

работы, так и по требуемым ресурсам. Автоматы же всегда затрачивают на работу время, пропорциональное длине входной ленты, независимо от сложности своего устройства. Поэтому единственным параметром, отличающим более предпочтительный автомат от менее предпочтительного, является количество состояний. Понятно, что автомат с меньшим числом состояний требует меньшего объема памяти, да и устроен он обычно проще.

По какой причине некоторый рассматриваемый нами автомат может оказаться неоптимальным (то есть содержащим больше состояний, чем необходимо для распознавания интересующего языка)? Во-первых, разрабатывая автомат с помощью карандаша и бумаги, вы естественным образом думаете о распознаваемом языке, а не о минимизации состояний. Во-вторых, нередки случаи (и мы с ними познакомимся), когда конечный автомат создается, простите за тавтологию, автоматически некоторой компьютерной программой. При этом полученное устройство почти всегда будет очень далеким от оптимального¹⁶.

К счастью, существует алгоритм, позволяющий получить оптимальный автомат, эквивалентный данному (то есть выполняющий ту же самую работу, но имеющий при этом минимально возможное количество состояний). Рассмотрим этот полезный алгоритм подробно.

Прежде всего нам потребуется понятие *эквивалентности состояний*. Два состояния называются эквивалентными (не вдаваясь в математические подробности), если дальнейшее поведение автомата в первом состоянии совпадает с дальнейшим поведением автомата во втором состоянии. Сказанное иллюстрирует рис. 2.5.

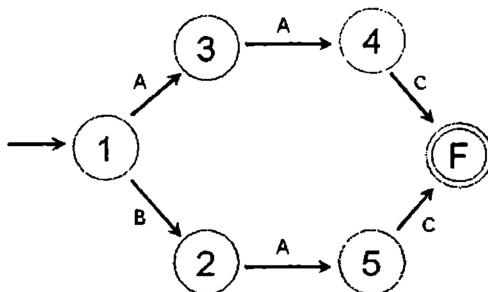


Рис. 2.5. Эквивалентность состояний

¹⁶ Заметьте, здесь речь идет не о машинной генерации текста программы, реализующего уже разработанный автомат, а о проектировании компьютером самой конфигурации автомата.

Состояния 2 и 3 эквивалентны. Не имеет значения, в каком из них находится автомат в данный момент времени — любые входные данные ленты будут обработаны одинаковым образом.

Цель минимизации состоит в объединении эквивалентных состояний в единственное новое (конечно, при этом придется подправить и таблицу переходов). На псевдокоде алгоритм минимизации выглядит так:

удалить недостижимые состояния и связанные с ними правила;

создать таблицу всевозможных пар состояний вида (p, q);

ОТМЕТИТЬ те пары, где одно из состояний является допускающим, а другое — нет;

DO

```
    found = false;
    ЕСЛИ существует неотмеченная пара (p, q), такая,
    что для некоторого элемента входного алфавита a
    пара ( $\delta(p, a)$ ,  $\delta(q, a)$ ) отмечена
        ОТМЕТИТЬ пару (p, q);
        found = true;
    WHILE found // то есть пока изменения происходят
```

заменить каждое множество эквивалентных друг другу состояний на единственное новое; соответствующим образом изменить таблицу переходов;

качестве комментариев к этому алгоритму нужно сказать следующее:

- ♦ В автомате могут существовать так называемые *недостижимые* состояния, то есть состояния, которые не могут быть достигнуты никакой последовательностью символов входной ленты (см. рис. 2.6). Разумеется, никто не будет вносить в автомат подобную бессмыслицу, рисуя его вручную. Но при машинной генерации автомата недостижимые состояния вполне могут появиться. Алгоритм минимизации, очевидно, должен уничтожать как все недостижимые состояния, так и правила, с ними связанные.
- ♦ Какое именно действие скрывается за словом «*отметить*»? Где-то в программе каждой паре состояний должно сопоставляться некоторое булево значение (true = отмечена / false = не отмечена). «*Отметить*» означает установить это значение в true.
- ♦ Алгоритм отмечает те или иные пары в соответствии со вполне определенным критерием: элементы отмечаемой пары не должны быть эквивалентными состояниями. Итоговая цель цикла DO... WHILE — отметить все такие пары. После этого должно быть понят-

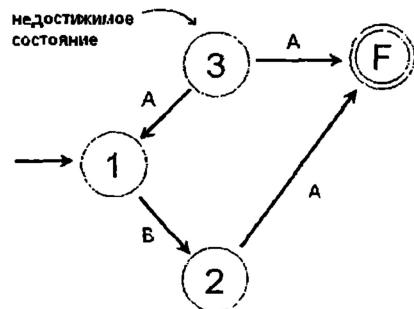


Рис. 2.6. Пример недостижимого состояния в конечном автомате

но, как именно работает алгоритм. Поскольку никакое допускающее состояние не может быть эквивалентно какому-либо обычному состоянию, мы можем сразу же отметить пары, состоящие из обычного и допускающего состояний. Далее, предположим, что в соответствии с таблицей правил автомат переходит из некоторого состояния p в состояние p' по входному символу a (то есть существует правило $(p, a) \rightarrow p'$). Допустим, существует также аналогичное правило для состояния q : $(q, a) \rightarrow q'$. Тогда если состояния p' и q' не являются эквивалентными, состояния p и q также заведомо неэквивалентны. Так как записи $\delta(p, a)$ и $\delta(q, a)$ обозначают правые части правил $(p, a) \rightarrow p'$ и $(q, a) \rightarrow q'$, условный оператор, приведенный в псевдокоде, претворяет эту идею в жизнь.

- Предположим, алгоритм нашел некоторую неотмеченную пару состояний (p, q) . Теперь его задача — перебирать все возможные символы входной ленты в поисках отмеченной пары $(\delta(p, a), \delta(q, a))$. А что будет, если для текущего рассматриваемого символа соответствующего правила не предусмотрено? Выше мы уже обсуждали подобную ситуацию. Напомню, реализация автомата на уровне программы — нечастый случай, когда отсутствие того или иного правила довольно легко сходит с рук (можно просто сообщить об ошибке). Алгоритмы, работающие с автоматами, нередко предполагают, что правило вида $(p, a) \rightarrow q$ существует для любых допустимых p и a . На практике это несколько (хотя и отнюдь не фатально) усложняет жизнь. Следует лишь ввести новое недопускающее состояние в автомат и направить в него все до этого момента не предусмотренные переходы. Предположим, что допустимыми символами ленты для автомата с рис. 2.5 являются латинские буквы A, B и C. Тогда перед минимизацией его придется модифицировать, как показано на рис. 2.7.
- После выполнения цикла DO...WHILE в нашем распоряжении оказывается список пар эквивалентных друг другу состояний. Но как заменить каждый класс эквивалентности (то есть множество эквивалентных друг другу состояний) единственным состоянием? Если

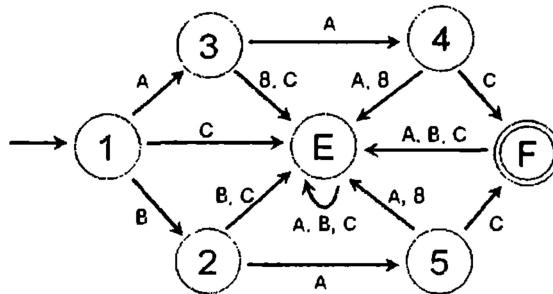


Рис. 2.7. Автомат с полным набором правил

состояния обозначаются целыми числами (предположим это), выявить классы эквивалентности поможет следующий код:

```

ЦИКЛ по всем состояниям q
  e_class[q] = q;
ЦИКЛ по всем неотмеченным парам (p, q)
  ЦИКЛ по всем состояниям s
    ЕСЛИ e_class[s] == p, ТО e_class[s] = q
  
```

Изначально каждое состояние представляет свой собственный класс эквивалентности ($e_class[q] = q$). Затем, если найдена какая-либо пара эквивалентных состояний (p, q), все представители класса состояния p «переписываются» в класс состояния q . В итоге для любого состояния соответствующее значение массива e_class будет равно номеру класса эквивалентности, к которому это состояние относится.

К сожалению, алгоритмы теории автоматов — не самое увлекательное чтение. Однако пройти мимо них просто нельзя; думаю, никого не надо убеждать в том, что, к примеру, минимизация автомата — вещь на редкость полезная.

Поскольку эта книга ориентирована скорее на практиков, мне хочется довести большинство важных алгоритмов до реально работающего кода. С другой стороны, загромождать книгу листингами нежелательно, поэтому реализация будет несколько ограниченной.

Во-первых, мы будем предполагать, что все состояния автомата обозначаются целыми, подряд идущими числами, начиная с единицы. Во-вторых, допустимыми элементами входной ленты считаются только стандартные символы из таблицы ASCII. В-третьих, в целях экономии места мы не будем заниматься удалением недостижимых состояний. В общих чертах удаление недостижимых состояний производится достаточно просто. Для этого надо завести очередь и сразу же добавить в нее стартовое состояние (которое, естественно, является достижимым). Далее выполняется алгоритм:

```
ПОКА очередь непуста
    извлечь очередной элемент S;
    пометить его как достижимый;
    внести в очередь все состояния, в которые существует
    прямой переход из S;
```

После выполнения цикла все недостижимые состояния останутся непомечеными; их и все связанные с ними правила следует удалить.

Программа минимизации читает входные данные (конфигурацию автомата) с консоли в таком формате:

```
количество состояний
список символов пенты
список допускающих состояний
стартовое состояние
правило1
правило2
...
правилоН
```

Таким образом, автомату, изображенному на рис. 2.7, соответствует ввод

```
7
A B C
6
1
1 A 3
1 B 2
1 C 7
2 A 5
2 B 7
2 C 7
3 A 4
3 B 7
3 C 7
4 A 7
4 B 7
4 C 6
5 A 7
```

```
5 B 7
5 C 6
6 A 7
6 B 7
6 C 7
7 A 7
7 B 7
7 C 7
```

Поскольку все состояния должны быть обозначены числами, я заменил
состояние F номером 6, а состояние E — номером 7.

Полный текст программы минимизации автомата приведен в листинге 2.2.

Листинг 2.2. Минимизация детерминированного конечного автомата

```
struct Pair      // пара состояний (p, q)
{
    public int p, q;

    public Pair(int thep, int theq) {p = thep; q = theq; }

    struct Leftside // левая часть правила - (состояние, символ)
    {
        public int state;
        public char symbol;

        public Leftside(int st, char sym) {state = st; symbol = sym;}
    }

    static void Main(string[] args)
    {
        // количество состояний
        int Nstates = Convert.ToInt32(Console.In.ReadLine());

        string[] symstr = (Console.In.ReadLine()).Split(' ');
        char[] symbols = new char[symstr.Length];
        for(int i = 0; i < symstr.Length; i++)
        // символы входной ленты
            symbols[i] = Convert.ToChar(symstr[i]);

        // список допускающих состояний
        string[] Flist = (Console.In.ReadLine()).Split(' ');
```

```
bool[] favorable = new bool[Nstates + 1];
for(int i = 1; i <= Nstates; i++)
    favorable[i] = false;
foreach(string id in Flist)
    favorable[Convert.ToInt32(id)] = true;

// стартовое состояние
int StartingState = Convert.ToInt32(Console.In.ReadLine());

// правила хранятся как (левая часть, состояние)
Hashtable rules = new Hashtable();
Hashtable pairs = new Hashtable();
                                // список пар состояний
string s;
while((s = Console.In.ReadLine()) != "")
    // считывание правил
{
    string[] tr = s.Split(' ');
    rules.Add(new Leftside(Convert.ToInt32(tr[0]),
        Convert.ToChar(tr[1])),
        Convert.ToInt32(tr[2]));
}

// ОТМЕТИТЬ те пары, где одно из состояний является
// допускающим, а другое - нет
for(int i = 1; i <= Nstates; i++)
    for(int j = 1; j <= Nstates; j++)
    {
        bool Marked = ((favorable[i] == true &&
            favorable[j] == false) ||
            (favorable[i] == false &&
            favorable[j] == true));
        pairs.Add(new Pair(i, j), Marked);
    }

bool found;
do
{
    found = false;
    foreach(IDictionaryEnumerator e = pairs.Get.GetEnumerator();
        e.MoveNext(); )
    {
        if((bool)e.Value == false)
            // найдена неотмеченная пара (p, q)
```

```
foreach(char a in symbols)
{
    int d1 = (int)rules[new Leftside(((Pair)e.Key).p, a)];
    int d2 = (int)rules[new Leftside(((Pair)e.Key).q, a)];

    // найдена отмеченная пара (d(p, a), d(q, a))
    if((bool)pairs[new Pair(d1, d2)] == true)
    {
        pairs[new Pair(((Pair)e.Key).p,
                        ((Pair)e.Key).q)] = true;
        found = true;
        goto exit;
    }
}
exit: ;
}

while(found);

// выявление классов эквивалентности
int[] e_class = new int[Nstates + 1];
for(int i = 1; i <= Nstates; i++)
    e_class[i] = i;

for(IDictionaryEnumerator e = pairs.Get Enumerator();
    e.MoveNext();)
    if((bool)e.Value == false)
        for(int i = 1; i <= Nstates; i++)
            if(e_class[i] == ((Pair)e.Key).p)
                e_class[i] = ((Pair)e.Key).q;

bool[] StatePrinted = new bool[Nstates + 1];
for(int i = 1; i <= Nstates; i++)
    StatePrinted[i] = false;

Console.Write("Состояния: ");      // вывод состояний
for(int state = 1; state <= Nstates; state++)
    if(!StatePrinted[e_class[state]])
        // если состояние еще не
        // выведено на печать
        {
            Console.WriteLine("{0} ", e_class[state]);
            StatePrinted[e_class[state]] = true;
        }
Console.WriteLine();                  // вывод особых состояний
```

```
Console.WriteLine("Стартовое состояние: {0}\n" +
                  "Допускающее состояние: {1}",
                  e_class[StartingState],
                  e_class[Convert.ToInt32(Flist[0])]);
Console.WriteLine("Правила:");
Hashtable RulePrinted = new Hashtable();
for (IDictionaryEnumerator e = rules.GetInternalEnumerator();
     e.MoveNext();)
{
    string rule = "(" + e_class[((Leftside)e.Key).state].ToString() +
                  ", " + ((Leftside)e.Key).symbol.ToString() +
                  ") -> " + e_class[(int)e.Value].ToString();
    if (RulePrinted.ContainsKey(rule) == false)
        // если правило еще не
        // выведено на печать
        {
            Console.WriteLine(rule);
            RulePrinted[rule] = true;
        }
}
}
```

Результирующий автомат для приведенного примера, построенный программой минимизации, содержит всего пять состояний:

```
Состояния: 1 3 5 6 7
Стартовое состояние: 1
Допускающее состояние: 6
Правила:
(7, C) -> 7
(7, B) -> 7
(7, A) -> 7
(6, C) -> 7
(5, C) -> 6
(6, B) -> 7
(5, B) -> 7
(6, A) -> 7
(5, A) -> 7
(3, C) -> 7
(3, B) -> 7
(3, A) -> 5
(1, C) -> 7
(1, B) -> 3
(1, A) -> 3
```

Граф этого автомата приведен на рис. 2.8.