Лекция 4. Методы программирования структур: графы и деревья

4.1. Граф

Граф представляет собой множество точек (*вершин, узлов*) вместе с линиями, соединяющими некоторые или все пары точек. Форма линии не играет роли, важно лишь ее наличие или отсутствие для конкретной пары вершин. Направленные линии со стрелками называют *дугами*, не имеющие направления - *ребрами*. В прикладных областях графы часто называют сетями.

В математике граф определяется как пара G=(V, E), где V - множество вершин, а E - множество пар вершин из V. Неупорядоченную пару вершин <u, v> называют ребром, упорядоченную пару (u, v) - дугой. Ребро также обозначают u-v, дугу u->v.

Граф, содержащий только ребра, - *неориентированный,*
содержащий дуги - *ориентированный (орграф*).

Говорят, что ребро u-v или дуга u->v соединяет вершины u и v. Дуга u->v начинается в вершине u и кончается в вершине v (ведет от вершины u к вершине v). При этом вершину v называют *преемником* вершины u, u - *предшественником* вершины v.

Вершины, соединенные ребром или дугой, называют *смежными*. Ребра, имеющие общую вершину, также называют смежными. Ребро (дуга) и любая из его двух вершин называются *инцидентными*.

Ребро (дуга), соединяющее вершину с ней же самой, называется *петлей*.

Степень вершины равна числу инцидентных ей ребер (дуг).
Степень графа - максимальная степень его вершин.

У *взвешенного графа* ребрам и дугам приписывают *вес* - числовую или символьную характеристику. У *размеченного графа* каждой вершине соответствует некоторая информация - *метка*.

Путь - это последовательность вершин, в которой каждая вершина соединена ребром или дугой со следующей вершиной. *Длина пути* равна количеству его ребер (дуг).

Циклом называют замкнутый путь, в котором все ребра (дуги) различны.

Применение графов в алгоритмических задачах

Графы широко применяются в математике, физике, химии, технике, экономике, биологии, психологии, социологии, лингвистике и других областях для описания строения разнообразных систем.

В информатике и кибернетике графы используются при изучении структур данных, языков программирования, в трансляторах, операционных системах, теории поиска, искусственном интеллекте, теории автоматов, теории управления, вычислительной технике и т. д.

Вершины графа соответствуют реальным или абстрактным объектам произвольной природы, а дуги и ребра описывают различные типы связей между ними: математические, физические, экономические, биологические, общественные, грамматические и т. п.

Граф задает *бинарное отношение* (связи между парами объектов) на множестве вершин. Неориентированный граф задает симметричное отношение: ребро можно рассматривать как две противоположные дуги между одной парой вершин.

Граф – удобная информационная структура для математических моделей разнообразных объектов.

Примеры графов:

Объект	*Представление в виде графа*
Электрическая цепь	взвешенный орграф: вершины - точки соединения проводов; дуги - резисторы, конденсаторы, диоды и другие электротехнические элементы; вес дуги - ее полное электрическое сопротивление; направление дуги - полярность подключения
Блок-схема алгоритма	размеченный орграф: вершины - блоки алгоритма; дуги - линии передачи управления; метка вершины - выполняемые в блоке операции

Операции над графами очень разнообразны:

- определение различных характеристик,
- добавление и удаление вершин и ребер,
- поиск путей и подграфов нужного вида,
- обход вершин в определенном порядке и т. п.

Представление графов

На примере графа из рис. 4.1 а рассмотрим основные методы представления графов, выбор зависит от конкретной задачи.

Обозначим: *n* - количество вершин графа, *m* - количество ребер (дуг) графа.

1. **Последовательность ребер (дуг)**, перед которой указывается количество вершин. Каждое ребро (дуга) задается парой номеров вершин (рис. 4.1 б). Удобна для внешнего представления при вводе. Требует памяти порядка O(m).

2. **Матрица смежности** - квадратная матрица *n*n* (рис. 4.1 в), строки и столбцы которой соответствуют вершинам, а элементы равны

$$a[i,j] = \begin{cases} 1, \text{ если в графе есть дуга } i\text{->}j \\ 0, \text{ если в графе нет дуги } i\text{->}j \end{cases}$$

3. **Матрица весов** - квадратная матрица n*n, с элементами w[i,j] = вес дуги i->j . В зависимости от задачи отсутствующим дугам (ребрам) приписывают нулевой или бесконечный вес. Например, при поиске пути с максимальным весом, чтобы программа не включала в него отсутствующие дуги, они должны иметь вес ноль; при поиске пути минимального веса - бесконечность.

Бесконечность представляется особым значением, например, =1, если вес неотрицателен. Можно использовать большое число, но это не всегда удобно.

Для неориентированного графа матрицы смежности и весов симметричны относительно главной диагонали: a[i,j] = a[j,i], и можно хранить только их половину: верхнюю или нижнюю треугольную матрицу. **Требуется память порядка $O(n^2)$.**

4. **Матрица инцидентности** - прямоугольная матрица n*m (строки соответствуют вершинам, столбцы – дугам или ребрам), с элементами для орграфа:

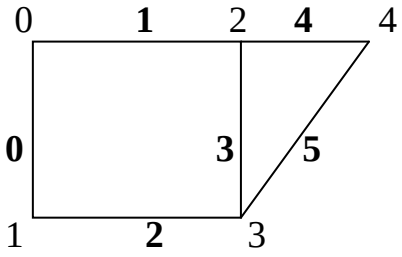
$$b[i,j] = \begin{cases} -1, \text{ если i-я вершина является началом } & \text{j-й дуги;} \\ 1, \text{ если i-я вершина является концом } & \text{j-й дуги;} \\ 0, \text{ если i-я вершина не инцидентна } & \text{j-й дуге;} \\ 2, \text{ если j-я дуга - петля } i\text{->}i. & \end{cases}$$

Для неориентированного графа (рис. 4.1 г):

$$b[i,j] = \begin{cases} 1, \text{ если i-я вершина инцидентна j-му ребру;} \\ 0, \text{ если i-я вершина не инцидентна j-му ребру;} \\ 2, \text{ если j-е ребро - петля } i\text{-}i. \end{cases}$$

Требует m*n бит, из которых m*(n-2) - нули. Много памяти: порядка O(n*m), $m \le n^2$. Неудобна для большинства операций.

а) Граф



б) Последовательность ребер

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

в) Матрица смежности

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	0	1	0
2	1	0	0	1	1
3	0	1	1	0	1
4	0	0	1	1	0

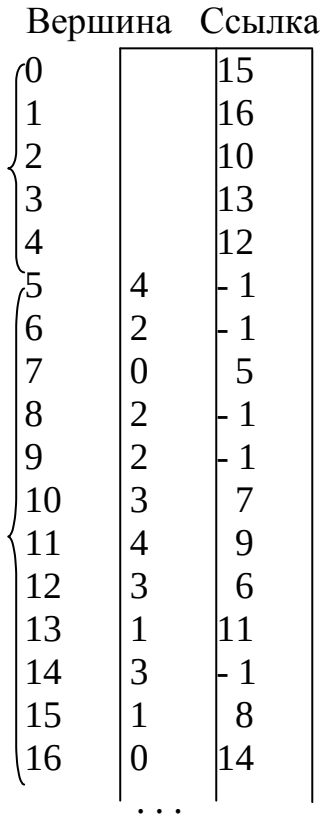
г) Матрица инцидентности

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

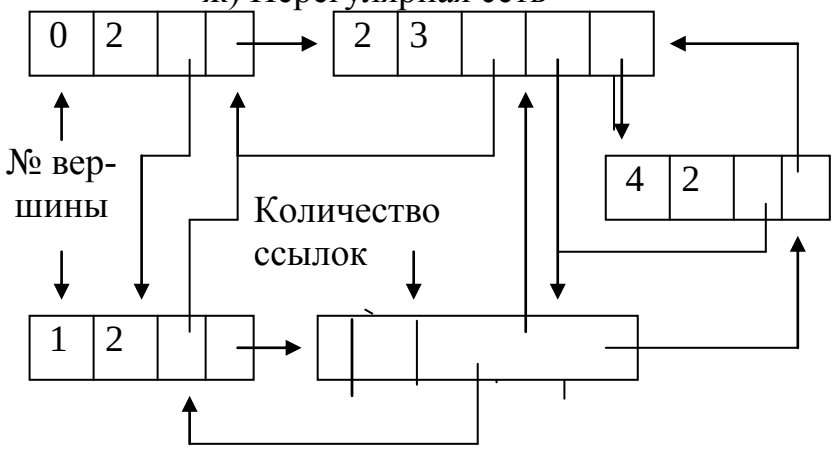
д) Векторы смежности

	0	1	2
0	1	2	
1	0	3	
2	3	0	4
3	1	4	2
4	3	2	

е) Списки смежности



ж) Нерегулярная сеть



з) Списковая структура

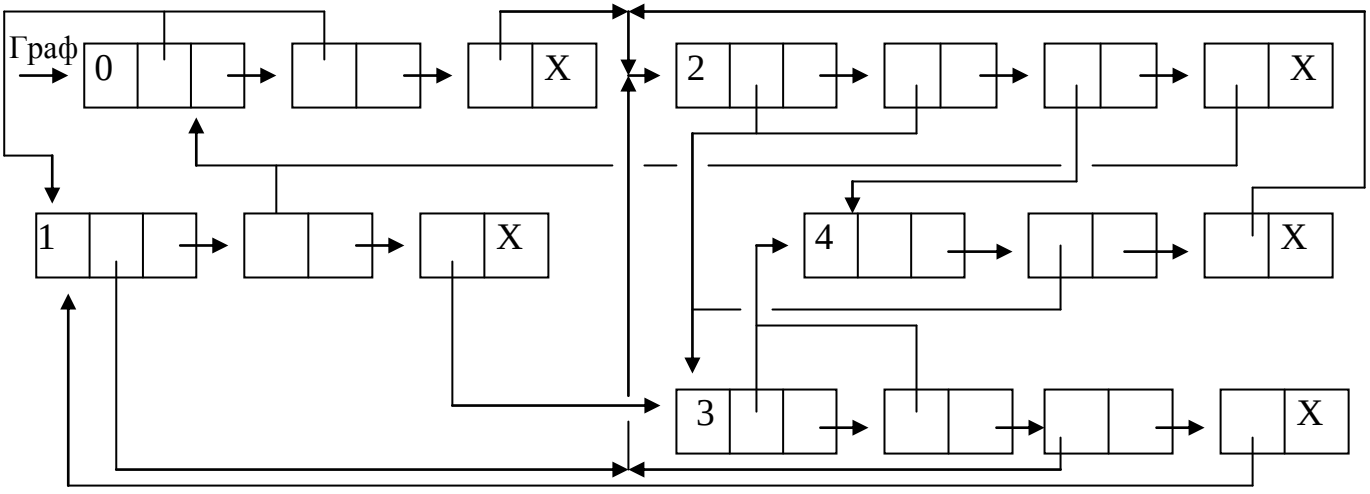


Рис. 4.1. Методы представления графа

5. **Структуры смежности**. Для каждой вершины x хранится множество номеров смежных с ней вершин *соседи*(x), в орграфе – *преемники*(x), в виде:

а) *вектора смежности* фиксированной длины (рис. 4.1 д) либо переменной длины;

б) *списка смежности*. Например, на рис. 4.1 е показаны списки смежности в параллельных массивах, полученные при вводе последовательности ребер из рис. 4.1 б по алгоритму 4.1. Память порядка O(n+m), удобны для различных операций, в том числе с несколькими графами.

Алгоритм 4.1. Ввод перечня ребер и получение списков смежности графа в параллельных массивах

```
#define PUSTO -1                    /* Пустая ссылка                          */
Ввод n;
for (i=0; i<n; i++)
      Ссылка[i] = PUSTO;            /* Создание пустых списков                */
for (i=n; не конец данных; i+=2)    /* Свободная позиция                      */
{   Ввод j, k;                      /* Ребро j-k = дуги j->k и k->j           */
   /* Включить вершину j в начало списка соседей k-й вершины                 */
   Вершина[i]=j;  Ссылка[i]=Ссылка[k];  Ссылка[k]=i;
   /* Включить вершину k в начало списка соседей j-й вершины                 */
   Вершина[i+1]=k;  Ссылка[i+1]=Ссылка[j];  Ссылка[j]=i+1;
}
```

6. **Сеть** - структура хранения, наиболее похожая на граф и допускающая большое разнообразие вариантов. На рис. 4.1 ж - нерегулярная сеть, на рис. 4.3 в - регулярная. Элементы сети полезно связать списком по возрастанию номеров вершин.

7. **Списковые структуры** для графа также могут быть разнообразными, например, списковая структура на рис. 4.1 з содержит список вершин, к которым прицеплены списки дуг. На рис. 4.3 б вершины не образуют списка и соединены только дугами.

8. **Алгоритм-перечислитель** (*итератор*), который перечисляет (выдает один за другим) всех преемников (соседей) заданной вершины графа. Этот способ возможен только для графа, построенного по определенной закономерности. Требуемая память не зависит от размера графа.

Сети и списки более удобны для добавления и удаления вершин, чем матрицы. Выбор представления графа может сильно влиять на эффективность алгоритма. С другой стороны, если время решения задачи порядка не менее,

чем $O(n^2)$, то оно мало зависит от способа представления, который можно изменить за время $O(n^2)$.

4.2. Дерево

Дерево – это связный неориентированный граф без циклов (или связный граф с минимальным числом ребер). *Связным* называют граф, в котором для любых двух вершин существует связывающий их путь. Дерево имеет n-1 ребер, где n - число вершин.

Дерево с одной выделенной вершиной (корнем) называют *корневым деревом*. В дальнейшем всюду в данном пособии под деревом понимается корневое дерево.

Дерево обычно рассматривают как ориентированное, причем от корня (реже - к корню), но изображают без стрелок. Преемников вершины называют ее *сыновьями*, предшественника – *отцом* (в этом случае более уместно называть вершину узлом – словом мужского рода).

Корень обычно изображают наверху, на уровне 0, остальные вершины - ниже, на уровнях, соответствующих расстоянию от корня. Сыновья вершин уровня i относятся к уровню i+1. Высотой дерева считают максимальный уровень его вершин.

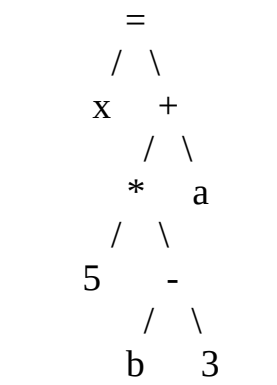
Степень вершины равна числу ее сыновей (в отличие от других графов, не учитывается входящая сверху дуга). Вершины без сыновей называют *терминальными* (концевыми) вершинами или *листьями*, вершины, имеющие сыновей, называют *внутренними*. Дерево называют *упорядоченным*, если сыновья каждой вершины каким-либо образом упорядочены.

Особо важную роль играет *бинарное дерево* - упорядоченное дерево степени два: вершина имеет не более двух сыновей, образующих корни ее левого и правого поддерева.

Деревья определяют очень распространенные *иерархические отношения* (подчиненность, старшинство, вложенность, родственные отношения и т. п.). Приведем примеры.

1. Структура книги, состоящей из разделов, подразделов и т. д. Вершина - часть книги, ее сыновья - вложенные в нее составные части. Аналогично представляется структура организации и ее подразделений; состав промышленного изделия из агрегатов и входящих в них узлов и деталей; программа из модулей, подпрограмм, блоков и операторов; скобочная структура выражения; грамматическая структура предложения и т. д.

На рис. 4.2 - структура оператора присваивания: листьями являются операнды; внутренняя вершина представляет операцию, а ее сыновья – операнды этой операции, которые могут быть результатом других операций.



x = 5 * (b – 3) + a

Рис. 4.2. Древовидная структура выражения

2. Генеалогические деревья: предки человека (бинарное дерево) и его потомки. В случае брака между потомками и наличия у них общего ребенка ветви дерева потомков срастаются, образуя орграф более общего вида, чем дерево.

3. История спортивного турнира с выбыванием побежденного - бинарное дерево: вершина - игра, сыновья - ее участники или определившие их игры.

Операции над деревьями такие же, как для графов.

Представление деревьев

Для деревьев можно использовать любые методы хранения графов, но матрица смежности неудобна, т. к. сильно разрежена (доля единиц меньше, чем 2 / n, где n - число вершин). Обычно используют списковые структуры и сети. На рис. 4.3 а - 4.3 в показано дерево и примеры его представления списковой структурой и регулярной сетью (см. также рис. 5.3). Списковая структура (рис. 4.3 б) содержит элементы двух типов: для вершин и для дуг.

Неудобство нерегулярной сети - разные размеры элементов из-за разного числа ссылок. У регулярной сети много места занимают пустые ссылки.

Этих недостатков можно избежать, если вместо исходного дерева хранить бинарное дерево с теми же вершинами, полученное из исходного дерева по принципу *“левый сын – правый брат”*. Каждая вершина бинарного дерева имеет две ссылки: к ее левому сыну и правому брату в исходном дереве. Это преобразование взаимно однозначно и по бинарному дереву можно восстановить исходное дерево.

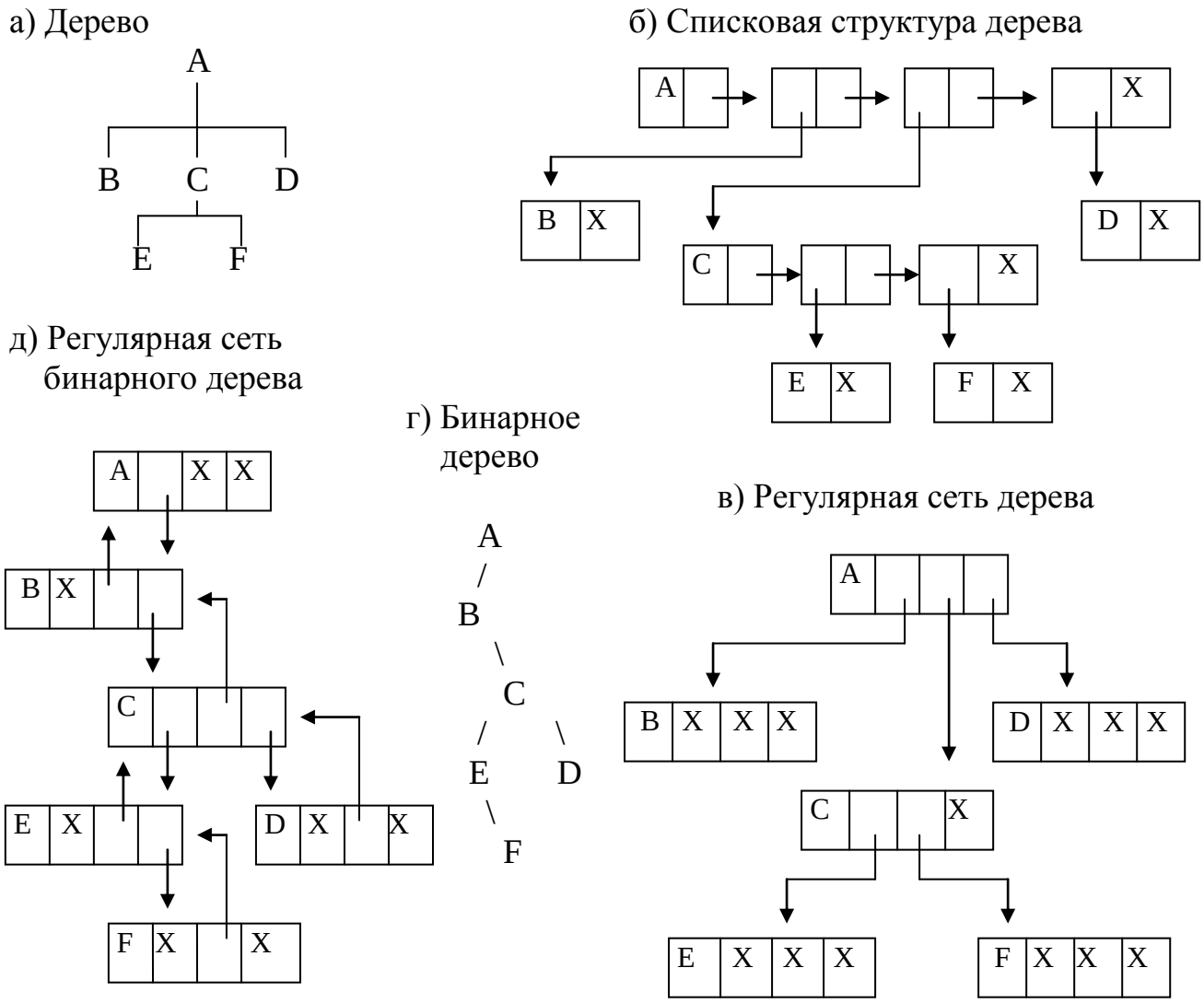


Рис. 4.3. Способы представления деревьев

Таким способом можно хранить и совокупность деревьев – *лес*, т.е. несвязный граф без циклов. Корни составляющих лес деревьев считаются братьями и упорядочиваются по какому-либо принципу.

На рис. 4.3 г показано бинарное дерево, соответствующее дереву рис. 4.3а; на рис. 4.3 д - представление бинарного дерева в виде регулярной сети с тремя ссылками: не только вниз к двум сыновьям, но и вверх к отцу.

Дополнительные ссылки расширяют пути доступа, но тратится память и время на их корректировку - нужен компромисс.

На рис. 4.6 дерево представлено вектором: у каждой вершины одна ссылка наверх, к отцу. В разделе 3.3.3 бинарные деревья представляются с помощью адресной арифметики.

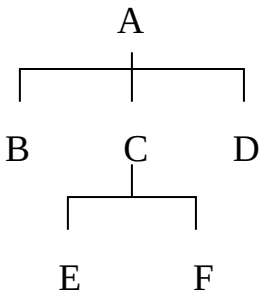
4.3. Обход дерева и графа

В некоторых задачах необходимо обойти дерево или граф в определенном порядке с посещением один раз каждой вершины для выполнения некоторой операции, например поиска чего-либо.

Обход (*поиск*) в графах и деревьях можно выполнять в ширину и в глубину. Для деревьев (особенно бинарных) часто используют еще обходы сверху, снизу и слева направо.

4.3.1. Обход в ширину

Обход дерева в ширину происходит по уровням: сначала посещается вершина уровня 0 (корень), затем вершины уровня 1, затем уровень 2 и т. д. Уровень i+1 включает сыновей вершин уровня i (рис. 4.4 а, 4.4 б; алг. 4.2).



б) Обход в ширину:
A, B, C, D, E, F

в) Обход в глубину:
A, B, C, E, F, D

Рис. 4.4. Обход дерева в ширину и в глубину

Алгоритм 4.2. Обход дерева в ширину

```
Пустая Очередь <== корень;
while (Очередь != пусто)
{ Очередь ==> t;  Посетить t;
   for (x ∈ сыновья(t))
     Очередь <== x;
}
```

Обозначение: for (x ∈ S) . . . - цикл, выполняемый для каждого элемента x из множества S (∈ - "принадлежит").

Трассировочная таблица обхода дерева из рис. 4.4 а:

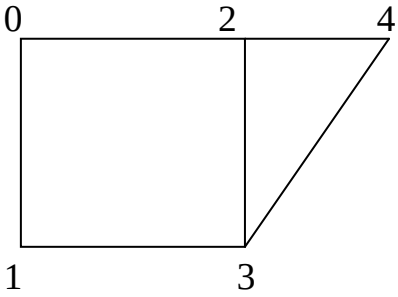
Посещение:	A	B	C	D	E	F
Очередь=	A	B C D	C D	D E F	E F	F	

Обход в ширину выполняется с помощью очереди вершин на посещение. Из головы очереди выбирается и посещается вершина, а ее сыновья заносятся в хвост очереди.

Пока из очереди выбираются и посещаются все вершины уровня i, за ними в очереди расположатся все вершины уровня i+1. Первоначально в очередь помещается единственная вершина уровня 0 – корень дерева. На удивление простой и изящный алгоритм!

Обход графа в ширину можно рассматривать как обход в ширину дерева его путей, начинающихся с какой-либо вершины. Он также выполняется по уровням: заданная начальная вершина, затем - вершины на расстоянии 1 от нее, затем - на расстояниях 2, 3 и т. д. (рис. 4.5).

Подобный обход иногда называют "*волновым алгоритмом*", т.к. последовательность обхода напоминает круговые волны от брошенного в воду камня.



Обходы в ширину:

0, 1, 2, 3, 4
1, 0, 3, 2, 4
2, 0, 3, 4, 1
3, 1, 2, 4, 0
4, 2, 3, 0, 1

Рис. 4.5. Обход графа в ширину

Алгоритм 4.2 обхода дерева в ширину для обхода графа дополняется учетом посещений: при записи в очередь на посещение вершина j отмечается в векторе p, чтобы избежать ее повторного посещения. Так получен алгоритм 4.3.

При этом удобно в p[j] записывать номер вершины v, предшествующей j на пути к j от начала обхода. Этот путь является кратчайшим, т. к. обход ведется по уровням, и на предыдущих уровнях вершина j не встретилась (т.е. за меньшее число шагов до нее дойти невозможно).

Первоначально вектор p заполняется значениями NOV, обозначающими новые вершины, еще не записанные в очередь на посещение.

В результате вектор p будет содержать дерево кратчайших путей от каждой вершины графа до начальной вершины (вот вам еще один метод хранения дерева – с одной ссылкой, дуги направлены к корню).

На рис. 4.6 показано дерево кратчайших путей от всех вершин графа до вершины 1, которая является началом обхода.

Алгоритм 4.3. Обход связного графа (орграфа) в ширину, начиная с вершины vn

```
#define NOV -1          /* Вершина новая - не была в очереди          */
int p[n];               /* Учет посещений; пути к началу обхода       */
    . . .
```

```
/* Подпрограмма обхода графа в ширину, начиная с вершины vn                */
void obhod_gr_sh (int vn)
{  Пустая Очередь <== vn; p[vn] = vn;
   do
   {  Очередь ==> v; Посетить v;                    /* выполняется n раз    */
      for (j ∈ Преемники(v))
         if (p[j]==NOV)                             /* выполняется m раз    */
         {   Очередь <== j; p[j] = v; }             /* выполняется n раз    */
   } while (Очередь != пусто);                      /* выполняется n раз    */
}
. . .
for (i=0; i<n; i++)   p[i] = NOV;          /* Инициализация p               */
obhod_gr_sh (vn);                          /* Обход графа, начиная с vn     */
```

Кратчайший путь из i в vn - последовательность v[1], ... , v[L], где v[1]=i, v[L]=vn, v[k]=p[v[k-1]] для k>1.

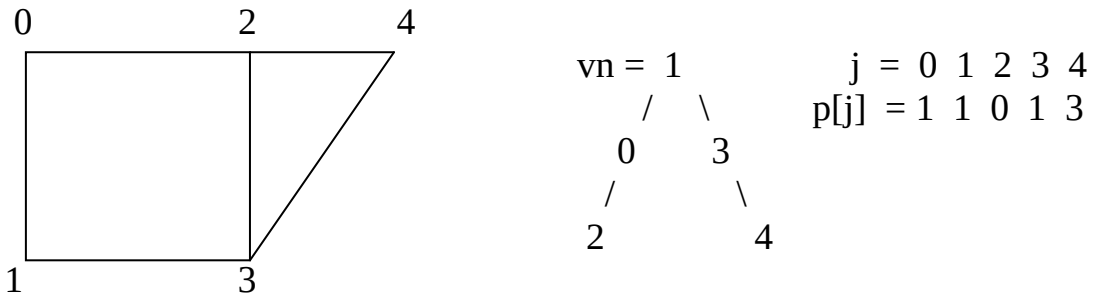


Рис. 4.6. Дерево кратчайших путей графа к вершине 1

Для несвязного графа алг. 4.3 посещает вершины только той компоненты связности, в которой начат обход (он позволяет находить компоненты связности). Посещение всех вершин графа обеспечивает алгоритм 4.4.

Алгоритм 4.4. Обход графа (орграфа) в ширину

```
for  (i=0; i<n; i++) p[i] = NOV;           /* Инициализация  p              */
for  (vn=1; vn<=n; vn++)
     if (p[vn]==NOV) obhod_gr_sh(vn);      /* Обход, начиная с vn           */
```

Обход в ширину требует времени порядка O(n+m), т. к. каждая из n вершин один раз посещается, попадает в очередь и удаляется из нее. Дополнительно один раз просматривается каждая из m дуг графа (при переборе преемников ее начальной вершины).

Для поиска пути между двумя вершинами графа обход в ширину удобнее, чем в глубину, т. к. дает кратчайший путь.

4.3.2. Обход в глубину

Деревья имеют рекурсивное строение: части дерева сами могут быть деревьями. Поэтому можно дать рекурсивное определение (корневого) дерева: *дерево* - либо пустое множество вершин, либо вершина (корень), связанная с конечным числом непересекающихся деревьев (называемых *поддеревьями*).

На рис. 4.4 а, 4.4 в показано дерево и последовательность его *обхода в глубину* по принципу: если можно – вперед (вглубь) к сыну, иначе - назад к отцу.

На основе рекурсивного определения удобно строить рекурсивные программы обработки деревьев. Примером является рекурсивный обход дерева в глубину (алг. 4.5).

Алгоритм 4.5. Рекурсивный обход в глубину дерева с корнем t

```
void obhod_der_gl (t)
{   if (дерево t != пусто)
    {  Посетить корень t;
       for (x ∈ поддеревья(t))        /*Для каждого x из поддеревьев t        */
          obhod_der_gl (x);           /* Обойти поддерево x                   */
    }                    ↑
}                    Возврат 2
```

Трассировочная таблица обхода дерева (рис. 4.4 а) по алгоритму 4.5:

Вызов: obhod_der_gl (A); /* Возврат 1 */

N вызова	1	2	1	3	4	3	5	3	1	6	1
t =	A	B	A	C	E	C	F	C	A	D	A
x =	B		C	E		F		D			
Посещение:	A	B		C	E		F			D	
Стек =											
t, возврат	A 1	A 1 B 2	A 1	A 1 C 2	A 1 C 2 E 2	A 1 C 2	A 1 D 2 F 2	A 1 C 2	A 1	A 1 D 2	A 1	

Алгоритм 4.5 удобен для представлений дерева с возможностью пустых ссылок на поддеревья (типа рис. 4.3 в, 4.3 д). Когда пустые поддеревья в явном виде отсутствуют (ссылки делаются только на непустые поддеревья, как на рис. 4.3 б или в нерегулярных сетях), удобнее алгоритм 4.5а, основанный на другом определении: (непустое корневое) *дерево* – это вершина (корень), связанная с корнями n деревьев ($n \geq 0$).

Алгоритм 4.5а. Рекурсивный обход в глубину непустого дерева с корнем t

```
void obhod_der_gl (t)
{   Посетить t;
    for (x ∈ сыновья(t))        /* Для каждого x из сыновей t                */
        obhod_der_gl (x);       /* Обойти поддерево с корнем x               */
}
```

Рекурсивный алгоритм часто проще итеративного, но тратит больше времени и памяти на рекурсивные вызовы. Итеративный алгоритм предпочтительнее, если он не слишком сложен.

Алгоритм 4.6. Итеративный обход в глубину непустого дерева

```
Посетить корень; Пустой Стек <== корень;
while (Стек != пусто)                                              Идея:
{   t = верх(Стек);                                                 2
    if (t имеет не посещенных сыновей, левый из них x)              ↕
    {   Посетить x; Стек <== x; }              /* Вглубь    */      ┼
     else
        Стек==>;                               /* Назад     */      1
}
```

Строку /* Назад */ алгоритма 4.6 можно заменить на три строки /* Назад или вправо */ (см. алг. 4.7): если невозможно двигаться вглубь, делается попытка перейти к брату и только при его отсутствии - назад. Это несколько ускоряет обход (меньше итераций) за счет усложнения алгоритма.

Трассировочная таблица обхода дерева (рис. 4.4 а) по алгоритму 4.6:

t=	A	A	B	A	C	E	C	F	C	A	D	A
Посещение:	A	B		C	E		F			D		
Стек=	A	A	A	A	A	A	A	A	A	A	A	
	↕	B		C	C	C	C	C		D		
					E		F					

Алгоритм 4.7. Итеративный обход в глубину непустого дерева

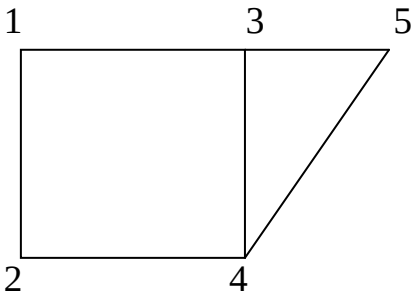
```
Посетить корень; Пустой Стек <== корень;
while (Стек != пусто)
{  t = верх(Стек);                                                  Идея:
   if (t имеет не посещенных сыновей, левый из них x)               3
   {    Посетить x; Стек <== x; }                 /* Вглубь   */   ↕
  else                                                              ┼─ 2
   {  Стек ==> t;                                 /* Назад    */
     if (t имеет правого брата x)                 /* или      */
      { Посетить x; Стек <== x; }                 /* вправо   */   1
   }
}
```

Трассировочная таблица обхода дерева (рис. 4.4 а) по алгоритму 4.7:

t=	A	A	B	C	E	F	C	D	A
Посещение:		B	C	E	F		D		
Стек=	A	A B	A C	A C E	A C F	A C	A D	A	

Пример 4.1. Рассмотрим обход графа в глубину на примере следующей задачи. Найти кратчайший цикл в графе.

На рис. 4.7 показан граф, содержащий циклы (замкнутые пути) длины пять: 1, 2, 4, 5, 3, 1; длины четыре: 1, 2, 4, 3, 1 и длины три: 3, 4, 5, 3. Длина пути - количество ребер.



Матрица смежности графа

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

Кратчайший цикл длины 3: 5 3 4 5

Рис. 4.7. Граф с циклами

Для этой задачи можно использовать универсальные методы перебора вариантов: обход графа в глубину (поиск с возвратом) и обход графа в ширину.

Возможные пути графа, начинающиеся с некоторой вершины, образуют дерево с корнем в этой вершине (в общем случае бесконечное из-за циклов). Вершины, смежные с некоторой вершиной графа, соответствуют ее сыновьям в дереве путей. Соединив все такие деревья с фиктивным корнем, получим дерево всех возможных путей графа (рис. 4.8).

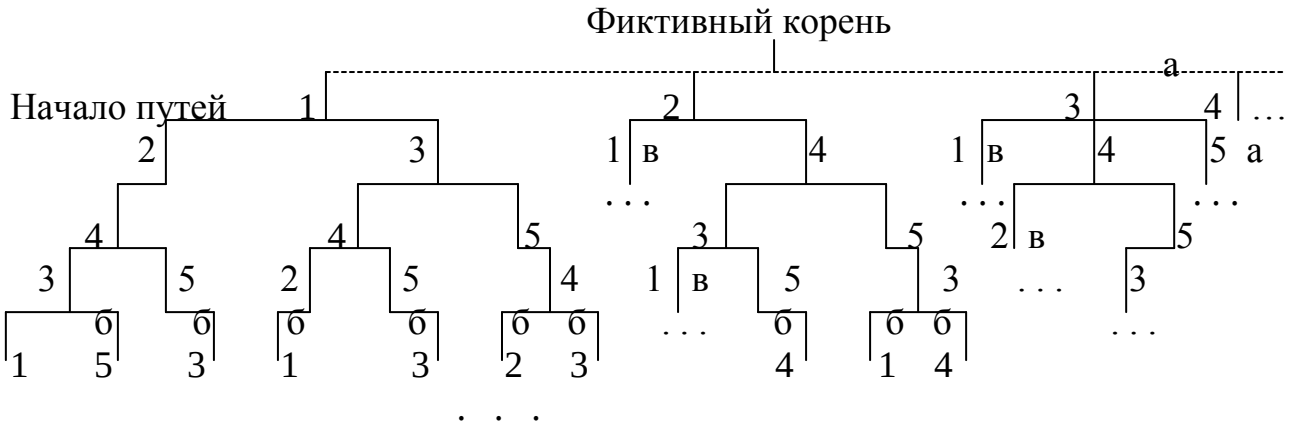


Рис. 4.8. Дерево путей графа, изображенного на рис. 4.7

Обход графа в глубину или в ширину можно рассматривать как обход в глубину или в ширину дерева путей этого графа.

Для поиска кратчайшего цикла графа организуем перебор всех его путей, среди них будут и все циклы. Этот перебор можно рассматривать как обход в глубину графа (дерева его путей), основанный на алгоритме 4.6.

Не существуют циклы длиной менее трех. Самый длинный простой цикл (без повторяющихся вершин, т. е. вложенных циклов), проходящий через все вершины, имеет длину n, т.к. при большей длине путь содержит повторяющиеся вершины. Таким образом, длина искомого кратчайшего цикла лежит в пределах от 3 до n. Возможен граф без циклов - *лес*; в частном случае – дерево.

Выбранный алгоритм 4.6 конкретизирован в виде подпрограммы pminc (алгоритм 4.8). В подпрограмме pminc для сокращения перебора путей используются следующие правила отсечения ветвей - эвристики (в скобках приведена соответствующая формальная запись). *Эвристика* (от греческого «эврика» - я нашел!) – это неформальный метод, основанный на догадках.

а) Прекратить поиск, обнаружив цикл длиной 3 (*dcmin==3).

б) Отвергать пути, длиннее минимального из найденных циклов или равные ему (k>= *dcmin). Поэтому, найдя цикл, можно удалить из стека сразу две вершины (k=k-3; ... k++;).

в) Отвергать пути, ведущие в вершину, ранее использованную в качестве начальной (vn < v[0]), т.к. все проходящие через нее циклы уже просмотрены. По этой причине перебор по возрастанию возможных номеров очередной вершины пути начинать не с 1, а с начальной вершины текущего пути (v[к+1] = v[0]).

На рис. 4.8 отсекаемые по этим эвристикам ветви дерева отмечены соответствующими буквами. Места использования эвристик показаны в комментариях к подпрограмме pminc.

Эти эвристики дают эффект не всегда, а только при соответствующих обстоятельствах, например, эвристика а) - только при наличии цикла длины 3, но всегда требуют времени на дополнительные проверки. Общий эффект использования эвристик зависит от вероятности возникновения благоприятных обстоятельств в исходных графах.

Алгоритм 4.9 представляет собой другую версию подпрограммы pminc для поиска кратчайшего цикл графа, использующую обход в ширину.

При обходе неориентированного графа в ширину цикл обнаруживается, когда повторно встречается вершина j, ранее уже включенная в очередь на посещение, т.е. найден второй путь от начальной вершины обхода vn до вершины j. По одному из этих путей можно от вершины vn добраться до вершины j, а по другому пути – вернуться обратно, т.е. замкнуть цикл. Для орграфа этот алгоритм не подходит (в обратную сторону двигаться нельзя).

Алгоритм 4.8. Подпрограмма поиска кратчайшего цикла графа обходом в глубину

```
/* Поиск минимального цикла в графе с количеством вершин n,                    */
/* матрицей смежности g. Значение функции: 0 - цикл найден,                    */
/* 1 - граф не имеет циклов. Номера вершин найденного цикла                    */
/* - в векторе c, его длина - dcmin (3..n).                                    */
/* Метод: итеративный обход графа в глубину.                                   */

#define NMAX 20                     /* Максимальное количество вершин графа    */

int pminc (int n, char g[][NMAX], int *dcmin, int c[])
{   int v[NMAX+1];                  /* Стек с номерами вершин текущего пути    */
    int k;                          /* Указатель стека v                       */
    int j;                          /* Номер строки матрицы смежности          */
    int vn;                         /* Номер очередной вершины текущего пути   */
   *dcmin = n + 1;                  /* Длина минимального цикла (3..n)         */
    for (v[0]=1; v[0]<=n && *dcmin>3;  )          /* Начальная вершина: 1..n   */
                        /*      Эвристика а); v[0]++ за счет v[k]++             */
   {     /* Обход в глубину дерева путей, начинающихся с v[0]                   */
         k = 1; v[1] = v[0]+1;    /*Начальный номер преемников v[0]             */
        do                  /*  Эвристика в)                                   */
        {   /* Найти вершину vn - очередного преемника v[k-1]                  */
            j = v[k-1];
            for (vn=v[k]; vn<n && (g[j][vn]==0 || k>1 && vn==v[k-2]); ) vn++;
            if (vn<n &&             /* Есть путь вперед                        */
                        k<*dcmin)   /* Эвристика б)                            */
            { v[k] = vn;            /* Вперед: vn - в стек                     */
              v[k+1] = v[0];        /* Начальный преемник v[k]                 */
                        /* ↑ Эвристика в)                                      */
              if (v[0]==v[k] && k>0)  /* Нашли цикл                            */
              {   *dcmin = k;
                  /* Запомнить цикл v[0]...v[k]                                */
                  for (j=0; j<=k; j++)  c[j] = v[j];
                  k = k - 3;                /* Назад: удалить 2 вершины пути   */
                        /*  Эвристика б)                                       */
                  v[k+1]++;                 /* Следующий преемник v[k]         */
                 }
               k++;
            }
           else                     /* Назад                                   */
           {  k--;                  /* Удалить v[k] из стека                   */
              v[k]++;                       /* Следующий преемник v[k-1]       */
            }
         }
        while (k>0 && *dcmin>3);    /* Стек не пуст && Эвристика а)             */
   }
   return *dcmin > n;               /* Если *dcmin>n, циклов нет               */
}
```

Алгоритм 4.9. Подпрограмма поиска кратчайшего цикла графа обходом в ширину

```
/* Поиск минимального цикла в графе с количеством вершин n,                  */
/* матрицей смежности g. Значение функции: 0 - цикл найден,                  */
/* 1 - граф не имеет циклов. Номера вершин найденного цикла                  */
/* - в векторе cmin, его длина - dcmin (3..n)                                */
/* Метод: обход графа в ширину.  Д.Г. Хохлов 18.04.95                        */
#define NMAX 20     /* Максимальное количество вершин графа                  */
#define NOV -1              /* Вершина новая, не встречалась                 */

int pminc (int n, char g[][NMAX], int *dcmin, int cmin[])
{  int p[NMAX];             /* Кратчайшие пути к начальной вершине           */
   int q[NMAX];             /* Очередь вершин на посещение (вектор)          */
   int in, ik;              /* Индексы начала и конца очереди                */
   int i, j;                /* Номера вершин                                 */
   int vn, v, vpr;          /* Номер начальной, текущей и предыдущей вершин  */
   int c[NMAX+1];    /* Текущий цикл                                         */
   /* Поиск кратчайшего цикла обходом графа в ширину                         */
   *dcmin = n + 1;          /* Длина минимального цикла (3..n)               */
   for (vn=0; vn<n && *dcmin>3; vn++)     /* Начальная вершина              */
   {  /* Обход в ширину дерева путей, начинающихся с vn                      */
      for (i=0; i<n; i++) p[i] = NOV;          /* Все вершины новые         */
      in=0; ik=1; q[0]=vn;                     /* Пустая Очередь <== vn     */
      do
      {  /* Взять из (непустой) очереди и посетить вершину v                */
         v=q[in]; ++in; /* q[0 .. n-1] */
         vpr=p[v];
         for (j=0; j<n; j++)
              if (g[v][j]==1 && j!=vpr)
                 if (p[j] == NOV)                  /* Вершина j не посещалась */
                 {  /* Очередь на посещение <== j                           */
                    q[ik]=j; ++ik; /* q[0 .. n-1] */
                    p[j] = v;                      /* Предыдущая вершина пути к vn */
                 } else                            /* Цикл из двух путей vn...j */
                 {   /* Создать список vn->...->v->j                        */
                     for (i=v; j!=vn;)
                     {  vpr=p[i]; p[i]=j; j=i; i=vpr; }
                     /* Запомнить цикл v->j...->v в векторе c               */
                     c[0]=v; i=v; j=0;
                     do { i=p[i]; c[++j]=i; } while (i!=v);
                     if (j < *dcmin)
                     {   /* Запомнить цикл c[0]...c[j]                      */
                        *dcmin = j;
                         while (j>=0) cmin[j]=c[j--];
                     }
                     in=ik; break;
                 }
      } while (in!=ik && *dcmin>3);              /* Очередь не пуста        */
    }
   return *dcmin > n;
}
```

4.3.3. Обход бинарного дерева

Для обработки бинарного дерева удобно использовать его рекурсивное определение: б*инарное дерево* – это либо пусто (пустое множество вершин), либо вершина (корень), соединенная с двумя бинарными деревьями – левым и правым поддеревом.

Обход бинарного дерева включает посещение корня дерева и (рекурсивный) обход двух поддеревьев.

Рассмотрим обход бинарного дерева на примере следующей задачи. Составить алгоритм вывода информации всех вершин бинарного дерева. Рекурсивная подпрограмма print_subtree(x) по одному разу выводит все вершины дерева с корнем kor (алг. 4.10). Главная программа содержит вызов print_subtree(корень).

Алгоритм 4.10. Обход бинарного дерева сверху

```
void print_subtree (ссылка kor);
{
   if (kor != NULL)
   {  Печать (kor -> информация);
      print_subtree (kor -> левая_ссылка);
      print_subtree (kor -> правая_ссылка);
   }
}
```

Приведенный алгоритм выводит сначала корень дерева, затем его левое поддерево, а затем правое поддерево. Три строки в операторе if можно переставить шестью способами, и каждый из этих способов дает свой порядок вывода вершин.

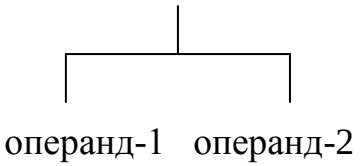
Наибольшее применение получили следующие три способа обхода бинарного дерева, названные в зависимости от того, когда посещается корень (в литературе встречаются и другие названия):

Прямой обход (обход сверху):	*Внутренний обход (обход слева направо):*	*Обратный обход (обход снизу):*
1. Корень	1. Левое поддерево	1. Левое поддерево
2. Левое поддерево	2. Корень	2. Правое поддерево
3. Правое поддерево	3. Правое поддерево	3. Корень

Прямой и обратный обходы возможны для любого (корневого) дерева.

На рис. 4.9 показано бинарное дерево, описывающее структуру выражения, и способы его обхода. Внутренняя вершина дерева соответствует операции, ребра ведут к ее операндам (листьям) или определяющим их операциям. Прямой и обратный обходы вершин дерева дают, соответственно, прямую и обратную польскую запись выражения (префиксную и постфиксную запись) – см. раздел 7.2.1.

а) Элементарное выражение

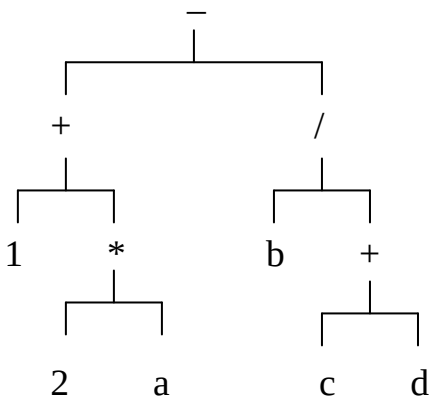


Прямой обход:
операция операнд-1 операнд-2

Внутренний обход:
операнд-1 операция операнд-2

Обратный обход:
операнд-1 операнд-2 операция

б) Выражение 1 + 2 * a - b / (c + d)



Прямой обход:
-+1*2a/b+cd

Внутренний обход :
1+2*a-b/(c+d)

Обратный обход:
12a*+bcd+/-

Рис. 4.9. Дерево выражения и способы его обхода

Вывод вершин дерева в порядке внутреннего обхода образует инфиксную запись выражения, в записи которого желательно использовать минимум скобок.

Перед выполнением каждой операции - корня некоторого дерева - необходимо знать ее операнды, т.е. выполнить (корневые) операции ее поддеревьев. Для этого выражение поддерева заключается в скобки, когда без скобок корневая операция дерева выполнилась бы раньше.

4.4. Кратчайшие пути и расстояния

У взвешенного графа *длина (вес) пути* определяется как сумма весов его дуг. *Расстояние* D[i,j] между вершинами i и j - длина кратчайшего пути между ними (может быть и отрицательной). Считают, что расстояние от вершины до нее самой равно нулю: D[i,i] = 0. В частном случае обычного, не взвешенного, графа вес дуги равен 1; длина пути и расстояние измеряются числом дуг.

Алгоритм Дейкстры

Для неотрицательных весов расстояния (и кратчайшие пути) от одной вершины до всех остальных можно получить обходом в ширину взвешенного орграфа по алгоритму Дейкстры (алг. 4.11.) с временем работы $O(n^2)$.

Алгоритм 4.11. Определение расстояний d[j] от вершины ist до всех вершин j взвешенного орграфа с неотрицательными весами дуг w[i][j] поиском в ширину (Э. Дейкстра, 1959)

```
for (j=0; j<n; j++)  d[j]=w[ist][j];
d[ist]=0;           /* Расстояние(ist,ist) = 0 по определению */
T = V - {ist};
while (T != пусто)
{  /* Инвариант: для всех j из V-T: d[j] = Расстояние (ist,j),*/
   /*            кратчайший путь проходит через V-T;          */
   /*            для всех j из T: d[j]=min длина путей ist-j, */
   /*            проходящих через V-T.                        */
   Найти в T вершину k с наименьшим d[k];
   T = T - {k};
   for (j ∈ T)
      if (d[k] + w[k][j] < d[j])
         d[j] = d[k] + w[k][j];
}
```

Обозначения:

n - количество вершин графа;

V - множество вершин графа;

T - множество вершин, расстояние до которых определено не окончательно;

V-T - (разность множеств V и T) множество вершин с окончательно определенными расстояниями.

Множество T уменьшается по следующему принципу: для вершины k из T, у которой число d[k] минимально, это число является окончательным значением искомого расстояния. Допустим, что есть более короткий путь. Как следует из инварианта, он должен проходить через вершины не только множества V-T, но и множества T. Рассмотрим первую вершину из T на этом пути – уже до нее путь длиннее! Здесь существенна неотрицательность весов.

Удалив из множества T вершину k, необходимо для всех вершин j из T скорректировать числа d[j], чтобы они оставались минимальными длинами путей ist-j, проходящих через V-T. При этом достаточно учесть лишь пути через вершину k длиной d[k]+w[k][j], в которых k является предпоследней вершиной пути, т.к. в любом пути вида ist-k-m-j, где m принадлежит V-T, участок ist-k-m можно заменить более коротким ist-m (без k), а все такие пути уже рассмотрены ранее.

Множество T можно представить характеристическим вектором, а для удобства проверки пустоты хранить количество его элементов.

Кратчайшие пути от вершины ist до всех вершин можно получить (по аналогии с вектором p алгоритма 4.3), если дополнить алгоритм Дейкстры построением вектора P, в котором P[j] - первая вершина кратчайшего пути из ist в j. Ускорение работы алгоритма Дейкстры рассмотрено в задаче 4.24.

Алгоритм Флойда

Алгоритм Дейкстры работает только для неотрицательных весов и получает расстояния (и кратчайшие пути) от одной вершины до всех остальных за время $O(n^2)$. Кратчайшие пути и расстояния между всеми вершинами быстрее определяются через матрицы. Алгоритм Флойда позволяет получить расстояния между всеми парами вершин, в том числе и для отрицательных весов, за время $O(n^3)$.

Обозначим $D_k[i,j]$ - длина кратчайшего пути от вершины i к вершине j через вершины из множества {0, ... , k}.

Тогда без промежуточных вершин (k = -1):

$$D_{-1}[i,j] = W[i,j] \qquad (4.1)$$

где W - матрица весов с бесконечными весами отсутствующих дуг.

Если кратчайший путь от вершины i к вершине j через множество вершин {0, ..., k-1, k} не содержит вершины k, то

$$D_k[i,j] = D_{k-1}[i,j],$$

иначе его можно разделить на два пути, проходящих через вершины множества {0,...,k-1}: от вершины i до вершины k и от вершины k до вершины j, причем:

$$D_k[i,j] = D_{k-1}[i,k]+D_{k-1}[k,j].$$

Отсюда:

$$D_k[i,j] = \min (D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]) \qquad (4.2)$$

При k = n-1, когда множество возможных промежуточных вершин путей охватит все вершины графа, расстояние между вершинами i и j примет окончательное значение, равное:

$$D[i,j] = D_{n-1}[i,j].$$

Отсюда получен алгоритм. 4.12 сложностью $O(n^3)$ операций. В нем используется то обстоятельство, что k-я строка и k-й столбец матрицы D не изменяются на шаге k.

Если в графе есть циклы с отрицательной длиной (уменьшающие длину путей при каждом прохождении), расстояния между некоторыми парами вершин не существуют (поскольку длина пути может стать меньше любого числа).

Алгоритм 4.12. Вычисление расстояний D[i,j] между всеми парами вершин взвешенного орграфа (Р. Флойд, 1962)

```
D = W;                          /* Матрица весов (отсутствие дуги - бесконечность)   */
for (i=0; i<n; i++) D[i,i] = 0;             /* если D[i,i] бесконечны                */
for (k=0; k<n; k++)
      for (i=0; i<n; i++)
            if (D[i,k] != бесконечность)
                for (j=0; j<n; j++)
                     D[i,j] = min (D[i,j], D[i,k]+D[k,j]);
```

Кратчайшие пути между всеми вершинами можно получить (по аналогии с вектором p алгоритма 4.3) в виде матрицы P в которой P[i,j] - первая вершина кратчайшего пути из i в j. Тогда кратчайший путь из i в j - последовательность v[1], ... ,v[r], где v[1]=i, v[r]=j, v[k]=P[k-1,j] для k>1. Для вычисления P в начало алг. 4.12 вставляется инициализация:

```
if (W[i,j] - бесконечность) P[i,j] = -1;        /* Пока нет пути                 */
else P[i,j] = j;
```

и тело цикла по j заменяется на строки, обеспечивающие запоминание нового начала кратчайшего пути:

```
if (D[i,k]+D[k,j] < D[i,j])
{   P[i,j] = P[i,k];   D[i,j] = D[i,k]+D[k,j];  }
```

В конце работы останется P[i,j] = -1, если не существует пути из i в j.

Алгоритм Уоршалла

Основная идея алгоритма Флойда (увеличивать максимальный номер промежуточных вершин k) совпадает с идеей алгоритма Уоршалла, предназначенного для получения матрица достижимости (связности) графа из его матрицы смежности.

Матрица достижимости C размера n*n содержит элементы

$$C[i,j] = \begin{cases} 1, \text{ если в графе есть путь из вершины i в j} \\ 0, \text{ если в графе нет пути из вершины i в j} \end{cases}$$

Матрица достижимости C получается аналогично матрице расстояний D алгоритма Флойда 4.12. Обозначим $C_k[i,j]$ - есть путь от i к j через вершины из множества {0, ... , k}.

Тогда без промежуточных вершин:

$C_{-1}[i,j] = A[i,j]$ (A - матрица смежности графа) (4.3)

Если через вершины из множества {1, ... , k-1} есть путь от i до j или есть два пути: от i до k и от k до j, то есть путь от i к j через вершины из множества {0, ... , k}. Отсюда:

$$C_k[i,j] = C_{k-1}[i,j] \;||\; C_{k-1}[i,k] \;\&\; C_{k-1}[k,j] \qquad (4.4)$$

Из равенства $C[i,j] = C_{n-1}[i,j]$

следует алг. 4.12, требующий $O(n^3)$ операций.

Алгоритм 4.12. Вычисление матрицы C достижимости вершин орграфа (С. Уоршалл, 1962)

```
C = A;                    /* A - матрица смежности орграфа                */
for (k=0; k<n; k++)
      for (i=0; i<n; i++)
          if (C[i,k])   C[i,*] = C[i,*] || C[k,*];
```

C[i,*] обозначает i-ю строку матрицы C. Если элемент матрицы C занимает один бит, операцию "или" над i-й и k-й строками матрицы (в теле цикла) можно выполнять поразрядно (битовой операцией |), т. е. очень быстро.

Транзитивным замыканием бинарного отношения называют наименьшее содержащее его отношение, обладающее свойством транзитивности (передаваться транзитом через промежуточные элементы, "по наследству"):

X->Y и Y->Z ==> X->Z

(если в отношении "->" находятся X с Y и Y с Z, то X с Z тоже состоят в отношении "->").

Примеры. Пусть x, y - люди. Тогда:

а) Отношение "x старше y" транзитивно;

б) Отношение "x - родитель y" не транзитивно. Его транзитивным замыканием является отношение "x - предок y".

Дуги графа задают бинарное отношение между вершинами. Матрицу достижимости графа часто обозначают A*, т. к. она определяет транзитивное замыкание бинарного отношения, задаваемого графом (матрицей смежности A). Матрицу C можно получить еще и следующим способом

$$C = A^* = A + A^2 + A^3 + ... + A^{n-1} \qquad (4.5)$$

Здесь при вычислении *произведения Z логических матриц* X и Y, состоящих из единиц и нулей, арифметическое сложение и умножение заменяются на логические операции ИЛИ и И (+ на ||, * на &&):

$$Z[i,j] = \bigvee_{k=0}^{n-1} X[i,k] \;\&\&\; Y[k,j] \qquad (4.6)$$

Вычисление A* по формулам (4.5), (4.6) требует $O(n^4)$ операций. Метод Уоршалла быстрее: $O(n^3)$ операций. Методом обхода графа в глубину или в ширину можно получить A* для неориентированного графа за время O(n+m).

4.8. Задачи

4.1. Для графов (орграфов), изображенных на рис. 4.10, показать их представление в виде последовательности ребер (дуг), матрицы смежности, матрицы весов, матрицы инцидентности, векторов смежности, списков смежности, сети и списковой структуры.

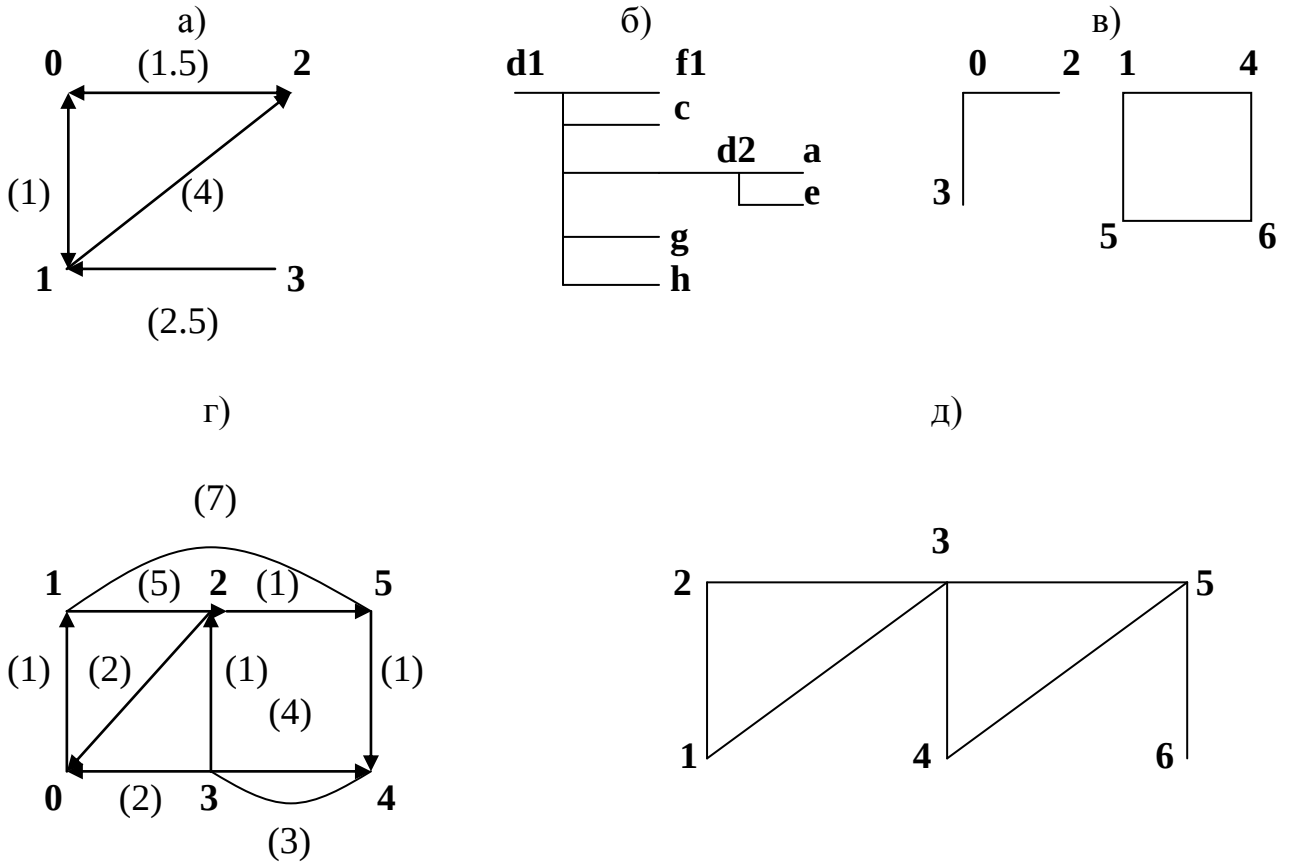


Рис. 4.10. Примеры графов

4.2. На рис. 4.11, показаны различные представления разных графов (орграфов). Изобразить эти графы и определить их основные характеристики: наличие / отсутствие ориентированности; наличие / отсутствие весов и меток; количество вершин и ребер (дуг); наличие / отсутствие связности; количество компонентов связности; наличие / отсутствие петель; наличие / отсутствие изолированных вершин; количество преемников и предшественников каждой вершины; степень каждой вершины и степень графа.

4.3. Дано количество вершин и последовательность дуг графа (орграфа):

а) 4, 3-0, 0-2, 2-3, 2-0, 1-1, 3-2, 0-3;

б) 5, 1-3-6, 4-0-1, 3-1-4, 2-3-5, 0-4-2, 3-2-5
(после каждой дуги указан ее вес);

в) 5, 4-2, 1-3, 0-4, 2-3, 1-4, 1-2, 4-3, 3-1.

Изобразить этот граф, определить, является ли он ориентированным или нет, и показать его представление (если таковое существует) в виде матрицы смежности, матрицы весов, матрицы инцидентности, векторов смежности, списков смежности, сети и списковой структуры.

а) Количество вершин 6, последовательность дуг:
0-1, 1-2, 3-5, 1-0, 3-2, 1-1, 0-2, 2-1, 5-3

б) Матрица смежности

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	0	1	0
2	1	0	0	1	1
3	0	1	1	0	1
4	0	0	1	1	0

в) Матрица инцидентности

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

г) Списки смежности в параллельных массивах

	Вершина	Ссылки
	. . .	
Указатели списков	10	0 26
	11	1 15
	12	2 18
	13	3 25
	14	4 23
Дуги	15	3 24
	16	4 -1
	17	1 21
	18	2 22
	19	0 -1
	20	4 17
	21	2 -1
	22	3 -1
	23	3 17
	24	0 -1
	25	4 19
	26	1 16
	. . .	

д) Матрица весов

	0	1	2	3	4
0	0	2	3	0	7
1	4	0	0	5	0
2	3	0	0	0	5
3	0	4	0	0	6
4	7	0	5	8	0

е) Векторы смежности

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

Рис. 4.11. Примеры представления графов

4.4. Для графов, изображенных на рис. 4.10 в, 4.10 г, 4.10 д, показать последовательность вершин

1) при обходе графа в глубину, начиная с вершины 1;

2) при обходе графа в ширину, начиная с вершины 1.

4.5. Для графов, изображенных на рис. 4.10 а, 4.10 в и 4.10 г, показать (изобразить)

1) матрицу связности;
2) матрицу расстояний;
3) матрицу кратчайших путей (как в алгоритме Флойда).

4.6. Описание списковой структуры. Составить описание данных для представления взвешенного орграфа в виде списковой структуры. Вес дуги - вещественное число.

4.7. Написать фрагмент программы вывода последовательности ребер данного неориентированного графа по его матрице смежности.

4.8. Получение списковой структуры. Написать фрагмент программы получения взвешенного орграфа, изображенного на рис. 10.1 а, и представленного в виде списковой структуры. Вес дуги - вещественное число.

4.9. Написать фрагмент программы (подпрограмму) получения

а) матрицы связности графа по алгоритму Уоршалла. Количество вершин графа не более 32. Биты каждой строки матрицы упакованы в виде переменной типа long.

б) матрицы расстояний и матрицы кратчайших путей графа по алгоритму Флойда , если количество вершин не превышает 20.

4.10. Составить описание данных для представления графа в виде

а) последовательности ребер (дуг),
б) матрицы смежности,
в) матрицы весов,
г) матрицы инцидентности,
д) векторов смежности,
е) списков смежности,
ж) сети,
з) списковой структуры.

Необходимые детали представления уточнить самостоятельно.

4.11. Составить фрагмент программы (подпрограмму) ввода графа и его преобразования из последовательности ребер (дуг) в

а) матрицу смежности; б) сеть; в) списковую структуру.

Необходимые детали представления уточнить самостоятельно.

4.12. Задан граф (орграф) в виде матрицы смежности. Составить программу (фрагмент программы, подпрограмму)

а) проверки, есть ли в графе петли;
б) поиска в графе изолированной вершины (не смежной с другими);
в) определения степени графа;
г) получения последовательности ребер.

4.13. Обработка сети. Задан орграф в виде регулярной сети с четырьмя ссылками, одна из которых указывает на следующую по порядку вершину, а остальные изображают дуги. Составить фрагмент программы подсчета

количества предшественников вершины с номером 1 (количества вершин, от которых ведут дуги к вершине1).

4.14. Задан орграф в виде регулярной сети с четырьмя ссылками, одна из которых указывает на следующую по порядку вершину, а остальные изображают дуги. Составить фрагмент программы (подпрограмму)

а) подсчета количества вершин, не имеющих преемников;

б) определения номера вершины с максимальным количеством предшественников;

в) получения матрицы смежности орграфа.

4.15. Привести пример дерева и показать разные способы его представления. Указать последовательность вершин при обходе дерева в глубину и в ширину.

4.16. Составить фрагмент программы (подпрограмму) определения высоты данного дерева, представленного регулярной сетью.

Необходимые детали представления уточнить самостоятельно.

Указание: требуется обход дерева (раздел 4.3).

4.17. Составить фрагменты программы (подпрограммы) преобразования друг в друга пар: а-в, а-д, б-г, д-е, б-ж, ж-з представлений графа из задачи 4.10.

4.18. Дан граф (орграф). Составить описание данных для его представления и фрагмент программы

а) получения кратчайших путей и расстояний между всеми вершинами по алгоритму Флойда и вывода кратчайшего пути от вершины А до вершины В.

б) получения матрицы связности по алгоритму Уоршалла и вывода номеров вершин каждой компоненты связности графа.

4.19. Составить трассировочную таблицу обхода дерева из рис. 4.9 б по алгоритму 4.10 (раздел 4.3.5).

4.20. Составить нерекурсивные алгоритмы решения задачи обхода бинарного дерева из раздела 4.3.5 для трех способов обхода дерева: прямого, внутреннего и обратного. Представление дерева выбрать самостоятельно.

4.21. Дан граф (орграф) без циклов. Составить описание данных для его представления и фрагмент программы (подпрограмму)

а) проверки, существует ли путь от вершины А к вершине В.

б) поиска какого-либо пути от вершины А к вершине В.

4.22. ***Расстояние до всех вершин***. Дана матрица весов взвешенного орграфа, причем веса всех дуг неотрицательны. Количество вершин не превышает 20. Написать фрагмент программы вычисления расстояний от заданной вершины до всех вершин.

4.23. ***Кратчайшие пути до всех вершин***. Дополнить вычисление расстояний в задаче 4.22 получением кратчайших путей.

Указание. Дерево кратчайших путей можно представить в виде вектора, как в алгоритме обхода в ширину обычного не взвешенного графа. В этом случае получается обратная последовательность вершин каждого пути: от конца к началу.

4.24. Решить задачу 4.22 более быстрым методом за время O(n log n).

Указание: для поиска минимального элемента за время O(log n), вместо O(n), представить массив расстояний d для множества T алгоритма Дейкстры в виде бинарного дерева с помощью адресной арифметики (см. примеры 3.22 и 3.23 из раздела 3.3.3). Это обеспечит для алгоритма Дейкстры время работы O(n log n).

4.25. Оценить необходимый объем памяти для структур данных из задач: 4.1 - 4.3, 4.8, 4.9, 4.12. Недостающие детали представления уточнить самостоятельно.

4.26. **Родственники**. Требуется составить программу, которая вводит и выполняет последовательность команд двух видов: сообщения и запросы. Сообщения о рождении человека поступают в хронологическом порядке и требуют запоминания заданной информации. По запросу программа выдает информацию о характере родственной связи между двумя людьми. *Сообщение* о рождении человека по имени X имеет вид

X сын Y Z

или X дочь Y Z

где Y – имя матери, Z – имя отца.

Запрос “кем является человек по имени X для человека по имени Y” имеет вид

кто X для Y

Два человека являются родственниками, если у них есть общий предок. Характер родственной связи X и Y программа определяет в виде пары чисел m, n, равных, соответственно, расстоянию (количеству поколений) от X и от Y до их ближайшего общего предка. При этом человек считается собственным предком поколения 0.

Для ближайших родственников указываются также следующие названия: *она сама*, *он сам*, *мать*, *отец*, *дочь*, *сын*, *бабушка*, *дедушка*, *внучка*, *внук*, *сестра*, *брат*, *тетя* (сестра матери или отца), *дядя* (брат матери или отца), *племянница* (дочь сестры или брата), *племянник* (сын сестры или брата), *кузина* (дочь тети или дяди) и *кузен* (сын тети или дяди).

Кроме того, прямой потомок или предок поколения более чем 2 именуется добавлением соответствующего количества приставок пра-, например: *правнучка* – дочь внучки или внука, *праправнук* – сын правнучки или правнука, *прабабушка* – мать бабушки или дедушки, *прапрадедушка* – отец прабабушки или прадедушки и т. д. *Входной файл* **с**одержит последовательность команд. В этой последовательности не более 2000 различных имен. Длина имени не превышает 50 символов, при этом средняя длина всех имен не более 10 символов. Имена людей и служебные слова (*сын*, *дочь*, *кто*, *для*), из которых состоит команда, разделяются произвольным количеством разделителей - пробелов и/или символов новой строки. Команды отделяются друг от друга таким же образом. Имя может содержать любые символы, кроме разделителей.

Выходной файл **с**остоит из ответов на запросы. Ответ начинается с новой строки и содержит числа m, n и название родственника (если имеется), разделяемые одним пробелом:

m n название

или слова “нет связи”, если отсутствует информация о родственной связи.

Пример входного файла:	*Выходной файл*
Апполон сын Латона Зевс	0 1 отец
кто Зевс для Апполон	1 0 сын
Афродита дочь Диона Зевс	1 1 сестра
кто Апполон для Зевс	нет связи
кто Афродита для Апполон	
кто Апполон для Диона	

4.27. Пробирки. Имеются три пробирки вместимостью по 100 мл. На двух пробирках нанесены одинаковые риски, третья без рисок. Возле каждой риски надписано целое число миллилитров, которое вмещается в пробирку до этой риски. Изначально одна из пробирок с рисками наполнена 100 мл воды, а остальные две пустые. Составить программу, которая выясняет, можно ли отмерить в пробирку без рисок ровно 1 мл воды, и если да, то находит минимально необходимое для этого число переливаний. Воду можно переливать из одной пробирки в другую до тех пор, пока либо первая из них не станет пустой, либо одна из пробирок не окажется заполненной до какой-нибудь риски.

Входной файл содержит количество рисок n ($n \le 20$) и значения, приписанные рискам V[1], ..., V[n] (последняя риска считается сделанной на верхнем крае пробирки, числа разделены пробелами и/или переводом строки).

Выходной файл должен содержать слово YES и найденное число или NO*:*

YES (или NO)

минимальное число переливаний

4.28. Знакомство [45]. K членов клуба решили познакомиться ($0<K\le100$), организовав два вечера встреч. Известно, кто с кем уже знаком. Составить

программу, определяющую, возможно или нет пригласить на каждую встречу только незнакомых друг с другом членов клуба (YES/NO). Если это возможно, то программа выдает номера членов клуба, приглашенных на каждую из встреч. Числа в файлах разделены пробелами и/или символами новой строки.

Входной файл:

K	Количество членов клуба
1 N[1,1] N[1,2] ...	Номера знакомых 1-го члена клуба
2 N[2,1] N[2,2] ...	Номера знакомых 2-го члена клуба
. . .	
K N[1,1] N[1,2] ...	Номера знакомых K-го члена клуба

Выходной файл (три строки):

YES или NO
Номера участников первого вечера (если ответ YES)
Номера участников второго вечера (если ответ YES)

Пример входного файла:

```
5
1 2 3 5
2 1
3 1
4 5
5 1 4
```

Выходной файл для этого примера:

```
YES
1 4
2 3 5
```