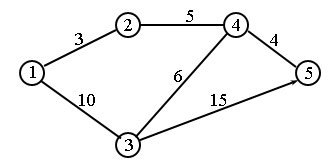


Рис.4. Пример сети

***Шаг 0.*** Начальные матрицы **D0** и **S0** строятся непосредственно по заданной схеме сети. Матрица **D0** симметрична, за исключением пары элементов **d35** и **d53**, где **d53** равно бесконечности, поскольку невозможен переход от узла 5 к узлу 3:

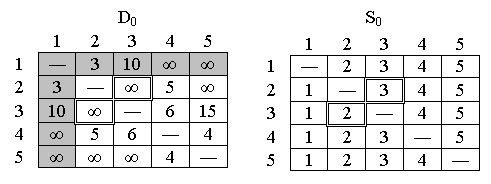


Рис.5. Начальное состояние

***Шаг 1.*** В матрице **D0** выделены ведущие строка и столбец **(k = 1)**. Двойной рамкой представлены элементы **d23** и **d32**, единственные среди элементов матрицы **D0**, значения которых можно улучшить с помощью треугольного оператора. Таким образом, чтобы на основе матриц **D0** и **S0** получить матрицы **D1** и **S1**, выполняем следующие действия.

1. Заменяем **d23** на **d21 + d13 = 3 + 10 = 13** и устанавливаем **s23 = 1**.
2. Заменяем **d32** на **d31 + d12 = 10 + 3 = 13** и устанавливаем **s32 = 1**.

Матрицы **D1** и **S1** имеют следующий вид:

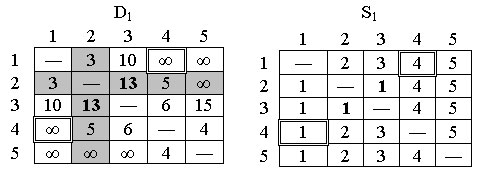


Рис.6. Матрицы **D1** и **S1**

***Шаг 2.*** Полагаем **k = 2**; в матрице **D1** выделены ведущие строка и столбец. Треугольный оператор применяется к элементам матрицы **D1** и **S1**, выделенным двойной рамкой. В результате получаем матрицы **D2** и **S2**:

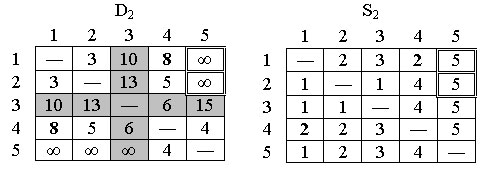


Рис.7. Матрицы **D2** и **S2**

***Шаг 3.*** Полагаем **k = 3**; в матрице **D2** выделены ведущие строка и столбец. Треугольный оператор применяется к элементам матрицы **D2** и **S2**, выделенным двойной рамкой. В результате получаем матрицы **D3** и **S3**:

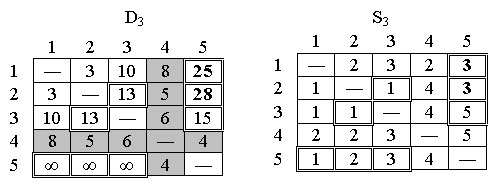


Рис.8. Матрицы **D3** и **S3**

***Шаг 4.*** Полагаем **k = 4**, ведущие строка и столбец в матрице **D3** выделены. Получаем новые матрицы **D4** и **S4**:

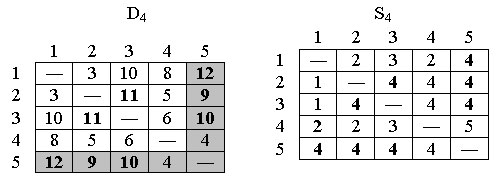


Рис.9. Матрицы **D4** и **S4**

***Шаг 5.*** Полагаем **k = 5**, ведущие строка и столбец в матрице **D4** выделены. Никаких действий на этом шаге не выполняем; вычисления закончены.

Конечные матрицы **D4** и **S4** содержат всю информацию, необходимую для определения кратчайших путей между любыми двумя узлами сети. Например, кратчайшее расстояние между узлами 1 и 5 равно **d15 = 12**.

Для нахождения соответствующих маршрутов напомним, что сегмент маршрута **(i, j)** состоит из ребра **(i, j)** только в том случае, когда **sij = j**. В противном случае узлы **i** и **j** связаны, по крайней мере, через один промежуточный узел. Например, поскольку **s15 = 4** и **s45 = 5**, сначала кратчайший маршрут между узлами 1 и 5 будет иметь вид **1->4->5**. Но так как **s14** не равно 4, узлы 1 и 4 в определенном пути не связаны одним ребром (но в исходной сети они могут быть связаны непосредственно). Далее следует определить промежуточный узел (узлы) между первым и четвертым узлами. Имеем **s14 = 2** и **s24 = 4**, поэтому маршрут **1->4** заменяем **1->2->4**. Поскольку **s12 = 2** и **s24 = 4**, других промежуточных узлов нет. Комбинируя определенные сегменты маршрута, окончательно получаем следующий кратчайший путь от узла 1 до узла 5: **1->2->4->5**. Длина этого пути равна 12 километрам.

Приведем текст программы, реализующей алгоритм Флойда.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** MaxNodes 5 **//Количество вершин.**

**//Описание типа узла стека.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

**int** Element;

svqz Sled;

};

**class** Spisok

{

**private**:

**int** Mas[MaxNodes][MaxNodes]; **//Матрица весов дуг.**

**int** DD[MaxNodes][MaxNodes]; **//Матрица расстояний.**

**int** SS[MaxNodes][MaxNodes]; **//Матрица последовательных узлов.**

svqz Stack; **//Указатель на рабочий стек.**

**void** UDALENIE (svqz \*, **int** \*);

**void** W\_S (svqz \*, **int**);

**void** Small\_Put (int,int);

**public**:

Spisok() {Stack = **NULL**;}

**void** Vvod\_Ves();

**void** Reshenie ();

};

**void** main()

{

Spisok A;

A.Vvod\_Ves();

A.Reshenie();

}

**void** Spisok::Small\_Put (**int** one, **int** two)

**//Нахождение кратчайшего пути.**

{

svqz St=NULL; **//Указатель на вспомогательный стек.**

svqz UkZv;

**int** Flag=FALSE; **//Флаг построения кратчайшего пути.**

**int** elem1,elem2,k;

**//Помещение в стек конечной и начальной вершин.**

W\_S (&Stack,two);

W\_S (&Stack,one);

**while** (!Flag)

{

**//Извлекли верхних два элемента.**

UDALENIE(&Stack,&elem1);

UDALENIE(&Stack,&elem2);

**if** (SS[elem1][elem2]==elem2) **//Если есть путь...**

**if** (elem2==two) **//и это конечный узел...**

{

Flag = TRUE; **//то кратчайший путь найден.**

W\_S (&St,elem1);

W\_S (&St,elem2);

}

**else** **//и это не конечный узел...**

{

W\_S (&St,elem1); **//В вспомогательный стек.**

W\_S (&Stack,elem2); **//Обратно в рабочий стек.**

}

**else** **//Если пути нет.**

{

W\_S (&Stack,elem2); **//Обратно в рабочий стек.**

k = SS[elem1][elem2];

W\_S (&Stack,k); **//Запомнить промежуточную вершину.**

W\_S (&Stack,elem1); **//Обратно в рабочий стек.**

}

}

UkZv = St;

**while** ( UkZv != **NULL** )

{ cout << (UkZv->Element+1) << " ";

UkZv = UkZv->Sled; }

cout << endl;

}

**void** Spisok::W\_S (svqz \*stk, **int** Elem)

**//Помещение Elem в стек stk.**

{

svqz q=new (Zveno);

(\*q).Element = Elem;

(\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::UDALENIE (svqz \*stk, **int** \*Klad)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметре Klad.**

{

svqz q;

**if** (\*stk==NULL) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*Klad = (\*\*stk).Element;

q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

**void** Spisok::Vvod\_Ves()

**//Ввод матрицы весов дуг заданного графа.**

{

cout << "Вводите элементы матрицы весов дуг по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout << "Введите Mas[" << (i+1) << "," << (j+1) << "]: ";

cin >> Mas[i][j];

}

}

**void** Spisok::Reshenie()

{

**int** one,two;

**int** i,j;

**//Инициализация.**

**for** (i=0;i<MaxNodes;i++)

**for** (j=0;j<MaxNodes;j++)

{

**if** (Mas[i][j]>0) SS[i][j]=j;

**else** SS[i][j]=0;

DD[i][j]=Mas[i][j];

}

cout << "\nНачальная вершина: ";

cin >> one; one--;

cout << "Конечная вершина: ";

cin >> two; two--;

**int** ved=0;

**while** (ved<MaxNodes)

{

**for** (i=0;i<MaxNodes;i++)

**for** (j=0;j<MaxNodes;j++)

**if** (i!=j && i!=ved && j!=ved &&

DD[i][ved]>0 && DD[ved][j]>0)

**if** (DD[i][ved]+DD[ved][j]<DD[i][j] || DD[i][j]==0)

{

DD[i][j]=DD[i][ved]+DD[ved][j];

SS[i][j]=ved;

}

ved++;

}

i=one;

**if** (SS[i][two]!=two && SS[i][two]!=0)

**while** (SS[i][two]!=two)

{

j=SS[i][two];

**while** (SS[i][j]!=j) j=SS[i][j];

i=j;

}

cout << "\nКратчайший путь (в обратном порядке): ";

Small\_Put (one, two);

cout << "Длина минимального пути между этими вершинами: " << DD[one][two];

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din124_1.zip).

Результат работы этой программы для сети, изображенной на рисунке 4, будет следующим:

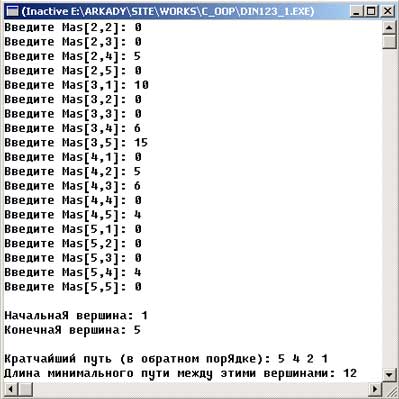


Рис.10. Результат работы приложения

На следующем шаге мы рассмотрим ***задачу о максимальном потоке***.

## Задача о максимальном потоке

На этом шаге мы сформулируем ***задачу о максимальном потоке***.

Рассмотрим сеть трубопроводов для транспортировки сырой нефти от буровых скважин до нефтеперегонных заводов. Для перекачки нефти предусмотрены магистральные насосные станции. Каждый сегмент трубопровода имеет свою пропускную способность. Сегменты трубопровода могут быть как однонаправленные (осуществляют перекачку нефти только в одном направлении), так и в двунаправленные. В однонаправленных сегментах положительная пропускная способность предполагается в одном направлении и нулевая - в другом. На рис. 1 показана типовая сеть нефтепроводов. Как определить оптимальную пропускную способность (т.е. максимальный поток) между нефтяными скважинами и нефтеперегонными заводами?

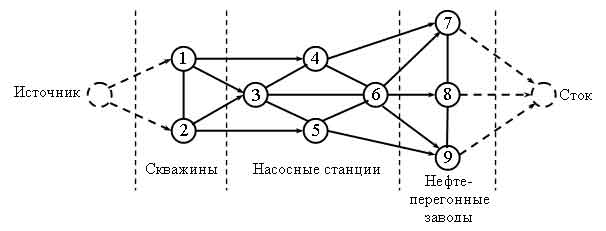


Рис.1. Типовая сеть нефтепроводов

При решении данной задачи исходную сеть необходимо свести к сети с одним источником и одним стоком. Этого можно достигнуть путем введения дополнительных дуг с бесконечной пропускной способностью от источника к скважинам и от нефтеперегонных заводов к стоку (на рис. 1 эти дуги показаны пунктирными линиями).

Для ребра **(i, j)**, где **i < j**, используем запись **(Cij,Cji)** для представления пропускных способностей в направлениях **i->j** и **j->i** соответственно. Во избежании недоразумений на схеме сети **Cij** будем располагать на ребре **(i, j)** ближе к узлу **i**, а **Cji** ближе к узлу **j**, как показано на рис. 2.



Рис.2. Иллюстрация пропускной способности

***Перебор разрезов***

***Разрез*** определяет множество ребер, при удалении которых из сети полностью прекращается поток от источника к столу. ***Пропускная способность разреза*** равна сумме пропускных способностей "разрезанных" ребер. Среди ***всех***разрезов сети разрез с ***минимальной пропускной способностью*** определяет максимальный поток в сети.

Пример. Рассмотрим сеть, показанную на рис. 3. На этом рисунке при обозначении пропускных способностей двунаправленных ребер придерживались соглашения, принятого ранее (рис. 2). Например, для ребра (3, 4) пропускная способность в направлении **3 -> 4** равна 10, а в направлении **4 -> 3** равна 5.

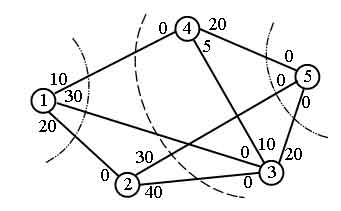


Рис.3. Пример сети

Разрезы, представленные на рис. 3, имеют следующие пропускные способности:

Разрез "Разрезанные" ребра Пропускная способность

1 (1, 2), (1, 3), (1, 4) 10 + 30 + 20 = 60

2 (1, 3), (1, 4), (2, 3), (2, 5) 30 + 10 + 40 + 30 = 110

3 (2, 5), (3, 5), (4, 5) 30 + 20 + 20 = 70

Вывод, который можно сделать из этих трех разрезов, заключается в том, что максимальный поток не может превышать 60 единиц. Но мы не можем сказать, какой максимальный поток на самом деле, так как не перебрали ***все возможные разрезы*** сети. К сожалению, перебор всех разрезов является непростой задачей. Поэтому для определения максимального потока в сети не используются алгоритмы, основанные на полном переборе разрезов.

На следующем шаге мы рассмотрим ***алгоритм нахождения максимального потока***.

## Алгоритм нахождения максимального потока

На этом шаге мы реализуем ***алгоритм нахождения максимального потока***.

Идея данного алгоритма состоит в нахождении сквозных путей с положительными потоками от источника к стоку.

Рассмотрим ребро **(i, j)** с (начальной) пропускной способностью **(Cij,Cji)**. В процессе выполнения алгоритма части этих пропускных способностей "забираются" потоками, проходящими через данное ребро, в результате каждое ребро будет иметь остаточную пропускную способность. Будем использовать запись **(cij, cji)** для представления остаточных пропускных способностей. Сеть, где все ребра имеют остаточную пропускную способность, назовем ***остаточной***.

Для произвольного узла **j**, получающего поток от узла **i**, определим метку **[aj, i]**, где **aj** - величина потока, протекающего от узла **j** к узлу **i**. Алгоритм нахождения максимального потока предполагает выполнение следующих действий.

***Шаг 1.*** Для всех ребер **(i, j)** положим остаточную пропускную способность равной первоначальной пропускной способности, т.е. приравняем **(cij, cji) = (Cij, Cji)**. Назначим **a1 = *бесконечности*** и пометим узел 1 меткой **[*бесконечность*, -]**. Полагаем **i = 1** и переходим ко второму шагу.

***Шаг 2.*** Определяем множество **Si** как множество узлов **j**, в которые можно перейти из узла **i** по ребру с положительной остаточной пропускной способностью (т.е. **cij > 0** для всех **j**, принадлежащих **Si**). Если **Si** не пустое множество, выполняем третий шаг, в противном случае переходим к шагу 4.

***Шаг 3.*** В множестве **Si** находим узел **k**, такой, что **cik = max {cij}** для всех **j**, принадлежащих **Si**. Положим **ak = cik** и пометим узел **k** меткой **[ak, i]**. Если последней меткой помечен узел стока (т.е. если **k = n**), сквозной путь найден, и мы переходим к пятому шагу.

***Шаг 4 (Откат назад).*** Если **i = 1**, сквозной путь невозможен, и мы переходим к шагу 6. Если **i** не равно 1, находим помеченный узел **r**, непосредственно предшествующий узлу **i**, и удаляем узел **i** из множества узлов, смежных с узлом **r**. Полагаем **i = r** и возвращаемся ко второму шагу.

***Шаг 5 (Определение остаточной сети).*** Обозначим через **Np = {1, k1, k2, …, n}** множество узлов, через которые проходит **p**-й найденный сквозной путь от узла источника (узел 1) до узла стока (узел **n**). Тогда максимальный поток, проходящий по этому пути, вычисляется как **fp = min {a1, ak1, ak2, ..., an}**.

Остальные пропускные способности ребер, составляющих сквозной путь, уменьшается на величину **fp** в направлении движения потока и увеличиваются на эту же величину в противоположном направлении. Таким образом, для ребра **(i, j)**, входящего в сквозной путь, текущие остаточные стоимости **(cij, cji)** изменятся следующим образом:

* **(cij - fp, cji + fp)**, если поток идет от узла **i** к узлу **j**,
* **(cij + fp, cji - fp)**, если поток идет от узла **j** к узлу **i**.

Далее восстанавливаем все узлы, удаленные на шаге 4. полагаем **i = 1** и возвращаемся ко второму шагу для поиска нового сквозного пути.

***Шаг 5 (Решение).***

* При **m** найденных сквозных путях максимальный поток вычисляется по формуле **F = f1 + f2 + ... + fm**.
* Имея значения начальных **(Cij, Cji)** и конечных **(cij, cji)** пропускных способностей ребра **(i, j)**, можно вычислить оптимальный поток через это ребро следующим образом. Положим **(a,b) = (Cij - cij, Cji - cji)**. Если **a > 0**, поток, проходящий через ребро **(i, j)**, равен **a**. Если же **b > 0**, тогда поток равен **b**. (Случай, когда одновременно **a > 0** и**b > 0**, невозможен.)

Процесс отката назад на четвертом шаге выполняется тогда, когда алгоритм должен "убить" промежуточный узел до момента реализации сквозного пути. Коррекцию пропускных способностей, выполняемых на шаге 5, можно пояснить на примере простой сети, показанной на рис. 1. На рис. 1а найден первый сквозной путь **N1 = {1, 2, 3, 4}** с максимальным потоком **f1 = 5**. После этого остаточные пропускные способности ребер (1, 2), (2, 3) и (3, 4) изменятся соответственно с (5, 0) на (0, 5). На рис. 1б показан второй сквозной путь **N2 = {1, 3, 2, 4}** с максимальным потоком **f2 = 5**. После коррекции пропускных способностей получаем сеть, показанную на рис. 1в, где уже невозможно построить сквозной путь. Почему так получилось? При вычислении остаточных пропускных способностей на шаге 5 при переходе от сети (б) к сети (в) невозможна организация потока в направлении **2 -> 3**. Получается, что алгоритм как бы "помнит", что поток в направлении **2 -> 3** уже был в предыдущих сквозных путях, и поэтому снова (на пятом шаге) изменяет пропускную способность с 0 до 5 в направлении от узла 3 к узлу 2.

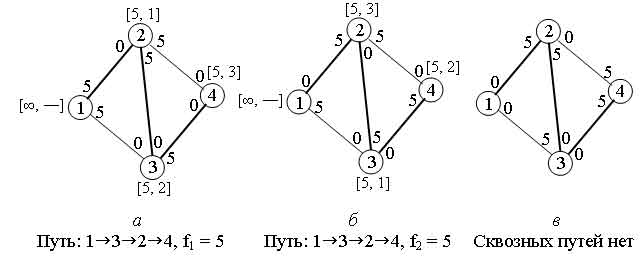


Рис.1. Иллюстрация сквозных путей

Пример. Найдем максимальный поток в сети из [примера предыдущего шага](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0125.html#1). На рисунке 2 предлагается графическая иллюстрация выполнения алгоритма. Считаем полезным сравнить описание выполняемых алгоритмом вычислительных итераций с их графическим представлением.

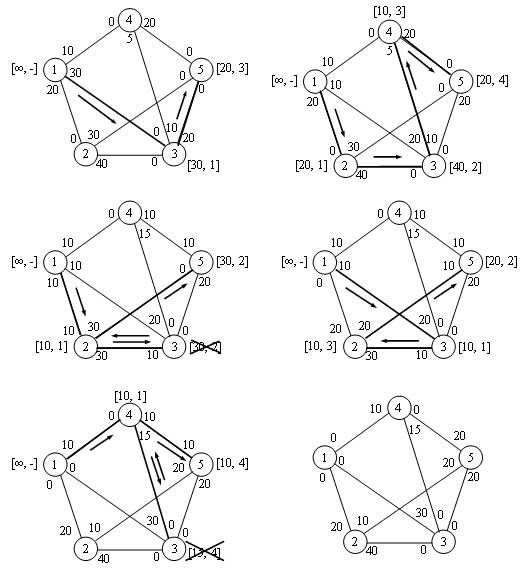


Рис.2. Графическая иллюстрация выполнения алгоритма

***Итерация 1.***

Положим остаточные пропускные способности **(cij, cji)** всех ребер равными первоначальным пропускным способностям **(Cij, Cji)**.

***Шаг 1.*** Назначаем **a1 = *бесконечности*** и помечаем узел 1 меткой **[*бесконечность*,-]**. Полагаем **i = 1**.

***Шаг 2.*** **S1 = [2, 3, 4]** (множество не пустое).

***Шаг 3.* k = 3**, поскольку **c13 = max {c12, c13, c14} = max {20, 30, 10} = 30**. Назначаем **a3 = c13 = 30** и помечаем узел 3 меткой **[30, 1]**. Полагаем **i = 3** и возвращаемся к шагу 2.

***Шаг 2.*** **S2 = [4, 5]**.

***Шаг 3.*** **k = 5** и **a5 = c35 = max {10, 20} = 20**. Помечаем узел 5 меткой **[20,3]**. Получен сквозной путь. Переходим к шагу 5.

***Шаг 5.*** Сквозной путь определяем по меткам, начиная с узла 5 и заканчивая узлом 1: **(5) -> [20, 3] -> (3) -> [30, 1] -> (1)**. Таким образом, **N1 = {1, 3, 5}** и **f1 = min {a1, a3, a5} = {*бесконечность*, 30, 20} = 20**. Вычисляем остаточные пропускные способности вдоль пути **N1**:

(c13, c31) = (30 - 20, 0 + 20) = (10, 20),

(c35, c53) = (20 - 20, 0 + 20) = (0, 20).

***Итерация 2.***

***Шаг 1.*** Назначаем **a1 = *бесконечности*** и помечаем узел 1 меткой **[*бесконечность*,-]**. Полагаем **i = 1**.

***Шаг 2.*** **S1 = [2, 3, 4]**.

***Шаг 3.*** **k = 2**, назначаем **a2 = c12 = max {20, 10, 10} = 20** и помечаем узел 2 меткой **[20, 1]**. Полагаем **i =2** и возвращаемся к шагу 2.

***Шаг 2.*** **S3 = [4]** (отметим, что **c35 = 0**, поэтому узел 5 не включается в **S3**).

***Шаг 3.*** **k = 4**, назначаем **a4 = c34 = 10** и помечаем узел 4 меткой **[10, 3]**. Полагаем **i = 4** и возвращаемся к шагу 2.

***Шаг 2.*** **S4 = [5]** (поскольку узлы 1 и 3 уже помечены, они не включаются в **S4**).

***Шаг 3.*** **k = 5** и **a5 = c45 = 20**. Помечаем узел 5 меткой **[20, 4]**. Получаем сквозной путь. Переходим к шагу 5.

***Шаг 5.*** **N2 = {1, 2, 3, 4, 5}** и **f2 = min {*бесконечность*, 20, 40, 10, 20} = 10**. Вычисляем остаточные пропускные способности вдоль пути **N2**:

(c12, c21) = (20 - 10, 0 + 10) = (10, 10),

(c*23*, c32) = (40 - 10, 0 + 10) = (30, 10),

(c34, c43) = (10 - 10, 5 + 10) = (0, 15),

(c45, c54) = (20 - 10, 0 + 10) = (10, 10).

***Итерация 3.***

***Шаг 1.*** Назначаем **a1 = *бесконечности*** и помечаем узел 1 меткой **[*бесконечность*,-]**. Полагаем **i = 1**.

***Шаг 2.*** **S1 = [2, 3, 4]**.

***Шаг 3.*** **k = 2**, назначаем **a2 = c12 = max {10, 10, 10} = 10** и помечаем узел 2 меткой **[10, 1]**. Полагаем **i = 2** и возвращаемся к шагу 2.

***Шаг 2.*** **S2 = [3, 5]**.

***Шаг 3.*** **k = 3** и **a3 = c23 = 30**. Помечаем узел 3 меткой **[30, 2]**. Полагаем **i = 3** и возвращаемся к шагу 2.

***Шаг 2.*** **S3** пусто, поскольку **c34 = c35 = 0**. Переходим к шагу 4.

***Шаг 4.*** Метка **[30, 2]** узла 3 показывает номер предшествующего узла **r = 2**. На этой итерации узел 3 в дальнейшем во внимание не принимается, его метку вычеркиваем. Полагаем **i = r = 2** и возвращаемся к шагу 2.

***Шаг 2.*** **S4 = [5]** (поскольку узел 3 удален из возможного сквозного пути).

***Шаг 3.*** **k = 5** и **a5 = c25 = 30**. помечаем узел меткой **[30, 2]**. Получаем сквозной путь. Переходим к шагу 5.

***Шаг 5.*** **N3 = {1, 2, 5}** и **f3 = min {*бесконечность*, 10, 30} = 10**. Вычисляем остаточные пропускные способности вдоль пути **N3**:

(c12, c21) = (10 - 10, 10 + 10) = (0, 20),

(c2, c52) = (30 - 10, 0 + 10) = (20, 10).

***Итерация 4.*** На этой итерации получен путь **N4 = {1, 3, 2, 5}** с **f4 = 10**.

***Итерация 5.*** На этой итерации получен путь **N5 = {1, 4, 5}** с **f5 = 10**.

***Итерация 6.*** Новые сквозные пути невозможны, поскольку все ребра, исходящие из узла 1, имеют нулевые остаточные пропускные способности. Переходим к шагу 6 для определения решения.

***Шаг 6.*** Максимальный объем потока в сети равен **F = f1 + f2 + … +f5 = 20 + 10 + 10 + 10 + 10 = 60** единиц. Значения потоков по различным ребрам вычисляются путем вычитания последних значений остаточных пропускных способностей (т.е. **(cij, cji)6**) из первоначальных значений пропускных способностей **(Cij, Cji)**. Результаты вычислений приведены ниже:

Ребро (Cij, Cji)-(cij, cji)6 Величина потока Направление

(1, 2) (20, 0) - (0, 20) = (20, -20) 20 1->2

(1, 3) (30, 0) - (0, 30) = (30, -30) 30 1->3

(1, 4) (10, 0) - (0, 10) = (10, -10) 10 1->4

(2, 3) (40, 0) - (40, 0) = (0, 0) 0 -

(2, 5) (30, 0) - (10, 20) = (20, -20) 20 2->5

(3, 4) (10, 5) - (0, 15) = (10, -10) 10 3->4

(3, 5) (20, 0) - (0, 20) = (20, -20) 20 3->5

(4, 5) (20, 0) - (0, 20) = (20, -20) 20 4->5

Приведем текст программы, решающей данную задачу:

**#include** <iostream.h>

**#include** <stdlib.h> **//Для функции abs().**

**#define** TRUE 1

**#define** FALSE 0

**#define** MaxNodes 5 **//Количество вершин.**

**#define** MaxInt 1000 **//Машинный эквивалент бесконечности.**

**//Описание типа узла.**

**struct** Uzel

{

**int** Element; **//Заданный номер вершины.**

**int** Propusk; **//Пропускная способность.**

**int** Metka; **//Помечена вершина или нет.**

};

**class** Spisok

{

**private**:

**int** C[MaxNodes][MaxNodes]; **//Матрица начальных пропускных способностей.**

**int** c[MaxNodes][MaxNodes]; **//Матрица конечных пропускных способностей.**

**int** Put[MaxNodes][MaxNodes]; **//Матрица сквозных путей.**

**int** Potok [MaxNodes]; **//Потоки.**

**int** Est (Uzel\*,int,int);

int Tpk (int\*,int,int);

**public**:

**void** Vvod\_Ves();

**int** Reshenie ();

**void** Vyvod(**int**);

};

**void** main()

{

Spisok A;

A.Vvod\_Ves();

A.Vyvod(A.Reshenie());

}

**void** Spisok::Vvod\_Ves()

**//Ввод матрицы пропускных способностей.**

{

cout << "Вводите пропускные способности ребер:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout << "Введите C[" << (i+1) << "," << (j+1) << "]: ";

cin >> C[i][j];

c[i][j] = C[i][j];

}

}

**void** Spisok::Vyvod(**int** n)

**//Вывод результатов.**

{

**//Вычисление максимального объема потока.**

**for** (**int** i=0,sum=0;i<=n;sum+=Potok[i++]);

cout << "\nМаксимальный объем потока в сети: " << sum;

cout << "\nЗначения потоков по различным ребрам:\n";

**for** (i=0;i<MaxNodes;i++)

**for** (**int** j=i;j<MaxNodes;j++)

**if** (C[i][j])

{

cout << "Ребро (" << (i+1) << "," << (j+1) <<"): ";

cout << "(" << C[i][j] << "," << C[j][i] << ")-(";

cout << c[i][j] << "," << c[j][i] << ")=(";

cout << (C[i][j]-c[i][j]) << "," << (C[j][i]-c[j][i]) << ") ";

cout << "Поток: " << abs(C[i][j]-c[i][j]) << " ";

**if** (C[i][j]-c[i][j]!=0)

{

cout << "Направление: ";

**if** (C[i][j]-c[i][j]>0)

cout << (i+1) << "->" << (j+1);

**else**

cout << (j+1) << "->" << (i+1);

}

cout << endl;

}

}

**int** Spisok::Reshenie()

{

Uzel SS[MaxNodes]; **//Множество узлов, в которые можно перейти.**

Uzel S[MaxNodes]; **//Путь.**

**int** i,j=0,k,el,mx,mn;

**int** m; **//Текущее количество вершин в пути.**

**int** nomer=-1; **//Текущее количество сквозных потоков.**

**int** Tupik[MaxNodes]; **//Перечень "тупиковых" вершин.**

**int** N\_Tupik; **//Количество элементов в массиве.**

**while** (j!=-1)

{

i=m=0;

S[i].Element=0;

S[i].Propusk=MaxInt;

S[i].Metka=TRUE;

el=0;

N\_Tupik=-1;

**while** (el!=MaxNodes-1)

{

j=-1;

**for** (k=0;k<MaxNodes;k++)

**if** (c[i][k]!=0) **//Если есть ненулевой поток...**

**if** (i>0) **//и в путь уже включены вершины...**

{

**if** (!Est(&S[0],m,k) && !Tpk(&Tupik[0],N\_Tupik,k))

**//то включаем текущую вершину,**

**//если ее нет в пути и если она не "тупиковая".**

{

SS[++j].Element=k;

SS[j].Propusk=c[i][k];

SS[j].Metka=FALSE;

}

}

**else**

**if** (!Tpk(&Tupik[0],N\_Tupik,k)) **//Не вернулись ли назад?**

**//Поток не нулевой, и это первая вершина.**

{ **//Включаем эту вершину в путь.**

SS[++j].Element=k;

SS[j].Propusk=c[i][k];

SS[j].Metka=FALSE;

}

**if** (j!=-1) **//Есть продолжение.**

{

mx=SS[0].Propusk;

el=0;

**for** (k=1;k<=j;k++)

**if** (SS[k].Propusk>mx)

{ el=k; mx=SS[k].Propusk; }

S[++m].Element=SS[el].Element;

S[m].Propusk=mx;

S[m].Metka=TRUE;

**if** (SS[el].Element==MaxNodes-1) **//Найден сквозной путь.**

{

nomer++;

**//Запоминаем сквозной путь.**

**for** (k=0;k<=m;k++)

Put[nomer][k]=S[k].Element;

**//Ищем минимальный поток.**

mn=S[0].Propusk;

el=0;

**for** (k=1;k<=m;k++)

**if** (S[k].Propusk<mn)

{ el=k; mn=S[k].Propusk; }

Potok[nomer]=mn; **//Запоминаем его.**

**//Вычисляем остаточные пропускные способности.**

**for** (k=0;k<m;k++)

{

c[S[k].Element][S[k+1].Element] -= Potok[nomer];

c[S[k+1].Element][S[k].Element] += Potok[nomer];

}

el=MaxNodes-1; **//Переход к следующей итерации.**

j=0;

}

**else** i=S[m].Element;

}

**else** **//Продолжения нет. Это возможно тогда, когда:**

{

**if** (i==0) **//а) все пропускные способности нулевые.**

**// В этом случае - выход**

el=MaxNodes-1;

**else** **//б) мы попали в тупик. Запомним тупиковую вершину**

**// в массиве и отступим назад на одну вершину.**

{

Tupik[++N\_Tupik]=S[m].Element;

m--;

i=S[m].Element;

}

}

}

}

**return** nomer; **//Возвращает количество сквозных потоков.**

}

**int** Spisok::Est(Uzel S[], **int** m, **int** k)

**//Функция проверяет, есть ли вершина k в пути S.**

**//m - текущее количество элементов в пути.**

**//Возвращает 1, если вершина есть, и 0 - в противном случае.**

{

**for** (**int** l=0;l<=m;l++)

**if** (S[l].Element==k) **return** 1;

**return** 0;

}

**int** Spisok::Tpk(**int** Tupik[],int N\_Tupik, int k)

**//Функция проверяет, есть ли вершина k в массиве "тупиковых" вершин.**

**//N\_Tupik - текущее количество вершин в массиве.**

**//Возвращает 1, если вершина есть, и 0 - в противном случае.**

{

**if** (N\_Tupik==-1) **return** 0;

**for** (**int** l=0;l<=N\_Tupik;l++)

**if** (Tupik[l]==k) **return** 1;

**return** 0;

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din126_1.zip).

Результат работы этой программы для сети, изображенной на рисунке 2, будет следующим:

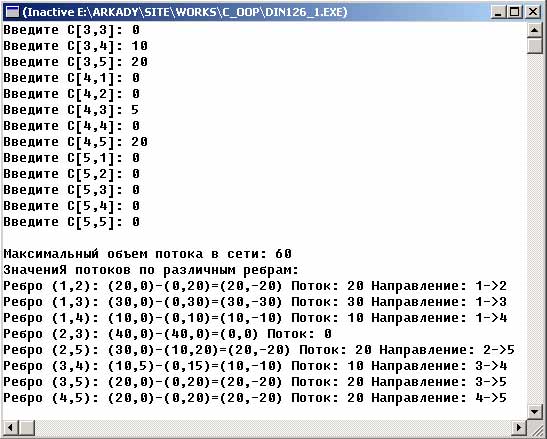


Рис.3. Результат работы приложения

Со следующего шага мы начнем рассматривать задачу о ***нахождении потока наименьшей стоимости***.

## Нахождение потока наименьшей стоимости (общие замечания)

На этом шаге мы рассмотрим (общих чертах) ***задачу нахождения потока наименьшей стоимости***.

Задача нахождения потока наименьшей стоимости в сети с ограниченной пропускной способностью обобщает задачу [***определения максимального потока***](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0126.html) по следующим направлениям.

1. Все ребра допускают только одностороннее направление потока, т.е. являются (ориентированными) дугами.
2. Каждой дуге поставлена в соответствие (неотрицательная) стоимость прохождения единицы потока по данной дуге.
3. Дуги могут иметь положительную нижнюю границу пропускной способности.
4. Любой узел сети может выступать как в качестве источника, так и стока.

В рассматриваемой задаче необходимо найти потоки по дугам, минимизирующие общую стоимость прохождения потока по сети; при этом должны удовлетворятся ограничения на пропускные способности дуг и на величины предложений и спроса отдельных (или всех) узлов. Сначала опишем сетевую модель с заданными стоимостями прохождения потока по дугам и сформулируем соответствующую задачу линейного программирования. Эта задача далее будет использована как основа для построения специального симплексного алгоритма, предназначенного для решения данной сетевой задачи.

Прежде чем приступить к рассмотрению поставленной задачи коротко изложим основные положения линейного программирования и симплекс-метода, упомянутые выше.

***Линейное программирование*** - это метод математического моделирования, разработанный для оптимизации использования ограниченных ресурсов. Задачи линейного программирования являются задачами на нахождение наиболее выгодного варианта: наилучшего плана перевозок, наиболее экономного рациона и т.д. С математической точки зрения в каждой задаче ищутся значения нескольких неизвестных, причем требуется, чтобы:

* эти значения были неотрицательны;
* эти значения удовлетворяли некоторой системе линейных уравнений или линейных неравенств;
* при этих значениях некоторая линейная функция обращалась в минимум (или максимум).

Число переменных и число условий могут быть какими угодно. В реальных задачах эти числа весьма велики. Общая математическая формулировка задачи линейного программирования выглядит следующим образом.

***Дана система линейных уравнений***

a11x1 + a12x2 + … + a1nxn = b1

a21x1 + a22x2 + … + a2nxn = b2 (\*)

. . . . . . . .

am1x1 + am2x2 + … + amnxn = bm

***и линейная функция***

f = c1x1 + c2x2 + ... + cnxn.

***Требуется найти такое неотрицательное решение***

x1>=0, x2>=0, ... , xn>= 0

***системы, при котором функция f принимает наименьшее значение (минимизируется).***

Естественно, что решение таких задач связано с большим объемом вычислений. Однако существуют методы, позволяющие найти решение любой задачи линейного программирования за конечное число шагов. К их числу относится прежде всего так называемый ***симплекс-метод***.

Для начала работы по симплекс-методу требуется, чтобы заданная система уравнений была приведена к такому виду, в котором какие-либо **r** неизвестных **(r<=m)** выражены через остальные, причем свободные члены этих выражений неотрицательны. Предположим, для определенности, что неизвестные, которые выражены через остальные, - это **x1, ..., xr**. Следовательно, система приведена к виду: приведена к виду:

x1 = a'1,r+1xr+1 + ... + a'1,nxn + b'1

x2 = a'2,r+1xr+1 + ... + a'2,nxn + b'2

. . . . . . . .

xr = a'r,r+1xr+1 + ... + a'r,nxn + b'r

где

b'1 >=0, b'2 >=0, ..., b'n >=0 (\*\*)

Неизвестные **x1, …, xr**, находящиеся в левой части этой системы, называются ***базисными***, а весь набор **{ x1, …, xr}**, который мы обозначим для краткости одной буквой ***Б***, - ***базисом***, остальные неизвестные называются ***небазисными***или ***свободными***. Подставляя в форму **f** вместо базисных неизвестных их выражения через небазисные из последней системы, мы можем и саму форму **f** также записать через небазисные неизвестные **xr+1, …, xn**:

f = c0 + с'r+1xr+1 + ... + с'nxn.

Положим все небазисные неизвестные равными нулю: **xr+1 = ... = xn = 0** и и найдем из последней системы значения базисных неизвестных: **x1 = b'1, …, xr = b'n**. Полученное таким путем решение системы:

(b'1, …, b'r, 0, …, 0)

будет, вследствие (\*\*), допустимым. Оно называется ***базисным решением***, отвечающим базису **x1, …, xr**. Для базисного решения значение формы **f** равно

fБ = с0.

Решение задачи при помощи симплекс-метода распадается на ряд шагов. Каждый из шагов заключается в том, что от данного базиса ***Б*** мы переходим к другому базису ***Б'*** с таким расчетом, чтобы значение **fБ** уменьшилось или, по крайней мере, не увеличилось: **fБ' <= fБ**. Новый базис ***Б'*** получается из старого ***Б*** весьма просто: из ***Б*** удаляется одна из неизвестных и взамен нее вводится другая (из числа прежних небазисных). Изменение базиса влечет за собой соответствующую перестройку системы (\*). После некоторого числа таких шагов мы или приходим к базису **Б(k)**, для которого **fБ(k)** есть искомый минимум формы **f**, а соответствующее базисное решение является оптимальным, или же выясняем, что задача решения не имеет.

При рассмотрении поставленной задачи необходимо знать термин ***двойственная задача***. Исходную задачу линейного программирования будем называть ***прямой***. ***Двойственная задача*** - это задача, формулируемая с помощью определенных правил непосредственно из прямой задачи. Переменные и ограничения двойственной задачи формируются путем преобразований прямой задачи по следующим правилам:

1. Каждому из **m** ограничений прямой задачи соответствует переменная двойственной задачи.
2. Каждой из **n** переменных прямой задачи соответствует ограничение двойственной задачи.
3. Коэффициенты при какой-либо переменной в ограничениях прямой задачи становятся коэффициентами ограничения двойственной задачи, соответствующей этой переменной, а правая часть формируемого ограничения равна коэффициенту при этой переменной в выражении целевой функции.
4. Коэффициенты целевой функции двойственной задачи равны правым частям ограниченной прямой задачи.

На следующем шаге мы рассмотрим ***пример построения сетевой модели***.

## Нахождение потока наименьшей стоимости. Сетевая модель

На этом шаге мы рассмотрим ***пример сетевой модели***.

Рассмотрим сеть **G = (N, A)** с ограниченной пропускной способностью, где **N** - множество узлов, **A** - множество дуг. Обозначим:

* **xij** - величина потока, протекающего от узла **i** к узлу **j**,
* **uij (lij)** - верхняя (нижняя) граница пропускной способности дуги **(i, j)**,
* **cij** - стоимость прохождения единицы потока по дуге **(i, j)**,
* **fi** - величина "чистого" результирующего потока, протекающего через узел **i**.

На рисунке 1 показано, как на схемах сетей будем отображать определенные параметры дуг. Метка **[fi]** указывает положительное (отрицательное) значение предложения (спроса), соответствующего узлу **i**:

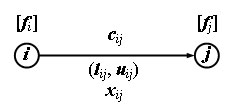


Рис.1. Отображение параметров дуг

Рассмотрим пример построения сетевой модели.

Пример. Компания **GrainCo** снабжает зерном из трех зернохранилищ три птицеводческие фермы. Предложение зернохранилищ составляет 100, 200 и 50 тысяч тонн зерна, а спрос ферм - 150, 80 и 120 тысяч тонн соответственно. Компания может транспортировать зерно по железной дороге, за исключением трех маршрутов, где используется автомобильный транспорт.

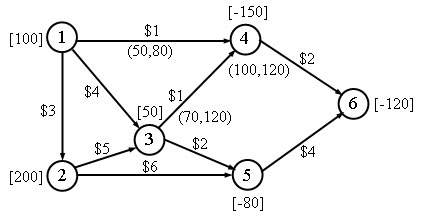


Рис.2. Схема маршрутов

На рисунке 2 показаны возможные маршруты между зернохранилищами и птицеводческими фермами. Зернохранилища представлены узлами 1, 2 и 3; их предложения указаны метками [100], [200] и [50] соответственно. Фермы обозначены узлами 4, 5 и 6 с величинами спроса [-150], [-80] и [-120]. Маршруты транспортировки зерна показаны на рисунке 2 дугами, соединяющими узлы сети. Дуги (1, 4), (3, 4) и (4, 6) соответствуют автомобильным маршрутам. Эти маршруты имеют верхние и нижние границы пропускных способностей. Например, по маршруту (1, 4) можно провести от 50 до 80 тысяч тонн зерна. Другие маршруты соответствуют железнодорожному транспорту, пропускная способность которого практически не ограничена. Стоимость транспортировки одной тонны зерна показана возле каждой дуги.

На следующем шаге мы рассмотрим ***сетевую модель как задачу линейного программирования***.

## Нахождение потока наименьшей стоимости. Сетевая модель как задача линейного программирования

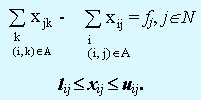
На этом шаге мы рассмотрим ***преобразование сетевой модели в задачу линейного программирования***.

Определение модели сети с ограниченной пропускной способностью как задачи линейного программирования необходимо для разработки симплексного алгоритма решения задач данного типа. Используя определения, данные в [шаге 127](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0127.html#1), мы можем записать задачу линейного программирования для сети с ограниченной пропускной способностью следующим образом.

Минимизировать:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris129_1.jpg

при ограничениях:



Результирующий "чистый" поток **fj**, протекающий через узел **j**, вычисляется по формуле

fj = (Величина потока, выходящего из узла j) -

(Величина потока, входящего в узел j).

Узел **j** выступает в качестве источника, если **fj > 0**, и как исток при **fj < 0**.

Нижнюю границу пропускной способности **lij** можно удалить из ограничений с помощью подстановки **xij = x'ij - lij**. Для нового потока верхней границей пропускной способности будет величина **uij - lij**. В этом случае результирующий поток через узел **i** будет равен **[fi] - lij**, а через узел **j** - **[fj] + lij**. На рисунке 1 показаны преобразования характеристик дуги **(i, j)** после исключения ее нижней границы пропускной способности.

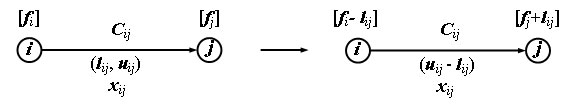


Рис.1. Преобразования характеристик

***Пример 1***. Запишем задачи линейного программирования для сети из рисунка 2 до и после исключения нижних границ пропускных способностей.

Основные ограничения формулируемой задачи линейного программирования связаны с определением входных и выходных потоков, протекающих через каждый узел, что порождает следующую задачу линейного программирования:

x12 x13 x14 x23 x25 x34 x35 x46 x56

Миними- 3 4 1 5 6 1 2 2 4

зировать

Узел 1 1 1 1 = 100

Узел 2 -1 1 1 = 200

Узел 3 -1 -1 1 1 = 50

Узел 4 -1 -1 1 =-150

Узел 5 -1 -1 1 = -80

Узел 6 -1 -1 =-120

Нижние 0 0 50 0 0 70 0 100 0

границы

Верхние Б Б 80 Б Б 120 Б 120 Б

границы

***Примечание.*** ***Б*** - бесконечность.

Сделаем замечание о структуре коэффициентов, формирующих ограничения. В столбце, соответствующем переменной **xij**, всегда в строке **i** стоит **+1**, а в строке **j** находится **-1**. Остальные коэффициенты равны нулю. Такая структура коэффициентов типична для сетевых моделей.

Для переменных, представляющих потоки через дуги, имеющие ненулевые нижние границы пропускных способностей, выполняем замену:

x14 = x'14 + 50,

x34 = x'34 + 70,

x46 = x'46 + 100.

В результате получаем следующую задачу линейного программирования:

x12 x13 x14 x23 x25 x34 x35 x46 x56

Миними- 3 4 1 5 6 1 2 2 4

зировать

Узел 1 1 1 1 = 50

Узел 2 -1 1 1 = 200

Узел 3 -1 -1 1 1 = -20

Узел 4 -1 -1 1 =-130

Узел 5 -1 -1 1 = -80

Узел 6 -1 -1 = -20

Верхние Б Б 30 Б Б 50 Б 20 Б

границы

***Примечание.*** ***Б*** - бесконечность.

Соответствующая сеть после исключения нижних границ пропускных способностей дуг показана на рисунке 2.

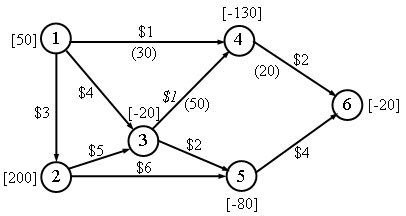


Рис.2. Сеть после исключения нижних границ

Отметим, что данную сеть можно получить непосредственно из сети, представленной на рисунке 3, с помощью преобразований, показанных на рисунке 1, причем без необходимости записи в виде задачи линейного программирования.

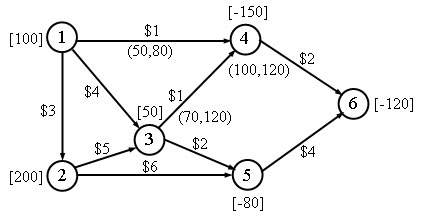


Рис.3. Схема маршрутов

В следующем примере представлена сетевая модель, которая исходно не удовлетворяет условию ***узлового потока***(т.е. условию, что результирующий поток, проходящий через узел, равен разности выходного и входного потоков), но которую можно преобразовать в модель, удовлетворяющую этому условию, путем изменения ограничений соответствующей задачи линейного программирования.

***Пример 2.*** Агентство по найму рабочей силы имеет заказ на рабочих на 4 месяца вперед (с января по апрель) согласно следующему графику:

Месяц Январь Февраль Март Апрель

К-во рабочих 100 120 80 170

Поскольку спрос на рабочих различен в разные месяцы, возможно, экономически целесообразно нанять больше рабочих, чем требуется, в текущем месяце. Стоимость найма рабочих и удержания их в "ждущем режиме" зависит от времени трудоустройства, как показано в следующей таблице:

Время трудоустройства (месяцы) 1 2 3 4

Стоимость на одного рабочего (рубли) 100 130 180 220

Обозначим через **xij** количество рабочих, нанятых на начало **i**-го месяца и освобожденных на начало **j**-го. Например, **x12** - это количество рабочих, нанятых в январе только на один месяц.

Чтобы сформулировать задачу линейного программирования, нужно добавить еще пятый месяц (май). Тогда, например, переменная **x45** будет обозначать количество рабочих, нанятых в апреле на один месяц. Естественно, на май нет заказа на рабочих.

Ограничения строятся так, чтобы спрос на рабочих в месяц **k** можно было бы удовлетворить за счет всех рабочих **xij**, где **i<=k<=j**. Обозначив через **Si** **(Si>=0)** количество рабочих, "лишних" в месяце **i**, получим следующую задачу линейного программирования:

x12 x13 x14 x15 x23 x24 x25 x34 x35 x45 S1 S2 S3 S4

100 130 180 220 100 130 180 100 130 100 Min

Янв. 1 1 1 1 -1 100

Фев. 1 1 1 1 1 1 -1 120

Март 1 1 1 1 1 1 -1 80

Апр. 1 1 1 1 -1 170

В данной задаче линейного программирования нет той специальной структуры коэффициентов ограничений, что была в модели из примера 1. Тем не менее, эта задача имеет свою специфику, которая позволит преобразовать ее в сетевую модель. Выполним следующие действия.

1. Из **n**-го уравнения (ограничения) задачи создадим новое **(n + 1)**-е уравнение, умножив **n**-е уравнение на -1.
2. Оставляем первое уравнение без изменений.
3. Для **i = 2, 3, …, n** последовательно заменяем **i**-е уравнение разностью **i**-го и **(i - 1)**-го уравнений.

Применение описанной процедуры к задаче данного примера приводит к следующей задаче линейного программирования, которая уже имеет структуру сетевой модели.

Таким образом, задачу распределения рабочих можно представить в виде эквивалентной задачи нахождения потока минимальной стоимости в сети с ограниченной пропускной способностью (рисунок 4):

x12 x13 x14 x15 x23 x24 x25 x34 x35 x45 S1 S2 S3 S4

100 130 180 220 100 130 180 100 130 100 Min

Янв. 1 1 1 1 -1 100

Фев. -1 1 1 1 1 -1 20

Март -1 -1 1 1 1 -1 -40

Апр. -1 -1 -1 1 1 -1 90

Май -1 -1 -1 -1 -1 -1 -170

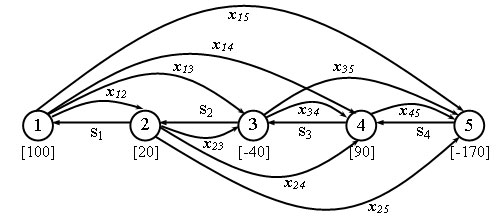


Рис.4. Сеть с ограниченной пропускной способностью

Задачу линейного программирования можно решить с помощью специализированных программных средств, например **Microsoft Excel**.

На следующем шаге мы рассмотрим ***симплексный алгоритм для сетей с ограниченной пропускной способностью***.

## Нахождение потока наименьшей стоимости. Симплексный алгоритм для сетей с ограниченной пропускной способностью

На этом шаге мы приведем ***пример использования симплексного метода при решении сетевых задач***.

Предлагаемый алгоритм повторяет в точности те же шаги, что и обычный симплексный метод. Вместе с тем здесь учитывается специальная структура сетевой модели.

Используя определения из [предыдущего шага](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0129.html), **fi** - результирующий поток через узел **i**, строим симплексный алгоритм на основе условия равенства нулю суммы **fi** при **i**, изменяющимся от 1 до **n**. Это условие означает, что в сети суммарный объем предложений равен суммарному объему спроса. Мы всегда можем удовлетворить данное условие путем введения фиктивного источника или стока, которые можно связать с остальными узлами сети дугами с нулевой стоимостью и бесконечной пропускной способностью. Однако сбалансированность сети не гарантирует существования допустимого решения, поскольку этому может воспрепятствовать ограниченность пропускных способностей дуг.

Опишем шаги алгоритма нахождения потока минимальной стоимости для сетей с ограниченной пропускной способностью.

***Шаг 0.*** Определяем для данной сети начальное базисное допустимое решение (множество дуг). Переходим к шагу 1.

***Шаг 1.*** С помощью условия оптимальности симплекс-метода определяем вводимую в базис переменную (дугу). Если на основе условия оптимальности определяем, что последнее решение оптимально, вычисления заканчиваются. В противном случае переходим к шагу 2.

***Шаг 2.*** На основе условия допустимости симплекс-метода определяем исключаемую из базиса переменную (дугу). Изменив базис, возвращаемся к шагу 1.

Сети с **n** узлами и нулевым результирующим потоком (т.е. при выполнении равенства **f1 + f2 +…+ fn = 0**) соответствуют **n - 1** ***независимым*** ограничениям в виде равенств. Поэтому базисное решение должно содержать **n - 1**переменных. Можно показать, что базисное решение соответствует ***остовному дереву*** данной сети.

Вводимая переменная (дуга) на шаге 1 определяется путем вычисления разностей **zij - cij** для всех текущих небазисных дуг **(i, j)**. Если для всех разностей **zij - cij <= 0**, тогда текущее базисное решение оптимально. Иначе в качестве вводимой в базис переменной выбираем дугу, которой соответствует наибольшее положительное значение разности **zij - cij**.

Вычисление разностей **zij - cij** основано на соотношениях двойственности, точно так же как в транспортной модели. Обозначим через **wi** переменную задачи, двойственной к задаче линейного программирования, которая (переменная) соответствует ограничению узла **i**. Тогда данная двойственная задача формулируется следующим образом.

Максимизировать

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris130_1.jpg

при выполнении условий

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris130_2.jpg

переменные **wi** произвольного знака (**i = 1, 2, …, n**).

Из теории линейного программирования следует, что **wi - wj = cij** для любой базисной дуги **(i, j)**. Поскольку исходная задача линейного программирования по определению имеет одно избыточное ограничение, мы можем присвоить произвольное значение одной из переменной двойственной задачи. Для определенности положим **w1 = 0**. Затем следует решить (базисную) систему уравнений **wi - wj = cij** для нахождения остальных переменных двойственной задачи. Далее вычисляем разности **zij - cij** для небазисных переменных согласно формуле

zij - cij = wi - wj - сij.

Теперь осталось показать, как определяется исключаемая из базиса переменная. Для этого рассмотрим следующий числовой пример.

Пример. Сеть трубопроводов связывает две станции опреснения воды с двумя городами. Ежедневное предложение опреснительных станций составляет 40 и 50 миллионов литров воды, города ежедневно потребляют 30 и 60 миллион литров воды. Каждая станция трубопроводами соединена с каждым городом непосредственно, однако они могут также перекачивать воду в города через специальную насосную станцию. Кроме того, станция 1 может транспортировать воду на станцию 2, а город 1 - в город 2. Данная сеть сбалансирована, так как в ней суммарный спрос равен суммарному предложению. Описанная сеть показана на рисунке 1.

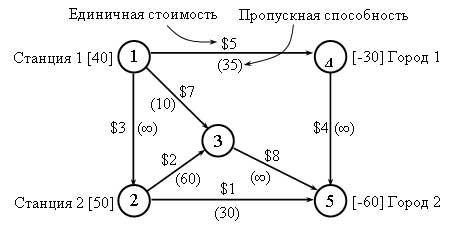


Рис.1. Пример сети

***Итерация 0.***

***Шаг 0.*** *Определение начального допустимого базисного решения*. Нетрудно построить остовное дерево (на рис. 2 показано сплошными дугами) для рассматриваемой сети. Отсюда получаем начальное допустимое базисное решение. Обычно для нахождения такого решения используется метод введения искусственных переменных.

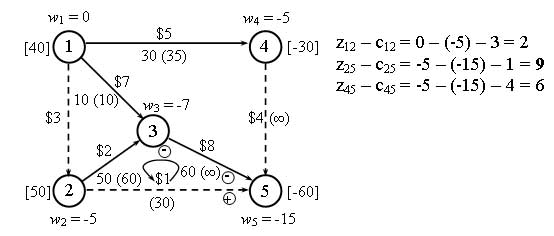


Рис.2. Остовное дерево

На рис. 2 показано, что базисному решению соответствуют дуги (1, 3), (1,4) и (3, 5) с потоками 30, 10, 50 и 60 единиц соответственно. Оставшиеся дуги (показаны пунктиром) представляют небазисные переменные. Запись **x(c)**показывает, что через соответствующую дугу с пропускной способностью **c** проходит поток **x**.

***Итерация 1.***

***Шаг 1***. *Определение вводимой в базис переменной*. При решении системы уравнений

w1 = 0,

wi - wj = cij для базисных дуг (i, j)

получим значения переменных двойственной задачи.

Дуга (1, 3): w1 - w3 = 7 ==> w3 = -7.

Дуга (1, 4): w1 - w4 = 5 ==> w4 = -5.

Дуга (2, 3): w2 - w3 = 2 ==> w2 = -5.

Дуга (3, 5): w3 - w5 = 8 ==> w5 = -15.

Теперь вычисляем разности **zij - cij** для небазисных переменных.

Дуга (1, 2): w1 - w2 - c12 = 0 - (-5) - 3 = 2.

Дуга (2, 5): w2 - w5 - c25 = (-5) - (-15) - 1 = 9.

Дуга (4, 5): w4 - w5 - c45 = (-5) - (-15) - 4 = 6.

Таким образом, дуга (2, 5) будет введена в базисное решение.

***Шаг 2.*** *Определение исключаемой из базиса переменной*. На рис. 2 видно, что дуга (2, 5) совместно с базисными дугами (2, 3) и (3, 5) образуют цикл. ***По определению остовное дерево не может содержать циклов***. Поскольку поток через дугу (2, 5) должен возрасти, необходимо выровнять потоки через дуги, составляющие цикл таким образом, чтобы новое решение осталось допустимым. Для этого поток через дугу (2, 5) пометим знаком "+", потоки через другие дуги цикла - знаком "+" или "-", в зависимости от того, будут ли совпадать направления потоков в этих дугах с направлением потока в дуге (2, 5) при обходе цикла против часовой стрелки. Пометки дуг цикла показаны на рис. 2.

При определении максимального потока, протекающего через дугу (2, 5), необходимо придерживаться следующих правил.

1. Новый поток в текущей базисной дуге не может быть отрицательным.
2. Поток через вводимую в базис дугу не может превышать ее пропускную способность.

Применение правила 1 показывает, что потоки через дуги (2, 3) и (3, 5) нельзя уменьшить более, чем на **min {50, 60} = 50** единиц. Из правила 2 следует, что поток через дугу (2, 5) не может превышать 30 единиц (пропускная способность этой дуги равна 30). Поэтому поток через дуги цикла изменится не более, чем на **min {30, 50} = 30**единиц. Таким образом, поток через дугу (2, 5) равен 30 единицам, через дугу (2, 3) - 50 - 30 = 20 единицам, а через дугу (3, 5) - 60 - 30 =30 единицам.

Поскольку никакая из текущих базисных переменных не приняла нулевого значения, дуга (2, 5) должна остаться небазисной, но с ненулевым значением в 30 единиц. Чтобы выполнить требование равенства нулю небазисных переменных, сделаем подстановку

x25 = 30 - x52, 0 <= x52 <= 30.

Эта подстановка изменит уравнения для потоков, протекающих через узлы 2 и 5.

Текущее уравнение для потоков узла 2: 50 + x12 = x23 + x25.

Текущее уравнение для потоков узла 5: x25 + x35 + x45 = 60.

После подстановки **x25 = 30 - x52** получим

Новое уравнение для потоков узла 2: 20 + x12 + x52 = x23.

Новое уравнение для потоков узла 5: x35 + x45 = x52 + 30.

Результаты этих изменений показаны на рисунке 3. Направление потока через дугу (2, 5) изменилось на обратное (от узла 5 к узлу 2), причем, как и ожидалось, **x52 = 0**. Описанная подстановка также требует изменения стоимости прохождения потока по дуге (2, 5) до -$1. Те дуги, направления потоков которых изменены на противоположные, помечены в сети звездочкой.

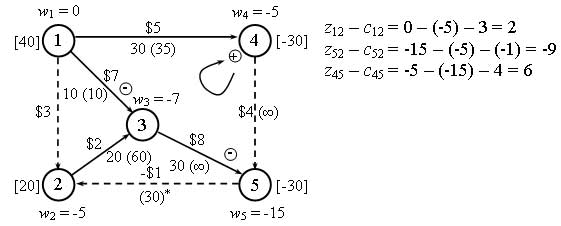


Рис.3. Результаты изменения сети

***Итерация 2.***

На рисунке 3 представлены новые значения разностей **zij - cij**. Очевидно, что в базис следует ввести дугу (4, 5). Введение в базис этой дуги также приводит к образованию цикла.

Величину потока через дугу (4, 5) можно увеличить до наименьшей из следующих величин.

1. ***Максимальный поток*** через дугу (4, 5), определяемый пропускной способностью, равен бесконечности.
2. ***Максимальное увеличение*** потока через дугу (1, 4) равно 35 - 30 = 5 единиц.
3. ***Максимальное уменьшение*** потока через дугу (1, 3) равно 10 единиц.
4. ***Максимальное уменьшение*** потока через дугу (3, 5) равно 30 единиц.

Таким образом, поток через дугу (4, 5) можно увеличить до 5 единиц; эта дуга входит в базис, а дуга (1, 4) с потоком в 35 единиц исключается из базиса.

Выполнив подстановку **x14 = 35 - x41**, получим сеть, показанную на рис. 4, где дуги (1, 3), (2, 3), (3, 5) и (4, 5) формируют остовное дерево сети (базисное решение). Для дуги (1, 4) с обратным направлением потока изменена стоимость прохождения потока до -$5.

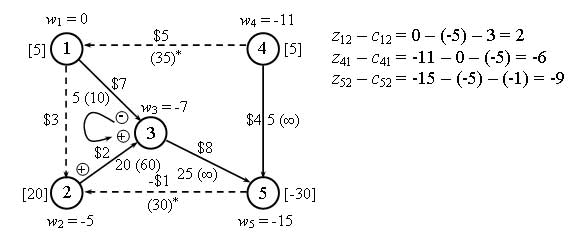


Рис.4. Результирующая сеть

***Итерация 3.***

Вычисленные новые значения разностей **zij - cij** для небазисных дуг (1, 2), (4, 1) и (5, 2) показаны на рис. 4. Из этих значений вытекает, что в базис следует ввести дугу (1, 2) с потоком в 5 единиц, тогда как дуга (1, 3) исключается из базиса с нулевым значением потока. Новое решение представлено на рисунке 5.

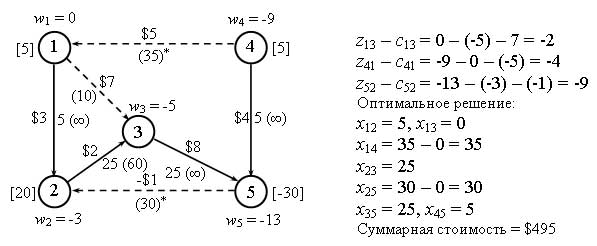


Рис.5. Новое решение

***Итерация 4.***

Из новых значений разностей **zij - cij** (рис. 5) видно, что последнее решение оптимально. Значения исходных переменных получаем путем обратной подстановки, как показано на рисунке 5.

Со следующего шага мы начнем знакомиться с ***методами сетевого планирования***.

## Методы сетевого планирования (общие сведения)

На этом шаге мы приведем ***общие сведения о методах сетевого планироваия***.

На основе сетевых моделей разработано множество методов планирования, составление временных расписаний и управления проектами. Наиболее известные - ***метод критического пути*** (**Critical Path Method**, сокращенно **CPM**), а также система планирования и руководства программами разработок (**Program Evaluation and Review Technique**, сокращенно [**PERT**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0093.html)). В этих методах проекты рассматриваются как совокупность некоторых взаимосвязанных процессов (видов деятельности, этапов или фаз выполнения проекта), каждый из которых требует определенных временных и других ресурсов. В методах **CPM** и **PERT** проводится анализ проектов для составления временных графиков распределения фаз проектов. На рисунке 1 в обобщенной форме показаны основные этапы выполнения этих методов. На первом этапе определяются отдельные процессы, составляющие проект, их отношения предшествования (т.е. какой процесс должен предшествовать другому) и их длительность. Далее проект представляется в виде сети, показывающей отношения предшествования среди процессов, составляющих проект. На третьем этапе на основе построенной сети выполняются вычисления, в результате которых составляется временной график реализации проекта.

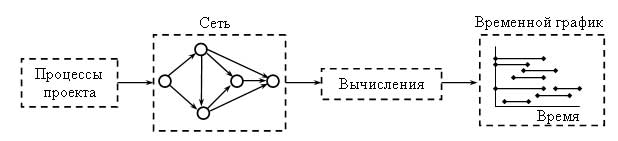


Рис.1. Основные этапы выполнения проектирования

Методы **CPM** и **PERT**, которые разрабатывались независимо друг от друга, отличаются тем, что в методе критического пути длительность каждого этапа проекта является ***детерминированной***, тогда как в системе планирования **PERT** - ***стохастической***.

На следующем шаге мы рассмотрим ***построение сети проекта***.

## Методы сетевого планирования. Построение сети проекта

На этом шаге мы рассмотрим ***пример построения сети проекта***.

Каждый процесс проекта обозначается в сети дугой, ориентированной по направлению выполнения проекта. Узлы сети (также называемые ***событиями***) устанавливают отношения предшествования среди процессов проекта.

Построение сети проекта основано на следующих правилах.

***Правило 1.*** *Каждый процесс в проекте представим одной и только одной дугой*.

***Правило 2.*** *Каждый процесс идентифицируется двумя концевыми узлами*.

На рисунке 1 показано, как с помощью фиктивного процесса можно представить два параллельных (конкурирующих) процесса **А** и **В**. По определению фиктивный процесс (который на схеме сети обычно обозначается пунктирной дугой) не поглощает временных или других ресурсов. Вставив фиктивный процесс одним из четырех способов, показанных на рисунке 1, мы получаем возможность идентифицировать процессы **А** и **В**, по крайней мере, одним уникальным концевым узлом (как требует правило 2).

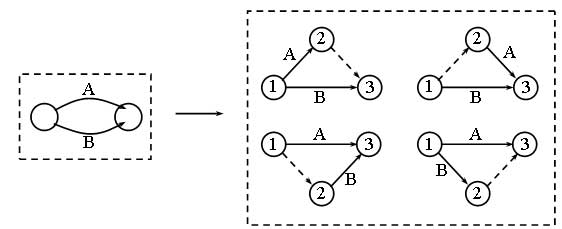


Рис.1. Представление конкурирующих процессов

***Правило 3.*** *Для поддержания правильных отношений предшествования при включении в сеть любого процесса необходимо ответить на следующие вопросы.*

* *Какой процесс непосредственно предшествует текущему?*
* *Какой процесс должен выполняться после завершения текущего процесса?*
* *Какой процесс конкурирует (выполняется параллельно) с текущим?*

Ответы на эти вопросы, возможно, потребует включения в сеть фиктивных процессов, чтобы правильно отобразить последовательность выполнения процессов. Предположим, например, что четыре процесса должны удовлетворять следующим условиям.

1. Процесс **С** должен начаться после завершения процессов **А** и **В**.
2. Процесс **Е** должен начаться непосредственно после завершения процесса **В**.

На рисунке 2а показано неправильное представление наших процессов, так как из него следует, что процесс **Е**должен начаться после завершения как процесса **В**, так и **А**. На рисунке 2б показано, как с помощью фиктивного процесса **D** разрешить эту коллизию.

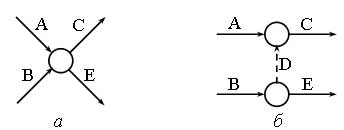


Рис.2. Представление процессов

Пример. Издатель имеет контракт с автором на издание его книги. Ниже представлена последовательность (упрощенная) процессов, приводящая к реализации проекта издания книги. Необходимо разработать сеть для этого проекта.

Процесс Предшествующий процесс Длительность (недели)

A: Прочтение рукописи редактором - 3

B: Пробная верстка отдельных страниц книги - 2

C: Разработка обложки книги - 4

D: Подготовка иллюстраций - 3

E: Просмотр автором редакторских правок и

сверстанных страниц А,В 2

F: Верстка книги (создание макета книги) Е 4

G: Проверка автором макета книги F 2

H: Проверка автором иллюстраций D 1

I: Подготовка печатных форм G,H 2

J: Печать и брошюровка книги C,I 4

На рисунке 3 показана сеть, представляющая взаимосвязь процессов данного проекта. Фиктивный процесс (2, 3) введен для того, чтобы "развести" конкурирующие процессы **А** и **В**. Номера узлов сети возрастают в направлении выполнения проектов.

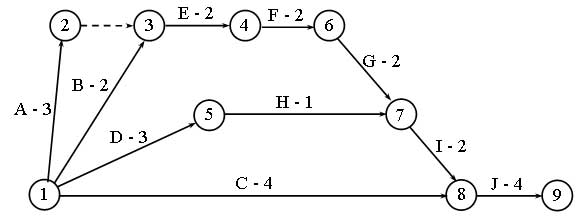


Рис.3. Взаимосвязь процессов

На следующем шаге мы рассмотрим ***метод критического пути***.

## Методы сетевого планирования. Метод критического пути

На этом шаге мы рассмотрим ***алгоритм определения критического пути***.

Конечным результатом применения метода критического пути будет построение временного графика выполнения проекта (см. рис. 1 [шага 131](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0131.html)). Для этого проводятся специальные вычисления, в результате чего получаем следующую информацию.

1. Общая длительность выполнения проекта.
2. Разделение множества процессов, составляющих проект, на критически и некритические.

Процесс является ***критическим***, если он не имеет "зазора" для времени своего начала и завершения. Таким образом, чтобы весь проект завершился без задержек, необходимо, чтобы все критические процессы начинались и заканчивались в строго определенное время. Для ***некритического*** процесса возможен некоторый "дрейф" времени его начала, но в определенных границах, когда время его начала не влияет на длительность выполнения всего проекта.

Для проведения необходимых вычислений определим ***событие*** как точку на временной оси, где завершается один процесс и начинается другой. В терминах сети, событие - это сетевой узел. Нам понадобятся также следующие определения и обозначения:

* **Aj** - самое раннее возможное время наступления события **j**,
* **Bj** - самое позднее возможное время наступления события **j**,
* **Dij** - длительность процесса **(i, j)**.

Вычисления критического пути включает два этапа (прохода). При ***проходе вперед*** вычисляются ***самые ранние времена наступления событий***, а при ***проходе назад*** - ***самые поздние времена наступления*** тех же событий.

***Проход вперед.*** Здесь вычисления начинаются в узле 1 и заканчиваются в последнем узле **n**.

***Начальный шаг.*** Полагаем **A1=0**; это указывает на то, что проект начинается в нулевой момент времени. ***Основной шаг* j.** Для узла **j** определяем узлы **p, q, …, v,** непосредственно связанные с узлом **j** процессами **(p, j), (q, j), …, (v, j)**, для которых уже вычислены самые ранние времена наступления соответствующих событий. Самое раннее время наступления события **j** вычисляется по формуле

Aj = max { Ap + Dpj, Aq + Dqj, …, Av + Dvj}.

Проход вперед завершается, когда будет вычислена величина **An** для узла **n**. По определению величина **j** равна самому длинному пути (длительности) от начала проекта до узла (события) **j**.

***Проход назад.*** В этом проходе вычисления начинаются в последнем узле **n** и заканчивается в узле 1.

***Начальный шаг.*** Полагаем **Bn = An**; это указывает, что самое раннее и самое позднее времена для завершения проекта совпадают.

***Основной шаг* j.** Для узла **j** определяем узлы **p, q, …, v,** непосредственно связанные с узлом **j** процессами **(j, p), (j, q), …, (j, v)**, для которых уже вычислены самые поздние времена наступления соответствующих событий. Самое позднее время наступления события **j** вычисляется по формуле

Bj = min {Bp - Djp, Bq - Djq, …, Bv - Djv}.

Проход назад завершается при вычислении величины **B1** для узла 1.

Процесс **(i, j)** будет ***критическим***, если выполняются три условия.

1. **Bi = Ai**.
2. **Bj = Aj**.
3. **Bj - Bi = Aj - Ai = Dij**.

Эти условия не выполняются, то процесс ***некритический***.

Критические процессы должны образовывать непрерывный путь через всю сеть от начального события до конечного.

Пример. Найдем критический путь для сети проекта, показанной на рисунке 1. Длительность всех процессов дана в днях.

***Проход вперед.***

***Узел 1***. Полагаем **A1 = 0**.

***Узел 2***. **A2 = A1 + D12 = 0 + 5 = 5**.

***Узел 3***. **A3 = max { A1 + D13, A2 + D23} = max {0 + 6, 5 + 3} = 8**.

***Узел 4***. **A4 = A2 + D24 = 5 + 8 = 13**.

***Узел 5***. **A5 = max { A3 + D35, A4 + D45} = max {8 + 2, 13 + 0} = 13**.

***Узел 6***. **A6 = max { A3 + D36, A4 + D46, A5 + D56} = max {8 + 11, 13 + 1, 13 + 12} = 25**.

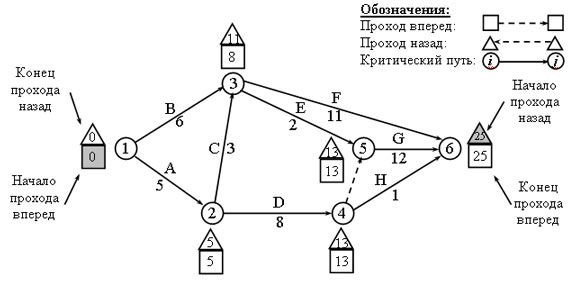


Рис.1. Пример проекта

Таким образом, расчеты показывают, что проект можно выполнить за 25 дней.

***Проход назад.***

***Узел 6***. Полагаем **A6 = B6 = 25**.

***Узел 5***. **B5 = B6 - D56 = 25 - 12 = 13**.

***Узел 4***. **B4 = min {B6 - D46, B5 - D45} = min {25 - 1, 13 - 0} = 13**.

***Узел 3***. **B3 = min {B6 - D36, B5 - D35} = min {25 - 11, 13 - 2} = 11**.

***Узел 2***. **B2 = min {B4 - D24, B3 - D23} = min {13 - 8, 11 - 3} = 5**.

***Узел 1***. **B1 = min {B3 - D13, B2 - D12} = min {11 - 6, 5 - 5} = 0**.

Вычисления без ошибок всегда приводят к результату **B = 0**.

Результаты вычислений, выполняемых при проходах вперед и назад, показаны на рисунке 1. Правила определения критических процессов показывают, что критический путь составляют процессы **1->2->4->5->6**, т.е. этот путь проходит от начального узла 1 до конечного узла 6. Сумма длительности критических процессов (1, 2), (2, 4), (4, 5) и (5, 6) равна длительности всего проекта (т.е. 25 дней). Отметим, что процесс (4, 6) удовлетворяет первым двум условиям критического пути (**B4 = A4 = 13** и **B6 = A6 = 25**), но не удовлетворяет третьему условию ( **A6 - B4** не равно **D46**). Поэтому данный процесс не является критическим.

Для проверки полученных результатов воспользуемся [программой, приведенной на 121 шаге](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0121.html#1), которая позволяет спланировать критический путь. Для заданных исходных данных результат вычислений по этой программе будет следующим:

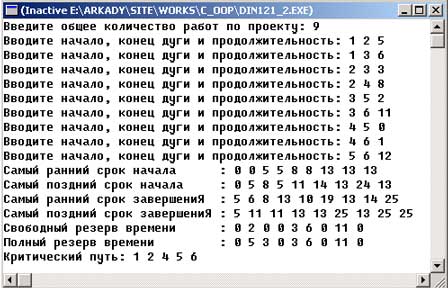


Рис.2. Результат работы приложения

На следующем шаге мы рассмотрим ***построение временного графика***.

## Методы сетевого планирования. Построение временного графика

На этом шаге мы рассмотрим ***правила построения временного графика***.

На этом шаге мы покажем, как на основе данных, полученных расчетным путем на предыдущем шаге, строится временной график последовательного выполнения проекта. Мы уже знаем, что **Ai** для процесса **(i, j)** указывает на с***амое раннее время начала этого процесса***, а **Bj** - на ***самое позднее время завершения*** процесса. Таким образом, пара величин **(Ai, Bj)** ограничивает максимальный интервал времени, в течение которого может выполняться процесс **(i, j)**.

***Построение предварительного графика***

Метод построения предварительного временного графика выполнения проекта покажем на следующем примере.

Пример. Построим временной график проекта на примере из [предыдущего шага](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0133.html#1).

Предварительный временной график проекта можно начертить, используя максимальные интервалы выполнения каждого процесса. В результате получим график, представленный на рисунке 1:

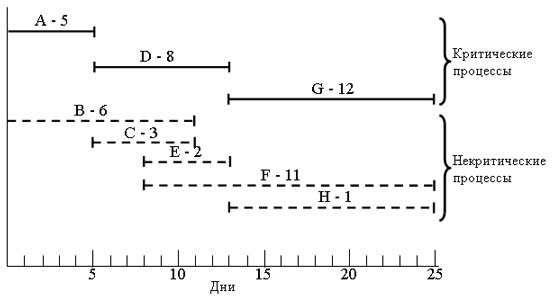


Рис.1. Время выполнения процессов

Сделаем два замечания.

1. Критические процессы (показаны на графике сплошными линиями) располагаются последовательно друг за другом без временных зазоров и перекрытий. Таким образом, их суммарная длительность равна длительности выполнения всего проекта (в данном случае, 25 дней).
2. Некритические процессы (показаны на графике пунктирными линиями) представлены максимальными интервалами выполнения, которые превышают реальную длительность выполнения этих процессов. Поэтому необходимо каким-то образом определиться с началом выполнения этих процессов.

Как выбрать время начала выполнения некритического процесса? Обычно предпочитают начинать некритические процессы (по возможности) в самый ранний срок. В этом случае остается запас времени (остаток максимального интервала выполнения), который можно использовать для решения неожиданно возникших во время выполнения процесса проблем. Вместе с тем при необходимости можно перенести начало выполнения какого-либо процесса. Допустим, если в нашем примере во время выполнения процессов **E** и **F** (рис. 1) используется одно и то же оборудование, причем в каждый момент времени его можно задействовать только для одного процесса, тогда можно исключить временное наложение этих процессов, начав процесс **F** после завершения **Е**.

Если на некритические процессы не накладываются какие-либо дополнительные ограничения и все они начинаются в самый ранний момент времени, то временной график проекта строится автоматически. Однако в этом случае могут нарушаться некоторые отношения предшествования. В частности, в нашем примере (см. рис. 1 [предыдущего шага](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0133.html#1)) процесс **С** должен быть завершен до начала процесса **Е**. Но максимальные интервалы времени выполнения этих процессов перекрываются, поэтому и реальные интервалы времени их выполнения также могут перекрываться. Поэтому необходимо предусмотреть какие-нибудь ***"красные флажки"***, которые автоматически указывали бы, когда тот или иной процесс может начинаться без нарушения отношений предшествования с другими процессами. Далее мы покажем, как для этого использовать ***запасы времени*** отдельных процессов.

***Определение запасов времени***

***Запас времени некритического процесса*** - это часть максимального интервала времени выполнения этого процесса (который, напомним, больше реальной длительности процесса). Различают ***общий запас времени*** и ***свободный запас времени*** процесса.

На рисунке 2 показана разность между этими запасами времени процесса **(i, j)** - общим (**TFij**) и свободным (**FFij**). Общий запас времени процесса **(i, j)** определяется как превышение над длительностью выполнения этого процесса интервала времени от самого раннего момента осуществления события **i** до самого позднего осуществления события **j**, т.е.

TFij = Bj - Ai - Dij.

Свободный запас времени процесса **(i, j)** определяется как превышение над длительностью выполнения этого процесса интервала времени - от самого раннего момента осуществления события **i** до самого раннего времени осуществления события **j**, т.е.

FFij = Aj - Bi - Dij.

По определению **FFij <= TFij**.

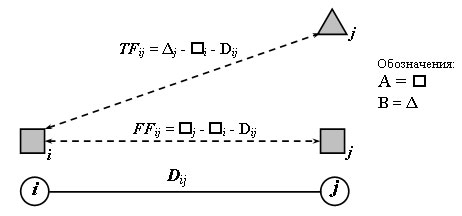


Рис.2. Разность между общим и свободным запасами времени

***Правило "красного флажка"***

***Для некритического процесса*** **(i, j)**

* ***если*** **FFij = TFij**, ***тогда данный процесс может выполняться в любое время внутри максимального интервала*** **(Ai, Bj)** ***без нарушения отношений следования***;
* ***если*** **FFij < TFij**, ***тогда без нарушения отношений следования данный процесс может начаться со сдвигом, не превышающим*** **FFij**, ***относительно самого раннего момента начала процесса*** **i**. ***Сдвиг начала процесса на величину времени, превышающую*** **FFij** ***(но не более*** **TFij*), должен сопровождаться равным сдвигом относительно*** **j** ***всех процессов, начинающихся с события*** **j**.

Это правило означает, что некритический процесс **(i, j)** помечается ***"красным флажком"*** только тогда, когда **FFij < TFij**. Этот флажок принимается во внимание при сдвиге начала процесса относительно самого раннего времени **i** на такую величину, при которой следует рассчитывать сдвиг процессов, следующих из узла **j**.

Пример. Вычислим запасы времени для некритических процессов в сети проекта из [примера шага 133](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0133.html#1) и на основе этих расчетов построим окончательный временной график проекта.

Общие и свободные запасы времени некритических процессов представлены ниже. Такие расчеты можно проводить непосредственно на сети проекта, как показано на рисунке 2.

Некритический Длительность Общий запас Свободный запас

процесс процесса времени (TF) времени (FF)

B (1, 3) 6 11 - 0 - 6 = 5 8 - 0 - 6 = 2

C (2, 3) 3 11 - 5 - 3 = 3 8 - 5 - 3 = 0

E (3, 5) 2 13 - 8 - 2 = 3 13 - 8 - 2 = 3

F (3, 6) 11 25 - 8 - 11 = 6 25 - 8 - 11 = 6

H (4, 6) 1 25 - 13 - 1 = 11 25 - 13 - 1 = 11

Правило ***"красного флажка"*** следует применять только к процессам **В** и **С**, поскольку для них **FF < TF**. Оставшиеся процессы (**E**, **F** и **H**) имеют **FF = TF**, поэтому они могут выполняться в любое время внутри своих максимальных интервалов времени выполнения.

Рассмотрим процесс **В**, помеченный ***"красным флажком"***. Поскольку для этого процесса **TF** = 5 дней, он может начаться в любой день из интервала 0 - 5 дней от начала выполнения всего проекта (рис. 1). Но если **FF** = 2 дня, то, поскольку процесс **В** начнется в 0-й, 1-й или 2-й день от начала выполнения проекта, это не окажет никакого эффекта на последующие процессы **E** и **F**. Однако если процесс **В** начнется в **(2 + *B*)**-й день **(2 + *B* < 5)**, начало выполнения процессов **E** и **F** необходимо сдвинуть от самого раннего срока их начала (8-й день от начала выполнения проекта) на величину, не меньшую ***B***; только при таком условии не нарушатся отношения следования между процессами **В**, **E** и **F**.

Для помеченного ***"красным флажком"*** процесса **С** имеем **FF = 0**. Это означает, что любой сдвиг начала выполнения этого процесса должен сопровождаться таким же (не меньшим) сдвигом начала выполнения процессов **E**и **F**.

На следующем шаге мы приведем ***программу, позволяющую визуально строить сетевые модели***.

## Приложение "Сетевые модели"

На этом шаге мы рассмотрим ***приложение, иллюстрирующее некоторые из рассмотренных ранее алгоритмов***.

Описанное на этом шаге приложение было выполнено в среде программирования **Delphi 6.0** студенткой факультета математики и информационных технологий Курганского государственного университета ***Н.Л.Скутиной*** в 2004 г.

Разработанная программа ***"Сетевые модели"*** позволяет изобразить сеть с помощью узлов и ребер (неориентированных и ориентированных), а также найти минимальное [остовное дерево](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0107.html) построенной сети и минимальный путь между любыми двумя узлами с помощью двух алгоритмов: [алгоритма Дейкстры](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0123.html) и [алгоритма Флойда](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0124.html).

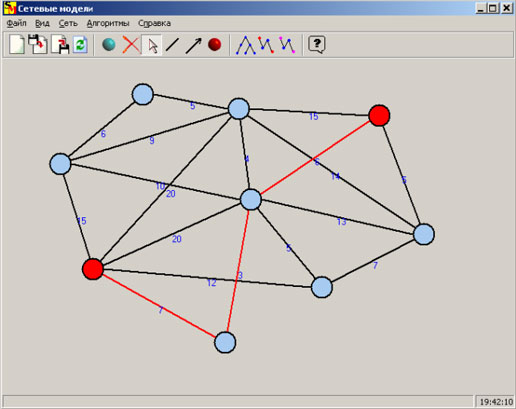


Рис.1. Внешний вид окна программы

Работу программы можно разделить на две части: ***построение сети*** и ***применение указанных алгоритмов к этой сети***.

***Построение сети***

Для того чтобы добавлять новые узлы должна быть нажата кнопка панели инструментов "Новый узел" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_2.jpg. Для ввода узла необходимо щелкнуть левой кнопкой мыши на рабочем поле.

Для того чтобы добавлять новые ребра должна быть нажата кнопка "Добавить путь" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_3.jpg или "Добавить направленный путь" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_4.jpg. Для ввода ребра необходимо щелкнуть левой кнопкой мыши на одном узле, затем на другом, при этом появляется диалоговое окно для ввода длины ребра (пока не введут длину перейти в окно программы нельзя).

При перемещении узла нажмите кнопку "Перемещение узла" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_5.jpg на панели инструментов, щелкните левой кнопкой мыши на перемещаемый узел, затем, не отпуская кнопки мыши, на свободное место, куда вы хотите переместить узел.

Для удаления узла нажмите кнопку "Удаление узла" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_6.jpg на панели инструментов, затем щелкните левой кнопкой мыши на узел, который хотите удалить.

***Замечание.*** *Пункт меню* ***"Сеть"*** *также содержит все перечисленные выше команды.*

Для сохранения сети в файле нажмите кнопку "Сохранить" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_7.jpg. В появившемся окне диалога введите имя файла и выберите директорию, затем нажмите кнопку **ОK**.

Для загрузки сети из файла нажмите кнопку "Открыть" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_8.jpg и в появившемся окне выберите нужный файл.

***Замечание.*** *Эти команды также можно выполнить с помощью соответствующих команд меню* ***"Файл"****.*

Чтобы очистить поле и ввести новую сеть нажмите кнопку "Новый" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris135_9.jpg или выберите соответствующую команду в меню "Файл". Для получения помощи нажмите кнопку "Справка" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris13510.jpg или выберите соответствующую команду меню "Справка". Для очистки результатов действий алгоритмов нажмите кнопку "Обновить" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris13511.jpg или выберите соответствующую команду пункта меню "Вид".

***Выполнение алгоритмов***

Для нахождения минимального остовного дерева сети щелкните по кнопке "Построить остовное дерево" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris13512.jpg.

Если выбрать две различные вершины (для этого необходимо щелкнуть по кнопке "Выделить узлы" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris13513.jpg или выбрать соответствующую команду меню "Сеть", а затем щелкнуть на те узлы, которые надо выделить) и нажать на кнопку "Выполнить алгоритм Дейкстры" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris13514.jpg или "Выполнить алгоритм Флойда" http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris13514.jpg, то программа найдет кратчайший путь между выделенными узлами. Если сеть является ориентированной, то очень важен порядок выделения узлов.

***Замечание.*** *Эти команды также можно выполнить с помощью соответствующих команд меню* ***"Алгоритмы"****.*

Для всех кнопок панели инструментов и команд главного меню в строке состояния отображается подсказка. При наведении мыши на кнопку панели инструментов появляется всплывающая подсказка. К некоторым командам есть горячие клавиши, которые отображаются справа от команды в главном меню или во всплывающей подсказке.

Текст этого приложения можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din135_1.zip).

Мы закончили изложение материала, связанного с динамическими структурами данных. Конечно, мы не охватили всего многообразия динамических структур. Может быть, в дальнейшем, мы к нему вернемся.