Статья опубликована в журнале «Компьютерные инструменты в образовании». 2006. № 4, с. 26 – 35.

Оршанский Сергей Александрович
Шалыто Анатолий Абрамович

ПРИМЕНЕНИЕ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ ПРИ РЕШЕНИИ ЗАДАЧ НА КОНЕЧНЫХ АВТОМАТАХ

ВВЕДЕНИЕ

В последнее время в программировании все чаще используются конечные автоматы [1–7]. Поэтому задача более глубокого исследования их свойств остается актуальной.

Эти исследования осуществляются с применением различных математических методов [1]. При этом представляется интересным использование для этой цели динамического программирования [8–10].

Цель настоящей работы – продемонстрировать эффективность применения динамического программирования для решения одной олимпиадной задачи на конечных автоматах, которая имеет название «Непоглощающий конечный автомат» [11].

1. КОНЕЧНЫЕ АВТОМАТЫ

Конечный автомат состоит из множества состояний и «управления», которое переводит автомат из одного состояния в другое в зависимости от получаемых извне «входных данных». Автоматы разделяются на два класса в зависимости от типа управления. Оно может быть «детерминированным» – автомат в каждый момент времени находится только в одном состоянии, и «недетерминированным» – автомат может одновременно находиться в нескольких состояниях.

Приведем классическое определение детерминированного конечного автомата (ДКА) [1]. ДКА – это упорядоченный набор $<\Sigma, U, s, T, \varphi>$, где Σ – конечное множество, называемое входным алфавитом, U – конечное множество состояний, s из U – начальное состояние, T – множество терминальных состояний из U, и, наконец, $\varphi: U \times \Sigma \rightarrow U$ – функция переходов.

Входом автомата является строка α над алфавитом Σ. Первоначально автомат находится в состоянии s. На очередном шаге он переходит из текущего состояния u в состояние $\varphi(u, c)$, где c – первый символ входной строки. После этого первый символ входной строки удаляется, и шаг повторяется. Если к моменту исчерпания входной строки автомат находится в терминальном состоянии, то говорят, что он допускает исходную строку α, а в противном случае – отвергает ее.

2. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Динамическое программирование позволяет решать задачи, разбивая каждую из них на подзадачи, аналогичные исходной задаче, и объединяя в дальнейшем решения этих подзадач. Подзадачи, в свою очередь, разбиваются на «подподзадачи» и т.д. Ключевым условием для применения динамического программирования является **наличие перекрывающихся подзадач**. Суммарное число всех встречающихся подзадач должно быть относительно невелико – например, должно полиномиально зависеть от размера входных данных.

Алгоритмы, основанные на динамическом программировании, используют перекрытие подзадач следующим образом: каждая подзадача решается один раз, и ответ заносится в специальную таблицу или запоминается иным способом. Когда эта подзадача встречается снова, программа не будет тратить время на ее повторное решение, а использует готовый ответ. Поэтому алгоритмы, основанные на динамическом программировании, оказываются существенно эффективнее алгоритмов, основанных на методе «Разделяй и властвуй» [12].

Обычно задача, решаемая с помощью динамического программирования, представляется как вычисление некоторой функции. В этом случае подзадачей, как правило, является поиск значений той же функции с меньшими значениями аргументов.

Как строится алгоритм, основанный на динамическом программировании? Основной шаг заключается в нахождении рекуррентных соотношений, связывающих значение функции для задачи со значениями функции для различных подзадач. Тогда, зная ответ для базовых случаев, можно вычислить ответ для интересующей задачи, предварительно найдя ответы для подзадач.

Примеры классических задач динамического программирования: перемножение матриц с минимальным количеством умножений, оптимальная триангуляция выпуклого многоугольника, наибольшая возрастающая подпоследовательность.

Систематическое изучение динамического программирования было начато Р.Беллманом в 1955 г. [8], хотя некоторые приемы такого рода были известны и ранее. О динамическом программировании много написано в работах [9, 10].

3. ЗАДАЧА «НЕПОГЛОЩАЮЩИЙ ДЕТЕРМИНИРОВАННЫЙ КОНЕЧНЫЙ АВТОМАТ»

Задача «Непоглощающий детерминированный конечный автомат».
(Non Absorbing Deterministic Finite Automaton (DFA)).

Автор задачи: Андрей Станкевич.
Источник: летние сборы команд-участниц чемпионата мира ACM по программированию. Петрозаводск, 2003.
Расположение: задача № 201 в архиве олимпиадных задач Саратовского государственного университета на сайте http://acm.sgu.ru

Условие задачи сформулировано на английском языке. Перевод на русский выполнен авторами настоящей работы. Аббревиатура «DFA» переводится как «ДКА» – детерминированный конечный автомат.

Ограничения и требования:

- ограничение по времени: 2с;
- ограничение по памяти: 64 Мб;
- входные данные: стандартный ввод;
- выходные данные: стандартный вывод.

В теории компиляторов и языков широко используются детерминированные конечные автоматы, определение которых приведено в разд. 1.

Иногда удобно расширять это определение понятием непоглощающих ребер. Для этого в дополнение к функции переходов φ также вводится функция поглощения $\chi: U\times\Sigma \rightarrow \{0,1\}$. Тогда при совершении перехода из состояния u по символу c, первый символ из входной строки удаляется, только если $\chi(u,c) = 0$. Если же $\chi(u,c) = 1$, то входная строка остается без изменений, и следующий переход производится из нового состояния, но по тому же символу c. В первом случае говорят, что произошел переход по поглощающему ребру, а во втором – по непоглощающему.

По определению, такой автомат допускает строку α, если после некоторого числа шагов строка оказывается пустой, а автомат при этом находится в терминальном состоянии.

Задача: найти количество строк данной длины N, допускаемых заданным ДКА с непоглощающими ребрами.

Формат входных данных

Первая строка входного файла содержит Σ – подмножество английского алфавита (несколько маленьких латинских букв).

Вторая строка содержит $K = |U|$ – количество состояний автомата $(1 \le K \le 1000)$. Состояния нумеруются от 1 до K.

Третья строка содержит S $(1 \le S \le K)$ – номер начального состояния и $L = |T|$ – количество терминальных состояний, а затем L различных целых чисел от 1 до K каждое – номера терминальных состояний.

Следующие K строк содержат по $|\Sigma|$ целых чисел каждая и определяют функцию φ.

Следующие за ними K строк определяют функцию χ тем же способом. Последняя строка входного файла содержит N $(1 \le N \le 60)$.

Формат выходных данных

Единственное число – количество строк длины N над алфавитом Σ, допускаемых данным ДКА.

Пример исходных данных для рассматриваемой задачи приведен в таблице.

Пример входных данных	Пример выходных данных
ab 2 1 1 2 2 1 1 2 0 1 0 0 3	2

Описанный в примере автомат допускает две строки длины три: “aaa” и “abb”.

4. РЕШЕНИЕ

4.1. Анализ задачи

Имеется ДКА с непоглощающими ребрами – ребрами, при совершении перехода по которым не удаляется первый символ входной строки. В этом случае следующий переход происходит из нового состояния, но по тому же символу. Это будет продолжаться до тех пор, пока не произойдет переход по поглощающему ребру, либо пока одно и то же состояние не повторится дважды, и процесс не зациклится. Важно, анализируя условие задачи, не упустить из виду возможность существования циклов из непоглощающих ребер.

ДКА с непоглощающими ребрами может быть сведен к ДКА без них. Рассмотрим произвольное непоглощающее ребро – пару (u,c): $\chi(u,c)=1$. В зависимости от текущего состояния u и первого символа входной строки c автомат либо войдет в цикл, либо рано или поздно пройдет по поглощающему ребру. Перед этим автомат, возможно, пройдет по цепочке непоглощающих ребер. Этого можно избежать, изменив функцию переходов соответствующим образом. Для того чтобы учесть возможность вхождения в цикл, добавим фиктивное состояние «Недопуск», не являющееся терминальным. Будем считать, что все ребра из состояния «Недопуск» ведут в него же. Поскольку после попадания в цикл из непоглощающих ребер очередной символ входной строки никогда не будет удален, то автомат по определению не допускает исходную строку. Положим $\varphi(u,c)$="Недопуск". При этом вместо перехода по непоглощающему ребру, приводящему в цикл, автомат совершит переход по поглощающему ребру в состояние «Недопуск», и исходная строка не будет допущена.

Обратимся к входным данным из примера. ДКА, описанный в нем, имеет два состояния. Первое состояние – начальное, а второе – терминальное. Необходимо посчитать количество строк из трех символов, допускаемых автоматом.

На рис.1 изображен ДКА из примера. В нем непоглощающее ребро выделено пунктиром.

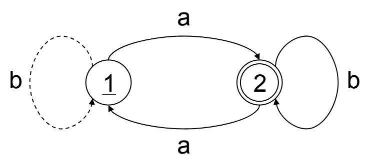

Рис.1. ДКА из примера с непоглощающими ребрами

Преобразуем автомат для того, чтобы избавиться от непоглощающих ребер. На рис. 2 изображен преобразованный ДКА, эквивалентный исходному. Состояние «недопуск» помечено буквой «Н».

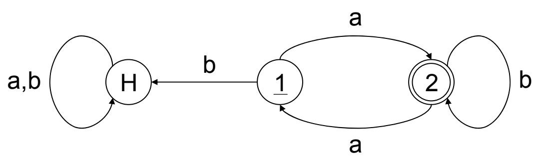

Рис. 2. ДКА из примера после удаления непоглощающих ребер

Найдем количество строк длины три, допускаемых рассматриваемым автоматом. Заметим, что в первом (начальном) состоянии на вход не должен подаваться символ “*b*”, поскольку это приведет к зацикливанию. Соответственно, первым символом должен быть “*a*”, по которому автомат переходит в состояние 2. Осталось подобрать оставшиеся два символа входной строки так, чтобы автомат через два шага оказался в единственном терминальном состоянии 2. Перебрав все варианты, выясняем, что автомат допускает две строки: “*aaa*” и “*abb*”. Действительно, ответ – два.

4.2. Общая схема решения

Выше было показано, что ДКА с непоглощающими ребрами можно преобразовать в ДКА без них. Однако пока неясно, насколько эффективно можно выполнить это преобразование. Построив ДКА, остается не ясным содержит ли он непоглощающие ребра (если не удается разработать эффективный алгоритм), или не содержащих их. Напомним, что ответ в задаче – количество допускаемых строк. Ясно, что получать ответ перебором и проверкой всех строк заданной длины нельзя, так как их в общем случае может оказаться очень много.

Как вообще можно посчитать количество строк, удовлетворяющих каким-то условиям? Первый существующий подход – использование хитрых комбинаторных соображений и непосредственное применение этих формул в программе. Второй (если система слишком сложна): найти некую рекуррентную формулу и выразить количество строк данной длины с какими-то свойствами через количество строк меньшей длины с теми же или другими свойствами. Затем применить динамическое программирование для последовательного вычисления всех этих значений. Опыт подсказывает, что для подсчета количества строк, допускаемых данным ДКА, следует использовать динамическое программирование.

Оценим практический масштаб задачи. Алфавит может состоять из 26 символов ($|A| = 26$), а строка может иметь максимальную длину в 60 символов. Рассмотрим конечный автомат, допускающий любую поданную на вход строку, например, автомат, состоящий из одного состояния, которое одновременно является начальным и допускающим. При этом ответ будет 26^{60}, что очень много. Насколько много? При наличии компьютера или хорошего калькулятора, на этот вопрос ответить легко. Определим $\log_{10}(26^{60}) = 60\log_{10}(26) = 60(\ln(26) / \ln(10)) \approx 60*1.415 \approx 84.9$. Следовательно, в ответе может быть 85 цифр в десятичном представлении. Такое длинное число не поместится в стандартный тип (`Integer`). Поэтому при написании решения на языке *Pascal* придется вручную реализовать арифметику повышенной точности.

Выделим подзадачи и составим рекуррентные соотношения. Ключевая идея: преимущество модели конечного автомата в том, что вся история выражается одним числом – номером состояния, а число этих состояний конечно. Для ответа на вопрос, допускается ли конкретная строка, имеет значение не только состояние автомата, но и количество поглощенных символов строки. Будем рассматривать ДКА без непоглощающих ребер. Тогда количество поглощенных символов совпадает с количеством прошедших шагов – тактов работы автомата.

Рассмотрим пару `(state,k)` – текущее состояние, которое описывается состоянием автомата `state` и количеством символов `k`, поглощенных автоматом к текущему моменту. Рассмотрим все строки длины `k`, обладающие следующим свойством: получив на вход такую строку, ДКА попадает из начального состояния в состояние `state`. Обозначим число таких строк через `f(state,k)`. Каждой паре `(state,k)` соответствует подзадача – вычисление `f(state,k)`. После решения этих подзадач останется только просуммировать значения `f(t,N)` по всем терминальным состояниям $t \in T$. Здесь `N` – длина строк, количество которых требуется найти. Как отмечалось выше, для применения динамического программирования необходимы рекуррентные соотношения, которые в данном случае строятся на указанных выше свойств конечного автомата:

$$f(state,0) = \begin{cases} 1, state = initial \\ 0, state \neq initial \end{cases};$$

$$f(state,k) = \sum_{\substack{st \in U, c \in \Sigma \\ \varphi(st,c)=state}} f(st,k-1),$$

где `initial` – начальное состояние ДКА. Здесь для вычисления `f(state,k)` перебираются все пары вида `(st,c)`, где `st` – предыдущее состояние, а `c` – символ, по которому произошел последний переход. Соответственно, `φ(st,c)=state`.

Сформулируем схему решения.

1. Превращаем ДКА с поглощающими ребрами в ДКА без них. Наличие решения с помощью динамического программирования для ДКА без непоглощающих ребер укрепляет уверенность в том, что эффективное устранение непоглощающих ребер также возможно.
2. Применяем динамическое программирование, используя вышеприведенные рекуррентные соотношения. Целесообразно, хотя и не обязательно, использовать динамическое программирование «Снизу вверх» [9].
3. Для нахождения ответа необходимо просуммировать $\sum_{t \in T} f(t,N)$ по всем терминальным состояниям `t`.

4.3. Удаление поглощающих ребер

Предположим, что ДКА находится в некотором состоянии, и переберем все символы из входного алфавита. Выполним переход по рассматриваемому символу, как будто он является очередным символом входной ленты. Если переход произошел по непоглощающему ребру, то выполним следующий переход по тому же символу. Если переход снова произошел по непоглощающему ребру, то повторим тот же «маневр». В результате либо символ будет поглощен, либо автомат войдет в цикл.

В первом случае можно изменить переход из исходного состояния, из которого автомат прошел по цепочке непоглощающих ребер, сразу поставив переход в конечное состояние. Во втором – можно поставить ребро в фиктивное состояние – «Недопуск», которое не является терминальным. Оно замкнуто на себя при переходе по всем символам. Попадание в это состояние будет соответствовать входу в цикл из непоглощающих ребер в исходном автомате. Рассмотрев таким образом все состояния автомата, построим конечный автомат без непоглощающих ребер, эквивалентный исходному.

Оценим эффективность программы, реализующей предложенный подход. Как определить, что произошло попадание в цикл? Все равно придется сделать порядка `|U|` шагов в худшем случае – можно сделать непосредственно `|U|` шагов, и если за это количество шагов ДКА не перейдет по непоглощающему ребру, следовательно, он вошел в цикл. Верхняя оценка времени работы: `|U|*|U|*|Σ|` – порядка $26*10^6$. Много это или мало? По меркам 2003 г., в котором была предложена задача, за секунду можно было выполнить 10^7 простых операций на языке высокого уровня, таком, как языки *Pascal* или *C++*. Следовательно, такой способ устранения непоглощающих ребер требует не более одной секунды, что в первом приближении допустимо, так как по условию задачи время на выполнение одного теста не должно превышать двух секунд.

Приведем псевдокод алгоритма удаления непоглощающих ребер. В псевдокоде множество всех состояний обозначено через State.

```
for c in Alpha do       // По всем символам алфавита
  for i in State do     // По всем состояниям автомата
begin
  cur := i              // Текущее состояние
  z := n                // Количество состояний
   // Пока ребро-переход из состояния k по символу j –
   // поглощающее, и еще сделано не очень много переходов
  while (χ[cur,c] = 1) and (z > 0) do
  begin
    cur := φ[cur][c] // Перейти
    z := z – 1       // Уменьшить счетчик
  end
  // Если все еще очередное ребро – непоглощающее
  if (χ[cur][c] = 1) then
    φ[i][c] := 0     // Следовательно – цикл
  else
    φ[i][c] := φ[cur][c] // Иначе переставляем ребро
  // Теперь снимаем пометку «непоглощающее ребро»
  χ[i][j] := 0
end
```

Отметим, что существует решение, которое устраняет непоглощающие ребра за O(|U|*|Σ|) с помощью поиска в глубину. Однако не следует искать его на этом этапе – ни в практическом программировании, ни в олимпиадном. Необходимо рассмотреть решение второй части, а потом уже решать, требуется ли более эффективно устранять непоглощающие ребра, или предложенный вариант приемлем.

4.4. Динамическое программирование и получение ответа

Перейдем к рассмотрению второго и третьего этапов. Теперь будем рассматривать только ДКА без непоглощающих ребер. Строим решение.

Динамическое программирование бывает двух видов: «Снизу вверх» и «Сверху вниз» [9]. Динамическое программирование «Снизу вверх», в свою очередь, удобно разделять на два вида. Будем называть первый из них, который больше похож на динамическое программирование «Сверху вниз», методом «Сзади сюда». В этом случае для каждой пары (state, k) значение f(state, k) находится через уже вычисленные значение функции для других пар аргументов. Второй метод («Отсюда вперед») состоит в следующем. Каждый раз выбирается пара (state, k), для который значение функции уже известно, и в таблице учитывается его вклад в значение функции для других пар аргументов, для которых верное значение функции еще не найдено.

В динамическом программировании «Сверху вниз» в каждом состоянии (state, k) придется найти пары, из которых был переход в рассматриваемое состояние. Следовательно, придется перебирать прошлые состояния и символы, по которым мог произойти переход, но суммирование производить лишь иногда. Аналогичная ситуация возникает при использовании динамического программирования «Снизу вверх», «Сзади сюда». Для варианта «Отсюда вперед» все проще: из данной пары переходим только в те состояния, в которые необходимо (рассматривая все символы), а прибавление будет происходить на каждой итерации цикла. Тем самым, полученное решение будет более эффективным. Итак, выбираем решение с помощью динамического программирования вида: «Снизу вверх», «Отсюда вперед».

Рассмотрим некоторые детали реализации. Заведем для значений функции f одноименный массив f[state, k].

Запишем на псевдокоде как применять рекуррентные соотношения, приведенные в разд. 4.2. Заметим, что цикл по всем состояниям включает фиктивное состояние «Недопуск» под номером ноль, добавленное после удаления непоглощающих ребер. Поэтому вместо множества состояний State будет встречаться состояние $State_0$, включающее еще и состояние «Недопуск».

```
// Один способ оказаться в начальном состоянии
f[init,0] := 1
// И ноль – во всех остальных состояниях
for i in State0 do
  if (i <> init) then
    f[i,0] := 0

// Считаем количество допускаемых n-символьных строк
for k := 1 to n do
for st in State0 do   // По всем состояниям
for c in Alpha do     // По всем символам алфавита
   f[φ[st,c],k] := f[φ[st,c],k] + f[st,k-1]

ans := 0
// Суммируем по всем терминальным состояниям
for st in TerminalState do ans := ans + f[st,n]
```

Отметим, что в этом тексте для простоты опущено обнуление всего массива f.

Зачем заводить двумерный массив? Ведь в каждый момент используется только предыдущее значение. Достаточно одномерного массива sum[state] – количество способов оказаться в данном состоянии на текущем шаге.

Изменим псевдокод, чтобы вместо двумерного массива использовался одномерный.

```
// Один способ оказаться в начальном состоянии
sum[init] := 1
// И ноль – во всех остальных состояниях
for i in State0 do
  if (i <> init) then
    sum[i] := 0

// Считаем количество допускаемых n-символьных строк
for k := 1 to n do
begin
  sum2 := sum
  for i in State0 do sum[i] := 0
  for i in State do
    for c in Alpha do
      sum[φ[i][c]] := sum[φ[i][c]] + sum2[i]
end
ans := 0
// Суммируем по всем терминальным состояниям
for st in TerminalState do
    ans := ans + sum[st]
```

Оценим эффективность этого решения. Длинные числа складываются N*|U|*|Σ| раз, что в худшем случае может достигать 60*1000*26 ≈ 1.5 миллионов раз. Второй секунды на это достаточно. И на копирования временного массива sum2 в sum – тем более. Скорее всего, для первой части не потребуется искать более эффективно решение.

Как уже было отмечено, ответ может быть очень большим. Поэтому должна применячться арифметика повышенной точности («длинная арифметика»). Для написания решения на языке *Pascal* (*Borland Delphi*) длинную арифметику придется реализовывать самостоятельно, используя при этом книгу Д. Кнута [13], в которой этому посвящена четвертая глава. В данной задаче для повышения эффективности разумно реализовать длинную арифметику по основанию 10^9 – хранить по девять десятичных цифр в одной ячейке массива, содержащего длинное число.

Исходный текст решения приведен в приложении.

ЗАКЛЮЧЕНИЕ

Применение динамического программирования позволило получить эффективное решение рассмотренной задачи, удовлетворяющее ограничениям по времени и памяти, поставленным в условии задачи.

Кроме рассмотренной, с применением динамического программирования могут быть решены также многие другие задачи на автоматах.

Перечислим некоторые из них:

– найти лексикографически минимальную строку, допускаемую данным автоматом;

– найти кратчайшую по количеству символов строку, допускаемую данным автоматом; найти, в каких состояниях может оказаться автомат после получения на вход строки данной длины;

– определить вероятность оказаться в каждом из состояний после получения случайной строки фиксированной длины, различные символы которой независимы в совокупности. При этом могут быть заданы вероятности появления каждого символа алфавита.

Перечислим еще несколько задач на автоматах:

– существует ли входная последовательность длины, не превышающей данную, на которой автомат выдает данный выход;

– существует ли входная последовательность, длина которой не превышает данную, на которой автомат выдает выход, допускаемый вторым автоматом (это динамическое программирование на произведении автоматов).

В этих задачах фразу «существует ли входная последовательность» можно заменить на фразу «найти лексикографически минимальную входную последовательность.

Аналогично можно посчитать количество строк данной длины, на которых автомат выдает данный выход и т. д.

Отметим, что применение динамического программирования на конечных автоматах, как и вообще на графах, является очень эффективным. Перекрывающиеся подзадачи автоматически выделяются: необходимо лишь отслеживать целевую функцию во всех состояниях автомата. В качестве примера можно привести задачу, при решении которой используется та же техника, что и при решении задачи, рассмотренной в настоящей работе: задача «Currency Exchange» («Обмен валюты»). Автор: Николай Дуров. Источник: четвертьфинал NEERC-2001, северный подрегион. Задача размещена на сайте http://acm.timus.ru под № 1162. Фактически математической моделью этой задачи является конечный автомат, в котором состояниями являются валюты, а переходами – обменные пункты.

Следует указать еще одну известную задачу: «Censored!» («Цензура!»). Автор: Николай Дуров. Источник: четвертьфинал NEERC-2001, северный подрегион. Задача размещена на сайте http://acm.timus.ru под № 1158. Ее решение состоит из двух этапов: построение конечного автомата, распознающего набор строк (на основе алгоритма Ахо – Корасика), и использование динамического программирования на полученном конечном автомате.

В заключение работы отметим, что исследования по теории автоматов проводятся на кафедре «Интеллектуальные системы» мехмата Московского государственного университета (http://intsys.msu.ru), а по вопросам применения автоматов в программировании – на кафедре «Технологии программирования» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СПбГУ ИТМО) (http://is.ifmo.ru).

ЛИТЕРАТУРА

1. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
2. *Непейвода Н.Н.* Стили и методы программирования. М.: Интернет-университет информационных технологий. 2005.
3. *Карпов Ю.Г.* Теория автоматов. СПб.: Питер, 2002.
4. *Шалыто А.А.* Технология автоматного программирования //Мир ПК. 2003. № 10, с.74–78. http://is.ifmo.ru/works/tech_aut_prog
5. *Казаков М.А., Шалыто А.А.* Использование автоматного программирования для реализации визуализаторов // Компьютерные инструменты в образовании. 2004. № 2, с. 19– 33. http://is.ifmo.ru/works/art_vis.pdf
6. *Беляев А.В., Суясов Д.И., Шалыто А.А.* Компьютерная игра «Космонавт». Проектирование и реализация // Компьютерные инструменты в образовании. 2004. № 4, с. 75–84. http://is.ifmo.ru/works/_cosmo_article.pdf
7. *Мазин М.А., Парфенов В.Г., Шалыто А.А.* Анимация. *FLASH*-технология. Автоматы // Компьютерные инструменты в образовании. 2003. № 4, с. 39–47. http://is.ifmo.ru/projects/flash/
8. *Беллман Р.* Динамическое программирование. М.: изд-во иностр. лит., 1960.
9. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦНМО, 1999.
10. *Скиена С., Ревилла М.* Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. М.: Кудиц-Образ, 2005.
11. *Станкевич А.С.* Непоглощающий детерминированный конечный автомат /Архив олимпиадных задач Саратовского государственного университета. Задача № 201. http://acm.sgu.ru
12. *Бобак И.* Алгоритмы: "возврат назад" и "разделяй и влавствуй" //Программист. 2002. № 3, с.29–32.
13. *Кнут Д.Э.* Искусство программирования. Том 2. Получисленные алгоритмы. М.: Вильямс, 2004.

ОБ АВТОРАХ

Оршанский Сергей Александрович – бакалавр прикладной математики и информатики СПбГУ ИТМО, чемпион мира по программированию ACM ICPC 2004, III место на чемпионате мира по программированию ACM ICPC 2005.

Шалыто Анатолий Абрамович – доктор технических наук, профессор СПбГУ ИТМО.

Приложение. Исходный текст решения задачи на *Borland Delphi*

```
{$apptype console}   // Создать консольное приложение

// Включить проверки переполнения и отключить оптимизацию
{$o-,q+,r+}

uses Math, SysUtils; // Подключить модули Math и SysUtils
```

```
           {Длинная арифметика. Сложение и вывод числа}

const
  pow = 9;            // Длинная арифметика по 9 десятичных цифр
  base = round(1e9); // 10^pow
  m = 10;             // Требуемое количество таких «цифр»
type
  long = array [0..m] of integer; // Тип: длинное число

// Процедура сложения двух длинных чисел a и b
// Результат записывается в a
// Перед параметром b стоит ключевое слово var,
// так как передавать массив по значению слишком медленно,
// по ссылке гораздо быстрее
procedure add(var a : long; var b : long);
var i, c : integer;
begin
  c := 0;
  for i := 0 to m do
  begin
    c := c + a[i] + b[i];
    if c >= base then
    begin
      a[i] := c - base;
      c := 1;
    end else
    begin
      a[i] := c;
      c := 0;
    end;
  end;
// Если вдруг “цифр” в типе long окажется недостаточно, то
// сигнализируется ошибка
  assert(c = 0);
end;

// Печать длинного числа
procedure print(var x : long);
var i, j : integer;
begin
  i := m;
  while (i > 0) and (x[i] = 0) do dec(i);
  write(x[i]);
  for j := i - 1 downto 0 do
    write(format('%.' + IntToStr(pow) + 'd', [x[j]]));
end;
```

```
                         {Константы и переменные}

const               // Две константы из условия задачи
  max_nst = 1000; // Максимальное количество состояний
  max_na = 26;    // Максимальный размер алфавита

var
  alfa : string;  // Входной алфавит автомата
  na : integer;   // Количество символов во входном алфавите
  nst : integer;  // Количество состояний автомата
  ist : integer;  // Начальное состояние автомата
  ntst : integer; // Количество конечных состояний автомата

  len : integer;  // Длина рассматриваемых строк
// Является ли состояние терминальным
  term : array [1..max_nst] of boolean;
// Функция переходов
  fi : array [1..max_nst, 1..max_na] of integer;
// Является ли ребро непоглощающим?
  ee : array [1..max_nst, 1..max_na] of boolean;

  sum, sum2 : array [0..max_nst] of long;
  i, j, k, z : integer; // Временные переменные
  ans : long;
```

```
                          {Ввод входных данных}

begin
  readln(alfa);      // Читаем алфавит
  na := length(alfa);// Имеет значение только размер алфавита
  read(nst);         // Читаем количество состояний
  read(ist);         // Читаем номер начального состояния
  read(ntst);        // Читаем количество терминальных состояний

// Читаем номера терминальных состояний
  for i := 1 to nst do
    term[i] := false;
  for i := 1 to ntst do
  begin
    read(j);
    term[j] := true;
  end;
// Читаем функцию переходов φ
  for i := 1 to nst do
  begin
    for j := 1 to na do read(fi[i][j]);
  end;
// Читаем функцию χ
  for i := 1 to nst do
  begin
    for j := 1 to na do
    begin
      read(k);
      ee[i][j] := k = 1;
    end;
  end;
// Читаем N – длину строк, количество которых необходимо найти
  read(len);
```

```
                    {Устранение непоглощающих ребер}

  for j := 1 to na do          // По всем символам алфавита
  for i := 1 to nst do         // По всем состояниям автомата
  begin
    k := i;                    // Текущее состояние
    z := nst;                  // Количество состояний

// Пока ребро-переход из состояния k по символу j –
// поглощающее, и еще сделано не очень много переходов
    while (ee[k][j]) and (z > 0) do
    begin
      k := fi[k][j];           // Перейти
      dec(z);                  // Уменьшить счетчик
    end;
// Если все еще очередное ребро – непоглощающее
    if ee[k][j] then
    begin
         fi[i][j] := 0;        // Значит – цикл
    end else
    begin
         fi[i][j] := fi[k][j];// Иначе переставляем ребро
    end;
   // И теперь снимаем пометку «непоглощающее ребро»
       ee[i][j] := false;
  end;
```

```
          {Динамическое программирование на конечном автомате}

// Инициализация
  for i := 0 to nst do
    fillchar(sum[i], sizeof(sum[i]), 0);
  sum[ist][0] := 1;
```

```
// Последовательное вычисление
  for k := 1 to len do
  begin
    sum2 := sum;
    for i := 0 to nst do
      fillchar(sum[i], sizeof(sum[i]), 0);
    for i := 1 to nst do
      for j := 1 to na do
      begin
        add(sum[fi[i][j]], sum2[i]);
      end;
  end;
```

```
// Получение ответа
// Для этого суммируем количество способов оказаться
// в каждом терминальном состоянии
  fillchar(ans, sizeof(ans), 0);
  for i := 1 to nst do
    if term[i] then
      add(ans, sum[i]);
```

```
                              {Вывод ответа}
  print(ans);
end.
```