МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский государственный институт электроники и математики
(Технический университет)

Л.Е. ЗАХАРОВА

АЛГОРИТМЫ ДИСКРЕТНОЙ МАТЕМАТИКИ

Утверждено Редакционно-издательским советом института в качестве Учебного пособия

Москва 2002

УДК
ББК
З

Рецензенты:
д-р. техн. наук, проф. И.П. Беляев (НИИИ информационных технологий);
д-р. техн. наук, проф. С.П. Плеханов (кафедра информатики и вычислительной техники МПУ).

Захарова Л.Е..
З- Алгоритмы дискретной математики: Учебное пособие. – Моск. гос. ин-т электроники и математики. М., 2002, 120 с.
ISBN

Предназначено для студентов при изучении дисциплины «Дискретная математика» и смежных с ней дисциплин. Будет полезным при подготовке к семинарам и контрольным работам. Каждая глава содержит алгоритмы дискретной математики, реализованные в виде программ на алгоритмическом языке Паскаль. Программы проверены на контрольных примерах.

ISBN **УДК**
ББК

Математики

Оглавление

Предисловие

Информатика, и в частности, алгоритмизация, последние десятилетия развивается очень бурно. Обычно развивающиеся дисциплины его очень трудно взять в рамки. Рамками, в данном случае, является дисциплина «Дискретная математика». При написании программ никто не задается целью алгоритмизовать дискретную математику, или другие дисциплины. Алгоритмизация происходит попутно при написании программ.

Пособие начинается с комбинаторики, хотя все книги по дискретной математике обычно начинаются с алгебры логики. Для алгоритмов дискретной математики алгоритмы комбинаторики являются основополагающими. Во всех последующих главах после комбинаторики используются ее алгоритмы. Все алгоритмы реализованы на языке Паскаль [6, 7].

Программы в пособии реализуют в основном, в соответствии с названием пособия, алгоритмы дискретной математики. Кроме того, реализовано несколько информационных моделей дискретной математики из не самых сложных. Реализованы в частности информационные модели компенсирования матриц и составления графика дежерств по заявкам.

Информационные модели занимают промежуточное положение между алгоритмами и математическими моделями. Алгоритм – это последовательность вычислительных действий, каждый шаг этой последовательности элементарен.: сложить, сравнить, перейти к, и т.д. Примеры алгоритмов: нахождение минимумом и максимумов в массиве, сортировка массива, вычисление ранга матрицы. Математические модели с помощью математики реализуют деятельность различных объектов: человеческого общества, машин и приборов, живых организмов и т.д. За тысячелетия в математике разработано много примеров формального описания объектов: интегральное и дифференциальное исчисления, геометрия, теория вероятностей и т.д. Молодая наука информатика не только использует для программирования все хитрости, придуманные математиками, но и разрабатывает свои, чисто информационные приемы, которые могут быть реализованы только с помощью ЭВМ: рекурсия, итерация, метод Монте-Карло и т.д. Информационные методы не надо путать с численными методами. Численные методы реализуют математические методы: метод Рунге-Кутта интегрирует систему дифференциальных уравнений, метод Симпсона берет интегралы и т.д. Информационные методы не реализуют никакого конкретного математического понятия, хотя могут быть использованы и в численных методах и в математических моделях. Информационная модель использует информационный метод, но не обязательно только его и не обязательно только один информационный метод.

Информационная модель может быть представлена в виде алгоритмов, и даже в виде информационной модели, если включает в себя другие информационные методы. Одна и та же задача может быть иногда решена и с помощью математической модели, и с помощью информационной модели, и просто в виде алгоритма. Тут важен результат, а не средства его достижения. Обычно применяются более быстродействующие и экономящие память ЭВМ средства. Выбор средств предоставлен в данном случае исполнителю. Разбиение на алгоритмы, математические и информационные модели во многом условно, т.к. в одной программе могут быть и математические и информационные методы. Кроме того, многие алгоритмы стали информационными моделями.

Пособие написано в объеме курса «Дискретная математика». Пособие не включает в себя весь курс лекций по этой дисциплине, и даже не включает в себя ни одной лекции полностью. Пособие содержит только некоторые из тех алгоритмов дисциплины, которые отсутствуют, или которые сложно найти в литературе. Некоторые алгоритмы, имеющиеся в литературе, в пособии представлены более доступно на более простых примерах.

При изучении дисциплины «Дискретная математика» одного этого пособия недостаточно, необходимы еще и книги, перечисленные в библиографическом списке. Многие алгоритмы из книг библиографического списка имеют более современные аналоги. Пособие – перечисляет некоторые, наименее разработанные, части дискретной математики с точки зрения алгоритмизации. Программы, включенные в пособие, невелики и написаны с учетом того, что в них будут разбираться.

1. Комбинаторика

Комбинаторика осуществляет пересчет и перечисление тех элементов в конечных множествах, которые обладают некоторыми заранее заданными свойствами [4]. Разницу между пересчетом и перечислением можно увидеть из следующего примера. Пересчитать все города в стране с населением более одного миллиона – это значит назвать цифру, а перечислить такие города – это огласить список городов. Алгоритмы комбинаторики легко позволяют решить задачи перечисления элементов множества. Некоторые из задач перечисления решаются без ЭВМ с помощью производящей функции (энумератора) [4], но далеко не все. Кроме того, использование производящей функции занимает гораздо больше времени, чем составление алгоритмов, написание и отладка программ. При необходимости и задачи пересчета могут быть решены с помощью алгоритмов комбинаторики, хотя большинство задач пересчета решается без ЭВМ. Надо только не забывать, что алгоритмы дают ответ в числах для каждого конкретного примера пересчета, а теория дает ответ в формулах, объединяющих группу примеров пересчета.

Рассмотрим (n,r)-выборки, т.е. выборки r элементов из n-элементного множества [1]. Упорядоченная (n,r)-выборка называется размещением из n по r, неупорядоченная – сочетанием из n по r. Естественно, что при одних и тех же из n и r размещений больше, чем сочетаний. Кроме того, из (n,r)-выборки могут быть с повторениями элементов в них, и без повторений. Число (n,r)-раэмещений с повторениями обозначается $\overline{A}_n^r$ и равно n^r. Программа razmp осуществляет перечисление всех n^r размещений с повторениями:

```
program razmp;
const n=5; r=3;
var i,j,k,a:integer;
begin
  a:=n;for i:=2 to r do a:=a*i;
  for i:=1 to a do
    begin
      k:=i-1;
      for j:=1 to r do
        begin
          write((k mod n)+1); k:=k div n;
```

```
        end;
      write(' ');
    end;
  writeln;
end
```

Здесь переменная a равна числу размещений с повторениями n^r. В данном случае выборка производится из множества, состоящего из n целых чисел от единицы до n. Если множество состоит из n элементов от b[1] до b[n], то программа будет иметь вид:

```
program razmp2;
const n=5; r=3;
var i,j,a,k:integer;b:array[1..n] of integer;
begin
  a:=n; for i:=2 to r do a:=a*n;
  for i:=1 to n do read(b[i]); readln;
  for i:=1 to a do
    begin
      k:=i-1;
      for j:=1 to r do
        begin
          write(b[(k mod n)+1]:1); k:=k div n;
        end;
      write(' ');
    end;
  writeln;
end.
```

Здесь массив b – множество целых чисел, но он может быть и множеством действительных, булевых, символьных, буквенных и т.д. переменных. Каждый элемент множества b и сам может быть массивом, множеством, страницей текста и т.д. Как видно, в программе razmp2 по сравнению с программой razmp добавился только ввод массива b и изменился вывод размещений на дисплей. Такие изменения нетрудно ввести в любую программу. Поэтому далее для удобства, где это возможно, будут рассматриваться только (n,r)-выборки из множества целых чисел от единицы до n.

Результатом работы программы razmp будет вывод всех чисел в n-ричной системе, имеющих r позиций, причем значения на позициях увеличены на единицу. Для $\overline{A}_5^3$, например, результат будет иметь вид: 111 211 311 411 511 121 … 521 131 … 551 112 … 555, всего 125 чисел, состоящих из трех позиций. Значения на каждой позиции меняются от единицы до пяти. Такой набор легко получить и без ЭВМ, и программа razmp написана не для этого. Программу нетрудно дописать так, чтобы она

выводила не все n^r размещений с повторениями, а те из них, которые обладают определенными свойствами. Надо определить, к примеру, размещения с повторениями, сумма значений на позициях у которых равна (больше, меньше) определенного числа. Для $\overline{A}_5^3$ программа, выводящая размещения с повторениями с суммой значений на позициях, равной восьми, имеет вид:

```
program razmp3;
const n=5; r=3;
var i,j,a,k,s,p:integer;
begin
  a:=n; for i:=2 to r do a:=a*n;
  for i:=1 to a do
    begin
      k:=i-1;s:=0;
      for j:=1 to r do
        begin
          s:=s+(k mod n)+1; k:=k div n;
        end;
      if s=8 then
        begin
          k:=i-1;for j:=1 to r do
            begin
              write(((k mod n)+1):1); k:=k div n;
            end;
          write(' ');
        end;
    end;
  writeln;
end.
```

Здесь переменная s – сумма значений на позициях. Условия на значениях на позиции в (n,r)-выборке могут быть самыми различными. Еще пример на размещения с повторениями при n больше r: надо вывести размещения с повторениями, содержащие на своих позициях все n элементов множества. Для $\overline{A}_3^5 = 3^5$ программа будет иметь вид:

```
program razmp4;
const n=3; r=5;
var i,j,a,k,s:integer;d:array[1..n] of integer;
begin
  a:=n; for i:=2 to r do a:=a*n;
  for i:=1 to a do
    begin
      k:=i-1;for j:=1 to n do d[j]:=0;for j:=1 to r do
```

```
        begin
          d[(k mod n)+1]:=1; k:=k div n;
        end;
      s:=d[1]; for j:=2 to n do s:=s*d[j];
      if s=1 then
        begin
          k:=i-1;for j:=1 to r do
            begin
              write(((k mod n)+1):1); k:=k div n;
            end;
          write(' ');
        end;
    end;
  writeln;
end.
```

Здесь d – вспомогательный массив, если i-ый элемент множества в выборке присутствует, то d[i]=1, иначе d[i]=0. Условия на значения на позициях в (n,r)-выборке должны быть корректными, т.е. не должно оказаться, что таких выборок не существует. Например, нет таких (5,3)-размещений с повторениями, сумма значений на позициях в которых больше пятнадцати.

На примере (5,3)-размещений с повторениями рассмотрим задачу пересчета. Следующая программа определяет, сколько существует таких выборок, что сумма значений на позициях в выборке равна восьми.

```
program razmp3;
const n=5; r=3;
var i,j,a,k,s,p:integer;
begin
  p:=0;a:=n; for i:=2 to r do a:=a*n;
  for i:=1 to a do
    begin
      k:=i-1;s:=0;
      for j:=1 to r do
        begin
          s:=s+(k mod n)+1; k:=k div n;
        end;
      if s=8 then p:=p+1;
    end;
  writeln('p=',p:8);
end.
```

Здесь s – сумма значений на позициях, а p – число размещений, удовлетворяющих заданному условию.

Все условия, которые можно задать с помощью производящей функции, можно задать и программах, но не наоборот. Энумераторы и денумераторы (производящии функции) позволяют только учесть любую корректную комбинацию условий, относящихся к наличию или отсутствию элементов множества в выборках. Таким образом, условия, например, на сумму значений на позициях в выборке с помощью производящей функции учесть нельзя. Отметим, что для размещений энумераторов не существует, а для сочетаний существуют. Производящие функции для пересчета (денумераторы) существуют как для размещений, так и для сочетаний.

Рассмотрим теперь размещения без повторений элементов множества в них. Число (n,r)-размещений без повторений обозначается A_n^r и равно n!/(n-r)!. Здесь r строго не больше n, тогда как в размещениях с повторениями n и r любые целые числа. Программа razm осуществляет перечисление всех (n,r)-размещений без повторений:

```
program razm;
const n=5; r=3;
var i,j,k,l,a,f:integer; m:array[1..n] of integer;
begin
  a:=1;for i:=n-r+1 to n do a:=a*i;
  for i:=1 to a do
    begin
      k:=i-1; for j:=1 to n do m[j]:=j;
      for j:=1 to r do
        begin
          l:=k mod (n-j+1)+1; k:=k div (n-j+1);
          write(m[l]:1); for f:=l to n-j do m[f]:=m[f+1];
        end;
      write(' ');
    end;
  writeln;
end.
```

Здесь a, как и раньше, общее число размещений. По номеру размещения i осуществляется вывод самого размещения. M[1..n] – вспомогательный массив для того, чтобы значения на позициях в размещении не повторялись. Программа razm основана на формуле n!/(n-r)!, также как программа razmp основана на формуле n^r. Здесь в цикле по j от 1 до r число l меняется от 1 до (n-j+1), а программе razmp это изменение не зависило бы от j, l менялось бы от 1 до n. Чтобы избежать повторений значений на позициях в размещении, на каждом следующем шаге по j от 2 до r получение значений на позициях осуществляется из числа элементов множества, которое на единицу меньше числа элементов для получения

значений на предыдущем шаге. На первом шаге в первой позиции значения получаются из всего n-элементного множества, на втором – из (n-1) элементов, … , на r-м шаге – из (n-r+1) элементов множества. После присвоения значению j-ой позиции l-го элемента множества, все элементы множества, начиная с (l+1)-го и кончая (n-j)-м, сдвигаются на один элемент вперед. M[l]=m[l+1], … , m[n-j]=m[n-j+1]. (n-j)-ый элемент будет последним при определении значения следующей (j+1)-й позиции. Таким образом, l-ый элемент множества уничтожается, при определении значений следующей позиции он рассматриваться не будет. Такой сдвиг с затиранием значений элементов множества увеличивает время работы программы.

Далее программу razm можно усложнять под любые корректные условия на элементы множества. Условия равенства суммы значений на позициях в размещениях без повторений могут быть корректными, а условия наличия всех n элементов в размещениях без повторений при r меньше n – нет.

Перейдем к сочетаниям без повторений. Число (n,r)-сочетаний без повторений обозначается C_n^r и равно n!/((n-r)!r!). Наша цель, как и раньше, по номеру сочетания i получить все значения, входящие в сочетание. Перечислим, например, все (5,3)-сочетания без повторений: 123, 124, 125, 134, 135, 145, 234, 235, 245, 345. Таких сочетаний 10 = 5!/(3!2!)= C_5^3. Заметим, что на j-ой позиции (j= 1, 2, 3) значение не больше (n-r+j). Для (5,3)-сочетаний без повторений при j=3 (последняя позиция) значение не больше пяти, при j=2 – не больше четырех, при j=1 – не больше трех. Программа строит следующее сочетание из предыдущего, а первое (тривиальное: 1, 2, … , r) задается.

Программа просматривает все значения на позициях в сочетании, начиная с последней r-ой позиции. Определяется позиция j, значение в которой m[j] меньше (n+j-r). Как видно из примера, такая позиция всегда существует для всех сочетаний без повторений, кроме последнего. Последнее сочетание в данном случае нас не интересует, так как после него нет сочетаний. После нахождения, начиная с конца сочетания, такой позиции j, значение в ней m[j] увеличивается на единицу, а значение в следующей (j+1)-ой позиции будет на единицу больше нового m[j]. Значение в последней r-ой позиции будет на (r-j) больше нового m[j]. Программа soch осуществляет перечисление всех (n,r)-сочетаний без повторений:

```
program soch;
const n=5; r=3;
var i,j,l,c,f:integer; m:array[1..n] of integer;
begin
  c:=r+1; for i:=r+2 to n do c:=c*i;
```

```
  f:=1;for i:=2 to n-r do f:=f*i;c:=c div f;
  for i:=1 to n do
    begin
      m[i]:=i; if i<=r then write(m[i]:1);
    end;
  write('  '); for i:=2 to c do
    begin
      f:=0; j:=r; repeat
        if m[j]<(n-r+j) then
          begin
            m[j]:=m[j]+1; for l:=j+1 to r do
            m[l]:=m[j]+l-j; f:=1;
          end;
        j:=j-1;
      until f=1; for j:=1 to r do write(m[j]:1); write('  ');
    end;
  writeln;
end.
```

Здесь c – общее число сочетаний, m[1..n] – массив значений на позициях, f – флаг, обозначающий нахождение нужной позиции в сочетании. Эта же задача может быть решена с помощью рекурсивной процедура nab.

```
program soch1;
const n=5; r=3;
var i,j,c,f:integer; m:array[1..n] of integer;
procedure nab(p:integer);
begin
  m[p]:=m[p]+1; if m[p]>n-r+p then
    begin
      f:=p-1;nab(f);
    end;
end;
begin
  c:=r+1; for i:=r+2 to n do c:=c*i;
  f:=1;for i:=2 to n-r do f:=f*i;c:=c div f;
  for i:=1 to n do
    begin
      m[i]:=i; if i<=r then write(m[i]:1);
    end;
  write('  '); for i:=2 to c do
    begin
      f:=r; nab(f); for j:=f+1 to r do m[j]:=m[j-1]+1;
      for j:=1 to r do write(m[j]:1); write('  ');
```

```
    end;
  writeln;
end.
```

Здесь f – номер нужной позиции в сочетании. Алгоритм в программах soch и soch1 один и тот же, только реализация разная.

Для перечисления всех сочетаний с повторениями воспользуемся формулой: $\overline{C}_n^r = C_{n+r-1}^r$. Для этого n-элементное множество целых чисел увеличим на (r-1) элемент, которые будут принимать значения от (n+1) до (n+r-1). Попадание элемента, равного (n+1), из увеличенного множества в сочетание с повторениями будет означать дублирование элемента, находящегося на первой позиции в сочетании. Попадание элемента, равного (n+j), из увеличенного множества в сочетание с повторениями будет означать дублирование элемента, находящегося на j-ой позиции в сочетании. Если элемент на последней позиции будет равен (n+r-1), то значит в последней r-ой позиции должен быть тот же элемент, что и на предпоследней.

Значение на первой позиции в сочетании не больше (n-r-1)-r+1=n, т.е. элементы дублирования из увеличенного множества на первой позиции никогда не окажутся. На второй позиции может оказаться первый элемент дублирования, равный (n+1). На третьей позиции в сочетании могут оказаться уже два элемента дублирования, равные (n+1) или (n+2). На последней позиции может оказаться любой их элементов дублирования.Если на первой позиции в сочетании число, а на всех остальных позициях элементы дублирования, то сочетание состоит из r одинаковых значений, равных значению на первой позиции в сочетании. Программа sochp осуществляет перечисление всех (n,r)-сочетаний с повторениями:

```
program sochp;
const n=5; r=3;
var i,j,l,c,f:integer; m:array[1..(n+r-1)] of integer;
mp:array[1..r] of integer;
begin
  c:=r+1; for i:=r+2 to n+r-1 do c:=c*i;
  f:=1;for i:=2 to n-1 do f:=f*i;c:=c div f;
  for i:=1 to n+r-1 do
    begin
      m[i]:=i; if i<=r then write(m[i]:1);
    end;
  write('  '); for i:=2 to c do
    begin
      f:=0; j:=r; repeat
        if m[j]<(n-1+j) then
```

```
          begin
            m[j]:=m[j]+1; for l:=j+1 to r do
            m[l]:=m[j]+l-j; f:=1;
          end;
        j:=j-1;
      until f=1; for j:=1 to r do
        begin
          mp[j]:=m[j]; if m[j]>n then
          mp[j]:=mp[m[j]-n];write(mp[j]:1);
        end;
      write(' ');
    end;
  writeln;
end.
```

Можно предложить алгоритм, основанный на способе получения формулы $\overline{C}_n^r = C_{n+r-1}^r$ [1]. Здесь каждому (n,r)-сочетанию с повторениями ставится в соответствие двоичный вектор длины (n+r-1) из r нулей и (n-1) единиц. Число нулей, находящихся между (i-1)-й и i-й единицей, где $2 \le i \le n-1$, равно числу значений i, входящих в (n,r)-сочетание с повторениями. Число нулей, стоящих перед первой единицей, равно числу значений 1, входящих в (n,r)-сочетание с повторениями. Число нулей, стоящих после (n-1)-й единиqs, равно числу значений n, входящих в (n,r)-сочетание с повторениями. Например, при n=5, r=3 (5,3)-сочетание с повторениями будет {2,2,4}, тогда двоичный вектор длины 7 будет 1001101; для {1,5,5}- 0111100. Это соответствие между (n,r)-сочетаниями с повторениями и двоичными векторами с (n-1) единицами и r нулями взаимно однозначно.

С другой стороны, число двоичных векторов с r нулями и (n-1) единицами равно числу ((n+r-1),r)-сочетаний без повторений, где каждое ((n+r-1),r)-сочетание состоит из номеров позиций с нулями в двоичном векторе длины (n+r-1). Для вышеприведенных примеров для двоичного вектора 1001101 (7,3)-сочетание без повторений будет {2,3,6}, а для 01111000 – {1,6,7}. Значит программа должна переводить {2,3,6} в {2,2,4}, а {1,6,6} – в {1,5,5}. Программа sochp1 переводит ((n+r-1),r)-сочетание без повторений в (n,r)-сочетание с повторениями.

```
program sochp1;
const n=5; r=3;
var i,j,l,c,f:integer; m:array[1..r] of integer;
begin
  c:=r+1; for i:=r+2 to n+r-1 do c:=c*i;
  f:=1;for i:=2 to n-1 do f:=f*i;c:=c div f;
  for i:=1 to n do
```

```
    begin
      m[i]:=i; if i<=r then write((m[i]-i+1):1);
    end;
  write(' '); for i:=2 to c do
    begin
      f:=0; j:=r; repeat
        if m[j]<(n-1+j) then
          begin
            m[j]:=m[j]+1; for l:=j+1 to r do
            m[l]:=m[j]+l-j; f:=1;
          end;
        j:=j-1;
      until f=1; for j:=1 to r do write((m[j]-j+1):1); write(' ');
    end;
  writeln;
end.
```

Здесь по сравнению с программой soch изменен вывод на дисплей и вместо n поставлено n+r-1. В выводе на дисплей вместо m[i] (i= 1..r) в программе soch вывоодится (m[i]-i+1). Значения на первых позициях (i=1) у (n,r)-сочетаний с повторениями и ((n+r-1),r)-сочетаний без повторений совпадают, а для произвольного i разница между значениями на i-х позициях у этих сочетаний равна i-1. Программы sochp и sochp1 тоже различаются только выводом на дисплей, и быстродействие у них одинаковое. Но кому-то понятнее один алгоритм, а кому-то – другой.

Теперь вернемся к размещениям без повторений. Следующий алгоритм будет более сложным, чем вышеизложенный алгоритм razm, но и более быстродействующим. $A_n^n = P_n$ - это число перестановок n-элементного множества. Перестановкой называется размещение с повторениями при n=r. Общее число перестановок P_n=n!. На рис. 1.1. представлен метод получения всех n! перестановок n-элементного множества, последующая перестановка получена из предыдущей обменом местами только двух значений на позициях, не обязательно соседних. На рис. 1.1. Меняющиеся местами значения на позициях подчеркнуты. Все 6 перестановок трехэлементного множества изображены полностью. 24 перестановки четырехэлементного множества разделены на четыре группы. Каждая группа состоит из шести (P_3) перестановок, первая позиция в группе неизменна. В каждой группе из четырех изображены первая и последняя (шестая) перестановки. Значения на позициях в каждой из четырех групп меняются местами так же, как и в трехэлементном множестве, причем значения на первой позиции в это трехэлементное множество не входят. Значения на позициях, меняющиеся местами при смене группы, подчеркнуты, одно из них обязательно находится на первой позиции.

Аналогично, все перестановки из i элементов (i≤n) разделены на i групп. В каждой группе значение на первой позиции не меняется, а оставшиеся (i-1) значений на позициях меняются местами как перестановки из (i-1) элементов. Представлены начало и конец каждой группы. Значения на позициях, меняющиеся местами при смене группы, подчеркнуты, одно из них обязательно находится на первой позиции.

Как видно, для перестановки из нечетного числа элементов множества при смене группы меняются местами только значения на соседних, первой и второй позициях. Для перестановок из четного числа элементов при смене группы первые два раза меняются местами значения на первой и второй позициях, далее на первой и четвертой позициях, на первой и

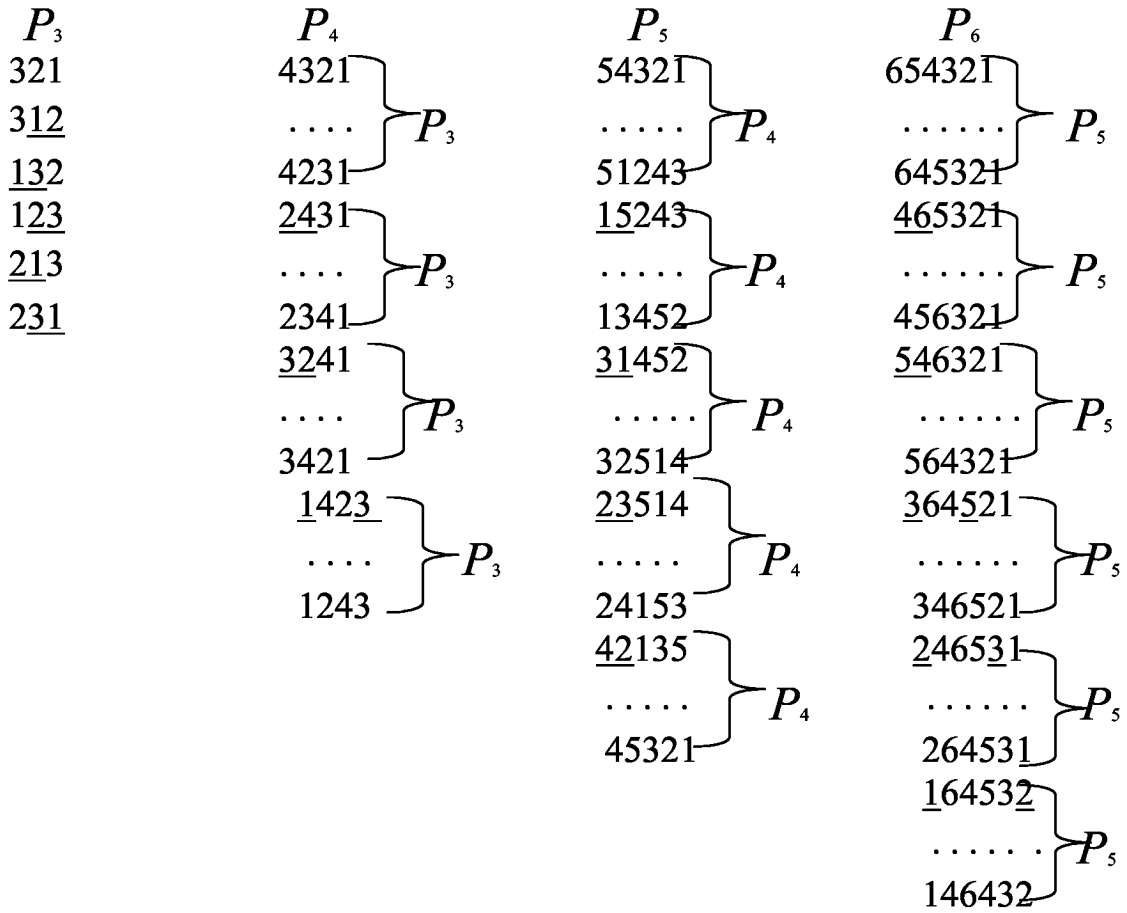


Рис. 1.1. Перестановки множества.

пятой, . . . , на первой и последней позициях при получении последней группы.

Рекурсивная процедура per осуществляет перечисление всех k! перестановок k-элементного множества (первые k элементов из массива m1[1..r]).

```
procedure per(k:integer);
var i,j,g:integer;
begin
  if k=3 then for i:=1 to 6 do
```

```
    begin
      if i mod 2 =0 then
        begin
          j:=m1[1]; m1[1]:=m1[2]; m1[2]:=j;
        end else if i>1 then
            begin
              j:=m1[2]; m1[2]:=m1[3]; m1[3]:=j;
            end;
      for j:=1 to r do write (m1[r-j+1]:1); write(' ');
    end else if k>3 then
      begin
        per(k-1); for i:=1 to k-1 do
          begin
            if k mod 2 = 1 then
              begin
                j:=m1[k]; m1[k]:=m1[k-1]; m1[k-1]:=j;
              end else begin
                g:=i; if g=2 then g:=1;j:=m1[k]; m1[k]:=m1[k-g]; m1[k-g]:=j;
              end;
            per(k-1);
          end;
      end;
end;
begin
for i:=1 to r do m1[i]:=i;
per(r);writeln;
end.
```

С перестановками связано много знаменитых задач [4]. Рассмотрим задачу о встречах: сколькими способами можно переставить k элементов так, чтобы каждый оказался не на своем месте. Другими словами, единица может быть везде, кроме первой позиции, двойка – везде кроме второй, и т.д. Решение существует при k>1. Перестановка, обладающая указанным свойством, называется беспорядком. Для того чтобы получить все беспорядки, надо в подпрограмму per вместо блока вывода на дисплей

```
      for j:=1 to r f do write (m1[r-j+1]:1); write(' ');
```

вставить другой блок вывода на дисплей:

```
      g:=0; for j:=1 to r do if m1[r-j+1]=r-j+1 then g:=1;
      if g=0 then
        begin
          for j:=1 to r f do write (m1[r-j+1]:1); write(' ');
        end;
```

Таким образом мы получили перечисление всех беспорядков. Аналогично можно устроить и их пересчет. Для этого в основной

программе prov, обращающейся к рекурсивной процедуре per, надо описать и обнулить до обращения к процедуре per целую переменную p и вывести ее на дисплей после обращения к per:

```
p:integer:
 …
p:=0; per(r);writeln('p=',p:10);
```

В самой же процедцре per надо блок вывода на дисплей заменить на:

```
g:=0; for j:=1 to r do if m1[r-j+1]=r-j+1 then g:=1;
if g=0 then p:=p+1;
```

Алгоритмы позволяют осуществить пересчет и перечисление и более сложных беспорядков: каждый элемент, например, не должен иметь своих привычных соседей, т.е. первый элемент – второго и последнего элементов, второй – третьего и первого и т.д. Или, наоборот, перестановки, в которых каждый элемент имеет хотя бы одного привычного соседа, или заранее заданные элементы имеют своих привычных соседей, а остальные элементы – нет.

Рассмотрим задачу Люка: сколько существует способов разместить за столом k супружеских пар так, чтобы мужчины чередовались с женщинами, причем мужья не должны сидеть рядом со своими женами. Решение существует при k>3. В этой формулировке задачи число перестановок мужчин не зависит от перестановок женщин. Задача сводится к задаче о перестановке мужей. Для перечисления всех таких перестановок надо блок вывода на дисплей в per заменить на:

```
g:=0; for j:=1 to r-1 do if (m1[r-j+1]=r-j+1) or
(m1[r-j+1]=r-j) then g:=1; if (m1[1]=1) or (m1[1]=r) then g:=1;
if g=0 then
  begin
    for j:=1 to r do write (m1[r-j+1]:1); write('  ');
  end;
```

Здесь все жены пронумерованы и сидят за круглым столом в соответствии со своими номерами: первая – на первом месте, … , последняя – на последнем рядом с первой и предпоследней женами. Значит первый муж не может оказаться на первой или на второй позициях, т.е. слева или справа от своей жены. Второй – на второй и третьей позициях перестановки, … , последний – на последней и первой позициях. Нетрудно осуществить и пересчет этой задачи. Решения для задачи Люка и для задачи о встречах получены теоретически (пересчет). Гораздо интереснее решать с помощью алгоритмов то, что без них не решается.

Если среди k пар существуют и другие связи: брат и сестра, сослуживцы и т.д., то можно усложнить задачу Люка: мужья не должны сидеть рядом с женами, сестрами, сослуживицами, соседками и т.д. В этой формулировке задачи число перестановок мужчин может оказаться зависимым от перестановок женщин. Для каждой перестановки женщин

надо будет определять подходящие перестановки мужчин. Знакомства между мужчинами и женщинами должны быть заданы отдельным массивом. Можно рассмотреть и другие усложненные задачи Люка: мужья не должны оказаться рядом с женой и напротив ее, или муж не должен оказаться рядом с женой и рядом с женщиной, чей муж сидит рядом с его женой, и т.д.

Можно так намудрить, что решение не будет существовать совсем, или будет сужествовать не при всех k. Суть в том, что алгоритмы позволяют учесть любые условия в перестновках, чего не скажешь про теорию.

Для получения всех размещений без повторений воспользуемся формулой $A_n^r = P_r \times C_n^r$. По этой формуле можно получить все размещения без повторений из всех сочетаний без повторений, переставив каждое полученное сочетание P_r раз. Программа razm1 осуществляет перечисление всех размещений без повторений более быстродействующим методом, чем программа razm.

```
program razm1;
const n=5; r=3;
var i,j,l,c,f:integer; m:array[1..n] of integer;m1:array[1..r] of integer;
procedure per(k:integer);
var i,j,g:integer;
begin
  if k=3 then for i:=1 to 6 do
    begin
      if i mod 2 =0 then
        begin
          j:=m1[1]; m1[1]:=m1[2]; m1[2]:=j;
        end else if i>1 then
            begin
              j:=m1[2]; m1[2]:=m1[3]; m1[3]:=j;
            end;
      for j:=1 to r do write (m1[r-j+1]:1); write('  ');
    end else if k>3 then
      begin
        per(k-1); for i:=1 to k-1 do
          begin
            if k mod 2 = 1 then
              begin
                j:=m1[k]; m1[k]:=m1[k-1]; m1[k-1]:=j;
              end else begin
                g:=i; if g=2 then g:=1;j:=m1[k]; m1[k]:=m1[k-g]; m1[k-g]:=j;
              end;
            per(k-1);
```

```
          end;
      end;
end;
begin
  c:=r+1; for i:=r+2 to n do c:=c*i;
  f:=1;for i:=2 to n-r do f:=f*i;c:=c div f;
  for i:=1 to n do
    begin
      m[i]:=i; if i<=r then m1[i]:=m[i];
    end;
  per(r); for i:=2 to c do
    begin
      f:=0; j:=r; repeat
        if m[j]<(n-r+j) then
          begin
            m[j]:=m[j]+1; for l:=j+1 to r do
            m[l]:=m[j]+l-j; f:=1;
          end;
        j:=j-1;
      until f=1; for j:=1 to r do m1[j]:=m[j]; per(r);
    end;
  writeln;
end.
```

Перестановки осуществляются на массиве m1 чтобы не портить массив m, который нужен для получения следующего сочетания. Вывод происходит только в процедуре per, т.е. выводятся только перестановки. В самой рекусивной процедуре per вывод происходит только при k=3, т.е. при перестановке значений на первых трех позициях. Все k! перестановок k-элементного множества состоят из k перестановок (k-1)-элементного множества, или из (k ×(k-1)) перестановок (k-2)-элементного множества, … , или из k!/3! перестановок трехэлементного множества. Так как при k=3 происходит вывод всех r позиций перестановки, а не трех, то значит выводятся полностью все r! перестановок из r позиций. Чтобы вывести P_r перестановок так, как это изображено выше для r=3, 4, 5, 6, вывод должен быть только перед и после per(k-1).

```
program prov;
const  r=6;
var m1:array [1..r] of integer;i:integer;
procedure per(k:integer);
var i,j,g:integer;
begin
  if k=3 then for i:=1 to 6 do
    begin
```

```
      if i mod 2 =0 then
        begin
          j:=m1[1]; m1[1]:=m1[2]; m1[2]:=j;
        end else if i>1 then
            begin
              j:=m1[2]; m1[2]:=m1[3]; m1[3]:=j;
            end;

    end else if k>3 then
      begin
      if k=r then begin for j:=1 to r do write (m1[r-j+1]:1); write(' ');end;
        per(k-1); if k=r then begin for j:=1 to r do write (m1[r-j+1]:1); write('
');end;
        for i:=1 to k-1 do
          begin
            if k mod 2 = 1 then
              begin
                j:=m1[k]; m1[k]:=m1[k-1]; m1[k-1]:=j;
              end else begin
                g:=i; if g=2 then g:=1;j:=m1[k]; m1[k]:=m1[k-g]; m1[k-g]:=j;
              end;
            if k=r then begin for j:=1 to r do write (m1[r-j+1]:1); write(' ');end;
            per(k-1); if k=r then begin for j:=1 to r do write (m1[r-j+1]:1); write('
');end;
          end;
      end;
end;
begin
for i:=1 to r do m1[i]:=i;
per(r);writeln;
end.
```

Отметим разницу между алгоритмами размещений и алгоритмаими сочетаний. В алгоритмах размещений размещение выводится по его номеру в цикле, а в алгоритмах сочетаний – по предыдущему сочетанию при заданном первом сочетании (так назаваемый алфавитный порядок [5]), что не очень удобно.

Теперь рассмотрим разбиения конечного множества с nn элементами на kk+1 подмножеств. При разбиении в первом подмножестве должно быть n1[1] элементов, во втором – n1[2], … , в предпоследнем – $(nn-\sum_{i=1}^{kk} n1[i])$. Набор подмножеств является упорядоченным, а сами подмножества – неупорядоченными. Если мощности (количества элементов) в подмножествах различны, то порядок в наборе подмножеств значения не

имеет. Упорядоченных наборов подмножеств тогда столько же, сколько неупорядоченных. Если в разбиении существуют подмножества с одинаковой мощностью, то упорядоченных наборов подмножеств больше, чем неупорядонных. Количество таких разбиений обозначается $C_{nn}^{n1[1],n1[2],...,n1[kk],nn-\sum n[i]}$ и равно

$$C_{nn}^{n1[1]} \times C_{nn-n1[1]}^{n1[2]} \times ... \times C_{nn-n1[1]-...-n1[kk-1]}^{n1[kk]} = (nn!)/(n1! \times n1[2]! \times ... \times n1[kk]! \times (nn-\sum_{i=1}^{kk} n1[i])!).$$

Здесь можно воспользоваться программой soch. Если число подмножеств разбиения заранее известно, то можно сделать kk вложенных циклов, но это неудобно. Гораздо грамотнее превратить программу soch в рекурсивную процедуру soc(k), все значения C_j^i подсчитать в основной программе:

```
program razb;
const nn=6; kk=2;
var i,j,s,n,f:integer; b:array[1..nn] of integer;
   cc,n1:array [1..kk] of integer;
procedure soc(k:integer);
var i,j,l,c,f,n,r:integer; m,bn,bb:array [1..nn] of integer;
begin
  n:=nn; r:=n1[k]; for i:= 1 to (k-1) do n:=n-n1[i];
  c:=cc[k]; if k<kk then
    begin
      if k>1 then for i:=1 to nn do bn[i]:=b[i]; soc(k+1);
    end else begin
      l:=n1[1]; f:=2;for i:=1 to nn do
        begin
          write(b[i]:1); if i=l then
            begin
              write(' ');if f<=kk then l:=l+n1[f];f:=f+1;
            end;
        end;
      write('  ');
    end;
  for i:=1 to n do m[i]:=i;
  for i:=2 to c do
    begin
      f:=0; j:=r; repeat
      if m[j]<(n-r+j) then
        begin
          m[j]:=m[j]+1; for l:=j+1 to r do
          m[l]:=m[j]+l-j; f:=1;
        end;
```

```
      j:=j-1;
      until f=1; j:=r+1; l:=1; for f:=1 to n do
      if m[l]=f then l:=l+1 else
        begin
          m[j]:=f; j:=j+1;
        end;
      if k<kk then
        begin
          for j:=1+nn-n to nn do bb[j]:=b[m[j-nn+n]+nn-n];
          for j:=1+nn-n to nn do b[j]:=bb[j]; soc(k+1);
        end else begin
          l:=n1[1];f:=2;for j:=1 to nn do
            begin
              if j<1+nn-n then write(b[j]:1) else
              write(b[m[j-nn+n]+nn-n]:1);
              if j=l then
                begin
                  write(' ');if f<=kk then l:=l+n1[f];f:=f+1;
                end;
            end;
          write('  ');
        end;
    end;
  if (k>1) and (k<kk) then for i:=1 to nn do b[i]:=bn[i];
end;
begin
  for i:=1 to nn do b[i]:=i-1;
  s:=0; for i:=1 to kk do
    begin
      read(n1[i]); s:=s+n1[i];
    end;
  readln; n:=nn; if s<nn then
    begin
      for i:=1 to kk do
        begin
          cc[i]:=n1[i]+1; for j:=n1[i]+2 to n do cc[i]:=cc[i]*j;
          f:=1; for j:=2 to n-n1[i] do f:=f*j; cc[i]:=cc[i] div f;
          n:=n-n1[i];
        end;
      i:=1; soc(i); writeln;
    end;
end.
```

Здесь в процедуре soc(k) вывод разбиений на дисплей происходит только при k=kk, подмножества в множестве отделяются одним пробелом, а разбиения друг от друга – двумя. В процедуре введено два вспомогапельных массива: bb и bn. В массиве bn запоминаются начальные значения массива b при входе в процедуру, а при выходе из нее массиву b возвращается из bn его начальное значение для данного k. Это сделано потому, что алгоритм определяет последующее сочетание по предыдущему, и значит предыдущее сочетание в рекурсивной процедуре soc должно быть сохранено. Массив bb введен для коррекции между массивом b и массивом m. Soc получает сочетания в первых элементах массива m, а позиции сочетания в массива b зависят от k – глубины рекурсии. Массив b при k=kk выводится на дисплей, а при k<kk – переставляется с помощью массива bb в соответствии с массивом m.

Алгоритмы позволяют вывести на дисплей только те разбиения, которые удовлетворяют заданным условиям. Можно, например, вывести только те разбиения, в которых первые (kk+1) элементов множества окажутся каждый в своем отдельном подмножестве, т.е. никакие бы два первых (kk+1) элементов не оказались бы в одном подмножестве. Или вывести только те разбиения, в которых заданный i-й элемент (1<i<nn) окажется в одном подмножестве хотя бы с одним из привычных соседей – с (i+1)-м или с (i-1)-м.

Рассмотрим случай, когда в каждом подмножестве, если его мощность больше единицы, должны быть хотя бы один четный и один нечетный элементы (хотя бы один с четным и один с нечетными номерами). Для этого в процедуре soc дополнительно вводится еще три целых переменных p, g1 b g2. Перед выводом на дисплей разбиения эти переменные обнуляются. Если в подмножестве существует нечетный элемент (с нечетным номером), то g1 становится единицей. Если в подмножестве существует четный элемент (с четным номером), то g2 становится единицей. Далее просматриваются все подмножества, мощности больше единицы, и если в каком-то из них нет четного или нечетного элемента, то p становится единицей. Если для разбиения p стало единицей, то оно не выводится на дисплей. Для этого в программе razb первый блок вывода на дисплей:

```
  l:=n1[1]; f:=2;for i:=1 to nn do
       begin
         write(b[i]:1); if i=l then
           begin
             write(' ');if f<=kk then l:=l+n1[f];f:=f+1;
           end;
       end;
     write('  ');
```

заменяется на:

```
l:=n1[1]; f:=2; p:=0; g2:=0; g1:=0;
    for i:=1 to nn do
      begin
        if b[i] mod 2 =1 then g1:=1 else g2:=1;
        if i=l then
          begin
            if (g1+g2<2) and (n1[f-1]>1) then p:=1;
            g1:=0; g2:=0; if f<= kk then l:=l+n1[f];
            f:=f+1;
          end;
      end;
    if p=0 then
      begin
        l:=n1[1]; f:=2;for i:=1 to nn do
          begin
            write(b[i]:1); if i=l then
              begin
                write(' ');if f<=kk then l:=l+n1[f];f:=f+1;
              end;
          end;
        write('  ');
      end;
```

Второй блок вывода на дисплей:

```
    l:=n1[1];f:=2;for j:=1 to nn do
          begin
            if j<1+nn-n then write(b[j]:1) else
            write(b[m[j-nn+n]+nn-n]:1);
            if j=l then
              begin
                write(' ');if f<=kk then l:=l+n1[f];f:=f+1;
              end;
          end;
        write('  ');
```

заменяется на:

```
    l:=n1[1]; f:=2; p:=0; g2:=0; g1:=0;
        for j:=1 to nn do
          begin
            if j<1+nn-n then a:=b[j] else a:=b[m[j-nn+n]+nn-n];
            if a mod 2 =1 then g1:=1 else g2:=1;
            if j=l then
              begin
                if (g1+g2<2) and (n1[f-1]>1) then p:=1;
                g1:=0; g2:=0; if f<= kk then l:=l+n1[f];
```

```
            f:=f+1;
          end;
      end;
    if p=0 then
      begin
        l:=n1[1];f:=2;for j:=1 to nn do
          begin
            if j<1+nn-n then write(b[j]:1) else
            write(b[m[j-nn+n]+nn-n]:1);
            if j=l then
              begin
                write(' ');if f<=kk then l:=l+n1[f];f:=f+1;
              end;
          end;
        write('  ');
      end;
```

Могут быть и случаи, когда в каждом подмножестве, если его мощность больше единицы, должен быть хотя бы один четный (нечетный) элемент.

2. Теория графов

Теория графов содержит много алгоритмов [3]. Некоторые из них представлены далее. Почти у каждого программиста есть своя программа для нахождения минимальных путей между вершинами в графе, и у автора этого пособия тоже. Программы компенсирования матриц и тестирования и восстановления автоматов являются достаточно редкими. Программа планарности иллюстрирует способы применения комбинаторики в алгоритмах теории графов.

2.1 Нахождение минимальных путей между вершинами в графе

Существует много способов построения минимальных путей между вершинами в графах и орграфах. Большинство из них основано на том, что любой минимальный путь сам состоит из минимальных путей. Предложенный далее алгоритм тоже основан на этом и строит минимальный путь между заданными вершинами в орграфе, или дает сообщение, что такого пути нет:

```
program way;
const n=20; pr=0.1; label 1,2;
var i,j,k,l,m,d:integer; s,a,b,an,p:array[1..n,1..n] of integer;
c:array[1..n] of integer;
begin
  randomize; for i:=1 to n do for j:=1 to n do a[i,j]:=0;
  if pr<0.9 then l:=trunc(n*n*pr) else l:=trunc(0.5*n*n); for i:=1 to l do
    begin
```

```
      1: j:=random(n*n+1); k:=(j-1) div n +1; j:=(j-1) mod n+1;
      if a[k,j]=1 then goto 1 else a[k,j]:=1;
    end;
  for i:=1 to n do for j:=1 to n do
    begin
      s[i,j]:=i*a[i,j]; p[i,j]:=a[i,j];b[i,j]:=a[i,j];
    end;
  for d:=2 to (n-1) do
    begin
      for i:=1 to n do for j:=1 to n do
        begin
          m:=0; for k:=1 to n do m:=m+b[i,k]*a[k,j];
          if m>0 then an[i,j]:=1;
        end;
      for i:=1 to n do for j:=1 to n do
      if (s[i,j]=0) and (an[i,j]=1) then
        begin
          p[i,j]:=d; for k:=1 to n do
          if b[i,k]*a[k,j]>0 then s[i,j]:=k;
        end;
      for i:=1 to n do for j:=1 to n do b[i,j]:=an[i,j];
    end;
  write('    ');for i:=1 to n do write (i:3);writeln;for i:=1 to n do
    begin
      write(i:3,'  ');for j:=1 to n do write(a[i,j]:3);writeln;
    end;
  2: readln(k,l); if (k*l>0) and (k<>l) and (p[k,l]>0) then
    begin
      j:=l; for i:=p[k,l] downto 2 do
        begin
          c[i]:=s[k,j]; j:=c[i];
        end;
      c[p[k,l]+1]:=l; c[1]:=k; for i:=1 to p[k,l]+1 do write(c[i]:3);
      writeln;
    end;
  if (k*l>0) and (k<>l) then
    begin
      if p[k,l]=0 then writeln('no way from ',k:3,' to ',l:3);
      goto 2;
    end;
end.
```

Орграф задается случайным образом своей матрицей смежности a[1..n,1..n]. a[i,j]=1 если существует дуга из i-й вершины в j-ю, и a[i,j]=0 в

противном случае. В программе way две константы: число вершин n и доля заполнения единицами матрицы смежности a – pr. Кроме двумерного массива a введено еще четыре двумерных массива: an, b, s, p. В an в зависимости от шага алгоритма d хранятся степени массива a. На втором шаге – вторая, на третьем – третья, … , на (n-1)-м – (n-1)-я. Последней степенью массива a является (n-1)-я, т.к. максимально возможный минимальный путь в орграфе равен (n-1), в этом случае он включает в себя все вершины орграфа. Степень массива a^k вычисляется по формуле: $a^k = a^{k-1} \times a$, где k = 2, 3, … , n-1. Известно, что для степеней матрицы выполняется соотношение: $a^k = a^{k-1} \times a = a \times a^{k-1}$.Докажем его методом индукции по k. Для k=2 это очевидно: $a^2 = a \times a = a \times a$. Для k=3 это тоже легко проверяется. Теперь пусть это верно для k=i-1: $a^{i-1} = a^{i-2} \times a = a \times a^{i-2}$ (индуктивное предположение). Получим из него, что $a^i = a^{i-1} \times a = a \times a^{i-2} \times a = a \times a^{i-1}$, т.е.это верно и для k=i. Соотношение доказано.

В алгоритме, однако, используется формула $a^k = a^{k-1} \times a$, по которой к минимальным путям длиной (k-1) пристраиваются пути длиной единица, а не формула

$a^k = a \times a^{k-1}$, по которой бы к путям длиной единица пристраивались бы минимальные пути длиной (k-1). В итоге по обеим формулам получились бы минимальные пути длиной k, но первая формула в данном случае удобнее. Массив b вспомогательный, в нем хранится предыдущая степень массива a. Предыдущая степень нужна при вычислении текущей степени (переменная d) массива a. Массив p хранит длины минимальных путей, в начале выполнения программы он обнуляется. Если за (n-1) шагов алгоритма элемент p[i,j] остался нулевым, то значит не существует пути из вершины i в вершину j. Таких нулевых элементов в массиве p будет много, если долю заполнения единицами (константа pr) массива a взять небольшой (для n=20 менее 0.1). На первом шаге массив p приравнивается массиву a, получаются минимальные пути длиной единица – дуги орграфа. На втором шаге, если в массиве an появились ненулевые элементы там, где в массиве a нули, то соответствующие элементы массива p становятся двойкой. На шаге d, если в массиве an появились ненулевые элементы там, где во всех предыдущих степенях массива a нули, то соответствующие элементы массива p становятся равными d.

В массиве s запоминаются номера предпоследних вершин минимальных путей, т.к. в алгоритме используется формула $a^k = a^{k-1} \times a$. Если бы в алгоритме использовалась формула $a^k = a \times a^{k-1}$, то в массиве s запоминались бы номера вторых от начала вершин минимальных путей.

Для путей длиной единица (дуг) номера предпоследних вершин - это номера вершин, из которых выходят дуги. Для путей длиной два – это номер средней вершины, и т.д. Таких вершин может быть несколько, если существует несколько путей минимальной длины между вершинами. Из этих путей подходит любой – они равносильны. В программе в массиве s запоминается та вершина, номер которой больше, чем у предпоследних вершин остальных минимальных путей, если они существуют. Если на шаге d в массиве an появился an[i,j]>0, тогда как на всех предыдущих шагах этот an[i,j] был нулем, то значит при умножении i-й строки массива b ((d-1)-я степень a) на j-й столбец массива a существовал k, такой что (b[i,k]×a[k,j])>0. Цикл поиска такого k производится от 1 до n, и s[i,j] присваивается значение k. Из нескольких k остается максимальное. Можно было сделать и так, что при нахождении первого такого k цикл поиска прекращался, и тогда бы в s запоминалась бы вершина с минимальныим номером. K была последней вершиной минимального пути из i в k длиной (d-1), а к этому пути пристроена дуга из k в j. В итоге получился минимальный путь из i в j длиной d.

После (n-1)-о шага алгоритма массивы s и p полностью заполнены. Далее по введенным номерам вершин k и l строится минимальный путь из k в l, если он существует, или дается сообщение: «no way from k to l» в противном случае. Для построения минимального пути из k в l используется одномерный массив c[1..n]. Длина пути известна, она равна p[k,l], и значит номерами вершин будет заполнено и выведено на дисплей (p[k,l]+1) элементов массива c. При этом всегда c[1]=k, а c[p[k,l]+1]= l. Промежуточные вершины берем из массива s. Сначала определяем предпоследнюю вершину j – это s[k,l]=j, затем предпоследнюю для пути из l в j - это s[k,j], и т.д. Если бы вместо формулы $a^k = a^{k-1} \times a$ использовалась бы формула $a^k = a \times a^{k-1}$, то сначала бы определяли вторую вершину j пути из k в l – это s[k,l]=j, затем вторую вершину пути из j в l – это s[j,l], и т.д.

Программа заканчивается, если вводится хотя бы один нуль вместо номера вершины k или l, или вводится две вершины с равными номерами k=l.

Покажем на примере, как работает алгоритм.

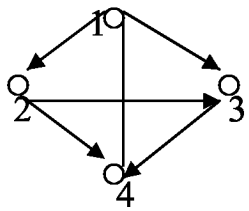


Рис. 2.1. Ориентированный граф.

Матрица смежности a и ее степени для графа, изображенного на рис.2.1. имеют вид:

a =

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

a^2 =

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

a^3 =

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

В степенях матрицы смежности выделены единичные элементы, которые в предыдущих степенях были нулями. Массивы p и s для трех шагов нашего примера имеют вид:

p=

0 1 1 .	0 1 1 2	0 1 1 2
. 0 1 1	2 0 1 1	2 0 1 1
. . 0 1	2 . 0 1	2 3 0 1
1 . . 0	1 2 2 0	1 2 2 0

s=

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

Здесь точками заменены еще не определенные на этом шаге элементы. Массив c построим на примере пути длиной 3 из третьей вершины во вторую:

3 2 1 1 1 1

c=3→2; c=3→1→2 ; c=3→4→1→2. Над стрелками указаны длины путей.

Для графов алгоритм тот же, только матрица смежности a должна быть симметричной. Надо заполнить единицами случайным образом верхний треугольник (половину) матрицы a, который выше главной диагонали, и скопировать его на нижний треугольник, или наоборот. Для графов все остальные двумерные массивы: an, b, p, s тоже будут симметричными, так что можно вычислять только их верхние (нижние) треугольники. Время выполнения программы уменьшится вдвое из-за этого. Но делать этого не обязательно, алгоритм будет работать и с симметричной матрицей a.

2.2. Компенсирование матриц

Скомпенсированной считается матрица, у которой сумма элементов по каждой строке равна сумме элементов по соответствующему столбцу. Симметричная матрица всегда будет скомпенсированной. Скомпенсированными могут быть и несимметричные матрицы. Большинство матриц можно превратить в скомпенсированные. Этот

процесс будет описан далее. Скомпенсированные матрицы используются в задачах тестирования, когда надо пройти все дуги орграфа так, чтобы сумма пройденных дуг была минимальной. Некоторые из дуг при этом могут быть пройдены неоднократно.

Матрицей смежности орграфа с n вершинами в теории графов называется квадратная матрица w размерности n ×n, где w[i,j]=k – числу дуг из i=й вершины в j-ю. Если w[i,j]=0, то значит дуги из i=й вершины в j-ю нет. Если матрицу w скомпенсировать путем добавления дуг только там, где они есть (w[i,j]>0), то тогда для каждой вершины число выходящих из нее дуг будет равно числу входящих в нее дуг. Число выходящих из i-й вершины дуг – это сумма элементов матрицы w по i-й строке, а число входящих в i-ю вершину дуг – это сумма элементов матрицы w по i-у столбцу. В скомпенсированной матрице w эти суммы равны. Если теперь по скомпенсированной матрице w построить скомпенсированный орграф, то можно решить задачу тестирования орграфа. Далее будет показано, как по скомпенсированной матрице строится замкнутый путь, содержащий все его дуги.

Цифровые устройства с памятью представляются в виде орграфа [2], и при тестировании проверяются все переходы между внутренними состояниями (памятью). Устройство тестируется по замкнутому пути, полученному по скомпенсированной матрице, и если устройство работает как эталонное устройство (или как математическая модель), то результат тестирования будет положительным. Задачи тестирования возникают и при проверке состояния авто и железных дорог в заданном районе. Периодически надо проехать все дороги и осуществить некий контроль над ними.

Полностью заполненную положительную матрицу всегда можно скомпенсировать:

```
program comp;
const n=8;k=2000; label 1,2;
var w:array [1..n,1..n] of integer; s1,s2:array[1..n] of integer;
i,j,k1,k2,r1,r2,g,s:integer;pb,p:array[1..k] of integer;
begin
  randomize;for i:=1 to n do
    begin
       s1[i]:=0; s2[i]:=0;
    end;
  for i:=1 to n do for j:=1 to n do w[i,j]:=random(9)+1;
  for i:=1 to n do for j:=1 to n do
    begin
       s1[i]:=s1[i]+w[i,j];s2[j]:=s2[j]+w[i,j];
    end;
  for i:=1 to n-1 do
```

```
    begin
        k1:=0;k2:=0; for j:=1 to n do
          begin
            if s1[j]<s2[j] then
                begin
                  k1:=j; r1:=s2[j]-s1[j];
                end;
            if s2[j]<s1[j] then
                begin
                  k2:=j; r2:=s1[j]-s2[j];
                end;
          end;
        if k1+k1=0 then goto 1;
        if r1>r2 then r1:=r2; w[k1,k2]:=w[k1,k2]+r1;
        s1[k1]:=s1[k1]+r1;s2[k2]:=s2[k2]+r1;
    end;
  1:writeln;for i:=1 to n do
    begin
        for j:=1 to n do write(w[i,j]:3);write(s1[i]:5);writeln;
    end;
  for j:=1 to n do write(s2[j]:3);writeln;
end.
```

Здесь n – число вершин в орграфе, w – его матрица смежности, заполненная случайным образом. Массивы s1 и s2 – суммы элементов матрицы w по строкам и по столбцам соответственно. Из n пар строка-столбец n1 имеет меньшую строку и n2 – меньший столбец, $n1+n2 \le n$. В программе k1 – номер пары строка-столбец с меньшей строкой, а k2 – номер пары с меньшим столбцом. Элемент w[k1,k2] увеличивается, все элементы w положительны и значит все они могут быть увеличены при компенсации. Строка k1 меньше столбца k1 на число r1, а столбец k2 меньше строки k2 на число r2. W[k1,k2] увеличивается на минимум из r1 и r2, при этом как минимум одна из двух пар строка-столбец компенсируется. Полностью заполненная положительная матрица компенсируется максимум за n-1 шагов, так как на последнем шаге компенсируется две пары строка-столбец. Если r1=r2, то на этом шаге тоже скомпенсируется две пары строка-столбец. Сумма элементов скомпенсированной матрицы для полностью заполненной положительной матрицы можно определить до компенсации – это сумма из n слагаемых, где каждое слагаемое равно максимуму из сумм по строке и соответствующему столбцу.

Если у матрицы n1=1 или n2=1, то матрица компенсируется единственным способом, в противном случае она может быть скомпенсирована несколькими способами. Компенсация происходит

максимум за n-1 шагов, при этом увеличивается максимум (n-1) элементов матрицы w на пересечении меньших строки и столбца из пар строка-столбец. Если n1=1, то n2=n-1, то таких элементов для увеличения всего в точности n-1=n1×n2. Если n1>1 и n2>1, то n1 ×n2>n-1, так как n1>1⇔n-n2>1⇔n>n2+1⇔n>(n2²-1)/(n2-1) ⇔ n×(n2-1)>(n2²-1) ⇔n×(n2-1)-n2²+1>0⇔ n2×n – n2²>n-1⇔n2×(n-n2)>n-1⇔ n2×n1>n-1. Значит из n1 ×n2 элементов матрицы w для увеличения n1×n2-n-1 останутся неувеличенными, и это могут быть каждый раз разные элементы. Более того, эти элементы могли быть нулями, и при этом матрица скомпенсируется по изложенному выше алгоритму. Результат компенсации в смысле сумм элементов по строкам и по столбцам будет одинаков, хотя сами скомпенсированные матрицы могут быть различными. Пример матрицы w и двух вариантов ее компенсации:

0	**1**	**1**	3	5 * (+2)		0	1	3	3	7		0	2	2	3	7
1	0	2	2	5		1	0	2	2	5		1	0	2	2	5
4	2	0	1	7		4	2	0	1	7		4	2	0	1	7
2	**1**	**1**	0	4 * (+2)		2	2	2	0	6		2	1	3	0	6
7	4	4	6			7	5	7	6			7	5	7	6	
	*	*														

(+1)(+3)

Здесь меньшие строки и столбцы помечены звездочкой (*). Элементы на пересечении меньших строки и столбца выделены. В первом варианте w[1,2] мог быть нулем, а во втором – w[4,2] и матрица скомпенсировалась бы по алгоритму comp. Элементы на главной диагонали w (так называемые петли) при компенсации не имеют значения, так как они входят как в сумму по строке так и в сумму по соответствующему столбцу.

Алгоритм кончается тогда, когда все пары строка-столбец скомпенсированы. Скомпенстроваенная матрица вместе с суммами по строкам и по столбцам выводится на дисплей. Теперь по скомпенстрованной матрице надо построить путь тестирования, состоящий из номеров вершин, которые в этот путь входят. Покажем на примере матрицы 3×3 (n=3) как этот путь p строится. Размерность одномерного массива минимального пути p равна сумме элементов скомпенсированной матрицы плюс единица. Матрица w до компенсации имеет вид:

0	1	2
2	0	1
2	3	0

, а после компенсации вид:

0	1	3
2	0	2
2	3	0.

Далее находим любой ненулевой элемент матрицы, пусть w[1,2]. Тогда p[1]=1, p[2]=2, а w[1,2] уменьшаем на единицу. Итого p=1,2 а w стало:

0 0 3
2 0 2
2 3 0 и мы находимся во втором состоянии. Далее находим во второй строке ненулевой элемент w[2,1] и возвращаемся в первое состояние. Теперь сразу несколько шагов построения p:

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

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

На каждом шаге один элемент матрицы уменьшается на единицу. Если путь на предыдущем шаге окончился в i-й вершине, то в i-й строке ищется ненулевой элемент. Если этот элемент находится в j-м столбце i-й строки, то к пути приписывается вершина j, а w[i,j] уменьшается на единицу. Если же в i-й строке все обнулено, то из уже пройденных строк находится необнуленная строка, и строится подпуть **pb.** В нашем примере мы перешли в первую обнуленную строку, и находим из пройденных строк необнуленную вторую строку:

pb= 2,**3**	2,**3,2**	2,**3,2,3**	2,**3,2,3,2**
0 0 0	0 0 0	0 0 0	0 0 0
0 0 1	0 0 1	0 0 0	0 0 0
0 2 0	0 1 0	0 1 0	0 0 0

Вставим подпуть **pb** в путь p: 1,2,1,3,1,3,1,3,2,**3,2,3,2**,1. Встроенный путь выделен более жирным шрифтом. Путей тестирования по одной матрице можно постоить несколько, но длина у них одинакова. Продолжение программы comp:

```
 s:=0;for i:=1 to n do s:=s+s1[i]; k1:=1;
for i:=1 to k do
  begin
```

```
        p[i]:=0; pb[i]:=0;
    end;
  r1:=0; r2:=0; for i:=1 to s+1 do
begin
        if i=1 then for j:=1 to n do if s1[j]>0 then
          begin
            k2:=1; while k2<=n do
                begin
                  if w[j,k2]>0 then
                      begin
                        w[j,k2]:=w[j,k2]-1; p[k1]:=j; p[k1+1]:=k2;
                        k1:=k1+2; s1[j]:=s1[j]-1; goto 2;
                      end;
                  k2:=k2+1;
                end;
          end;
        if i>1 then if s1[k2]>0 then for j:=1 to n do
        if w[k2,j]>0 then
          begin
            w[k2,j]:=w[k2,j]-1; p[k1]:=j; k1:=k1+1;
            s1[k2]:=s1[k2]-1; k2:=j;goto 2;
          end;
        if i>1 then if (s1[k2]=0) and (r1>0) then if s1[r2]=0 then
          begin
            for j:=k1-1 downto g+1 do p[j+r1-1]:=p[j];
            for j:=g+1 to g+r1-1 do p[j]:=pb[j-g];
            k1:=k1+r1-1; r2:=0; r1:=0;
          end;
       if i>1 then if (s1[k2]=0) and (r1=0) then
         begin
           r1:=1; g:=k1; repeat g:=g-1; until s1[p[g]]>0;
           for j:=1 to n do if w[p[g],j]>0 then
               begin
                 w[p[g],j]:=w[p[g],j]-1; pb[r1]:=j; r1:=r1+1;
                 s1[p[g]]:=s1[p[g]]-1; r2:=j; goto 2;
               end;
         end;
       if i>1 then if (s1[k2]=0) and (r1>0) then if s1[r2]>0 then
       for j:=1 to n do if w[r2,j]>0 then
          begin
            w[r2,j]:=w[r2,j]-1; pb[r1]:=j; r1:=r1+1;
            s1[r2]:=s1[r2]-1; r2:=j;goto 2;
          end;
```

```
  2: end;
 writeln('s=',s:5);for i:=1 to s+1 do write(p[i]:2);writeln;
```

Здесь константа k – максимальная длина пути тестирования, s – сумма элементов матрицы w, g – вспомогательная переменная, используется при вставке массива pb в массив p. Построение пути тестирования происходит по изложенному в примере алгоритму. Если компенсация в comp происходит только для полностью заполненной положительной матрицы w, то путь тестирования будет строиться и для скомпенсированной матрицы с некоторыми нулевыми элементами.

Как уже отмечено выше, некоторые неотрицательные матрицы, имеющие на пересечении меньших строки и столбца из пар строка-столбец нули, могут быть скомпенсированы по алгоритму, изложенному в программе comp. Для этого из элементов матрицы w, находящихся на пересечении меньших строки и столбца из пар строка-столбец составляется матрица v размерности n1× n2, n1+n2≤ n. Массив t1[1..n1] хранит номера меньших строк из пар строка-столбец, а массив t2[1..n2] – номера меньших столбцов из пар строка-столбец. V[i,j]=0, если w[t1[i],t2[j]]=0, и v[i,j]=1 в противном случае. Массив p1[1..n1] хранит разности между большими столбцами и меньшими строками из пар строка-столбец с меньшей строкой. Массив p2[1..n2] хранит разности между большими строками и меньшими столбцами из пар строка-столбец с меньшим столбцом. Переменная c хранит сумму элементов массива p1, она равна и сумме элементов p2, так как c – это разность между суммами элементов скомпенсированной по алгоритму comp матрицы w и матрицы w до компенсации.

По массивам p1 и t1 построен массив y1[1..c] по следующей схеме: p1[1] раз в y1 повторяется t1[1], далее p1[2] раз повторяется t1[2], … , p1[n1] раз в конце y1 повторяется t1[n1]. Аналогично по массивам p2 и t2 строится массива y2[1..c]. p2[1] раз в y2 повторяется t2[1], далее p2[2] раз повторяется t2[2], … , p2[n2] раз в конце y2 повторяется t2[n2]. Массивы v, p1, p2 используются затем для построения массива m[1..c,1..c]. Элементы массива v превращаются в подмассивы массива m. Массив m делится на n1× n2 подмассивов. Размерность подмассива № i×j, т.е. i-го по счету подмассива сверху и j-го слева матрицы m будет p1[i] × p2[j]. Если v[i,j]=0, то весь подмассив № i×j в массиве m будет нулевым, если v[i,j]=1, то весь этот подмассив будет состоять из единиц. Далее приведен пример массива w с суммами по сторокам и по столбцам, потом полученные из него массивы v, t1, p1, t2, p2 и числа c, n1 и n2. Последним приведен массив m с массивами y1 и y2:

```
0  4  5  1  3  1  0  6    20
3  4  4  0  0  2  5  2    20
5  0  0  0  4  4  4  6    23
```

	1	0	0	4	0	0	0	2	7
W	4	2	2	0	0	6	4	0	18
	3	4	1	0	0	2	3	1	14
	0	1	0	5	4	0	0	0	10
	0	1	0	0	4	5	0	1	11
	16	16	12	10	15	20	16	18	

					t1	p1
	1	0	0	0	4	3
	1	1	1	0	6	6
V	0	1	0	1	7	6
	0	1	0	1	8	7

t2	1	2	3	5
p2	4	4	11	3

c= 22 n1= 4 n2= 4

																							y1
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	6
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	6
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	6
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	6
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	6
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	6
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
m	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	8
y2	1	1	1	1	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	5	5	5	

Задача компенсации матрицы w сведена к задаче нахождения совершенного паросочетания в двудольном графе [3], для которого массив m является четвертью матрицы смежности. Первая доля двудольного графа – строки матрицы m, вторая – её столбцы. Сначала опишем первый алгоритм нахождения совершенного паросочетания. Выберем в двудольном графе вершину с минимальной степенью. Она может оказаться в любой из двух долей. Если таких вершин несколько, то берем или любую, или ту, у которой сумма степеней связанных с ней вершин минимальна. Эту сумму степеней можно считать вторичной степенью вершины. Выбранная вершина минимальной степени может быть связана с несколькими вершинами из другой доли (иметь степень, большую единицы). Из этих вершин, в свою очередь, выбираем вершину с минимальной степенью. Если таких вершин несколько, то или берем любую вершину, или берем вершину с минимальной вторичной степенью.

После того, как одна пара соединение – ресурс выбрана, из двудольного графа удаляем обе этих вершины вместе со всеми их ребрами. Степени оставшихся вершин при этом могут измениться (как и вторичные степени). Алгоритм составлен так, чтобы на каждом шаге (выборе очередной пары) как можно меньше вместе с парой удалять ребер, чтобы на последующих шагах было из чего выбирать. Для этого выбираются вершины с минимальной степенью (минимальной вторичной степенью). Если после удаления пары вместе с их ребрами останется вершина, не имеющая ребер (с нулевой степенью), то процесс нахождения совершенного парасочетания можно закончить, все равно результат будет отрицательным.

Далее, после удаления первой пары, опять ищем вершину с минимальной степенью и находим ей пару из другой доли с минимальной степенью из тех, с кем она имеет ребра. После получения второй пары также удаляем и саму вторую пару и все ее ребра и переходим к нахождению третьей пары. Если все вершины получат свою пару, то получим совершенное парасочетание.

Совершенное паросочетание может и не существовать, и их может быть несколько.

По этому алгоритму была написана и отлажена программа deg, и во всех случаях, когда существовало совершенное паросочетание, программа его находила.

```
program deg;
const n=8;
var m:array[1..n,1..n] of integer;
k,l,c,p,s,i1,j2,i,j,m1,m2:integer;
d,s1,s2:array[1..n] of integer;
begin
  {randomize; for i:=1 to n do for j:=1 to n do m[i,j]:=random(2);}
```

```
for i:=1 to 3 do for j:=1 to 4 do m[i,j]:=1;
for i:=5 to 8 do for j:=6 to 8 do m[i,j]:=1;
m[4,4]:=1;m[4,5]:=1;m[5,5]:=1;
for i:=1 to n do
  begin
    s1[i]:=0;s2[i]:=0; d[i]:=0
  end;
c:=0; for i:=1 to n do for j:=1 to n do
  begin
    s1[i]:=s1[i]+m[i,j]; c:=c+m[i,j]; s2[j]:=s2[j]+m[i,j]
  end;
for l:=1 to n do if c>0 then
  begin
    m1:=n; m2:=n; for i:=1 to n do
      begin
        if (s1[i]<m1) and (s1[i]>0) then
          begin
            m1:=s1[i]; i1:=i
          end;
        if (s2[i]<m2) and (s2[i]>0) then
          begin
            m2:=s2[i]; j2:=i
          end
      end;
    for i:=1 to n do if (s1[i]=m1) and (i<>i1) then
      begin
        s:=0; p:=0; for k:=1 to n do
          begin
            s:=s+m[i,k]*s2[k]; p:=p+m[i1,k]*s2[k]
          end;
        if s<p then i1:=i
      end;
    for i:=1 to n do if (s2[i]=m2) and (i<>j2) then
      begin
        s:=0; p:=0; for k:=1 to n do
          begin
            s:=s+m[k,i]*s1[k]; p:=p+m[k,j2]*s1[k]
          end;
        if s<p then j2:=i
      end;
    if m1<m2 then
      begin
        s:=n; for i:=1 to n do if (s2[i]<s) and (s2[i]>0) and
```

```
        (m[i1,i]>0) then
          begin
            s:=s2[i]; p:=i
          end;
        d[i1]:=p; for i:=1 to n do
          begin
            s2[i]:=s2[i]-m[i1,i]; s1[i]:=s1[i]-m[i,p]
          end;
        s1[i1]:=0; s2[p]:=0; for i:=1 to n do
          begin
            c:=c-m[i1,i]; m[i1,i]:=0;
            c:=c-m[i,p]; m[i,p]:=0
          end
      end  else
      begin
        s:=n; for i:=1 to n do if (s1[i]<s) and (s1[i]>0) and
        (m[i,j2]>0) then
          begin
            s:=s1[i]; p:=i
          end;
        d[p]:=j2; for i:=1 to n do
          begin
            s2[i]:=s2[i]-m[p,i]; s1[i]:=s1[i]-m[i,j2]
          end;
        s1[p]:=0; s2[j2]:=0; for i:=1 to n do
          begin
            c:=c-m[p,i]; m[p,i]:=0;
            c:=c-m[i,j2]; m[i,j2]:=0
          end
      end;
  end;
 writeln('Grafik'); write('  n% day  n% day  n% day  n% day');
 writeln('  n% day  n% day  n% day  n% day');
 for i:=1 to n do if d[i]>0 then
 write(i:5,d[i]:5); writeln
end.
```

При этом двудольный граф (массив m) задавался в программе случайным образом с помощью датчика случайных чисел. Массив s1 – степени вершин одной доли, s2 – другой. C – сумма элементов массива m. Переменная m1 – минимальная степень вершин в первой доле, m2 – во второй. D – массив полученного паросочетания. Существуют, однако, такие двудольные графы, в которых есть совершенное паросочетание, а этот алгоритм его не находит.

Возьмем, например, два полных двудольных графа с нечетным числом вершин, большим пяти. Полными двудольными графами в теории графов называются такие двудольные графы, в которых все вершины соединены со всеми вершинами другой доли. Рассмотрим два полных двудольных графа, каждый из семи вершин и двенадцати ребер. В этих графах в одной доле три вершины, а в другой – четыре. Три вершины одной доли имеют степень четыре, а четыре другой – степень три. Это минимальные графы, иллюстрирующие наш пример. Еще одно общее правило для таких двудольных графов: степени вершин должны быть больше двух.

Соединим эти два полных двудольных графа цепочкой из трех ребер и двух вершин (рис. 2.2.). Цепочка соединит вершины третьей степени полных двудольных графов. Получим двудольный граф, в каждой доле которого по восемь вершин. Совершенное паросочетание в этом графе существует – это два крайних ребра соединительной цепочки и по три ребра из каждого из полных двудольных графов. Вышеизложенный алгоритм начнет с того, что уберет из соединительной цепочки центральное ребро, после чего совершенного парасочетания получить не удастся. После удаления центрального ребра останутся те же два полных двудольных графа с нечетными числами вершин в них. Из этих графов можно получить только шесть пар (по три из каждого графа) вместо нужных семи.

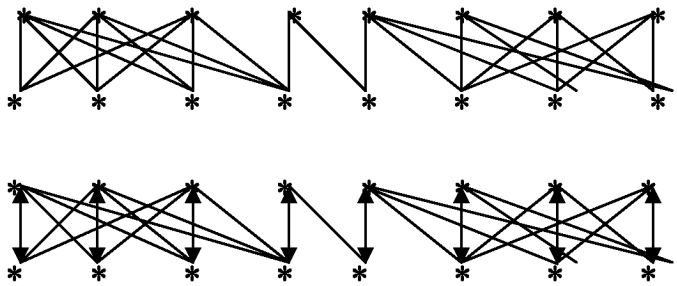


Рис. 2.2. Пример двудольного графа с совершенным паросочетанием.

Можно предложить и более точный алгоритм поиска совершенного паросочетания, лишенный недостатков первого алгоритма. Второй алгоритм найдет совершенное паросочетание в графе, составленном из двух полных двудольных графов, как описано выше. Второй алгоритм находит такие наборы их k вершин в каждой из долей, которые имеют ребра в совокупности точно с k вершинами другой доли. Поиск таких наборов начинается с k=1. Если вершина первой степени (висячая) существует в двудольном графе, то она вместе с соединенной с ней вершиной (не обязательно первой степени) составят пару для совершенного паросочетания и будут удалены из графа вместе со всеми своими ребрами. После удаления всех висячих вершин, если они есть, наличие такого набора при k=2 будет означать, что существует две вершины второй степени, соединенных с двумя вершинами другой доли

(рис. 2.3.). Степени двух вершин другой доли, входящих в рассматриваемую четверку, могут быть и больше двух. Обнаруженная для k=2 четверка вершин составит две пары для совершенного паросочетания и будет удалена из графа вместе со своими ребрами. Заметим, что в этой четверке существует два различных сочетания пар, подходящих для совершенного паросочетания. После удаления четверки вершин поиск наборов для k вершин в уменьшенном графе снова начинается с k=1.

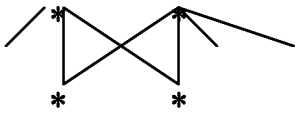


Рис. 2.3. Набор из четырех вершин в двудольном графе.

При k=3 наличие такого набора будет означать, что существует три вершины со степенями два или три, соединенные с тремя вершинами другой доли (рис. 2.4.). Кроме того, наличие набора для k=3 означает, что наборы для k=1 и k=2 не существуют, или уже убраны. Заметим, что для k=3 в шестерке вершин существует более одного сочетания пар, подходящих для совершенного паросочетания. И т.д. В общем случае в интересующий нас набор из k вершин могут входить вершины со степенями от двух до k. Для составления пар для паросочетания из обнаруженного набора из k вершин в каждой доле (всего 2 ×k вершин) можно применить первый алгоритм. Если минимальная степень вершин в графе i, то начинать надо с k=i.

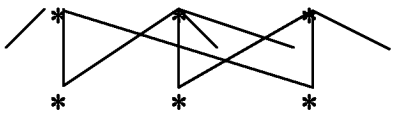


Рис. 2.4. Набор из шести вершин в двудольном графе.

Второй алгоритм занимает больше времени, чем первый. Для k=2 во втором алгоритме рассматриваются все пары из вершин только второй степени в двух долях графа, для k=3 – второй и третьей степени, для произвольного k – степени не более k и не менее двух. Второй алгоритм более точный, чем первый, но приложим не ко всем двудольным графам. Если в графе существует несколько совершенных паросочетаний, то может оказаться так, что для каждого набора из k вершин в каждой доле имеется k+h (h>0) соединенных с ними вершин другой доли (рис. 2.5.). В этом случае можно применить первый алгоритм ко всему двудольному графу. Можно также найти набор с минимальным h, применить первый алгоритм к этому минимальному набору, а затем полученный набор из k пар удалить вместе с ребрами из графа. И т.д.

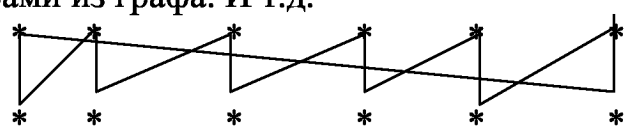


Рис. 2.5. Двудольный граф с двумя совершенными паросочетаниями.

В теории графов существует понятие частичного паросочетания [3]. Это когда из m необходимых пар удается найти m-h пар, т.е. на h меньше. Частичное паросочетание m-h существует, если для каждого набора из k вершин должно найтись в совокупности не менее k-h вершин другой доли ($1 \le k \le m$,). Второй алгоритм позволяет найти и частичное паросочетание.

```
program deg2;
const n=30;pr=0.1;label 1, 2, 3;
type ma=array[1..n] of integer;nm=array[1..2,1..n] of integer;
var m:array[1..n,1..n] of integer;
k,l,c,p,s,i1,w,j2,i,j,m1,m2,y,np:integer;d:ma;nom,ss:nm;
procedure soc(q,nn,r,s:integer;var y:integer);
label 1;var ii,i,j,l,cc,f,k,x:integer; nx,t:array[1..n] of integer;
procedure ob1(i1,i:integer);
var j:integer;
begin
  d[nom[1,i1]]:=nom[2,i]; for j:=1 to n do
  begin ss[1,j]:=ss[1,j]-m[j,i];ss[2,j]:=ss[2,j]-m[i1,j];end;np:=np-1;
  ss[1,i1]:=0; ss[2,i]:=0; for j:=1 to n do
    begin
        c:=c-m[i1,j]; m[i1,j]:=0;
        c:=c-m[j,i]; m[j,i]:=0
    end
end;
begin
  if nn=r then cc:=1 else begin cc:=r+1; for i:=r+2 to nn do cc:=cc*i;
  f:=1;for i:=2 to nn-r do f:=f*i;cc:=cc div f;end;y:=0;
  for i:=1 to nn do t[i]:=i; x:=0; for i:=1 to n do nx[i]:=0;
  for j:=1 to n do
    begin
        k:=0;for i:=1 to r do if q=1 then k:=k+m[s+t[i]-1,j]
        else k:=k+m[j,s+t[i]-1];
        if k>0 then begin x:=x+1;nx[x]:=j;end;
    end;
  if x=r then
    begin
        for i:=1 to r do if q=1 then ob1(s+t[i]-1,nx[i])
        else ob1(nx[i],s+t[i]-1); y:=1;goto 1;
    end;
  if cc>1 then for ii:=2 to cc do
    begin
        f:=0; j:=r; repeat
          if t[j]<(nn-r+j) then
            begin
```

```
                t[j]:=t[j]+1; if j<r then for l:=j+1 to r do
                t[l]:=t[j]+l-j; f:=1;
              end;
            j:=j-1;
          until f=1; x:=0;for j:=1 to n do nx[j]:=0;
          for j:=1 to n do
            begin
              k:=0;for i:=1 to r do if q=1 then k:=k+m[s+t[i]-1,j]
              else k:=k+m[j,s+t[i]-1];
              if k>0 then begin x:=x+1;nx[x]:=j;end;
            end;
          if x=r then
            begin
              for i:=1 to r do if q=1 then ob1(s+t[i]-1,nx[i])
              else ob1(nx[i],s+t[i]-1); y:=1;goto 1;
            end;
      end;
1:end;
procedure sort(j,e,o:integer);
var c,r,a:integer;
procedure obm(e,o:integer);
var i,b:integer;
begin b:=ss[j,e];ss[j,e]:=ss[j,o];ss[j,o]:=b; for i:=1 to n do
  if j=1 then begin
    b:=m[e,i];m[e,i]:=m[o,i];m[o,i]:=b;
  end else  begin
    b:=m[i,e];m[i,e]:=m[i,o];m[i,o]:=b
  end;
b:=nom[j,e];nom[j,e]:=nom[j,o];nom[j,o]:=b;
end;
begin
  if e<o then
    begin
        c:=e;r:=o;a:=ss[j,o];
        repeat
          while ss[j,c]<a do c:=c+1;
          while (ss[j,r]>=a) and (r>c) do r:=r-1;
          if c<r then obm(c,r);
        until c=r;
        obm(c,o);sort(j,e,(c-1));sort(j,(c+1),o);
    end
end;
procedure ob(i1,i:integer);
```

```
var j:integer;
begin
  d[nom[1,i1]]:=nom[2,i]; for j:=1 to n do
  begin ss[1,j]:=ss[1,j]-m[j,i];ss[2,j]:=ss[2,j]-m[i1,j];end;np:=np-1;
  ss[1,i1]:=0; ss[2,i]:=0; for j:=1 to n do
    begin
      c:=c-m[i1,j]; m[i1,j]:=0;
      c:=c-m[j,i]; m[j,i]:=0
    end
end;
begin
  {for i:=1 to 3 do for j:=1 to 4 do m[i,j]:=1;
  for i:=5 to 8 do for j:=6 to 8 do m[i,j]:=1;
  m[4,4]:=1;m[4,5]:=1;m[5,5]:=1;}
  for i:=1 to n do for j:=1 to n do m[i,j]:=0;y:=0;randomize;
  if pr<0.9 then l:=trunc(n*n*pr) else l:=trunc(0.5*n*n); for i:=1 to l do
    begin
      1: j:=random(n*n+1); k:=(j-1) div n +1; j:=(j-1) mod n+1;
      if m[k,j]=1 then goto 1 else m[k,j]:=1;
    end;
  for i:=1 to n do
    begin
      ss[1,i]:=0;ss[2,i]:=0; d[i]:=0;nom[1,i]:=i;nom[2,i]:=i;
    end;
  c:=0; for i:=1 to n do for j:=1 to n do
    begin
      ss[1,i]:=ss[1,i]+m[i,j]; c:=c+m[i,j]; ss[2,j]:=ss[2,j]+m[i,j]
    end;np:=n;
  for l:=1 to n do if c>0 then
    begin
      m1:=n; m2:=n;y:=0; for i:=1 to n do
        begin
          if (ss[1,i]<m1) and (ss[1,i]>0) then
            begin
              m1:=ss[1,i]; w:=i
            end;
          if (ss[2,i]<m2) and (ss[2,i]>0) then
            begin
              m2:=ss[2,i]; j2:=i
            end;
        end;
      k:=m2;l:=m1;i1:=2;if m1<m2 then begin i1:=1;k:=m1;l:=m2;end;
      if k=1 then
```

```
        begin
          if m1=1 then for i:=1 to n do if m[w,i]>0 then
              begin ob(w,i); goto 2; end;
          if m2=1 then for i:=1 to n do if m[i,j2]>0 then
              begin ob(i,j2); goto 2; end;
        end else begin
          sort(2,1,n);sort(1,1,n);if k<l then for i:=k to l-1 do
              begin
                p:=0; for j:=1 to n do if (ss[i1,j]>0)
                and (ss[i1,j]<=i) then
                    begin p:=p+1;if p=1 then s:=j; end;
                if p>=i then soc(i1,p,i,s,y);if y=1 then goto 2;
              end;
          if l<= np then for i:=l to np do
          for i1:=1 to 2 do
                begin
                  p:=0; for j:=1 to n do if (ss[i1,j]>0)
                  and (ss[i1,j]<=i) then
                    begin p:=p+1;if p=1 then s:=j; end;
                  if p>=i then soc(i1,p,i,s,y);if y=1 then goto 2;
                end;
             goto 3;
        end;
   2:end;
  3: writeln;write(' Grafik ');writeln; write('  n% day  n% day  n% day  n%
day');
  writeln('  n% day  n% day  n% day  n% day');
  for i:=1 to n do if d[i]>0 then
  write(i:5,d[i]:5); writeln;
end.
```

Здесь процедура soc(g,nn,r,s,y) перебирает все сочетания из nn по r, определяет по массиву m число связанных с r вершинами одной доли вершин другой доли, и если это число равно r, то 2×r вершин удаляются из массива m вместе со всеми своими ребрами. Для построения совершенного паросочетания в этом удаленном двудольном подмассиве m размерности r×r можно использовать первый алгоритм Процедура sort осуществляет быструю сортировку массива m как по сумме элементов по строкам, так и по сумме элементов по столбцам. В отсортированном по суммам элементов по строкам и по столбцам массиве удобнее находить вершины со степенями от 2-х до k. Процедура ob(i1,i) удаляет из массива m висячую верщину и связанную с ней вершину другой доли.

Массив d[1..c] – массив полученного паросочетания. Если массив m пуст, то и в массиве d будут нули. Если в массиве m есть хоть одна

единица, то и в массиве d появится хоть один ненулевой элемент. Если i-ю вершину первой доли (строка i) алгоритм соединит с j-й вершиной второй доли (столбцом j), то d[i] станет равно j. Если i-й вершине пара найдена не будет, то d[i] останется нулем. Если найдено совершенное паросочетание, то все элементы массива d будут больше нуля.

Массив nom[1..c,1..c] хранит номера строк и столбцов при сортировке. Массив ss[1..2,1..c] – массив сумм по строкам и по столбцам элементов массива m. Чтобы рекурсивные процедуры sort и soc и процедура ob были одни и те же для строк и для столбцов, суммы элементов по строкам и по столбцам пришлось объединить в один двумерный массив ss. Фиктивные параметры g в soc и j в sort принимают значение 1, если надо обработать строку, и 2 – если столбец. Результатом работы программы deg2 является массив d. Паросочетание, полученное с помощью первого алгоритма для массива m из примера имеет вид:

Grafik

n%	d	n%	d	n%	d	n%	d	n%	d	n%	d	n%	d	n%	d
1	1	2	2	3	3	4	4	5	9	6	10	7	11	8	12
9	13	10	5	11	6	12	7	13	8	14	20	15	21	16	22

Теперь осталось из массива d с помощью массивов y1 и y2 получить полностью или частично скомпенсированный маассив w:

```
for i:=1 to c do if d[i]>0 then
w[y1[i],y2[d[i]]]:= w[y1[i],y2[d[i]]]+1;
```

Частично скомпенсированный массив w из примера вместе с суммами элементов по строкам и по столбцам имеет вид:

	0	4	5	1	3	1	0	6	20
	3	4	4	0	0	2	5	2	20
	5	0	0	0	4	4	4	6	23
	4	0	0	4	0	0	0	2	10
W	4	2	2	0	0	6	4	0	18
	4	4	6	0	0	2	3	1	20
	0	5	0	5	6	0	0	0	16
	0	1	0	0	5	5	0	1	12
	20	20	17	10	18	20	16	18	

Если массив w скомпенсировался частично, можно провести компенсацию с построением минимальных путей там, где они существуют, в нулевых элементах массива v. Если v[i,j]=0 и существует минимальный путь длиной l из t1[i] в t2[j], то v[i,j] станет равным l. Если v[i,j]=0 и не существует пути из вершины t1[i] в вершину t2[j], то v[i,j] останется нулем. Как и раньше, из массива v получаем массив m и ищем совершенное паросочетание методом прямого перебора вариантов соединения строк со столбцами. Каждому варианту будет соответствовать

своим набором элементов массив d. Сначала первый вариант будет тривиальным: каждая i-я строка образует пару со своим собственным i-м столбцом, т.е. для каждого i d[i]=i. Для перебора всех вариантов надо получить все перестановки (их c!=P_c) массива d. Рекурсивная процедура per перечисления всех перестановок приведена в алгоритмах комбинаторики. Массив m1 приравняем массиву d, а блок печати в процедуре per :

```
  For j:=1 to r do write(m1[r-j+1]:1); write(' ');
```

Заменим на:

```
  S:=0; for j:=1 to r do
    Begin
        If m[j,m1[j]]>0 then s:=s+m[j,m1[j]] else
            Begin
                S:=0; goto 1;
            End;
    End;
  If (s>0 and (sm>s) then
     Begin
         Sm:=s; for j:=12 to r do dm[j]:=m1[j];
      End;
```

Кроме этого в основной программе, обращающейся к процедуре per, надо вставить:

```
  S,sm:integer; dm:array[1..c] of integer;
.....
  Sm:=0; for i:=1 to c do for j:=1 to c do sm:=sm+m[i,j];
  For i:=1 to c do dm[i]:=0;
```

Здесь критерием является сумма длин всех путей по i от 1 до c из строки i в столбец d[i]. Если хотя бы одного такого пути нет, то m[i,d[i]] будет нулем и весь вариант не рассматривается. Из рассматриваемых выбирается тот вариант, сумма длин путей (переменная s) которого минимальна. Массив dm хранит оптимальный вариант

2.3.Планарность

Программа Planar проверяет планарность графа с n вершинами, заданного матрицей смежности m[1..n,1..n]. Симметричный массив m задается в программе случайным образом, переменная pr – процент заполнения массива m единицами, деленный на два. Массив s[1..n] – степени вершин графа, он же является суммой элементов массива m по строкам и по столбцам, так как в симметричной матрице эти суммы равны. Сначала в графе удаляются ребра у висячих вершин (со степенью единица) и одновременно осуществляется операция, обратная подразбиению ребра, с вершинами второй степени. У вершины со степенью два удаляются оба

ребра, а между двумя смежными с этой вершиной второй степени вершинами строится новое ребро, если его еще не было. Если ребро между вершинами, смежными с вершиной второй степени, уже существовало, то операция, обратная подразбиению ребра, состоит в удалении двух ребер. После удаления висячей вершины могут возникнуть как новые висячие вершины, так и новые вершины второй степени. После операции, обратной подразбиению ребра, могут возникнуть и новые висячие вершины и новые вершины со второй степенью, поэтому эти две операции итерационно осуществляются в графе до тех пор, пока в нем вершин со степенью меньше трех не останется.

Далее массив m сортируется по возрастанию массива s методом быстрой сортировки (процедура sorts). При быстрой сортировке симметричного массива одновременно со строками переставляются и соответствующие им столбцы. Здесь при сортировке номера строк (столбцов) не сохраняются, так как не важно, какие именно вершины служат причиной непланарности графа. Если номера вершин, вызвавших непланарность, нужны, то тогда при сортировке номера вершин надо будет хранить в отдельном массиве. После сортировки первыми в массиве m будут нулевые строки (и столбцы, так как матрица симметрична), затем строки (столбцы) с суммой элементов три, четыре и т.д.

Граф непланарен, если он содержит подграфы, гомеоморфные K_5 и $K_{3,3}$ (теорема Куратовского) [3]. K_5 – это полный граф с пятью вершинами. $K_{3,3}$ – это полный двудольный граф по три вершины в каждой доле. Сначала программа проверяет наличие среди $K_{3,3}$ подграфов графа со степенями вершин не ниже трех. Для этого надо иметь 6 вершин со степенями не ниже трех. Переменная (nn-1) в программе – это число нулевых строк в массиве m, тогда (n-nn+1) – это число вершин со степенями не ниже трех. Если таких вершин не менее шести, то процедура soc33 проверяет наличие $K_{3,3}$ среди подграфов.

Процедура soc33 находит все сочетания из (n-nn+1) по шести. Здесь cc – число таких сочетаний. Оно определяется по другому алгоритму, чем в главе 1. Там решалась задача перечисления всех сочетаний, и поэтому число сочетаний было сравнительно небольшим. Число cc сочетаний из s по r там определялось так:

```
if s=r then cc:=1 else
    begin
      cc:=r+1; if s>r+1 then for i:=r+2 to s do
      cc:=(cc*i); f:=1; if s>r+1 then for i:=2 to s-r do f:=f*i; cc:=cc div f;
    end;
```

Там два цикла, в первом считается s!/r!, а во втором – (s-r)!. Так как число s небольшое, то в обоих циклах переполнения не было. Теперь число s – количество вершин в графе – может быть и большим. Уже при s=40 может

быть переполнение в обоих циклах, хотя сам результат – число сочетаний из s по r -будет в пределах 16-ти разрядного двоичного числа. Чтобы этого избежать переменные f и cc сделаны 32-х разрядными, а число сочетаний из s по r считается в одном цикле:

```
if s=r then cc:=1 else
    begin
      cc:=s; f:=1;if s>r+1 then for i:=s-1 downto r+1do
          begin
             cc:=(cc*i); f:=f*(s+3-i); cc:=cc div f;
          end;
    end;
```

Результаты расчетов cc по обоим алгоритмам совпадут, т.к. если число (s!/r!)состоит из двух сомножителей, то оно делится на 2, …, если из p сомножителей, то оно делится на p!. Из двух целых чисел, стоящих по порядку (отличающихся друг от друга на 1), одно обязательно четное, и значит произведение этих двух чисел делится нацело на два. Из трех целых чисел, стоящих по порядку (т.е. таких как k, k+1, k+2), одно обязательно четное и одно обязательно делится нацело на три, и значит произведение этих трех чисел делится нацело на шесть (3!). Аналогично получаем, что произведение p стоящих по порядку чисел делится нацело на p!.

В программе planar число s=n-nn+1, а число r равно пяти или шести в зависимости от процедуры, в которой определяется число сочетаний из s по r. В soc5 r=5, а в soc33 – r=6. Эти две процедуры объединять в одну нет смысла, так как в них столько же общего, сколько и различного. Общим у них является алгоритм подсчета числа состояний, а различными – алгоритмы нахождения K_5 и $K_{3,3}$ среди подграфов.

Процедура soc нахождения числа сочетаний описана в главе 1. В каждом из найденных наборов из шести вершин проводятся все возможные разбиения на две группы по три вершины. Таких разбиений 10 - C_5^2. Для получения всех таких разбиений рассматривается все время группа, в которой находится первая вершина из найденного набора из шести вершин. Чтобы получить все возможные варианты таких групп из оставшихся пяти вершин определяют все сочетания по две вершины. В каждом из полученных десяти разбиений считаются ребра, имеющие концы в разных группах по три вершины каждая. Если таких ребер девять, то $K_{3,3}$ является подграфом графа без висячих вершин и вершин второй степени, и тогда исходный граф непланарен.

Если $K_{3,3}$ нет среди подграфов, то программа проверяет наличие K_5 среди подграфов графа со степенями не ниже трех. Можно было сначала проверять наличие K_5, а затем, при необходимости, и наличие $K_{3,3}$.

Теперь переменная (nn-1) в программе – это число вершин со степенями не ниже четырех. Для K_5 надо иметь не менее пяти вершин со степенями не ниже четырех, а n-nn+1 – число таких вершин. Если n-nn+1 не менее пяти, то процедура soc5 проверяет наличие K_5 среди подграфов. Она находит все сочетания из (n-nn+1) по пяти. В каждом из полученных сочетаний считаются ребра, имеющие концы в этих пяти вершинах. Если таких ребер 10 – то K_5 является подграфом графа с вершинами, степени которых не ниже трех, и значит исходный граф непланарен.

В конце программы planar на дисплей выдается сообщение о планарности (или непланарности) графа без указания номеров вершин, составивших $K_{3,3}$ или K_5.

```
program planar;
const n=50; pr=0.48; label 1;
var i,j,nn,l,k,k1,k2,y:integer; s:array [1..n] of integer;
m:array[1..n,1..n] of integer;a:real;
procedure soc5(nn:integer;var y:integer);
label 1; var ii,i,j,l:integer; t:array[1..n] of integer;cc,f:longint;
begin
  if n-nn+1=5 then cc:=1 else
    begin
      cc:=6; if n-nn+1>6 then for i:=7 to n-nn+1 do
      cc:=(cc*i) div (i-5);
    end;
  y:=0; for i:=1 to n-nn+1 do t[i]:=i;
  f:=0; for i:=1 to 5 do for j:=1 to 5 do if i<>j then
  f:=f+m[t[i]+nn-1,t[j]+nn-1]; if f=20 then
    begin
      y:=1; goto 1;
    end;
  if cc>1 then for ii:=2 to cc do
    begin
      f:=0; j:=5; repeat
        if t[j]<n-nn-4+j then
          begin
            t[j]:=t[j]+1; if j<5 then for l:=j+1 to 5 do
            t[l]:=t[j]+l-j; f:=1;
          end;
        j:=j-1;
      until f=1;
      f:=0; for i:=1 to 5 do for j:=1 to 5 do if i<>j then
      f:=f+m[t[i]+nn-1,t[j]+nn-1]; if f=20 then
        begin
```

```
            y:=1; goto 1;
          end;
      end;
1:end;
procedure soc33(nn:integer;var y:integer);
label 1; var ii,i,j,l,k,f:integer; t:array[1..n] of integer;cc:longint;
begin
  if n-nn+1=6 then cc:=1 else
    begin
      cc:=n-nn+1; if n-nn+1>7 then for i:=n-nn downto 7 do
      cc:=(cc*i) div (n-nn+2-i);
    end;
  y:=0; for i:=1 to n-nn+1 do t[i]:=i;
  for i:=2 to 5 do for j:=i+1 to 6 do
    begin
      f:=0; for k:=1 to 6 do if (k<>j) and (k<>i) then
      f:=f+m[t[k]+nn-1,t[1]+nn-1]+m[t[k]+nn-1,t[i]+nn-1]
      +m[t[k]+nn-1,t[j]+nn-1]; if f=9 then
        begin
          y:=1; goto 1;
        end;
    end;
  if cc>1 then for ii:=2 to cc do
    begin
      f:=0; j:=6; repeat
        if t[j]<n-nn-5+j then
          begin
            t[j]:=t[j]+1; if j<5 then for l:=j+1 to 5 do
            t[l]:=t[j]+l-j; f:=1;
          end;
        j:=j-1;
      until f=1;
     for i:=2 to 5 do for j:=i+1 to 6 do
       begin
         f:=0; for k:=1 to 6 do if (k<>j) and (k<>i) then
         f:=f+m[t[k]+nn-1,t[1]+nn-1]+m[t[k]+nn-1,t[i]+nn-1]
         +m[t[k]+nn-1,t[j]+nn-1]; if f=9 then
           begin
             y:=1;goto 1;
           end;
       end;
    end;
1:end;
```

```
procedure sorts(e,o:integer);
var c,r,a:integer;
procedure obm(e,o:integer);
var i,b:integer;
begin b:=s[e];s[e]:=s[o];s[o]:=b; for i:=1 to n do
  begin
    b:=m[e,i];m[e,i]:=m[o,i];m[o,i]:=b;
    b:=m[i,e];m[i,e]:=m[i,o];m[i,o]:=b;
  end;
end;
begin
  if e<o then
    begin
        c:=e;r:=o;a:=s[o];
        repeat
          while s[c]<a do c:=c+1;
          while (s[r]>=a) and (r>c) do r:=r-1;
          if c<r then obm(c,r);
        until c=r;
        obm(c,o);sorts(e,(c-1));sorts((c+1),o);
    end
end;
begin
  for i:=1 to n do for j:=1 to n do m[i,j]:=0;a:=n*n;
  randomize; if pr<=0.49 then l:=round(a*pr) else
  l:=round(0.25*a); for i:=1 to l do
    begin
      1: j:=random(n*n+1); k:=(j-1) div n +1; j:=(j-1) mod n +1;
      if (m[k,j]=1) or (m[j,k]=1) then goto 1 else
        begin
          m[j,l]:=1; m[k,j]:=1;
        end;
    end;
  for i:=1 to n do s[i]:=0;
  for i:=1 to n do for j:=1 to n do s[i]:=s[i]+m[i,j];
  repeat
    l:=0; for i:=1 to n do
      begin
        if s[i]=1 then
          begin
            l:=1; s[i]:=0; for j:=1 to n do
              begin
                m[i,j]:=0; m[j,i]:=0;
```

```
            end;
          end;
        if s[i]=2 then
          begin
            k1:=0; k2:=0; for j:=1 to n do
            if m[i,j]=1 then
              begin
                m[i,j]:=0; m[j,i]:=0;
                if k1=0 then k1:=j else k2:=j;
              end;
            if m[k1,k2]=0 then
              begin
                m[k1,k2]:=1; m[k2,k1]:=1;
              end else begin
                s[k1]:=s[k1]-1; s[k2]:=s[k2]-1; l:=1;
              end;
            s[i]:=0;
          end;
      end;
    until l=0; sorts(1,n); nn:=1;for i:=1 to n do if s[i]=0 then
    nn:=nn+1; y:=0; if n-nn+1>=6 then soc33(nn,y);
    if y=0 then
      begin
        nn:=1; for i:=1 to n do if s[i]<4 then nn:=nn+1;
        if n-nn+1>=5 then soc5(nn,y);
      end;
    if y=1 then writeln(' no planar') else writeln(' planar');
end.
```

2.4. Тестирование и восстановление авторматов

Программа vost восстанавливает автомат Мура [2]. Выходной вектор в автомате Мура зависит только от текущего состояния автомата и не зависит от входного вектора. Поскольку выходной вектор является функцией состояния автомата, то для различных состояний автомата Мура значения выходной функции могут совпадать, что надо учитывать при восстановлении автомата. Зато для двух различных значений выходной функции значения состояний автомата Мура также различны, так как у каждого внутреннего состояния автомата Мура может быть только одно значение выходной функции.

Простейшим автоматом Мура будет автомат, когда значение выходной функции совпадает с внутренним состоянием автомата. Для восстановления используется таблица переходов и выходов автомата, которая для автомата Мура вырождается в таблицу переходов.

Сначала рассмотрим автомат Мура, у которого для каждого состояния автомата существует своё значение выходной функции, то есть нет таких двух различных состояний автомата, у которых совпадали бы значения выходных функций. Для восстановления такого автомата используем таблицу переходов размерностью n*m , где m – число входных векторов автомата (обычно это 2 в степени, равной числу входов автомата), а n – число выходных векторов (значений выходной функции автомата и, в данном случае, число внутренних состояний автомата). Число значений выходной функции n обычно не превышает 2 в степени, равной числу выходов автомата. При восстановлении берётся максимальное значение n. Эти обозначения совпадают с принятыми в теории графов обозначениями: n – число вершин графа, m – число ребер графа. При восстановлении – n – число вершин в графе переходов и выходов автомата, а m – число выходящих из каждой вершины этого графа дуг. Общее число дуг (направленных ребер) у графа переходов в автомате Мура – m*n.. Поскольку при восстановлении автомата Мура, представленного «черным ящиком», можно получить только выходной вектор в ответ на входной вектор, а внутренние состояния нам недоступны, то считаем значение выходов совпадающими со значениями внутренних состояний. Значения внутренних состояний сами по себе и не нужны, важно знать значение соответствующих им выходных векторов.

Таблица переходов автомата t[1..m,1..n] задается в vost случайным образом. На практике таблица переходов получается из «черного ящика» - при имеющемся выходном векторе i (i-й столбец таблицы переходов) подаем на входы входной вектор j (j-я строка таблицы) и получаем на выходах новое значение выходного вектора (значение элемента t[i,j] таблицы). При восстановлении «черного ящика» (прибор) массив t будет не нужен, его заменят физические данные с прибора. Массив t нужен для отладки программы. Можно было массив t брать и из файла, в котором смоделирован автомат Мура. Массив v[1..m,1..] хранит восстановленные из массива t данные, в начале программы он обнуляется. В конце программы оба массива выводятся для сравнения на дисплей. Они показывают одинаковый набор элементов.

Восстановление v начинается с первой строки первого столбца. После заполнения v[1,1] переходим в v[1,1]-й столбец и восстанавливаем или первую строку, если v[1,1] не равно 1, или вторую в противном случае. Массив q[1..n] для каждого состояния хранит номера строк, которые надо восстанавливать при попадании в это состояние. При попадании при восстановлении в i-е состояние надо восстанавливать q[i]-ю строку (подать

q[i]-й входной вектор на «черный ящик»). В начале программы все элементы массива q приравниваются единице. Если i-й столбец восстановлен полностью, то q[i] станет равно m+1. При попадании в восстановленное состояние ищется еще не восстановленное j-е состояние, у которого 1<q[j]<m+1. Если все состояния восстановлены, т.е. нет в массиве v полупустых столбцов, а только полностью заполненные или пустые, то программа заканчивается. Если же было найдено не восстановленное состояние, то надо организовать в него переход из текущего восстановленного состояния. Ведь если q[i]>1, то значит в процессе восстановления программа уже была в этом состоянии, и может опять туда попасть. Если из восстановленного i-о состояния не существует пути в не восстановленное j-е состояние, то тогда в j-е состояние существует путь из первого состояния. Первое состояние – это то состояние, в которое автомат Мура попадает при включении питания. Если из i-о состояния не существует пути в j-е состояние, то тогда из i-о состояния не существует пути и в первое состояние, в первое состояние можно попасть только выключив и снова включив питание. Операция по выключению/включению питания в программе обозначена как входной вектор, равный 2*m.

Для построения пути во время восстановления составляется матрица смежности графа (массив a[1..n,1..n]). В начале программы массив a обнуляется, а затем, если из i-о состояния существует дуга в j-е состояние, то a[i,j] приравнивается 1. Таких дуг из одного состояния в другое может быть несколько, но a[i,j] все равно будет равно 1. Какой именно входной вектор перевел автомат Мура из одного состояния в другое, пока не важно, главное, что этот переход существует. Для автоматов Мура матрица смежности a несимметрична.

После заполнения в первый раз одного из столбцов (i-о, например) восстановленной таблицы переходов v понадобится переход в не восстановленное состояние (j-е, например). Если a[i,j]=1, то определятся номер входного вектора, переводящего автомат из i-о состояния в j-е. Если таких входных векторов несколько, то берется любой. Для программы vost номер входного вектора не нужен, он нужен для восстановления по «черному ящику», а не по массиву t. Если a[i,j]=0, то происходит обращение к процедуре way(i,j), которая строит минимальный путь из i-о состояния в j-е. Путь строится так же как в параграфе 2.1. Здесь p[1..n,1..n] – массив длин минимальных путей, а s[1..n,1..n] – массив предпоследних вершин в минимальном пути. При первом обращении к way массивы p и s нулевые. Если p[i,j]=0, то массивы p и s заполняются по тому массиву a, который получился до первого обращения к процедуре way.

Если после заполнения массивов p и s элемент p[i,j] остался равным нулю, то строится путь из первого состояния в j-е. Путь строится в массиве c[1..n]. c[1] приравнивается i, … , c[p[i,j]+1] приравнивается j. Путь в

массиве c состоит из номеров вершин (состояний), а в массиве vm[1..n,1..n] – из номеров входных векторов. Номера входных векторов, кроме первого, берутся из массива t. Номер первого входного вектора - это 2*m (выкл./вкл. питания). Длина пути, состоящего из входных векторов, равна p[i,j]+1. Массив пути из входных векторов нужен для «черного ящика», а не для массива t.

Если после заполнения массивов p и s элемент p[i,j] стал больше нуля, то строится путь из i-о состояния в j-е. Сначала строится путь из номеров вершин в массиве c длины p[i,j]+1, а затем по таблице переходов в массиве vm длины p[i,j]. После построения всех минимальных путей v[i,j] приравнивается t[i,j].

При последующих обращениях к процедуре way матрица смежности a может иметь уже больше единиц, чем при предыдущих обращениях, а массивы p и s останутся построенными по старой матрице a. Если старый элемент p[i,j]>0, то заново для новой a массивы p и s при этом обращении к way не заполняются. Если старый элемент p[i,j]=0, то массивы p и s заполняются для новой матрицы смежности a. Далее порядок построения минимальных путей происходит так же, как и при первом обращении.

Если граф переходов не связный, то некоторые столбцы матрицы v останутся пустыми при остальных полностью заполненных. Полностью заполненными будут только те столбцы, соответствующие которым вершины находятся в одной компоненнте связности с первой вершиной. При случайном заполнении матрицы t несвязного графа переходов ни разу не получилось. Заполненные столбцы матрицы v должны совпасть с соответствующими столцами матрицы t.

```
program vost;
const n=20; m=10; label 1,2;
var v,t:array [1..m,1..n] of integer; i,j,k,l:integer;
vm,c,q:array[1..n] of integer; s,p,a:array[1..n,1..n] of integer;
procedure way(j,l:integer); label 1;
var x,an:array[1..n,1..n] of integer; i,k,r,y,z:integer;
begin
  if p[j,l]=0 then
     begin
       for i:=1 to n do for k:=1 to n do
         begin
           an[i,k]:=a[i,k]; x[i,k]:=0;
         end;
       for r:=2 to (n-1) do
         begin
           for i:=1 to n do for k:=1 to n do
           if (an[i,k]=0) and (i<>k) then
             begin
```

```
            z:=0; x[i,k]:=0; for y:=1 to n do
              begin
                if an[i,y]*an[y,k]>0 then z:=y;
                x[i,k]:=x[i,k]+an[i,y]*a[y,k];
              end;
            if x[i,k]>0 then
              begin
                p[i,k]:=r; x[i,k]:=1; s[i,k]:=z;
              end;
          end;
        for i:=1 to n do for k:=1 to n do
        if (an[i,k]=0) and (i<>k) and (x[i,k]>0) then
          begin
            an[i,k]:=1; x[i,k]:=0;
          end;
      end;
  end;
  if p[j,l]=0 then r:=1 else r:=j;
  k:=1; for i:=p[r,l] downto 2 do
    begin
      c[i]:=s[r,k]; k:=c[i];
    end;
  c[p[r,l]+1]:=l; c[1]:=r;
  for i:=1 to p[r,l] do
    begin
      for k:=1 to m do if v[k,c[i]]=c[i+1] then
        begin
          if p[j,l]=0 then vm[i+1]:=k else vm[i]:=k; goto 1;
        end;
    1: end;
  if p[j,l]=0 then vm[1]:=2*m;
end;
  begin
    randomize; for i:=1 to m do for j:=1 to n do
      begin
        t[i,j]:=random(n)+1; v[i,j]:=0;write(t[i,j]:3);if j=n then writeln;
      end;
    for i:=1 to n do
      begin
        c[i]:=0; q[i]:=1; vm[i]:=0;
      end;
    for i:=1 to n do for j:=1 to n do
      begin
```

```
      a[i,j]:=0;s[i,j]:=0; p[i,j]:=0;
    end;
  j:=1; for i:=1 to n*m do
    begin
      if q[j]<m+1 then
        begin
          2: v[q[j],j]:=t[q[j],j]; q[j]:=q[j]+1;
          k:=j; j:=v[q[j]-1,j]; a[k,j]:=1;
          p[k,j]:=1; s[k,j]:=k;
        end else begin
          l:=0; for k:=1 to n do if (q[k]<m+1) and
           (q[k]>1) then l:=k; if l>0 then way(j,l)
          else goto 1; j:=l; goto 2;
        end;
    end;
  1:writeln(i:10);
  for i:=1 to m do for j:=1 to n do
    begin
      write(v[i,j]:3);if j=n then writeln;
    end;
end.
```

Поскольку мы рассматривали автомат Мура, у которого каждому выходному вектору соответствует одно состояние, то двузначности (многозначности) таблицы переходов в этом случае не было. Это значит, что всегда после любой пары, состоящей из входного и выходного вектора, следовал только один выходной вектор. Если бы какому–то выходному вектору соответствовало бы два внутренних состояния, то значение следующего выходного вектора зависило бы и от внутреннего состояния. Чаще всего внутренние состояния различны, то есть существуют вектора, при воздействии которых автомат переходит в различные состояния. Отметим, что если внутренние состояния совпадают для всех входных векторов и имеют одинаковые выходные вектора, то они неразличимы и их можно считать за одно. Значит, если выходной вектор зависит и от внутреннего состояния автомата, то в таблице переходов возможна двузначность (многозначность), когда после пары, состоящей из выходного и входного векторов, может следовать несколько выходных векторов в зависимости от внутреннего состояния автомата.

Теперь рассмотрим автомат Мура, у которого таблица переходов для одного выходного вектора, например номер К, будет двухзначной, то есть у этого вектора два внутренних различных состояния автомата и существует как минимум один входной вектор (или входные вектора), при котором таблица переходов в К–ом столбце будет двузначной. Возьмём пустую таблицу переходов размерности m*n и начнём её заполнять. Для

перехода в ещё незаполненные столбцы таблицы используется уже заполненная ранее часть таблицы. Вместо того, чтобы получить после выходного вектора номер К под воздействием входного вектора номер i выходной вектор номер j, мы получим выходной вектор номер *l*. Проявила себя двузначность таблицы переходов в К–ом столбце при переходе в не восстановленное состояние. Обнаружив, что у выходного вектора номер К два внутренних состояния, добавляем к таблице ещё один столбец и нумеруем его К'. В качестве числа К' можно взять любое число, большее n. Для различия выходных векторов номер К и К' служит входной вектор i. Если после пары, состоящей из входного и выходного векторов номер i и номер К соответственно следует выходной вектор j, то это действительно выходной вектор К. Если же после пары, состоящей из i и К, следует *l*, то выходной вектор не К, а К'.

После обнаружения двузначности таблицы переходов, заполненные числом К её элементы в таблице переходов v обнуляются. Восстановление очищенной от значений К таблицы продолжается. Если получим выходной вектор номер К, то прежде чем занести его в таблицу, применяем входной вектор номер i, чтобы знать, какой из двух номеров занести – К или К'. После этого мы окажемся в одном из двух состояний – j или l, что надо учитывать при восстановлении. Еще надо учитывать, что столбцы таблицы переходов заполнены теперь ненулевыми значениями не подряд, как раньше. Процесс восстановления проводится так обычно, только для расширенной на один столбец таблицы переходов.

Расширять таблицу переходов можно постепенно, если нумеровать выходные вектора по мере их появления при проведении восстановления, а не в соответствии со значениями выходов. Таблица переходов тоже заполняется этими номерами. Тогда каждому номеру выходного вектора будет соответствовать выходной вектор, а некоторым номерам – набор из трёх векторов – выходного, входного и выходного. Это для тех номеров векторов, которые имеют совпадающие выходные вектора и нужен ещё дополнительный входной и выходной векторы для их различия. Отметим, что в этом наборе из трёх векторов первые два совпадают.

Если таблица переходов постепенно, по мере необходимости расширялась при восстановлении, то пустых столбцов в ней нет. В расширенной таблице переходов номера выходных векторов получались из их значений, а здесь в нерасширенной таблице – выходные вектора нумеровались по мере их появления на выходах автомата при его восстановлении. Если номера выходных векторов считать внутренними состояниями, соответствующими этим векторам, то получим восстановленную таблицу переходов автомата. Если теперь в таблице переходов состояния заменить соответствующими им выходными векторами, то получим восстановленную таблицу выходов автомата.

Можно сначала в нерасширенной таблице переходов заменить номера выходных векторов их значениями, взяв для номера К’ значение выходного вектора, оставшегося при восстановлении без номера. Если такого ненумерованного выходного вектора нет, то взять несуществующее значение выходного вектора. Такое может быть, если внутренних состояний у автомата больше, чем выходных векторов. Такую же замену надо сделать и в номерах столбцов. После этих замен и, при необходимости, перестановки столбцов в соответствии с их новыми номерами, получим восстановленную таблицу переходов. В этой таблице номера внутренних состояний равны номерам выходных векторов. Восстановленная таблица выходов получается из восстановленной таблицы переходов автомата, если заменить в ней несуществующее значение выходного вектора К’ реальным значением К.

Для многозначной таблицы переходов, когда у одного выходного вектора несколько различных (отличающихся) внутренних состояний, процедура восстановления аналогична. У многозначного К–ого столбца появятся несколько столбцов: К’,К’’,К’’’ и так далее. У каждого из этих новых столбцов будет, возможно, своя пара из входного и выходного векторов, по которой при восстановлении они будут отличаться друг от друга. Может быть сразу не удастся обнаружить входной вектор, позволяющий различать все состояния К, К’,К’’,К’’’ и так далее. Система различия состояний может быть более сложной. Например, входной вектор i позволяет отличать состояния К и К’ от других состояний, а после применения входного вектора i входной вектор i’ позволяет различать К и К’, и так далее. Ведь многозначность таблицы переходов обнаружится не сразу. Сначала обнаружится её двузначность при векторе i и состоянии К, потом трёхзначность при векторе i’ и состоянии К’ и так далее. У состояний К’’ и К’ будет теперь набор из четырёх векторов: входного, выходного, входного и выходного (i, *l*, i', j' у состояния К’ и i, *l*, i', *l*` у состояния К’’). У состояния К при этом останется набор из двух векторов: входного i и выходного j. Суть восстановления заключается в том, что у совпадающих по значению выходного вектора состояний имеются наборы из конечного числа входных и выходных векторов для их различия. Процесс восстановления при этом несколько усложняется.

Таблица переходов автомата Мура в общем случае может быть многозначной для нескольких входных векторов. Как частный случай, она может быть многозначной и для всех своих выходных векторов. И в этом случае восстановить автомат Мура возможно, ставя в соответствие каждому новому появившемуся состоянию конечный набор входных и выходных векторов.

Восстановление таблицы переходов в любом случае надо проводить до тех пор, пока в таблице не останется полупустых (неполностью заполненных) столбцов.

Восстановление таблицы выходов по восстановленной многозначной таблице переходов происходит аналогично восстановлению для двузначной таблицы переходов, всё равно, для какого количества столбцов таблица переходов является многозначной. Отметим, что для выявления многозначности таблицы переходов надо после её заполнения заполнить её ещё много раз, сменив порядок подачи входных векторов. В первый раз, например, входные вектора для каждого состояния будут начинаться с единицы и увеличиваться до m, а во второй – уменьшаться с m до единицы. Если при восстановлении многозначность не была обнаружена, или была обнаружена не везде, то восстановленный автомат Мура будет работать неправильно.

Стратегия восстановления необходима для автомата Мура с состояниями, имеющими одинаковые значения выходов (многозначных состояний). Для примера возьмем автомат Мура с четырьмя входными векторами (m=4) и тремя внутренними состояниями (n=3), два из которых (второй и третий) имеют одинаковые значения выходных векторов – двойку (рис. 2.6.а.). Если обратить внимание на таблицу выходов (рис. 2.6.б.), то состояния 2 и 3 различаются для всех входных векторов, кроме второго, т.е. любой из оставшихся трех векторов (1, 3 или 4) может служить для различия внутренних состояний 2 и 3 друг от друга. Если при восстановлении входные вектора подавать в порядке увеличения их номера, то получим одну восстановленную таблицу переходов с двумя внутренними состояниями (рис. 2.6.в.). Порядок восстановления для возрастающего порядка входных векторов отмечен нижним индексом на таблице переходов t. Массив c, состоящий из номеров состояний, которые автомат проходит при восстановлении, для этого примера имеет вид: 1, 1, 2, 1, 2, 2, 2, 1, 1. Массив vm, состоящий из номеров входных векторов, которые автомат проходит при восстановлении, для этого примера имеет вид: 1, 2, 1, 3, 2, 3, 4, 4. Двузначности таблицы переходов обнаружено не было несмотря на то, что внутренние состояния 2 и 3 имеют много различающих входных векторов. Это потому, что повторного обращения к восстановленной таблице переходов v для перехода в не восстановленное состояние не было.

А. Таблица переходов t.					Б. Таблица выходов					В. Таблица переходов v		Г. Таблица переходов v	
Выходы Вн. сост.		1	2	2	Выходы Вн. сост.		1	2	2	Вн. сост.		Вн.сост.	
		1	2	3			1	2	3	1	2	1	2
Вх	1	1_1	1_3	2	Вх	1	1	1	2	1	1	0	2

О Д Ы	2	2_2	3	2_5	О Д Ы	2	2	2	2	2	2	2	2
	3	3_4	2_6	1		3	2	2	1	2	2	2	1
	4	1_7	1_8	3		4	1	1	2	1	1	1	2

Рис. 2.6. Восстановление автомата Мура.

Если при восстановлении входные вектора подавать в порядке уменьшения их номера, начиная с четвертого в каждом внутреннем состоянии, то двузначность таблицы переходов t обнаружится при построении пути из второго восстановленного состояния в первое не восстановленное. При подаче входного вектора 3 мы получим не 1 на входах, как следует из восстановленной таблицы v, а 2. Не полностью восстановленная таблица переходов v для второго случая восстановления приведена на рис. 2.6.г., а массивы c и vm имеют вид соответственно: 1, 1, 2, 2, 1, 2, 2, 2, 2 (вместо 1) и 4, 3, 4, 3, 2, 2, 1, 3. Для убывающего порядка подачи входных векторов двузначность обнаружена при построении пути в не восстановленное состояние. Отметим, что уже при начале восстановления с убывающим порядком входных векторов, имея восстановленную таблицу переходов для возрастающего порядка входных векторов, двузначность сразу обнаружится при сравнении этих таблиц. Входной вектор 4 переводит автомат из 1-о состояния в 1-е при возрастающем порядке, и из 1-о во 2-е при убывающем.

Очевидно, что чем меньше различающих входных векторов у внутренних состояний, тем труднее их различить. Поэтому при защите каких-то устройств или программных продуктов от подделки или восстановления делают проверку на автомат Мура с малоразличающимися состояниями, т.к. их труднее восстановить.

Предположим, что таблица v была восстановлена при возрастающем порядке входных векторов, а затем была проверена при убывающем порядке входных векторов. При входном векторе 4 во втором внутреннем состоянии была обнаружена двузначность таблицы переходов. Значит существует два внутренних состояния 2 и 2’ с одинаковым выходным вектором 2. Для различия 2 и 2’ используем входной вектор 4. Получив на выходах значение 2, подаем входной вектор 4 и, если на выходах 1, то это было 2-е состояние, а если на выходах 2 – это было 2’. Для продолжения восстановления важно знать в случае получения 2 на выходах: какое это внутреннее состояние – 2 или 2’. Один раз проверив, получим, что это 2’, т.е. входной вектор 4 переводит из состояния 2 в состояние 1, а из состояния 2 – в состояние 2’. Такую проверку делаем один раз до начала восстановления. Таблица переходов мала (n=3, m=4) и поэтому в наборе

входной вектор и выходной вектор оказалось двузначное состояние. При большой таблице переходов надо брать различающий вектор, переводящий автомат в однозначные состояния. Этапы восстановления двузначной таблицы переходов изображены на рис. 2.7. Различающий входной вектор 4 уже внесен до восстановления в обнуленную таблицу v. Заметим, что при возвращении в
не восстановленное состояние проверки на состояние 2 или 2’ не производится, т.к. внутреннее состояние уже известно. Тут нижний индекс у состояний показывает последовательность восстановления таблицы переходов. Над стрелками показаны входные вектора, стрелки соединяют внутренние состояния, 2? означает двузначное второе состояние. Видно, что

Таблица переходов v (однозначная)				Обнуленная таблица переходов v (двузначная)					Таблица переходов v (двузначная)				
Вн. сост.		1	2	Вн. сост.		1	2	2’	Вн. сост.		1	2	2’
Входы	1	1	1	Входы	1	1	0	0	Входы	1	1	1_4	2_3
	2	2	2		2	0	0	0		2	2_1	$2'_5$	2_6
	3	2	2		3	0	0	0		3	$2'_2$	2_7	1_8
	4	1	1		4	1	2	2'		4	1	1	2'

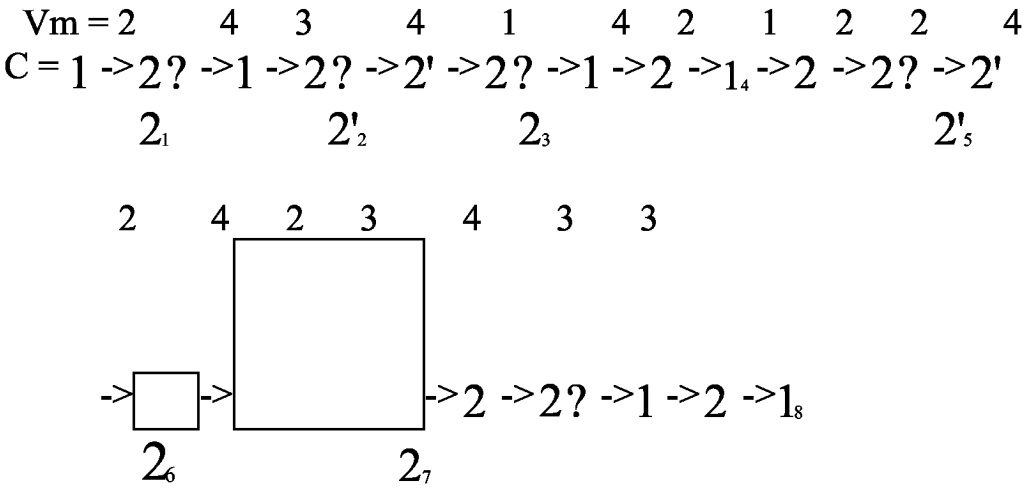


Рис. 2.7. Этапы восстановления двузначной таблицы переходов.

восстановленная таблица v отличается от таблицы t только в обозначениях 2’ и 3. Процесс восстановления двузначной и многозначной таблицы переходов тоже автоматизируется, только программа будет сложнее, чем программа vost. Существуют и другие стратегии восстановления. Можно

задавать входные вектора случайным образом, при этом можно задавать и восстановленный входной вектор для данного состояния, многозначность тогда может определиться и не при переходе в не восстановленное состояние. Можно задавать входной вектор случайным образом из оставшихся не восстановленных векторов для каждого состояния, и тогда многозначность определиться только при переходе в не восстановленное состояние. Можно восстанавливать таблицу любым способом, а затем случайным образом задавать входные вектора и проверять правильность таблицы.

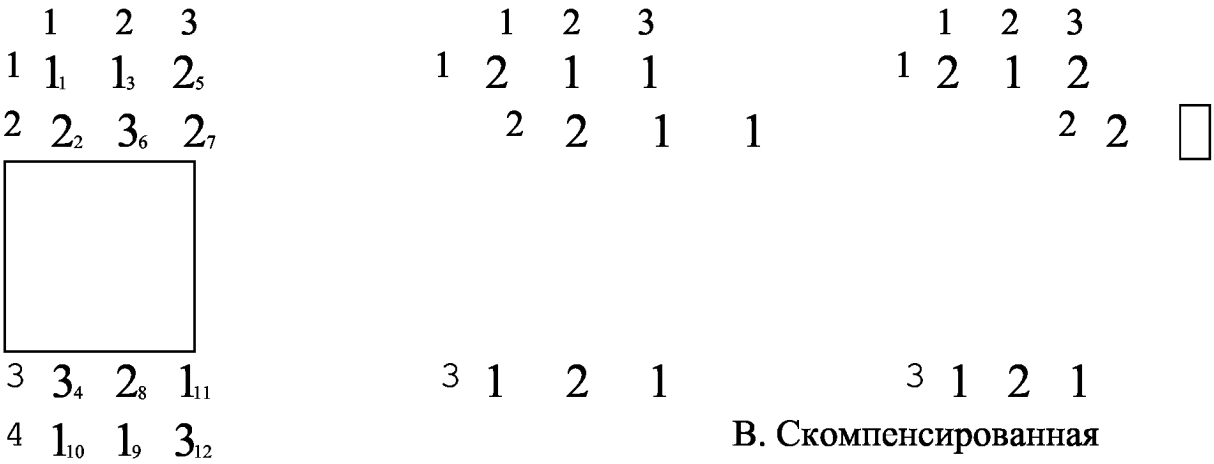
Тестирование автомата Мура проводить проще, чем его восстановление. Тестировать можно по тому же принципу, что и восстанавливать, т.е.: тестировать каждое состояние по порядку входных векторов, а при попадании в протестированное состояние осуществлять переход в недотестированное состояние. Для этого же автомата Мура на рис. 2.6 последовательность состояний и входных векторов при тестировании изображена на рис. 2.8. Нижним индексом помечены состояния, в которых проводится сравнение тестируемого устройства с математической моделью или с эталонным устройством. Сравниваются выходные вектора и, при возможности, внутренние состояния. Два состояния не имеют нижних индексов, так как был переход в недотестированное 3-е состояние. Значение нижних индексов показывает порядок тестирования. Таких переходов во время тестирования было два. Длина пути тестирования из вершин – 15, а из входных векторов – 14. Это не оптимальный процесс тестирования (рис. 2.9.а.).

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

$1 \to 1_1 \to 2_2 \to 1_3 \to 3_4 \to 2_5 \to 3_6 \to 2_7 \to 2_8 \to 1_9 \to 1_{10} \to 3 \to 1_{11} \to 3 \to 3_{12}$

Рис.2.8. Неоптимальное тестирование автомата Мура .

Составим матрицу смежности а графа переходов по таблице переходов (рис. 2.9.б.). Но в этой матрице смежности отражено число входных векторов, переводящих автомат из состояния в состояние. В отличие от процесса восстановления теперь это важно. Теперь при тестировании матрица а будет состоять, возможно, не из одних единиц (рис. 3.4.). Скомпенсируем её как в главе 2 и построим путь с из вершин: 1, 1, 1, 2, 2, 1, 3, 3, 1, 3, 2, 3, 2, 1. Скомпенсированная матрица смежности приведена на рис. 2.9. в. Длина пути из вершин при оптимальном тестировании равна 14, на единицу меньше не оптимального пути, т.к. при компенсировании добавили 1, а не 2, как при неоптимальном тестировании. Путь оптимального тестирования из входных векторов составляется из пути с и таблицы переходов, он
имеет длину 13: 1, 4, 2, 3, 1, 3, 4, 3, 3, 1, 2, 2, 4. Поскольку при построении



а. Матрица переходов. Б. Матрица смежности а. матрица смежности. В. Скомпенсированная

Рис. 2.9. Процесс тестирования автомата Мура.

пути из входных векторов должны быть использованы все переходы для всех входных векторов, нельзя брать любой входной вектор, переводящий из состояния в состояние как при восстановлении, или как при построении минимального пути. Если таких переходов несколько, то все они должны быть отражены в пути, состоящем из входных векторов (массив vm).

```
program test;
const n=3; m=4; label 1;
var i,j,k,p:integer; t: array[1..m,1..n] of integer;
c,vm:ARRAY[1..10*M*N] of integer; A:ARRAY[1..n,1..n] of integer;
procedure komp;
. . . . . .
end;
begin
  . . . . . .
  for i:=1 to n do for j:= 1 to n do a[i,j]:=0;
  for i:=1 to m do for j:= 1 to n do a[j,t[i,j]]:=a[j,t[i,j]]+1;
  komp; for i:=1 to k-1 do vm[i]:=0;
  for i:=1 to m do for j:= 1 to n do
     begin
        for p:= 1 to k-1 do
        if (vm[p]=0) and (c[p]=j) and (c[p+1]=t[i,j]) then
          begin
             vm[p]:=i; goto 1;
          end;
    1: end;
  for p:=1 to k-1 do if vm[p]=0 then
  for i:=1 to m do if t[i,c[p]]=c[p+1] then vm[p]:=i;
end.
```

Здесь массив t задается случайным образом, как и раньше. Процедура komp компенсирует матрицу смежности a, строит путь тестирования из вершин c и находит длину пути из вершин k. Затем путь из входных векторов vm обнуляется и просматривается последовательно весь массив t. Для каждого элемента массива t находится нулевой элемент массива vm, который соответствует этому элементу массива t. Затем элемент vm запоминает значение входного вектора из массива t. После этого в vm останется некоторое количество нулевых элементов, равное разности между суммой элементов скомпенсированной матрицы a и суммой элементов матрицы a. Эти нулевые элементы заполняются любыми подходящими значениями из массива t. Главное, чтобы все значения из t получили свое отражение в vm.

Если просто при заполнении vm обнулять уже использованные элементы t, то для элементов vm, полученных из-за компенсирования a, останется нулевая t. Поэтому вводится новый массив t1, равный t.

```
program test1;
const n=3; m=4;
var i,j,k,p,r:integer; t,t1: array[1..m,1..n] of integer;
c,vm:ARRAY[1..10*M*N] of integer; A:ARRAY[1..n,1..n] of integer;
. . . . . .
  komp; for i:=1 to k-1 do vm[i]:=0;
  for i:=1 to m do for j:= 1 to n do t1[i,j]:=t[i,j];
  for i:=1 to k-1 do
    begin
      r:=0; for p:=1 to m do if t1[p,c[i]]=c[i+1] then r:=p;
      if r>0 then
        begin
          vm[i]:=r; t1[r,c[i]]:=0;
        end else
          for p:=1 to m do if t[p,c[i]]=c[i+1] then vm[i]:=p;
    end;
end.
```

По программе test1 путь из входных векторов получился другой: 4, 1, 2, 3, 4, 3, 4, 3, 3, 2, 2, 1, 1. Но это только внешнее различие. Все переходы графа переходов автомата отражены в каждом из путей.

Для данной таблицы переходов экономия в оптимальном пути тестирования составила 1 вектор, а для реальных таблиц это значение может превысить в несколько раз саму таблицу (m*n). Разумеется, имеет смысл проводить оптимальное тестирование, когда тестируется несколько одинаковых устройств. Аналогично можно тестировать и автоматы Мили, когда выходы зависят от входного вектора и от внутреннего состояния. Для автомата Мура при тестировании достаточно один раз проверить соответствие между выходным вектором и внутренним состоянием, а

затем проверять только внутренние состояния, если они доступны. Для автомата Мили на каждом входном векторе надо проверять и выходные вектора и внутренние состояния, если они доступны.

3. Случайные процессы

Метод Монте-Карло [4], использующий датчики случайных чисел, часто дает хороший результат. Этот метод применяется в случае, когда

1) нет аналитических способов решения задачи,
2) аналитические способы существуют, но очень сложны и требуют много времени и сил на их изучение и применение,
3) для проверки решений, полученных другими методами.

Используя метод Монте-Карло, надо иметь в виду, что размер цикла, в котором проводится усреднение и получается решение, должен быть достаточно большим. Так что программа, использующая метод Монте-Карло может быть небольшой, а выполняться может долго.

В задачах, решаемых теорией массового обслуживания, часто моделируются случайные процессы. Многое процессы в обществе, экономике и технике можно считать случайными. Поэтому социологи, работники управления, экономисты и инженеры должны уметь их моделировать.

Для решения такого рода задач используются датчики случайных чисел. Случайные числа могут выдаваться по равномерному закону с равной вероятностью в заданном диапазоне, или по закону Гаусса (нормальному) с заданными параметрами. Датчики случайных чисел имеются как в прикладных математических пакетах, так и различных алгоритмических языках.

1. Сколько дверей надо открыть в театре, чтобы не было скопления зрителей перед входом. В театре N мест, проход одного зрителя через дверь занимает t секунд, зрители начинают собираться за b минут до начала представления.

Будем считать, что все билеты проданы, то есть в театр придут N человек. Процесс прихода зрителей можно описать как равномерным, так и нормальным законом. При использовании нормального закона самый большой поток зрителей идет за (b/2) минут до начала представления и дверей потребуется открыть больше, чем при использовании равномерного закона.

Если изобразить времена прихода зрителей, смоделированные по случайному закону, как точки на отрезке длиной b минут, то для решения первой задачи надо найти отрезок длиной t секунд с максимальным числом точек на нем. Это и будет необходимое количество открытых дверей.

Массив времен прихода зрителей m[1..n] получим с помощью датчика случайных чисел (далее в программе равномерного, а можно и нормального). Затем массив m упорядочим методом быстрой сортировки, так как он имеет большую размерность и упорядочивать методом "пузырька" его долго.

Вспомогательный массив m1[1..n] получим из упорядоченного массива m: для каждого i от 1 до n m1[i]=m[i]+t. Таким образом, массив m1 - это массив освобождения зрителями дверей после прохода в здание. Массив m1 уже упорядочен. Эти два массива можно соединить в один, вдвое большей размерности, и тоже упорядоченный. Далее можно последовательно рассматривать элементы этого соединенного массива.

Массивы m и m1 можно и не соединять, а рассматривать каждый раз минимальный из элементов двух массивов (как при их соединении). Здесь k – указатель массива m, а k1 – массива m1. Цикл перебора идет от двух до (2 x n). Если m[k]<m1[k1], то к указателю k прибавляется единица, и к числу открытых дверей p тоже прибавляется единица. Если m[k]>m1[k1], то к указателю k1 прибавляется единица, и из p вычитается единица. Если m[k]=m1[k1], то оба указателя (k1 и k) увеличиваются на единицу.

Происходит суммирование n единичных функций, когда для каждого i от 1 до n в m[i] начало единичной функции, а в m1[i] ее конец. В итоге получается ступенчатая функция, ступеньки высотой 1 и различной ширины. Каждый зритель добавляет в эту ступенчатую функцию прямоугольник, длиной t и высотой 1. При выполнении программы происходит как бы очерчивание этой ступенчатой функции. Общая площадь под ступенчатой функцией будет (n x t). У этой ступенчатой функции есть свои локальные минимумы и максимумы. Поскольку определяется максимум этой функции, то надо до конца рассматривать только элементы массива m (прибавляющего). **Программа 1:**

```
program doors1;
const n=500;t=5;b=30;
type ma=array[1..n] of integer;
var m,m1:ma;k,d,p,i,k1:integer;
procedure sort(var m:ma;e,o:integer);
var c,r,a:integer;
procedure obm(var m:ma;e,o:integer);
var b:integer;
begin b:=m[e];m[e]:=m[o];m[o]:=b end;
begin
  if e<o then
    begin
      c:=e;r:=o;a:=m[o];
      repeat
        while m[c]<a do c:=c+1;
```

```
        while (m[r]>=a) and (r>c) do r:=r-1;
        if c<r then obm(m,c,r);
      until c=r;
      obm (m,c,o);sort(m,e,(c-1));sort(m,(c+1),o);
    end
end;
begin
  randomize; for i:=1 to n do m[i]:=random(b*60+1);sort(m,1,n);
  for i:=1 to n do m1[i]:=m[i]+t;
  k:=2;k1:=1;d:=1;p:=1;
  for i:=2 to (2*n) do if k<=n then
    begin
      if m[k]<m1[k1] then
        begin
          k:=k+1;p:=p+1;if p>d then d:=p
        end;
      if m[k]>m1[k1] then
        begin
          k1:=k1+1;p:=p-1
        end;
      if m[k]=m1[k1] then
        begin
          k1:=k1+1;k:=k+1
        end;
    end;
  writeln(d:8);
end.
```

Здесь p – текущее число открытых дверей, а d – максимальное.

Можно решить первую задачу несколько иначе. Считаем все двери равноправными, так как зрителю все равно, через какую дверь войти. Для моделирования процесса прихода воспользуемся еще одним массивом m1[1..n]. Сначала этот массив пуст. После заполнения и сортировки массива m начинаем заполнять массив m1. Берём время прихода первого зрителя m1[1] и начинаем последовательно сравнивать с ним времена прихода следующих зрителей m[2], m[3] и т.д. Если (m[2]-m[1])<t, то элементу m1[1] присваиваем значение m[2]. Если (m[3]-m[1])<t, то элементу m1[2] присваиваем значение m[3] и так далее., пока не найдем первое m[i], такое что (m[i]-m[1])>t. Тогда значение m[1] заменяем значением m[i] и последующие элементы массива m сравниваются с m[i]. Заполнения массива m1 далее продолжается аналогично. Параллельно идет подсчет числа занятых элементов массива m1 (целая переменная (k1-1)).

После того, как массив m обработан до конца, массивы m и m1 меняются местами. Смоделирована работа первой двери, переходим к

моделированию работы второй двери. Теперь берем m1[1] и сравниваем с ним последующие элементы массива m1. Заметим, что обнулять массив m нет необходимости, так как ведется подсчет числа занятых элементов массива m (целая переменная (k-1)).

Далее массивы опять меняются местами и моделируется работа третьей двери до тех пор пока переменные k или k1 не станут равными единице. Это будет означать, что процесс моделирования закончен, больше проходить через дверь некому. **Программа 2:**

```
program doors2;
const n=500;t=5;b=30;
type ma=array[1..n] of integer;
var m,m1:ma;k,d,p,i,k1:integer;
procedure sort(var m:ma;e,o:integer); … end;
begin
  randomize; for i:=1 to n do m[i]:=random(b*60+1);sort(m,1,n);
  k:=n+1; d:=0;
  repeat
    p:=m[1];k1:=1;d:=d+1;
    for i:=2 to (k-1) do
      begin
        if m[i]<p+t then
          begin
            m1[k1]:=m[i];k1:=k1+1
          end
        else
          p:=m[i]
      end;
    if k1>1 then
      begin
        p:=m1[1];k:=1;d:=d+1;
        for i:=2 to (k1-1) do
          begin
            if m1[i]<p+t then
              begin
                m[k]:=m1[i];k:=k+1
              end
            else
              p:=m1[i]
          end
      end
    until (k1=1) or (k=1);
    writeln(d:8);
end.
```

Здесь d – число открытых дверей, а p – вспомогательная переменная, используеится при составлении масивов m1 и m, в ней запоминается значение предыдущего элемента массивов m или m1 соответственно.

Здесь и далее рекурсивная процедура sort заменена одной строкой: procedure sort(var m:ma;e,o:integer); … end;

В программе 2 двери получаются разной загруженности, первая по очереди моделирования – самая загруженная, … , последняя – меньше всего загруженная. Если у ступенчатой функции собрать все единичные ступеньки, образующие максимумы, то получим работу последней двери (самую незагруженную) при моделировании вторым способом. Если из ступенчатой функции убрать единичные ступеньки, образующие максимумы, то получим новую ступенчатую функцию с максимумами на единицу меньше. . Если у новой ступенчатой функции собрать все единичные ступеньки, образующие новые максимумы, то получим работу предпоследней двери при моделировании вторым способом. И т.д.

Для проверки правильности работы первой и второй программ каждый вариант заполнения датчиком случайных чисел массива m использовался для получения результатов в каждой из программ, и количества открытых дверей в программах совпали.

Минимизация числа открытых дверей экономит труд уборщиц и охранников, если они есть, и сохраняет тепло в помещении зимой. Зрители, разуметься, тоже не должны быть обижены. На закрытых дверях должны быть таблички: "Закрыто. Открыто рядом" и стрелка, направленная к открытой двери. В этом случае у зрителей не будет потерь времени на поиски открытой двери.

Очевидно, что не имеет смысла каждый день определять число открытых дверей, исходя из числа проданных билетов. Надо составить таблицу, по которой, имея число проданных билетов, можно определить число дверей, которое необходимо открыть. Для этого программу 1 или программу 2 надо оформить в виде процедуры, зависящей от количества зрителей n и определяющей число дверей d.

```
program tabl1;
const n=500;t=5;b=30;
type ma=array[1..n] of integer;
var d,d1,l:integer;m,m1:ma;
procedure door(var d:integer;nn:integer);
var k,p,i,k1:integer;
procedure sort(var m:ma;e,o:integer); …  end;
begin
   for i:=1 to nn do m[i]:=random(b*60+1);sort(m,1,nn);
  for i:=1 to nn do m1[i]:=m[i]+t;
  k:=2;k1:=1;d:=1;p:=1;
  for i:=2 to (2*nn) do if k<=nn then
```

```
    begin
      if m[k]<m1[k1] then
        begin
          k:=k+1;p:=p+1;if p>d then d:=p
        end;
      if m[k]>m1[k1] then
        begin
          k1:=k1+1;p:=p-1
        end;
      if m[k]=m1[k1] then
        begin
          k1:=k1+1;k:=k+1
        end;
    end;
end;
begin
  randomize;d1:=1; for l:=1 to n do
    begin
      door(d,l); if d>d1 then
        begin
          d1:=d;writeln(d:8,l:10)
        end
    end
end.
```

При больших n может оказаться важно, какая из программ 1 или 2 более быстродействующая. Большую часть времени занимает заполнение массива m с помощью датчика случайных чисел random. Поэтому заполнение массива m может быть вынесено из процедуры door. Чтобы получить более достоверные результаты, надо несколько раз выполнить программу tabl, а результаты усреднить. Такая таблица особенно важна, если используется не равномерный случайный закон, а какой либо другой.

Если заполнение массива m вынесено из процедуры, то сортировка его все равно происходит в процедуре door. В процедуру берется часть неупорядоченного массива. Кроме того, случайно задается элемент массива m, начиная с которого необходимое количество элементов передаются в процедуру для моделирования. Для этого массив m как бы зацикливается, то есть после номера n следует номер 1.

```
program tabl2;
const n=500;t=5;b=30;
type ma=array[1..n] of integer;
var d,d1,l:integer;m,m1,m2:ma;
procedure door(var d:integer;nn:integer);
```

```
var k,p,i,k1:integer;
procedure sort(var m2:ma;e,o:integer);
var c,r,a:integer;
procedure obm(var m2:ma;e,o:integer);
var b:integer;
begin b:=m2[e];m2[e]:=m2[o];m2[o]:=b end;
begin
  if e<o then
    begin
      c:=e;r:=o;a:=m2[o];
      repeat
        while m2[c]<a do c:=c+1;
        while (m2[r]>=a) and (r>c) do r:=r-1;
        if c<r then obm(m2,c,r);
      until c=r;
      obm (m2,c,o);sort(m2,e,(c-1));sort(m2,(c+1),o);
    end
end;
begin
  k1:=random(n)+1;
  for i:=1 to nn do m2[i]:=m[(i+k1-2) mod n +1];sort(m2,1,nn);
 k:=nn+1; d:=0;
  repeat
    p:=m2[1];k1:=1;d:=d+1;
    for i:=2 to (k-1) do
      begin
        if m2[i]<p+t then
          begin
            m1[k1]:=m2[i];k1:=k1+1
          end
        else
          p:=m2[i]
      end;
    if k1>1 then
      begin
        p:=m1[1];k:=1;d:=d+1;
        for i:=2 to (k1-1) do
          begin
            if m1[i]<p+t then
              begin
                m2[k]:=m1[i];k:=k+1
              end
            else
```

```
            p:=m1[i]
          end
      end
    until (k1=1) or (k=1);
end;
begin
  randomize; for l:=1 to n do m[l]:=random(b*60+1);
  d1:=1; for l:=1 to n do
    begin
      door(d,l); if d>d1 then
        begin
          d1:=d;writeln(d:8,l:10)
        end
    end
end.
```

Аналогичные задачи возникают в случаях определений чисел телефонных линий в справочных службах, количеств бригад скорой помощи и аварийных служб и т.д.

2. Сколько надо иметь бригад ремонтников, чтобы при возникновении аварии не было задержки ее ликвидации. Время ликвидации аварии колеблется от t1 до t2 минут. Время прибытия бригады на место аварии входит во время ликвидации. В день в среднем бывает n аварий.

При определении числа бригад не будем учитывать восьмичасовую продолжительность рабочего дня, сорокачасовую рабочую неделю и т.д. Просто определим, сколько надо иметь в каждый момент бригад ремонтников, готовых к ликвидации аварии. С учетом продолжительности рабочего дня, рабочей недели и т.д. таких бригад, конечно, потребуется больше. Этот учет можно сделать и без датчиков случайных чисел, поэтому в настоящей статье он не рассматривается.

Время возникновения аварии (массив m[1..n]) и время ликвидации аварии (массив t[1..n]) получим с помощью датчика случайных чисел.

```
program brig1;
const n=20;t1=30;t2=100;
type ma=array[1..n] of integer;
var m,m1,t:ma;k,d,p,i,k1:integer;
procedure sort(var m:ma;e,o:integer); … end;
begin
  randomize; for i:=1 to n do m[i]:=random(24*60+1);sort(m,1,n);
  randomize; for i:=1 to n do t[i]:=t1+random(t2-t1+1);
  for i:=1 to n do m1[i]:=m[i]+t[i];sort(m1,1,n);
  k:=2;k1:=1;d:=1;p:=1;
  for i:=2 to (2*n) do if k<=n then
```

```
    begin
      if m[k]<m1[k1] then
        begin
          k:=k+1;p:=p+1;if p>d then d:=p
        end;
      if m[k]>m1[k1] then
        begin
          k1:=k1+1;p:=p-1
        end;
      if m[k]=m1[k1] then
        begin
          k1:=k1+1;k:=k+1
        end;
    end;
  writeln(d:8);
end.
```

Эта программа мало чем отличается от первой. Здесь времена ликвидаций аварий различны, тогда как время прохода через дверь у всех зрителей одинаково. В отличие от программы 1, массив m1 сразу не получается упорядоченным, и значит его надо после получения упорядочить. Тут опять возникает ступенчатая функция, только каждая i-ая авария приносит ступень высотой 1 и шириной t[i]. В этом случае площадь под ступенчатой функцией будет равно сумме от 1 до n элементов массива t. В среднем каждое t[i] равно (t1+t2)/2, а сумма – (n x (t1+t2))/2. Аналогично можно переделать и вторую программу под эту задачу. Вместо переменной t надо поставить соответствующий элемент массива t[i]. Кроме того, необходим массив tt[1..n] для запоминания времен ликвидации при моделировании работы каждой последующей бригады. Массив tt получается из массива t (и наоборот t из tt), аналогично тому, как массив m1 получается из массива m (и наоборот m из m1).

```
program brig2;
const n=20;t1=30;t2=100;
type ma=array[1..n] of integer;
var m,m1,t,tt:ma;k,d,p,i,k1,td: integer;
procedure sort(var m:ma;e,o:integer); … end;
begin
  randomize; for i:=1 to n do m[i]:=random(24*60+1);sort(m,1,n);
  randomize; for i:=1 to n do t[i]:=t1+random(t2-t1+1);
  k:=n+1; d:=0;
  repeat
    p:=m[1];k1:=1;d:=d+1; td:=t[1];
    for i:=2 to (k-1) do
```

```
      begin
        if m[i]<p+td then
          begin
            m1[k1]:=m[i];tt[k1]:=t[i];k1:=k1+1
          end
        else
          begin
            p:=m[i]; td:=t[i];
          end;
      end;
    if k1>1 then
      begin
        p:=m1[1];k:=1;d:=d+1; td:=tt[1];
        for i:=2 to (k1-1) do
          begin
            if m1[i]<p+td  then
              begin
                m[k]:=m1[i];t[k]:=tt[i];k:=k+1
              end
            else
              begin
                p:=m1[i];td:=tt[i];
              end
          end
      end
    until (k1=1) or (k=1);
    writeln('          ',d:8);
end.
```

Здесь в переменной td запоминается значение предыдущего элемента массивов t или tt соответственно.

Для проверки правильности работы двух вариантов программ для второй задачи каждый вариант заполнения датчиком случайных чисел массива m использовался для получения результатов в каждом из вариантов программ, и количества бригад в вариантах программах совпали.

Теперь рассмотрим вторую задачу с задержкой ликвидации аварии.

2а. Сколько надо иметь бригад ремонтников, чтобы при возникновении аварии задержка ее ликвидации была не более t3 минут. Время ликвидации аварии колеблется от t1 до t2 минут. Время прибытия бригады на место аварии входит во время ликвидации. В день в среднем бывает n аварий.

Иногда задержка крайне нежелательна, и даже невозможна, как например, в скорой помощи. А вот без кабельного ТВ, к примеру, некоторое время жители могут обойтись.

Если бы задержка была не ограничена сверху, то n аварий в день занимали бы ((t1+t2) x n)/2 минут (площадь под ступенчатой функцией). Каждая бригада работает (24 x 60) минут. Итого, надо в каждый момент иметь ((t1+t2) x n)/(2 x 24 x 60) бригад. Это число ограничивает результат снизу, меньше бригад в программе получиться не должно.

Для решения задачи 2а обратимся к программе 2. При моделировании работы каждой бригады возникают временные интервалы, когда одна авария ликвидирована, на вызовы, которые были во время ликвидации, направлены другие бригады, и надо ждать следующего вызова. Постараемся, где это возможно, убрать интервалы в работе бригад, за счет чего повысить эффективность их работы. Если вызовы были за время, меньшее t3 до конца ликвидации аварии, то этот вызов может обслужить и бригада, находящаяся на ликвидации. Тогда один интервал в работе бригады исчезнет. Если таких вызовов нет, то интервал останется. Очевидно, что чем больше t3, тем меньше остается интервалов и при большом t3 результатом будет нижняя оценка.

```
program brig2a;
const n=20;t1=30;t2=100;t3=60;
type ma=array[1..n] of integer;
var m,m1,t,tt:ma;k,d,p,i,k1,td,pm:integer;
procedure sort(var m:ma;e,o:integer); … end;
begin
  randomize; for i:=1 to n do m[i]:=random(24*60+1);sort(m,1,n);
  randomize; for i:=1 to n do t[i]:=t1+random(t2-t1+1);
  k:=n+1; d:=0;
  repeat
    p:=m[1];k1:=1;d:=d+1;td:=t[1];pm:=p+24*60;
    for i:=2 to (k-1) do
      begin
        if (m[i]<p+td-t3) or (p>pm) then
          begin
            m1[k1]:=m[i];tt[k1]:=t[i];k1:=k1+1
          end
        else
          begin
            if m[i]<p+td then m[i]:=p+td;p:=m[i];td:=t[i]
          end;
      end;
    if k1>1 then
      begin
```

```
      p:=m1[1];k:=1;d:=d+1;td:=tt[1];pm:=p+24*60;
      for i:=2 to (k1-1) do
        begin
          if  (m1[i]<p+td-t3) or  (p>pm) then
            begin
              m[k]:=m1[i];t[k]:=tt[i];k:=k+1
            end
          else
            begin
              if m1[i]<p+td then m1[i]:=p+td;p:=m1[i];td:=tt[i];
            end;
        end;
    end;
  until (k1=1) or (k=1);
  writeln('          ',d:8);
end.
```

Проводилось сравнение с нижней оценкой для разных n. Для больших задержек число бригад совпадает с нижней оценкой.

Поскольку при сдвиге времени начала ликвидации аварии существует опасность выйти за пределы суток (24 * 60 минут), то для каждой бригады делается ограничение сверху продолжительности ее работы, равное pm.

По первой программе тоже можно решить задачу 2а. В суммарной ступенчатой функции рассмотрим локальные минимумы и максимумы. Максимумы будем уменьшать, а минимумы увеличивать. Постараемся таким образом сгладить ступенчатую функцию. Экстремумы, шириной не более t3, можно удалить, передвинув вперед на ширину ступеньки один отрезок из образующих экстремум. Если экстремальные ступеньки шире t3, то придется двигать несколько отрезков. К этому еще надо иметь массив, запоминающий сдвиги отрезков вперед. Иначе можно случайно сдвинуть отрезок на время, большее t3. Программа будет выполняться за несколько итераций, сглаживая локальные экстремумы, пока это возможно. При таком сглаживании экстремумов площадь под ступенчатой функцией не меняется.

Алгоритм, составленный по первой программе, получается не такой простой, как составленный по второй программе. Это доказывает, что всегда лучше иметь несколько алгоритмов на одну задачу, так как при усложнении задачи какой-то из алгоритмов может оказаться более удобным.

2б. Какое максимальное число аварий приходится ликвидировать одной бригаде, если при возникновении аварии на ее ликвидацию направляется бригада, дольше всех ждущая вызова. Таким образом достигается равномерная загруженность бригад. При возникновении аварии не было задержки ее ликвидации. Время ликвидации аварии

колеблется от t1 до t2 минут. Время прибытия бригады на место аварии входит во время ликвидации. В день в среднем бывает n аварий.

Естественно желание бригады обеспечить себя на весь рабочий день всем необходимым для ликвидации аварий. Поэтому задача 2б актуальна.

Ранее у всех бригад получалась разная загруженность. Существует алгоритм, по которому у бригад получается равномерная загруженность. Вводится массив z[1..n], содержащий номера бригад, обслуживающих аварии. Если аварию m[i] обслуживает j-ая бригада, то z[i]=j. Вводится еще массив ok[1..n], в котором упорядочены номера уже законченных аварий. Аварии нумеруются в порядке их возникновения и начала ликвидации. I-ая авария возникла в m[i] минут от начала суток, ликвидировала ее бригада № z[i]. Чтобы в упорядоченном массиве m1 найти время окончания ликвидации i-ой аварии, надо в массиве ok найти j такое что ok[j]=i. Все элементы массива ok различны и находятся в интервала от 1 до n. Тогда i-ая авария закончится во время m1[j], j-ой по очереди из ликвидированных аварий.

В ok[1] находится номер аварии, которая ликвидирована первой, … , в ok[n] находится номер аварии, которая ликвидирована последней. При этом m1[j] – время конца ликвидации аварии № ok[j], которая была ликвидирована j-ой. Ликвидацию производила z[ok[j]] бригада. Если d – число бригад, то для i в интервале от (d+1) до n m[i] не больше m1[i-d], так как d – максимум разности между количествами начатых и законченных аварий.

```
program brig1b;
const n=20;t1=30;t2=100;
type ma=array[1..n] of integer;
var m,m1,t,ok,z,ch:ma;k,d,p,i,k1,td,j,pm:integer;
procedure sort(var m:ma;e,o:integer); … end;
procedure sort1(var m:ma;e,o:integer);
var c,r,a:integer;
procedure obm1(var m:ma;e,o:integer);
var b:integer;
begin b:=m[e];m[e]:=m[o];m[o]:=b;b:=ok[e];ok[e]:=ok[o];ok[o]:=b end;
begin
  if e<o then
    begin
      c:=e;r:=o;a:=m[o];
      repeat
        while m[c]<a do c:=c+1;
        while (m[r]>=a) and (r>c) do r:=r-1;
        if c<r then obm1(m,c,r);
      until c=r;
```

```
      obm1 (m,c,o);sort1(m,e,(c-1));sort1(m,(c+1),o);
    end
end;
begin
  randomize;for j:=1 to 300 do
    begin
      for i:=1 to n do m[i]:=random(24*60+1);sort(m,1,n);
      for i:=1 to n do t[i]:=t1+random(t2-t1+1);
      for i:=1 to n do
        begin
          m1[i]:=m[i]+t[i];ok[i]:=i;ch[i]:=0;
        end;
      sort1(m1,1,n);
      k:=2;k1:=1;d:=1;p:=1;
      for i:=2 to (2*n) do if k<=n then
        begin
          if m[k]<m1[k1] then
            begin
              k:=k+1;p:=p+1;if p>d then d:=p
            end;
          if m[k]>m1[k1] then
            begin
              k1:=k1+1;p:=p-1
            end;
          if m[k]=m1[k1] then
            begin
              k1:=k1+1;k:=k+1
            end;
        end;
      for i:=1 to d do z[i]:=i;for i:=d+1 to n do z[i]:=z[ok[i-d]];
      for i:=1 to n do ch[z[i]]:=ch[z[i]]+1;if j=1 then pm:=ch[1];
      if (d>1) and (j=1) then for i:=2 to d do if ch[i]>pm then pm:=ch[i];
      if j>1 then for i:=1 to d do if ch[i]>pm then pm:=ch[i];
    end;
  writeln(pm:10);
end.
```

Здесь кроме процедуры сортировки sort существует еще и процедура sort1, в которой другая процедура обмена obm1. В obm1 вместе с перестановкой элементов массива m1 аналогично переставляются и элементы массива ok. В массиве ok упорядочены номера аварий, которые закончились. В ok[1] – номер аварии, которая ликвидирована первой, … , в ok[n] – номер аварии, которая ликвидирована последней. Перед

сортировкой массива m1 каждому i-ому элементу массива ok присваивается значение i.

По этому алгоритму для каждой бригады можно определить и общее время занятости на авариях, и максимальный и минимальный перерыв между авариями и т.д.

Теперь после решения задачи 2б можно решить задачу 2а, используя первую программу. Это решение будет более сложное, но и более точное в смысле моделирования реального процесса. Массив z более не понадобится, а массив ok и процедура sort1 будут нужны.

Как уже было отмечено, при d+1<=i<=n m[i] должна быть не меньше m1[i-d]. Теперь, с задержкой t3, если m1[i-d]-m[i]<=t3 и m1[i-d]>m[i], то m[i] можно сдвинуть, приравняв ее к m1[i-d]. При это м надо иметь в виду, что рассматривается ликвидация аварий в суточном временном интервале, и сдвиг возможен, если m1[i-d]<24*60 (минут). При сдвиге элемента массива m должен изменится и соответствующий ему элемент массива m1. Находим j такое что ok[j]=i и аналогично увеличиваем m1[j]. Далее массив m1 опять сортируем по возрастанию от элемента j до элемента n. При сортировке m1 переставляется и массив ok. Массив m сортировать после сдвига m[i] не нужно. Если m[i] и превысит соседние элементы массива m, то они в свою очередь подвергнутся сдвигу при последующей обработке так, что после всех сдвигов массив m останется упорядоченным по возрастанию.

Если теперь взять число бригад d, полученное при решении задачи 2, то сдвигать массивы не понадобится. Поэтому будем последовательно, на единицу за шаг, уменьшать число бригад d, полученное при решении задачи 2, пока на какой-то аварии сдвиг станет необходим, а сделать его будет невозможно (флаг k1 из нуля станет равным единице). В этом случае минимальное число бригад при возможной задержке t3 будет минимальном из таких, при которых сдвиг еще возможен.

```
program brig1a;
const n=20;t1=30;t2=100;t3=50;
type ma=array[1..n] of integer;
var m,m1,t,ok:ma;k,d,p,i,k1,td,j:integer;
procedure sort(var m:ma;e,o:integer); … end;
procedure sort1(var m:ma;e,o:integer); … end;
begin
  randomize;for i:=1 to n do m[i]:=random(24*60+1);sort(m,1,n);
  for i:=1 to n do t[i]:=t1+random(t2-t1+1);
  for i:=1 to n do
    begin
      m1[i]:=m[i]+t[i];ok[i]:=i;
    end;
  sort1(m1,1,n);
```

```
k:=2;k1:=1;d:=1;p:=1;
for i:=2 to (2*n) do if k<=n then
  begin
    if m[k]<m1[k1] then
      begin
        k:=k+1;p:=p+1;if p>d then d:=p
      end;
    if m[k]>m1[k1] then
      begin
        k1:=k1+1;p:=p-1
      end;
    if m[k]=m1[k1] then
      begin
        k1:=k1+1;k:=k+1
      end;
  end;
p:=d; k1:=0;repeat
  p:=p-1; for i:=p+1 to n do if k1=0 then if m[i]<m1[i-p] then
   if (m1[i-p]-m[i]<=t3) and (m1[i-p]<24*60) then
    begin
      j:=0;k:=0; repeat j:=j+1; if ok[j]=i then
        begin
          k:=1; m1[j]:=m1[j]+m1[i-p]-m[i]; m[i]:=m1[i-p]; sort1(m1,j,n);
        end; until k=1;
    end else k1:=1;
    until (k1=1) or (p=1); if k1=1 then p:=p+1; writeln(p:6);
end.
```

С другой стороны, нельзя решить задачу 2б, используя программу 2, т.к. по второму алгоритму бригады будут загружены работой неравномерно. Для некоторых задач эта неравномерность значения не имеет, а для задачи 2б – имеет. По второму алгоритму у бригад имеются различные коэффициенты приоритета в направлении на аварию. На ликвидацию аварии направляется бригада из незанятых на ликвидации, которая имеет больший (меньший) коэффициент приоритета. Чтобы результаты расчетов по программам bpig2a и brig1a совпали , в программу brig1a надо ввести коэффициенты приоритета при направлении на ликвидацию аварии для бригад. Надо иметь в виду, что эти коэффициенты должны быть различными для всех бригад. Никакие две бригады не должны иметь равные коэффициенты приоритетов. Направляться на ликвидацию в программе brig1a будет та бригада, у которой больше приоритет, из числа освободившихся бригад и бригад, которые освободятся не позднее времени t3 от вызова на ликвидацию аварии.

Результаты расчетов по программам brig1a и brig2a в большинстве случаев совпадают. При несовпадениях половина случаев, когда результаты расчетов по одной программе превышают результаты расчетов по другой, и половина протовоположных случаев.

Не лишена интереса и задача 3, как бы обратная ко второй:

3. Имеется d бригад ремонтников. Время ликвидации аварии колеблется от t1 до t2 минут. Время прибытия бригады на место аварии входит во время ликвидации. В день в среднем бывает n аварий. Какова максимальная задержка ликвидации аварии t3 в минутах.

```
 program zader;
const n=120;t1=30;t2=100;d=6;label 1;
type ma=array[1..n] of integer;
var m,m1,t:ma;k,p,i,k1,j,l,t3:integer;
procedure sort(var m:ma;e,o:integer);
var c,r,a:integer;
procedure obm(var m:ma;e,o:integer);
var b:integer;
begin b:=m[e];m[e]:=m[o];m[o]:=b end;
begin
  if e<o then
    begin
      c:=e;r:=o;a:=m[o];
      repeat
        while m[c]<a do c:=c+1;
        while (m[r]>=a) and (r>c) do r:=r-1;
        if c<r then obm(m,c,r);
      until c=r;
      obm (m,c,o);sort(m,e,(c-1));sort(m,(c+1),o);
    end
end;
procedure sort1(var m,tt:ma;e,o:integer);
var c,r,a:integer;
procedure obm(var m:ma;e,o:integer);
var b:integer;
begin b:=m[e];m[e]:=m[o];m[o]:=b;b:=tt[e];tt[e]:=tt[o];tt[o]:=b end;
begin
  if e<o then
    begin
      c:=e;r:=o;a:=m[o];
      repeat
        while m[c]<a do c:=c+1;
        while (m[r]>=a) and (r>c) do r:=r-1;
```

```
        if c<r then obm(m,c,r);
      until c=r;
      obm (m,c,o);sort(m,e,(c-1));sort(m,(c+1),o);
    end
end;
begin
  j:=((t1+t2)*n) div (48*60); k:=((t1+t2)*n) mod (48*60);
  if k>0 then j:=j+1;
  if d<j then writeln('Little') else
    begin
      randomize; for i:=1 to n do m[i]:=random(24*60+1);sort(m,1,n);
      randomize; for i:=1 to n do t[i]:=t1+random(t2-t1+1);
      for i:=1 to n do m1[i]:=m[i]+t[i];sort(m1,1,n);
      k:=2;k1:=1;p:=1;t3:=0;
      for i:=2 to (2*n) do if k<=n then
        begin
          1: if m[k]<m1[k1] then if p<d then
            begin
              k:=k+1;p:=p+1;
            end else begin
              if (m1[k1]-m[k])>t3 then t3:=m1[k1]-m[k];
              m[k]:=m1[k1]; sort1(m,t,k,n);
              for l:=1 to n do m1[l]:=m[l]+t[l];
              sort(m1,k1,n); goto 1;
            end;
          if m[k]>m1[k1] then
            begin
              k1:=k1+1;p:=p-1
            end;
          if m[k]=m1[k1] then
            begin
              k1:=k1+1;k:=k+1
            end;
        end;
      writeln(t3:8);
    end;
end.
```

Здесь обозначения такие же, как и в предыдущих задачах. Программа zader работает по первому ступенчатому алгоритму. Если встречается ступенька выше d (числа бригад), то соответиствующий элемент массива начала работ m сдвигается (задержка), далее массив m упорядочивается вместе с массивом t (процедура sort1), потом заново вычисляется и упорядочивается массив конца работ m1. Задержка, если она превышает t3,

запоминается. После всех действий элементы массива m с первого по (k-1)-й и массива m1 с первого по k1-й не изменятся. Заново происходит сравнение нового m[k] и старого m1[k1] и, если ступенька опять превышает d, то вышеописанные действия повторяются. В начале программы проверяется нижняя оценка количества бригад, необходимых для ликвидации n аварий при заданных t1 и t2. Если бригад мало, то на дисплей выдается соответствующее сообщение.

Датчики случайных чисел нужны и при моделировании задач с допусками и посадками.

4. Два горизонтальных размера молочного пакета (квадрат) имеют математическое ожидание m и дисперсию d (нормальный закон). В ящике уложено n1 x n2 молочных пакетов. Какие должны быть математические ожидания у ящиков с пакетами, чтобы в 9999 случаях из 10000 все пакеты в ящик входили. Дисперсия у ящиков задана – d2.

Аналогичные задачи могут быть и для круглых предметов, и для цилиндрических. Способ укладки этих предметов в ящик может быть различным.

Тут как бы обратная задача – по полученному в расчетах набору данных получить параметры случайного закона. Считаем, что вид случайного закона для размеров ящика совпадает с видом случайного закона для размеров пакетов (нормальный).

Известно, что математическое ожидание суммы нормально распределенных одинаковых случайных величин равно сумме мат. ожиданий, а дисперсия суммы – квадратному корню из суммы величин, умноженной на дисперсию одной величины. Таким образом, мат. ожидание и дисперсия набора из n1 на n2 молочных пакетов известны, и можно воспользоваться датчиком нормально распределенных чисел randn. Здесь mo+n1*m (mo+n2*m) – мат. ожидания размеров ящика. Функция zab равна вероятности v невхождения пакетов в ящик для данного mo. Эта функция определяется с помощью датчика случайных чисел. Методом последовательного деления отрезка пополам находится значение mo, такое что вероятность v невхождения пакетов в ящик равна заданной (0.0001).

```
program rand1;
const k=30000;n1=3;n2=4;v=0.0001;D2=0.05;m=5.0;
var i,j:integer; a,b,mo,d,q,p,s,r,g,d1,p1,p2 :real;
function randn(var mo,d:real):real;
var i:integer;x:real;
begin
  x:=0.0000;for i:=1 to 10 do x:=x+random(11);
  x:=x/10.0000; randn:=(x-5.0000)*d+mo;
end;
function zab(mo:real):real;
var c,t:real; i:integer;
```

```
begin
  b:=1.00/k;c:=0.0;a:=0.00;t:=d2;
  for i:=1 to k do if randn(c,d1)>=randn(mo,t) then a:=a+b;
  zab:=a;
end;
begin randomize; d:=0.1000;
    d1:=sqrt(n1)*d;
    if (v>0.00) AND (v<1.00) THEN
    begin
      s:=1.5*(d1+d2);p:=s/2.00;
      while abs(zab(s)-v)>v/10.0 do
        begin
          if zab(s)>v then s:=s+p else if zab(s)<v then s:=s-p;p:=p/2.00;
        end;
    end;write(s+n1*m:6:4);
   d1:=sqrt(n2)*d;
    if (v>0.00) AND (v<1.00) THEN
    begin
      s:=1.5*(d1+d2);p:=s/2.00;
      while abs(zab(s)-v)>v/10.0 do
        begin
          if zab(s)>v then s:=s+p else if zab(s)<v then s:=s-p;p:=p/2.00;
        end;
    end;writeln('       ',s+n2*m:6:4);
end.
```

Результаты расчетов выводятся на печать. Альтернативой к этой программе, или ее проверкой, является программа, вычисляющая машинный (табличный) интеграл вероятности:

```
program rand2;
const k1=100;n1=3;n2=4;v=0.0001;D2=0.05;m=5.0;
var i,j:integer; a,b,mo,d,q,p,s,r,g,d1,p1,p2 :real;
function randn(var mo,d:real):real;
var i:integer;x:real;
begin
  x:=0.0000;for i:=1 to 10 do x:=x+random(11);
  x:=x/10.0000; randn:=(x-5.0000)*d+mo;
end;
function zav(mo:real):real;
var s,s1,x,y:real;
  begin
    if p1<mo+4.0*d2 then
      begin
        s:=0.0; x:=p1; while x<= p2 do
```

```
          begin
            if x<mo+4.0*d2 then
              begin
                s1:=0.0; y:=mo-4.0*d2; while y<= x do
                  begin
                    s1:=s1+exp(-((y-mo)*(y-mo)*b)); y:=y+d2/k1;
                  end;
              end else s1:=sqrt(2.0*3.1415)*k1;
            s:=s+s1*exp(-(x*x*a))/k1; x:=x+d1/k1;
          end;
      end else s:=k1*2.0*3.1415;
    zav:=s/(k1*2.0*3.1415);
  end;
begin
  d:=0.1000;d1:=sqrt(n1)*d;
  p1:=-4.0*d1; if (mo-4.0*d2)>p1 then p1:=mo-4.0*d2;
  p2:=4.0*d1;
  a:=1.0/(2.0*d1*d1); b:=1.0/(2.0*d2*d2);
    if (v>0.00) AND (v<1.00) THEN
    begin
      s:=1.5*(d1+d2);p:=s/2.00;
      while abs(zav(s)-v)>v/10.0 do
        begin
          if zav(s)>v then s:=s+p else if zav(s)<v then s:=s-p;p:=p/2.00;
        end;
    end;write(s+n1*m:6:4);
  d1:=sqrt(n2)*d;
  p1:=-4.0*d1; if (mo-4.0*d2)>p1 then p1:=mo-4.0*d2;
  p2:=4.0*d1;
  a:=1.0/(2.0*d1*d1); b:=1.0/(2.0*d2*d2);
    if (v>0.00) AND (v<1.00) THEN
    begin
      s:=1.5*(d1+d2);p:=s/2.00;
      while abs(zav(s)-v)>v/10.0 do
        begin
          if zav(s)>v then s:=s+p else if zav(s)<v then s:=s-p;p:=p/2.00;
        end;
    end;writeln('      ',s+n2*m:6:4);
end.
```

Здесь значение интеграла вероятности в зависимости от mo вычисляет функция zav. Интеграл считается на отрезке, равном восьми, умноженном на величину дисперсии. Четвертая задача была просчитана по двум программам и результаты совпали. В первой программе точность расчетов

зависит от числа опытов k, а во второй от числа делений k1 отрезка при вычислении определенного интеграла.

5. Метод ветвей и границ

Обычно метод ветвей и границ применяется для решения сложных задач. Для примера: алгоритм Литтла для решения задачи коммивояжера. Использование метода для решения сложных задач доказывает его эффективность, но затрудняет понимание самого метода. Далее будет показано применение метода ветвей и границ для решения простых и доступных для понимания задач. Эти простые задачи могут быть решены и другими методами, может быть не так изящно и эффективно, как методом ветвей и границ. Но для понимания сущности метода можно допустить и “стрельбу из пушек по воробьям”.

Простейшей иллюстрацией метода ветвей и границ является нахождение обратного значения непрерывной монотонной функции. Далее будет подразумеваться, что рассматривается непрерывная функция, если нет особой оговорки. Случаи разрывной и дискретной функций будут оговариваться особо. Если имеется сложная, не имеющая, например, аналитического выражения, но монотонная функция одной переменной и надо найти значение t, такое что
f (t)=v с заданной точностью e, то применяется метод последовательного деления отрезка пополам. Для этого нужно иметь две точки x и z, такие что f (x)<v и f (x)>v, или наоборот f (x)>v и f (x)<v. Точки x и z или задаются как исходные данные, или определяются в программе с помощью датчика случайных чисел. Если f (x)>v, то значения x и z меняются местами и в итоге обмена получается что f (x)<v<f (z). Затем отрезок между x и z делится пополам точкой t, t=(x+z)/2, и если f (t)>v, то z присваивается значение t и x остается прежним, а если f (t)<v, то x присваивается значение t и z остается прежним. Таким образом отрезок [x,z] уменьшился вдвое, а соотношение f (x)<v<f (z) осталось верным. Далее отрезок [x,z] опять делится пополам до тех пор, пока он больше e. Если abs(x-z)<e, то

процесс останавливается и в качестве t берется любое из значений x и z. В качестве критерия для непрерывной функции можно взять и модуль разности между f (x) и f (z) – abs(f (x)-f (z)) – и процесс продолжается до тех пор, пока abs(f (x)-f (z))>e. Программа del находит обратное значение как возрастающей, так и убывающей монотонной функции.

```
program del;
const e=0.001;label 1;
var x,z,t,v:real;
function f(x:real):real;
begin
  f:=x*x*x+x*x+x;
end;
begin
  readln(x,z,v); if ((v>f(x)) and (v<f(z))) or ((v<f(x)) and (v>f(z))) then
    begin
      if v<f(x) then
        begin
          t:=x; x:=z; z:=t;
        end;
      while abs(x-z)>e do
        begin
          t:=(z+x)/2; if abs(f(t)-v)<e then goto 1;
          if f(t)<v then x:=t;if f(t)>v then  z:=t;
        end;
      1:writeln('t=',t:10:3);
    end else
      writeln('granici');
end.
```

Здесь точки x и z задаются как исходные данные. Обозначения в программе del соответствуют обозначениям в статье. Если точки x и z заданы некорректно, т.е. значения функции f в них обоих или меньше или больше v, то на дисплей выдается сообщение.

Примером сложной непрерывной монотонной функции является зависимость размера годового бюджета страны от возраста выхода работников на пенсию. В условиях дефицита рабочей силы такая задача актуальна. Очевидно, что тем дольше человек работает до выхода на пенсию, тем больший доход он приносит “казне”. Получили, что функция зависимости бюджета от пенсионного возраста непрерывная, монотонная и возрастающая. Саму функцию получить трудно, но возможно с помощью бухгалтеров и социологов. С другой стороны удлинение рабочего периода у людей не очень хорошо для их здоровья, поэтому сразу и на много увеличивать возраст выхода на пенсию нельзя. Надо оценить необходимое увеличение бюджета, без которого страна не может обойтись (террор,

эпидемия и т.д.), и методом ветвей и границ получить неизбежное увеличение рабочего периода у людей.

Метод последовательного деления отрезка пополам напоминает артиллерийскую стрельбу в неподвижную мишень: перелет, недолет, в “яблочко”. Неподвижная мишень берется в “вилку”, как и монотонная функция при использовании метода ветвей и границ. Если вычисление значений функции занимает много времени, то можно ускорить процесс, несколько изменив метод.

На отрезке [x,z] функцию f можно экстраполировать прямой g, т.е. между точками с координатами (f (x),x) и (f (z),z) проводится прямая, и точка t берется такой, что значение экстраполирующей прямой в ней равно v. g(y)=a×y+b, где a=(f (x)-f (z))/(x-z), b=(z×f (x)-x×f (z))/(z-x). При этом будет g (x)=f (x) и g (z)=
f (z). Теперь t берется равным (v-b)/a, а далее процесс идет как и раньше: t на каждом шаге заменяет или x или z. В качестве экстраполирующей функции g можно брать и параболу, и логарифм, и любую функцию, которая, по мнению программиста, лучше всего приближает функцию f. Чем лучше g экстраполирует f, тем быстрее находится значение обратной функции. Программа del1 находит обратное значение монотонной функции с использованием экстраполирующей прямой:

```
program del1;
const e=0.001;label 1,2;
var x,z,t,a,b,v:real;
function f(x:real):real;
begin
  f:=x*x*x+x*x+x;
end;
begin
  readln(v);2:x:=random(100);z:=random(100);
   if ((v>f(x)) and (v<f(z))) or ((v<f(x)) and (v>f(z))) then
     begin
       if v<f(x) then
         begin
           t:=x; x:=z; z:=t;
         end;
       while abs(x-z)>e do
         begin
           a:=(f(x)-f(z))/(x-z); b:=(z*f(x)-x*f(z))/(z-x);
           t:=(v-b)/a; if abs(f(t)-v)<e then goto 1;
           if f(t)<v then x:=t;if f(t)>v then  z:=t;
         end;
       1:writeln('t=',t:10:3);
     end else
```

```
    goto 2;
end.
```

Здесь точки x и z задаются с помощью генератора случайных чисел random.

Если функция f дискретна или разрывна, но монотонна, то точного значения обратной функции может и не найтись. В этом случае берется ближайшее к v значение функции, а процесс деления отрезка происходит до тех пор, пока x и z на станут соседними точками, т.к. дискретная функции определена не на всех точках. Отрезок делить для дискретной функции тоже можно любым способом: как пополам, так и в соответствии с экстраполирующей функцией. При делении отрезка можно и не попасть в одну из точек, где определена дискретная монотонная функция, в этом случая надо брать ближайшую к точке деления точку, где функция определена. Программа dei находит или обратное значение дискретной монотонной целочисленной функции или два самих близких к нему значения:

```
program dei;
label 1;
var x,z,t,v:integer;
function f(x:integer):integer;
begin
  f:=x*x*x+x*x+x;
end;
begin
  readln(x,z,v); if ((v>f(x)) and (v<f(z))) or ((v<f(x)) and (v>f(z))) then
    begin
      if v<f(x) then
        begin
          t:=x; x:=z; z:=t;
        end;
      while abs(x-z)>1 do
        begin
          t:=(z+x) div 2; if f(t)=v then goto 1;
          if f(t)<v then x:=t;if f(t)>v then  z:=t;
        end;
      1:write('t=',t:10);if f(t)=v then writeln;
      if (f(t)>v) and (f(t+1)<v) then writeln((t+1):10);
      if (f(t)>v) and (f(t-1)<v) then writeln((t-1):10);
      if (f(t)<v) and (f(t+1)>v) then writeln((t+1):10);
      if (f(t)<v) and (f(t-1)>v) then writeln((t-1):10);
    end else
      writeln('granici');
end.
```

Программа dei в сущности отличается от программы del целочисленным делением div. Программа dei выводит на дисплей две ближайшие по значению функции к v точки, если не существует значения функции точно равного v.

В случае с монотонной функцией для каждого отрезка, на котором она задана, легко определить min и max функции на этом отрезке, так как они находятся на концах отрезка. Это свойство может служить и определением монотонной функции. Монотонная функция – это такая функция, которая на любом отрезке имеет min и max на его концах. Отсюда вывод: что если для некоторой сложной функции существует простой способ находить min и max ее на любом отрезке, где она определена, то к этой функции применим метод ветвей и границ для нахождения обратного значения этой функции. В более общем случае, если существует простой способ получать на каждом наборе значений функции верхнюю и нижнюю оценки, то можно применять метод ветвей и границ для нахождения обратного значения этой функции.

Метод ветвей и границ в отличие от метода прямого перебора позволяет с помощью верхних и нижних значений на каждом наборе значений функции отсекать от дальнейшего поиска некоторые варианты без их конкретного рассмотрения.

Для двух сложных монотонных функций аналогично методом ветвей и границ может быть решена задача нахождения точки, где значения этих функций совпадают. Если одна функция возрастающая, а другая – убывающая, то точка совпадения их значений может быть только одна, или такой точки может совсем не существовать. Если обе функции возрастающие, или обе убывающие, то точек совпадения может быть несколько, или такой точки может совсем не существовать. В любом случае надо находить такие две точки, в одной из которых первая функция меньше второй, а в другой наоборот – вторая функция меньше первой (брать в “вилку”). Брать в “вилку” функции можно, например, с помощью датчика случайных чисел. Если в “вилку” взять не удалось, то значит точки совпадения функций не существует. Если же удалось – то точка совпадения, хотя бы одна, существует и метод ветвей и границ ее определит. Если точек совпадения несколько – то будет найдена одна из них.

Возрастающая монотонная функция со знаком минус станет убывающей монотонной функцией. Отсюда разность между возрастающей и убывающей монотонными функциями будет возрастающей монотонной функцией, т.к. сумма двух возрастающих монотонных функций является возрастающей монотонной функцией. Аналогично убывающая монотонная функция со знаком минус станет возрастающей монотонной функцией. Отсюда разность между убывающей и возрастающей монотонными функциями будет убывающей монотонной функцией, т.к. сумма двух

убывающих монотонных функций является убывающей монотонной функцией.

Для случая разности разнотипных монотонных функций задача нахождения точки их совпадения сводится к нахождению обратного значения нуля (v=0). Однако практически получить разность двух сложных функций бывает затруднительно, а то и невозможно. Каждая из этих функций может быть задана отдельной процедурой, которые могут быть написаны разными программистами на различных алгоритмических языках, и т.д.

Для случая разности двух однотипных монотонных функций результат вычитания может и не оказаться монотонной функцией, но метод ветвей и границ в этом случае все равно применим. Программа razn получает точку совпадения с заданной точностью e двух возрастающих монотонных функций f (y)=sin(y)+y и u (y)=$\sqrt{y}+\sqrt[0.25]{y}$:

```
program razn;
const e=0.001;label 1,2;
var x,z,t:real;
function f(x:real):real;
begin
  f:=sin(x)+x;
end;
function u(x:real):real;
begin
  u:=sqrt(x)+sqrt(sqrt(x));
end;
begin
  2:x:=random(100)/10.0+0.0;z:=random(100)/10.0+0.01;
  if ((u(x)>f(x)) and (u(z)<f(z))) or ((u(x)<f(x)) and (u(z)>f(z))) then
    begin
      if u(x)<f(x) then
        begin
          t:=x; x:=z; z:=t;
        end;
      while abs(x-z)>e do
        begin
          t:=(z+x)/2; if abs(f(t)-u(t))<e then goto 1;
          if f(t)<u(t) then x:=t;if f(t)>u(t) then  z:=t;
        end;
      1:writeln('t=',t:10:3);
    end else
      goto 2;
end.
```

Программа razn по существу отличается от программы del только заменой переменной v на функцию u(x). Метод ветвей и границ можно аналогично применить и для нахождения определенного значения a, например, разности двух монотонных функций, для чего надо найти значение разности, большее a, и значание разности, меньшее a. Аналогично могут быть рассмотрены и сумма, и произведение, и произведение степеней двух монотонных функций. Главное – это взять в "вилку".

Метод ветвей и граних применим и для уровневых задач. Рассмотрим уровневую задачу: спонсор перечислил в пенсионный фонд (массив пенсий p) сумму s с условием: поднять минимальный уровень пенсии на этот месяц.

Метод решения этой задачи с помощью итераций может быть представлен и как метод ветвей и границ. Не всегда итерации можно представить как метод ветвей и границ. При моделировании сложных схемотехнических цифровых устройств с обратными связями итерации нужны для реализации процесса установления значений сигналов на выходах и их нельзя представить как метод ветвей и границ.

Упорядочим пенсии в массиве p1. Пусть имеем упорядоченный массив пенсий по возрастанию p1[10]:
800 870 900 1150 1200 1220 1300 1500 1550 1700. Сумма s=1000. Средняя пенсия y равна 1219. После распределения спонсорской помощи s среди низкооплачиваемых пенсионеров средняя пенсия y1 станет равной 1319. При этом несколько изменится структура упорядоченного по возрастанию массива пенсий, так что в начале массива p1 будет несколько одинаковых новых значений, а далее значения массива не изменятся. Если помощь была небольшой, т.е. меньше разности между первой и второй пенсиями, то в массиве p1 изменится (возрастет) только первое значение. Если помощь большая, такая что новая средняя пенсия больше последнего элемента в массиве p1, то все элементы нового массива p1 станут одинаковыми и равными этой новой средней пенсии. В общем случае спонсорская помощь распределится между несколькими новыми значениями массива p1. Новая средняя пенсия будет не ниже этих одинаковых значений, т.к. в вычислении нового среднего учитываются все пенсии, а некоторые (большие) не всегда будут увеличиваться. Получили, таким образом, верхнюю оценку увеличенных с помощью спонсора небольших пенсий – это новая средняя пенсия, для нашего примера это 1319. Отсюда вывод: пенсии, большие увеличенного среднего не будут увеличиваться. Для нашего примера это три самых больших пенсии – 1500, 1550 и 1700. Осталось в качестве кандидатов для увеличения семь первых в массиве p пенсий, повторим процесс уже для семи пенсий с той же суммой s. Средним значением для семи первых (самых маленьких) пенсий будет 1063, а после распределения спонсорской помощи среди этих семи

пенсионеров средняя пенсия станет равной 1206. Это новое значение средней пенсии позволит не рассматривать еще две пенсии – 1220 и 1300, так как они выше его. Повторяем опять процесс для пяти самих низкооплачиваемых пенсионеров. Средним значением для пяти первых пенсий будет 984, а после распределения спонсорской помощи среди этих пяти пенсионеров средняя пенсия станет равной 1184. Это новое значение средней пенсии позволит не рассматривать еще одна пенсию – 1200, так как она выше его. Повторяем опять процесс для четырех самих низкооплачиваемых пенсионеров. Средним значением для четырех первых пенсий будет 930, а после распределения спонсорской помощи среди этих четырех пенсионеров средняя пенсия станет равной 1180. Это новое значение средней пенсии и будет результатом, так как все четыре оставшиеся пенсии ниже его. Таким образом новый упорядоченный массив пенсий p1 после распределения спонсорской помощи s=1000 будет иметь вид:
1180 1180 1180 1180 1200 1220 1300 1500 1550 1700. Спонсорская сумма s распределилась следующим образом: 380 – первому пенсионеру, 310 – второму, 280 – третьему и 30 – четвертому. 380+310+280+30=1000=s. Все сошлось. Метод ветвей и границ в приложении к уровневым задачам дал верный результат. Программа sponsor2 решает уровневую задачу изложенным выше способом:

```
program sponsor2;
label 1;
const N=20;
var P:array[1..N] of integer;
   I, S, N1, N2 : integer;
   y, y1 : real;
begin
  randomize;
  for i:=1 to N do P[I]:=random(800)+700;
  readln(S); y:=0.0; for i:=1 to N do
  y:=y+P[I]; y:=(y+S)/N; N2:=n;
1: y1:=0.0; N1:=0; for i:=1 to N do
  if P[I] < y then
     begin
       N1:=N1+1; y1:=y1+P[I]
     end;
  y1:=(y1+S)/N1; if N1 < N2 then
     begin
       N2:=N1; y:=y1; goto 1
     end;
  writeln('namber   old      new');
  for I:=1 to N do if P[I]>y then
```

```
  writeln(I,'       ',P[I],'   ',P[I]) else
  writeln(i,'       ',P[I],'   ',y);
end.
```

В представленном уровневом алгоритме нет множества ветвей, но зато есть границы.В этой простой задаче существует одна ветвь, и по ней по полученной границе проводятся отсечения. Можно считать, что при поиске обратного значения монотонной функци ветвей было две – в одну сторону оси координат и в другую. Небольшое количество ветвей в этих алгоритмах объясняется просторой алгоритмов.

Можно решить уровневую задачу со спонсором методом ветвей и границ несколько иначе. В массиве R[N-1] для каждого **i** (0<**i**<N) R[**i**] равно сумме, необходимой для того, чтобы **i** самых низкооплачиваемых пенсионеров поднять до уровня P1[**i**+1]. Очевидно, что R - упорядоченный по возрастанию массив. Y - это уровень, на который сумма S даст возможность поднять минимальный уровень пенсии в этом месяце.

Алгоритм определяет положение суммы S относительно элементов массива R[N-1] методом последовательного деления отрезка пополам. S может оказаться больше или равной последнему элементу массива R и тогда пенсия у всех пенсионеров увеличится и станет равной P1[N]+(S-R[N-1])/N. Если S меньше первого элемента массива R, то пенсия у самого низкооплачиваемого пенсионера увеличится и станет равной P1[1]+S. Причем (P1[1]+S)<P1[2] в этом случае. И, наконец, в общем случае S окажется между R[i] и R[i+1], или совпадет с R[i], и тогда пенсия увеличится у (i+1) (i - при S=R[i]) самых низкооплачиваемых пенсионеров. Все они получат в этом случае пенсию, равную P1[i+1]+(S-R[i])/(i+1).

```
program sponsor;
const n=20;
var p,p1:array[1..n] of integer;
r:array[1..n] of integer;
k,i,j,s:integer;y:real;
begin
  randomize;
  for i:=1 to n do
    begin
      p[i]:=random(300)+80; p1[i]:=p[i]
    end;
    write('s=');
  readln(s);for i:=1 to n-1 do for j:=1 to n-i do
  if p1[j]>p1[j+1] then
    begin
      k:=p1[j];p1[j]:=p1[j+1];p1[j+1]:=k
    end;
  for i:= 1 to n do write(p1[i]:4); writeln;
```

```
  r[1]:=p1[2]-p1[1];for i:=2 to n-1 do r[i]:=(p1[i+1]-p1[i])*i+r[i-1];
  if s<r[1] then y:=p1[1]+s;if s>=r[n-1] then y:=p1[n]+(s-r[n-1])/n;
  if (s>=r[1]) and (s<r[n-1]) then
    begin
      i:=n div 2;J:=(i+1) div 2; while j>0 do if (s>=r[i]) and (s<r[i+1]) then
        begin
          j:=0;y:=p1[i+1]+(s-r[i])/(i+1)
        end
      else
      begin
        if r[i]>s then
          begin
            i:=i-j;if i<1 then i:=1
          end
        else
          begin
            i:=i+j;if i>(n-2) then i:=n-2
          end;
          j:=(j+1) div 2
      end
    end;
  writeln('namber old    new');
  for i:=1 to n do if p[i]>y then
  writeln(i,'      ',p[i],'   ',p[i]) else
  writeln(i,'      ',p[i],'   ',y)
end
```

Метод деления отрезка пополам применяется чаще экстраполирующей функции. Он может быть применен и для уровневых задач. Та же задача со спонсорской помощью пенсионерам может быть решена и методом деления отрезка пополам.

Введем целочисленную функцию f (i)=((∑ $p1[j]$ +s)/i, где i- номер в массиве пенсий. Чтобы решить задачу надо найти i, такое что p1[i]<f (i)≤ p1[i+1]. Если для некоторого k f (k)>p1[k+1], то k надо увеличивать. Если для некоторого k f (k))≤p1[k], то k надо уменьшать. Программа spond находит и выводит на дисплей новый упорядоченный массив пенсий p1 после распределения спонсорской помощи s:

```
program spond;
label 1;
var x,z,a,b,s:integer;
p1: array [1..10] of integer;
function f(x:integer):integer;
var i,j:integer;
```

```
begin
  j:=p1[1];for i:=2 to x do j:=j+p1[i];f:=(j+s) div x;
end;
begin
  for x:=1 to 10 do read(p1[x]);readln(s);
  if (p1[10]>=f(10)) and (p1[2]<f(1))  then
    begin
      x:=1; z:=10;
      a:=(z+x) div 2;b:=(abs(x-z)) div 2;
      while b>1 do
        begin
          b:=b div 2;
          if (f(a)<=p1[a+1]) and (p1[a]<f(a)) then goto 1;
          if f(a)>p1[a+1] then  a:=a+b;
          if f(a)<=p1[a] then  a:=a-b;
          if (b=1)  then goto 1;
        end;
      1:writeln('a=',a:10,f(a):10);
      for x:=1 to 10 do if x<=a then write(f(a):5) else write (p1[x]:5);
      writeln;
    end else
      writeln('granici');
end.
```

Работа программы spond была проверена на вышеизложенном и решенном примере и был получен правильный результат. Программа spond получена из программы dei заменой сравнения функции f (t)<v на условие p1[a]<f(a)≤p1[a+1]. Программу spond можно сделать более эффективной, если для каждого нового значения a не считать заново всю сумму пенсий, а только ту ее часть, которая отличается от предыдущей суммы при вычислении значения
функции f.

Как видно по уровневым задачам, метод ветвей и границ может быть применен к одной и той же задаче разными способами. Метод ветвей и границ применяется и при решении ресурсных задач [2]. Пусть, например, имеется ресурс, который необходим при выполнении четырех работ (башенный кран). Потребность в других видах ресурсов не учитывается, считается, что их достаточно. Сеть комплекса работ изображена на рис. 1.а., где жирным шрифтом выделена ресурсная работа (средняя). Все четыре

I.	10 ~ **5** ~ 3	10 ~ ~	10 ~ ~	~ ~ 3	~ ~	~ ~
II. .	12 ~ **7** ~ 10	~ ~	~ ~	12 ~ ~	12 ~ ~	12 ~ ~
III.	28 ~ **4** ~ 6	~ ~	~ ~ 6	~ ~	~ ~	~ ~ 6
IV.	23 ~ **7** ~ 3	~ ~ 3	~ ~	~ ~	~ ~ 3	~ ~

а.) б.) в.) г.) д.) е.)

Рис.1. Сеть комплекса работ

работы состоят из трех этапов и у всех средний этап – ресурсный. Начинаются все работы одновременно. Цифры изображают продолжительность этапов. Требуется указать такой порядок выполнения ресурсных работ, чтобы время выполнения комплекса из четырех работ было минимальным.

Всего имеется 24 варианта последовательности выполнения ресурсных работ (4!=24). Метод ветвей и границ можно применить и при прямом переборе всех 24-х вариантов, когда каждый последующий вариант считается до тех пор, пока он меньше уже достигнутого минимума. Первый вариант, естественно, считается полностью. Так как каждый вариант состоит из последовательного сложения чисел, такое отсечение возможно. Сложение проводится до тех пор, пока сумма меньше достигнутого минимума, в противном случае происходит переход к следующему варианту.

Можно применить метод ветвей и границ к этой задаче и более эффективно. Общее время выполнения всех ресурсных этапов 23 = 5 + 7 + 4 + 7 (часа, минуты, дня – это безразлично). Все 24 варианта содержат это время. Отличаются варианты друг от друга всегда началом или концом, а то и началом и концом, и, возможно, временами задержек начал выполнения ресурсных этапов 2-й, 3-й и 4-й работ в каждом варианте. Задержка происходит тогда, когда ресурс свободен, а время использовать его в данной работе еще не пришло. Начало варианта – это первый этап первой работе в варианте, конец варианта – это третий этап четвертой работы в варианте. Сначала рассмотрим те работы, у которых сумма начала и конца минимальна. Тут будем считать, что задержек выполнения ресурсных этапов 2-й, 3-й и 4-й работ в варианте нет. Это два варианта, ресурсный этап I работы идет в нем первым, а ресурсный этап IV работы – последним. Сумма начала и конца у этих двух вариантов минимальна – 13 (рис.1.б.). В середине этих двух вариантов выполняются ресурсные этапы II и III работ. Теперь учтем задержеки выполнения ресурсных этапов 2-й, 3-й и 4-й работ в вариантах. Если порядок выполнения ресурсных этапов работ – I, II, III, IV – то к 13 добавится еще 6. Ресурсный (второй) этап работы II начнется в 15, когда кончится ресурсный этап работы I, а мог бы начаться и в 12, тут задержки ресурса нет. Зато ресурсный этап работы III начнется в 28, а ресурсный этап работы II кончится в 22, тут задержка ресурса 6. Ресурсный этап работы IV начнется в 32, когда кончится ресурсный этап работы III, а мог бы начаться и в 23, тут задержки ресурса нет. Если порядок выполнения ресурсных этапов работ – I, III, II, IV – то к 13 добавится еще 13. Пока имеем минимальный вариант из двух (I, II, III, IV) с временем выполнения комплекса работ 19 (+23, естественно).

Теперь необходимо рассмотреть еще варианты, у которых сумма начала и конца <19 , они могут дать время выполнения комплекса работ меньше 19. Таких вариантов 8: с суммой 16=10+6 (I - начало, III – конец, рис. 1.в.), с суммой 15=12+3 (II - начало, I – конец, рис. 1.г.), с суммой 15=12+3 (II - начало, IV – конец, рис. 1.д.), с суммой 18=12+6 (II - начало, III – конец, рис. 1.е.). Оставшиеся 14 вариантов уже имеют время выполнения комплекса работ > 19, так как у них сумма начала и конца уже больше 19. Призошло отсечение 14 вариантов.

Рассмотрим вариант с суммой начала и конца, равной 16 - I, II, IV , III – у него время выполнения комплекса работ 17=16+1, и это отсекает еще два варианта с суммой начала и конца, равной 18. Другой вариант с суммой 16 - I, IV , II, III – дает время 24=16+8. Четыре варианта с суммой 15 дают, соответственно, 23=15+9, 19=15+4, 24=15+9, 19+15+4, что больше минимального. Таким образом получили результат, рассмотрев 8 вариантов из 24-х. В [2] приведен несколько иной способ решения этой же задачи методом ветвей и границ. Там 24 варианта делились на 4 ветви, у каждой ветви первым из ресурсных этапов выполняется ресурсный этап одной и той же работы. Каждая из этих 4-х ветвей имеет, в свою очередь, 3 ветви, ветвление тут происходит по номеру ресурсного этапа, выполняемым вторым. Каждая из этих 3-х ветвей имеет 2 ветви, ветвление тут происходит по номеру ресурсного этапа, выполняемым третьим. У этого дерева ветвления с одним корнем 24 листа. Отсечение проводилось по оценке снизу, а именно по сумме начала и конца, описанной выше. Самое сложное в методе ветвей и границ с использованием оценок (сверху при поиске максимума и снизу при поиске минимума) это найти оценку для дальнейшего отсечения, а дерево ветвления придумать легче.

Ресурсные задачи решаются не только при планировании работ, но и при составлении расписаний. Ресурсным можно считать проведение лабораторных работ в дисплейных классах, например. Читается, скажем, 4 курса, каждый с проведением лабораторных работ в одном и том же дисплейном классе. До начала лабораторных работ лектор должен начитать материал, и после проведения лабораторных работ нужно время на их анализ и оценку.

Методом ветвей и границ можно решить и задачу построения минимального пути из i-й вершины ненагруженного орграфа в j-ю по матрице смежности орграфа. Можно строить все дерево путей из i-й вершины и отсекать те его ветви, где какая-то вершина встретится дважды. Из корня этого дерева (i-й вершины) на первом этапе строятся дуги (пути длиной единица), исходящие из него. На втором этапе из вершин на конце дуг опять строятся исходящие из них дуги, и так до тех пор, пока не встретится вершина j, или длина от корня до листа не станет равной n-1 ((n-1)-й этап), где n – число вершин в орграфе. Но можно производить и более сложное и эффективное отсечение, исходя из того, что минимальный

путь сам состоит из минимальных путей. На каждом этапе отсекать будем не только пути, получившие на этом этапе одну из уже пройденных этим путем вершин, но и пути, получившие на этом этапе одну из уже пройденных в том числе и другими путями на предыдущих этапах вершин. Другими словами, минимальный путь должен получать на каждом этапе новую, еще не пройденную никакими путями на предыдущих этапах вершину, в противном случае это будет не минимальный путь. Программа wayb находит минимальный путь из i-й вершины в j-ю методом ветвей и границ:

```
program wayb;
const n=20;
var G:array[1..n,1..n] of integer;
q,p:array[1..n] of integer;
i,j,k,l,m:integer;
begin
  for i:=1 to n do for j:=1 to n do g[i,j]:=random (2);
  readln(i,j); if i<>j then
    begin
      for k:=1 to n do q[k]:=0; q[1]:=1;
      for k:=1 to n do if g[i,k]=1 then q[k]:=2;
      for m:=2 to (n-1) do for k:=1 to n do
      if q[k]=m then for l:=1 to n do
      if (g[k,l]=1) and (q[l]=0) then q[l]:=m+1;
      if q[j]>0 then
        begin
          p[1]:=i; p[q[j]]:=j; for k:=q[j]-1 downto 2 do
          for l:=1 to n do if q[l]=k then
          if g[l,p[k+1]]=1 then p[k]:=l;
          write('lingth=',(q[j]-1):6,' way=');
          for k:=1 to q[j] do write(p[k]:3); writeln;
        end
   end else writeln('no way');
end.
```

Здесь g – матрица смежности орграфа, p – массив минимального пути из i в j, q – вспомогательный массив, сохраняющий для каждой вершины номер этапа ветвления, на котором она появилась (была пройдена) впервые. В начале программы массив q обнуляется, и только q[i] приравнивается единице. На первом этапе если существует дуга из i в k, то q[k] становится равным 2, и т.д. Если на (n-1)-м этапе q[j] останется нулем, то пути из i в j не существует. Если q[j] не равна нулю, то существует минимальный путь длиной (q[j]-1) и строится он, начиная с конца. P[1]=i, p[q[j]]=j а затем находится p[q[j]-1]. Для этого в массиве q определяются элементы, равные q[j]-1, и из них выбирается такой, чтобы вершина с

аналогичным номером имела дугу в вершину j. Если таких элементов (вершин) несколько, то выбирается любая. Если строить минимальный путь в другом направлении, т.е. из j в i-ю но с учетом обратного направления дуг, то массив q будет другим, а массив p будет строится, начиная сначала.

Бибилиографический список

1. Нефедов В.Н., Осипова В.А. Курс дискретной математики – М.: Изд-во МАИ, 1992
2. Кузнецов О.П., Адельсон-Вельский Г.М. Дискретная математика для инженера – М.: Энергоатомиздат, 1988
3. Уилсон Р. Введение в теорию графов – М.: Мир, 1977
4. Кофман А. Введение в прикладную комбинаторику – М.: Наука, Гл. ред. физ.-мат. лит., 1975
5. Липский В. Комбинаторика для программистов – М.: Мир, 1988
6. Форсайт Р. Паскаль для всех – М.: Машиностроение, 1986
7. Грэхем Р. Практический курс языка Паскаль для микроЭВМ – М.: Радио и связь, 1986