Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра “Компьютерные технологии”

А. А. Пестов, А. А. Шалыто

Преобразование недетерминированного конечного автомата в детерминированный

Работа выполнена
в рамках
“Движения за открытую проектную документацию”
http://is.ifmo.ru

Санкт-Петербург
2003

Содержание

Введение

В последнее время развивается SWITCH-технология, удобная для систем управления техническими объектами. Эта технология использует понятие “детерминированный конечный автомат” (ДКА). Однако существуют задачи, для которых исходную формализацию целесообразно выполнять с помощью недетерминированных конечных автоматов (НКА) [1, 2].

Кратко поясним, что такое ДКА, и чем он отличается от НКА.

1. Постановка задачи

ДКА - это конечный автомат, в котором каждый переход строго регламентирован. В таком автомате при поступлении входного воздействия однозначно определяется следующее состояние.

В НКА допускаются состояния, в которых по одному входному воздействию могут произойти переходы в различные состояния. При появлении входного воздействия НКА “размножается” - создаются копии автомата, в каждой из которых выполняется соответствующий переход из рассматриваемого состояния [1]. Считается, что НКА достигает конечного состояния, если хотя бы одна из его “копий” перейдет в свое конечное состояние. При проектировании конкретных программ, в отличие от теории, применять НКА невозможно, так как на практике нельзя “хаотично” создавать копии.

НКА бывает полезно использовать на ранних стадиях проектирования, когда еще не требуется добиваться “полной” четкости в описании поведения системы, так как это упрощает формализацию. При этом переход от НКА к ДКА осуществляется на последующих стадиях. НКА могут применяться при обработке текстов, например, при поиске подстрок. Именно эта задача и будет рассматриваться в данной работе в качестве примера.

В дальнейшем будем считать, что если для состояния, в котором находится автомат входное воздействие не определено, то он не “погибает”, а остается в этом состоянии.

Настоящая работа посвящена разработке программы преобразования НКА к ДКА на основе алгоритма К. Томпсона [1]. Программа позволяет также устранять вершины, недостижимые из начальной.

Программа написана на языке программирования C++ с использованием библиотеки классов MFC.

2. Краткое описание алгоритмов построения детерминированного автомата по недетерминированному

2.1. Первый алгоритм

Основной идеей этого алгоритма является то, что каждое состояние ДКА – набор некоторых состояний НКА. Создадим ДКА, граф переходов которого содержит 2^N вершин (N – количество состояний НКА), следующим образом.

Каждой вершине ДКА сопоставляется двоичный вектор (двоичное разложение номера вершины ДКА), описывающий набор состояний НКА, объединением которых является рассматриваемая вершина ДКА. Определим переходы из каждой вершины в созданном автомате. При этом объединяются переходы из состояний, отмеченных единицей в векторе. После этого переходы из одной вершины, помеченные одинаковыми входными воздействиями, заменяются на переход в вершину, вектор которой является объединением векторов вершин, в которые были направлены эти переходы.

Используем метаязык для описания алгоритмов. На этом языке вышеприведенный алгоритм описывается следующим образом [1]:

```
<первый алгоритм преобразования в детерминированный автомат>
    <создать 2^N состояний>
    <цикл по всем состояниям> BEGIN
        <разложить номер в двоичный вектор>
        <объединить переходы состояний исходного автомата,
        отмеченных единицами в векторе>
        <все переходы, соответствующие одному входному
        воздействию, объединить в один переход, который помечен
        этим же воздействием, но направлен в состояние, номер
        которого объединение векторов (номеров) состояний, куда
        вели начальные переходы>
    END
```

Рассмотренный алгоритм имеет недостаток – количество созданных состояний растет экспоненциально относительно количества состояний в исходном НКА. Однако большая часть этих состояний оказываются заведомо недостижимыми из начальной вершины, поэтому для уменьшения сложности описания автомата их следует удалить.

2.2. Второй алгоритм

Этот алгоритм предполагает те же действия, что и изложенные выше, однако он не создает сразу все 2^N состояний, а обходит “в ширину” граф переходов, соответствующий создаваемому ДКА. По сути, этот алгоритм последовательно рассматривает состояния НКА, превращая каждое из них в “детерминированное”, за счет создания дополнительных состояний. Алгоритм строит АВЛ – дерево (дерево, названное по первым буквам фамилий их исследователей – Адельсона-Вельского Г.М. и Ландиса Е.М.). Такое дерево хранит информацию (вектор и номер) о тех вершинах, которые помечаются в результате обхода в

ширину. Описание алгоритма на метаязыке выглядит почти так же, как и в предыдущем случае, но программный код более сложен:

```
<второй алгоритм преобразования в детерминированный автомат>
    <цикл по всем известным состояниям> BEGIN
        <отсортировать переходы из этого состояния по номеру
        входного воздействия>
        <для всех переходов с одинаковым входным воздействием>
            <создать вектор – объединение векторов состояний, куда
            ведут эти переходы>
            <найти по вектору номер состояния в АВЛ – дереве>
            <если такое состояние не нашлось, то создать
            состояние, соответствующее данному вектору>
            <заменить переходы, соответствующие одному входному
            воздействию, одним новым переходом, ведущим в
            найденное (созданное) состояние>
    END
```

Тексты основных классов программы, написанной на основе второго алгоритма, приведены в разд. 8.

3. Краткое описание классов программы

Программа преобразования недетерминированного автомата в детерминированный написана по второму алгоритму, и ее тексты приведены в разд. 8. Она состоит из двух частей, первая из которых отвечает за конкретное представление автомата и его состояний, а вторая - за графическую часть (интерфейс пользователя), которая в работе не рассматривается.

Первая часть является основной и содержит следующие классы:

- *CAutomaton* - класс, который хранит автомат. Он позволяет считывать из потока данных описание НКА, проверять детерминированность, приводить к ней и удалять недостижимые (лишние) состояния, а также выводить полученное описание ДКА в поток;

- *CState* – класс хранит состояние автомата. Он содержит данные о переходах из этих состояний, а также различные методы, вспомогательные по отношению к методам класса *CAutomaton*;

- *CLexems* - класс предназначен для разбора файлов на лексемы. Он используется для чтения входного потока и его разбора;

- *C2MapBitSet* – класс предназначен для работы с парами, каждая из которых состоит из двух элементов - множества чисел и одного числа. При этом допустимо искать пару, как по первому элементу, так и по второму. Пары хранятся в виде АВЛ - дерева;

- *CBitSet* - класс предназначен для работы с множествами (объединение, включение и исключение элементов);

- *CListWord* – класс предназначен для хранения пары “строка и число” в АВЛ – дереве.

В файле <vecutils.h> также описана процедура для удаления элементов вектора по шаблону.

Диаграмма рассмотренных классов изображена на рис. 1. Заголовки описаний приведены в разд. 7, а листинги – в разд. 8.

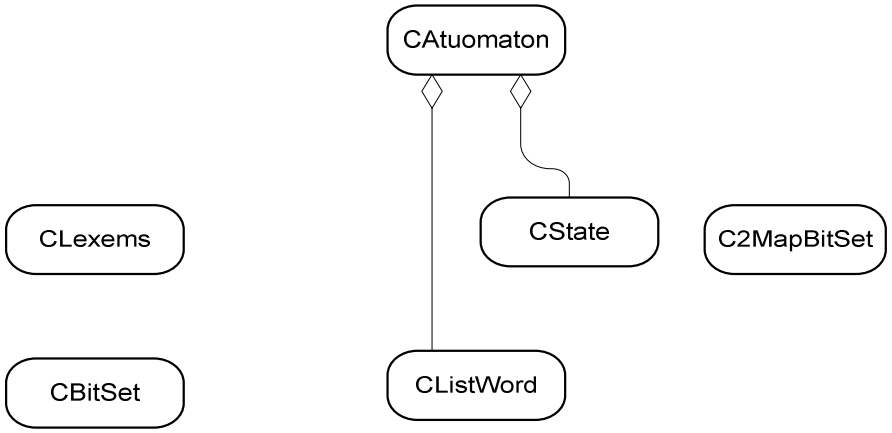

Рис. 1. Диаграмма классов

4. Описание формата представления автоматов

Программа позволяет считывать и сохранять описание автомата в текстовом виде.

Опишем формат записи автомата в форме БНФ (форма Бэкуса-Наура). Этот формат является набором правил. Поясним основные обозначения:

- под словами, записанными в фигурных скобках, подразумевается нетерминал – выражение, которое должно быть раскрыто с помощью описанных в формате правил;
- выражения, разделенные символом “|” и записанные в круглых скобках, под которыми понимают выбор одного из этих выражений:

  <a> = “Россия -” (“чемпион” | “вперед!” | “наша страна”)

  При этом выводятся одна из следующих строк:

  “Россия – чемпион”,

  “Россия – вперед!”,

  “Россия – наша страна”;

- для обозначения необязательности выражения оно заключается в квадратные скобки:

  <a> = [“почти”] “утро”

  Это правило выводит одну из следующих строк:

  “утро”,

  “почти утро”;

- для обозначения повторения используются фигурные скобки, которые означают повторение выражения от нуля до бесконечного числа раз:

  <a> = “Кто твои” {“пра”} “деды?”

  Это правило выводит строки:

  “Кто твои деды?”,

  “Кто твои прадеды?”,

  “Кто твои прапрадеды?” и т. д.

Перейдем к рассмотрению предлагаемого формата описания автоматов, которое выглядит следующим образом:

<автомат> = <имя автомата> "*{*" <список состояний> "*}*"

<список состояний> = { <состояние> }

<состояние> = ["*start*"] ["*final*"] "*state*" <имя состояния> "*{*" <список переходов> "*}*"

<список переходов> = { <переход> " *;* " }

<переход> = (<имя входного воздействия> | “ELSE” | “ANY”) <имя следующего состояния>

<имя> = (<буква> | <подчеркивание>) {<символ>}

<символ> = (<буква> | <подчеркивание> | <цифра>)

Под именем будем понимать слово, состоящее из латинских букв, цифр и знака подчеркивания. При этом имя не может начинаться с цифры.

Ключевое слово ***start*** (***final***) записывается перед именем состояния и предназначено для выделения начальных (конечных) состояний. В случае, если не указано ни одного начального состояния, считается, что любое состояние может быть начальным.

Формат допускает использование комментариев в стиле языка C – *//* (комментарий до конца строки) и */* комментарий */*.

Под именем входного воздействия ***ELSE*** понимаются все воздействия, для которых не описаны переходы в текущем состоянии, а под именем входного воздействия ANY понимаются все воздействия.

5. Алгоритм построение автомата для задачи поиска подстроки

Пусть требуется построить детерминированный автомат, ищущий некий образец R, как подстроку в строке *S*. Эта строка подается на вход автомата посимвольно.

Решим эту задачу следующим образом – создадим вначале НКА, который ищет (принимает) образец в подаваемой на вход строке, а затем с помощью программы, которая являлась целью работы, приведем этот НКА к ДКА.

Тогда алгоритм построения ДКА для поиска фиксированного образца R в строке запишется на метаязыке следующим образом:

```
<создать автомат из (length(R) + 1) состояний>
<сделать последнее состояние конечным>
<сделать нулевое состояние начальным>
for i = 0 to length(R) - 1
   <добавить переход (i, i+1, s[i])>
for i = 1 to length(R) - 1
    <добавить переход (i, 0, ELSE)>
<добавить переход(0, 0, ANY)>
<преобразовать в детерминированный автомат>
<удалить недостижимые вершины>
```

Переход описывается тройкой - *откуда совершен переход, куда и входное воздействие по которому он совершается.*

Отметим, что, как было указано выше, в классической трактовке [1] при неопределенном входном воздействии автомат “погибает”. Поэтому не требуются “обратные” связи в соответствующем графе переходов, а в алгоритме построения НКА удаляется второй цикл *for*. Однако, ДКА, построенные на основе этих трактовок, эквивалентны.

В следующем разделе приведен пример построения ДКА по НКА указанным выше способом.

6. Пример построения автомата для задачи поиска подстроки

Пусть требуется построить детерминированный автомат, ищущий слово *оборотень* (образец), как подстроку в строке *S*. Эта строка подается на вход автомата посимвольно.

Построим недетерминированный автомат с десятью состояниями (число букв в слове *оборотень* плюс одно). Номер состояния означает, что поданная на вход строка заканчивается на выбранные начальные буквы образца, где количество выбранных букв совпадает с номером состояния.

Последнее состояние, соответствующее случаю, когда слово найдено, является заключительным.

Например, пусть на вход автомата уже было подано 11 символов (обораааабо) строки *S*.

Автомат может одновременно находиться в состояниях с номерами 0, 1, 3, что соответствует выбору слов: “пустое”, "**о**", "**обо**". Ниже визуально отображено сопоставление конца поданной строки и начала образца:

```
обораааабо              обораааабо              обораааа обо
          оборотень               оборотень               оборотень

  состояние № 0           состояние № 1           состояние № 3
```

Для каждой вершины графа переходов, кроме последней (девятой), добавим дугу в следующую по номеру вершину. Разместим на этих дугах буквы в соответствии с их расположением в образце (рис. 2).

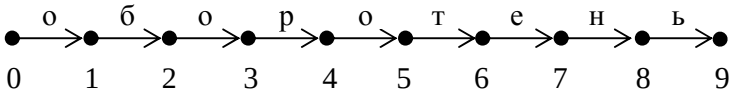

Рис. 2. Остов недетерминированного графа переходов

К построенному остову добавим переходы по воздействию *ELSE (иначе)* в нулевую вершину из каждой вершины (кроме нулевой и девятой). Добавим также переход из нулевой вершины в нулевую по внешнему воздействию *ANY (любое)*. В результате получим недетерминированный автомат (рис. 3). Для компактности он изображен в нелинейной форме.

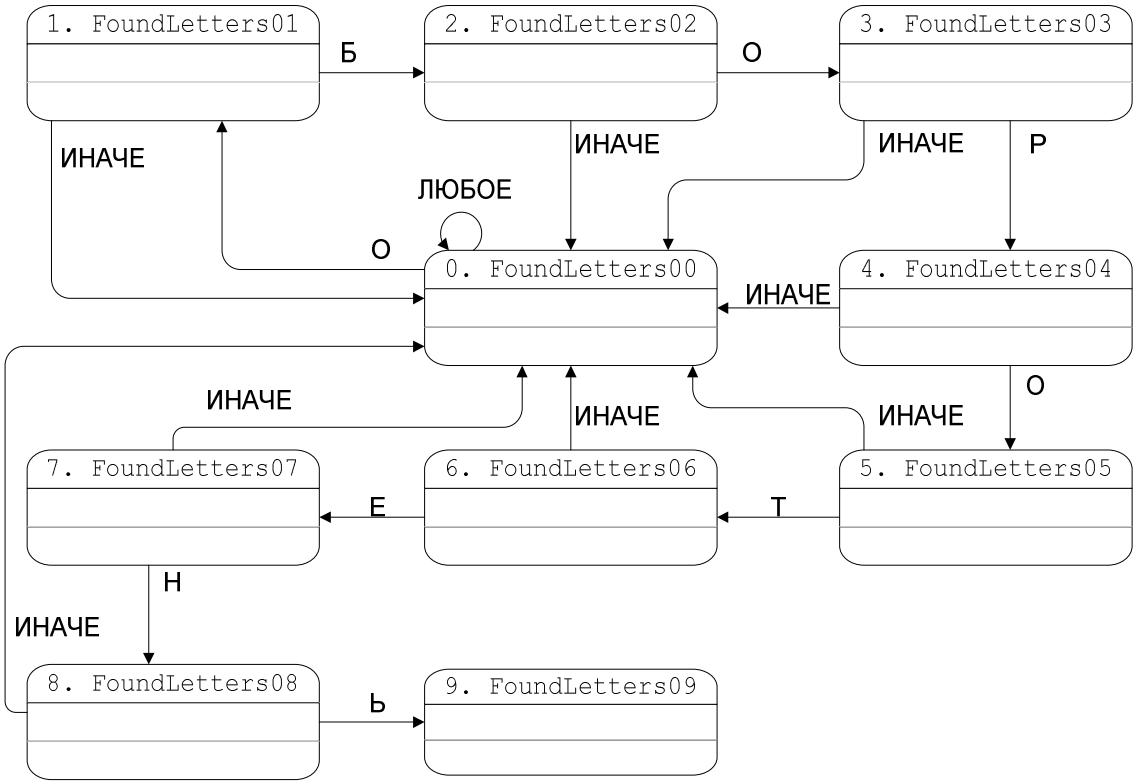

Рис. 3. Недетерминированный автомат для поиска слова *оборотень*

В формате представления автоматов построенный НКА запишем следующим образом:

```
Substring_Search {

    start state FoundLetters00 {
        letterO FoundLetters01;
        ANY FoundLetters00;
    }

    state FoundLetters01 {
        letterB FoundLetters02;
        ELSE FoundLetters00;
    }

    state FoundLetters02 {
        letterO FoundLetters03;
        ELSE FoundLetters00;
    }

    state FoundLetters03 {
        letterR FoundLetters04;
        ELSE FoundLetters00;
    }

    state FoundLetters04 {
        letterO FoundLetters05;
        ELSE FoundLetters00;
    }

    state FoundLetters05 {
        letterT FoundLetters06;
        ELSE FoundLetters00;
```

```
    }
    state FoundLetters06 {
        letterE FoundLetters07;
        ELSE FoundLetters00;
    }

    state FoundLetters07 {
        letterN FoundLetters08;
        ELSE FoundLetters00;
    }

    state FoundLetters08 {
        letterb FoundLetters09;
        ELSE FoundLetters00;
    }

    final state FoundLetters09 {
    } // Success
}
```

Применив к построенному недетерминированному автомату разработанную программу устранения “недетерминированности” (разд. 8), получим детерминированный автомат, описываемый следующим образом:

```
Substring_Search {

    start state FoundLetters00 {
        letterO CState10;
        ELSE    FoundLetters00;
    } // End of state FoundLetters00

    state FoundLetters01 {
        letterB FoundLetters02;
        ELSE    FoundLetters00;
    } // End of state FoundLetters01

    state FoundLetters02 {
        letterO FoundLetters03;
        ELSE    FoundLetters00;
    } // End of state FoundLetters02

    state FoundLetters03 {
        letterR FoundLetters04;
        ELSE    FoundLetters00;
    } // End of state FoundLetters03

    state FoundLetters04 {
        letterO FoundLetters05;
        ELSE    FoundLetters00;
    } // End of state FoundLetters04

    state FoundLetters05 {
        letterT FoundLetters06;
        ELSE    FoundLetters00;
    } // End of state FoundLetters05

    state FoundLetters06 {
        letterE FoundLetters07;
        ELSE    FoundLetters00;
    } // End of state FoundLetters06
```

```
    state FoundLetters07 {
        letterN FoundLetters08;
        ELSE    FoundLetters00;
    } // End of state FoundLetters07

    state FoundLetters08 {
        letterb FoundLetters09;
        ELSE    FoundLetters00;
    } // End of state FoundLetters08

    final state FoundLetters09 {
        ELSE    FoundLetters09;
    } // End of state FoundLetters09

    state CState10 {
        letterO CState10;
        letterB CState11;
        ELSE    FoundLetters00;
    } // End of state CState10

    state CState11 {
        letterO CState12;
        ELSE    FoundLetters00;
    } // End of state CState11

    state CState12 {
        letterO CState10;
        letterB CState11;
        letterR FoundLetters04;
        ELSE    FoundLetters00;
    } // End of state CState12

}  // End of Substring_Search
```

Этому автомату соответствует граф переходов, представленный на рис 4.

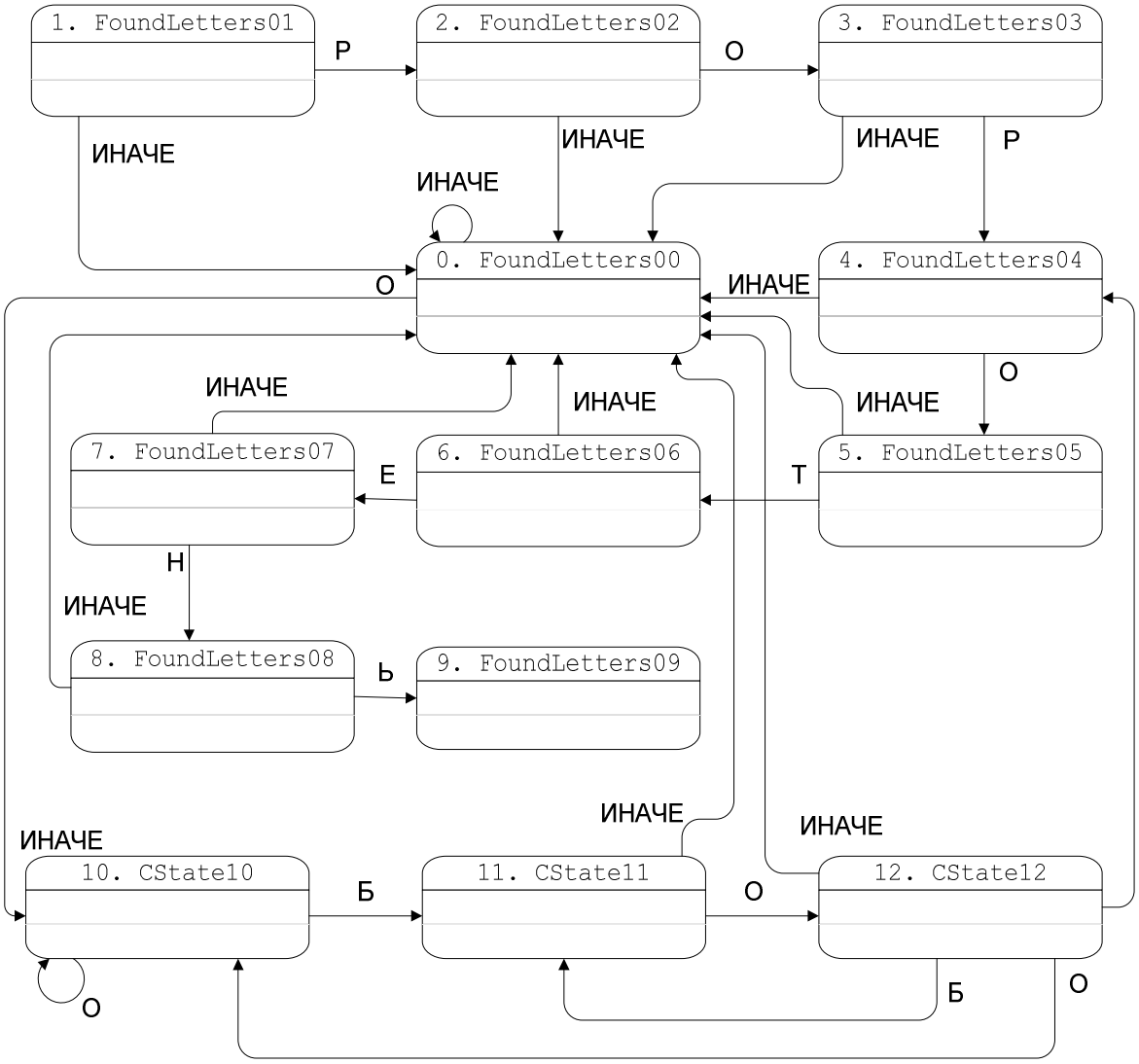

Рис. 4. Детерминированный автомат для поиска слова *оборотень*

Программа позволяет удалять недостижимые вершины. Поэтому после удаления первой, второй и третьей вершин, недостижимых из начальной вершины, получим описание упрощенного графа переходов:

```
Substring_Search {

    start state FoundLetters00 {
        letterO CState10;
        ELSE    FoundLetters00;
    } // End of state FoundLetters00

    state CState10 {
        letterO CState10;
        letterB CState11;
        ELSE    FoundLetters00;
    } // End of state CState10
```

```
    state FoundLetters04 {
        letterO FoundLetters05;
        ELSE    FoundLetters00;
    } // End of state FoundLetters04

    state FoundLetters05 {
        letterT FoundLetters06;
        ELSE    FoundLetters00;
    } // End of state FoundLetters05

    state FoundLetters06 {
        letterE FoundLetters07;
        ELSE    FoundLetters00;
    } // End of state FoundLetters06

    state FoundLetters07 {
        letterN FoundLetters08;
        ELSE    FoundLetters00;
    } // End of state FoundLetters07

    state FoundLetters08 {
        letterb FoundLetters09;
        ELSE    FoundLetters00;
    } // End of state FoundLetters08

    state CState11 {
        letterO CState12;
        ELSE    FoundLetters00;
    } // End of state CState11

    state CState12 {
        letterO CState10;
        letterB CState11;
        letterR FoundLetters04;
        ELSE    FoundLetters00;
    } // End of state CState12

    final state FoundLetters09 {
        ELSE    FoundLetters09;
    } // End of state FoundLetters09

}  // End of Substring_Search
```

Граф переходов упрощенного автомата приведен на рис. 5.

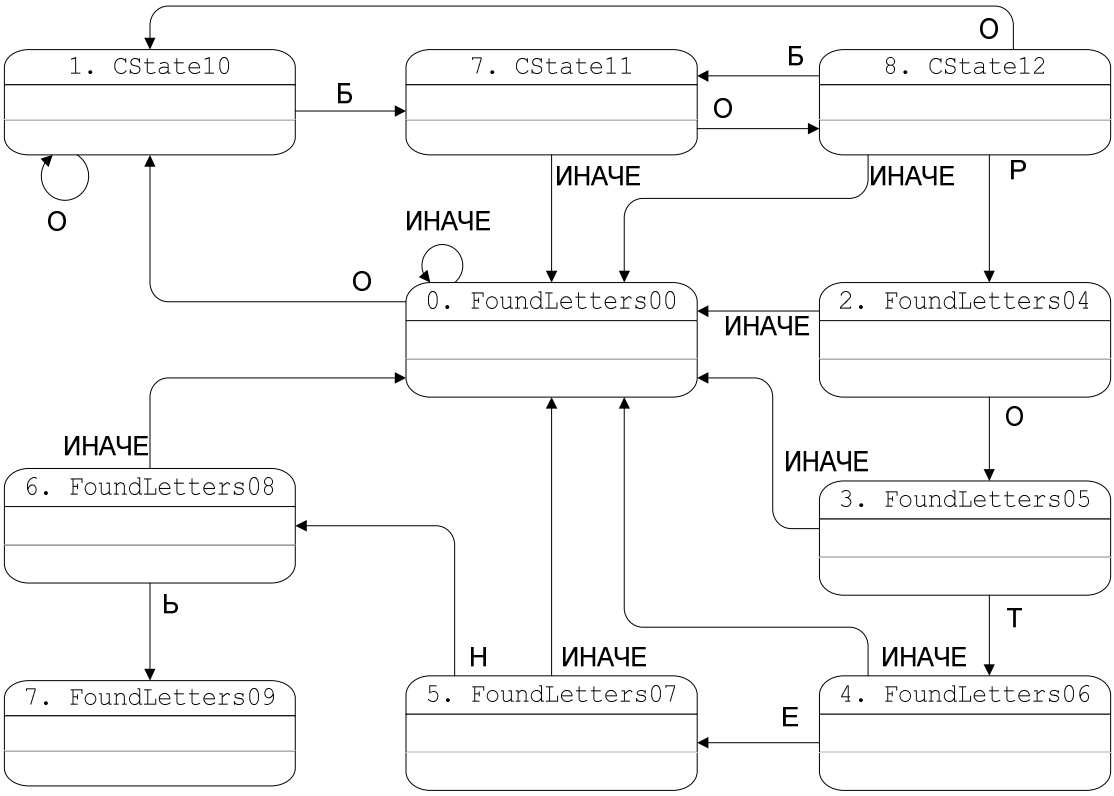

Рис. 5. Упрощенный детерминированный автомат для поиска слова *оборотень*

6. Сравнение алгоритмов построения автоматов поиска подстроки

Проведем качественный анализ автоматов для поиска фиксированной подстроки, которые получаются при различных способах построения детерминированного конечного автомата.

Сравниваемые автоматы строятся на основе двух подходов:

- рассмотренного выше;
- алгоритма Кнута-Мориса-Пратта (КМП) [3].

Детерминированный конечный автомат на основе КМП-алгоритма по строке *S* строится следующим образом:

```
<создать автомат из (length(s) + 1) состояний>
<сделать последнее состояние конечным>
<сделать первое состояние начальным>
for i = 1 to length(s)
   for ch = <все буквы>
      <добавить переход (i - 1, GetPrefix(i, ch), ch)>
```

Опишем функцию GetPrefix(i, ch) = max k • P(s, k) = S(P(s, i-1) + ch, k).

В этой функции:

- P(s, k) – функция префикса(первые k символов строки s);
- S(s, k) – функция суффикса (последние k символов строки s).

При первом подходе число состояний в общем случае растет экспоненциально от числа одинаковых букв в слове. Второй подход строит детерминированный автомат с количеством состояний, равным длине слова плюс один.

По количеству состояний автомата выигрывает второй подход, но первый значительно проще применять, так как построение НКА не вызывает проблем.

Отметим, что для рассмотренного примера оба подхода строят автоматы с одинаковым числом вершин.

7. Заголовки описаний классов

7.1. Класс “CAutomaton”

```
/* CAutomaton - класс, который хранит автомат. Он позволяет считывать из потока данных описание НКА, проверять детерминированность, приводить к ней и удалять недостижимые (лишние) состояния, а также выводить полученное описание ДКА в поток */

class CAutomaton
{
    friend class CState;

private:
    void scanState(CLexems *lex);
        // Считывание состояния из потока lex
    void scanAutomaton(CLexems *lex);
        // Считывание автомата из потока lex
    void scanLink(CLexems *lex, CState *state);
        // Считывание переходов из потока lex и запись их в состояние state

    char* m_automateName; // Имя автомата

    CListWord* m_statesnames; // Набор имен состояний
    CListWord* m_eventsnames; // Набор имен событий

    vector<CState*> m_vstates;
    // Вектор содержит состояния, в которых хранятся
    // только номера, и никаких имен

    void createState();
    // Создать состояние

public:
    CAutomaton(void);
    CAutomaton(const char* name);
    ~CAutomaton(void);


    int countStates(); // Количество состояний
    int countEvents(); // Количество входных воздействий
    int countUnachievable(); // Количество недостижимых состояний
```

```
void getNamesStates(vector<const char*>& list);
    // Получить список имен состояний
void getNamesEvents(vector<const char*>& list);
    // Получить список имен событий
void getNamesUnachievable(vector<const char*>& list);
    // Получить список имен недостижимых вершин

CState* getStateByIndex(int index);
    // Получить состояние по его id (индексу)
CState* getStateByName(const char* name);
    // Получить состояние по его имени
const char* getStateNameByIndex(int index);
    // Получить имя состояния по его индексу

int getEventIndexByName(const char* name);
    // Получить индекс события  по имени
const char* getEventNameByIndex(int idevent);
    // Получить имя входного воздействия по индексу

CBitSet* findAchievable();
    // Найти множество всех достижимых вершин

bool loadFromFile(const char *name);
    // Считать описание автомата из файла
bool saveToFile(const char *name);
    // Сохранить описание автомата в файл

bool checkDeterminate();
    // Проверить детерминированность
bool turnToDeterminate();
    // Привести к детерминированному
void clearAutomaton();
    // “Очистить” автомат
CState* createStateRandom();
// Создать состояние со "случайным" именем

bool deleteUnachievable();
    // Удалить недостижимые вершины
void deleteStatesByMask(CBitSet* set);
    // Удалить состояния по маске
```

```
    ostream& operator<< (ostream&);
        // Вывести описание автомата в поток
    istream& operator>> (istream&);
        // Считать описание автомата из потока
};

ostream& operator<< (ostream& out, CAutomaton* state);
ostream& operator<< (ostream& out, CAutomaton& state);
istream& operator>> (istream& sin, CAutomaton* state);
istream& operator>> (istream& sin, CAutomaton& state);
```

7.2. Класс “CState”

```
/* CState - класс хранит состояние автомата. Он содержит данные о переходах из этих состояний, а также различные методы, вспомогательные по отношению к методам класса CAutomaton */

class CState
{
    friend class CAutomaton;

private:
    CAutomaton* m_liststates;  // Указатель на "свой" автомат
    bool m_property_final;     // Свойство конечное состояние
    bool m_property_start;     // Свойство начальное состояние

    vector <POINT> m_idevents; // Список внешних воздействий

    int m_myOwnId;             // Индекс состояния в списке всех состояний

    void sortLinks();
    // Отсортировать список внешних воздействий
    void processAnyState();
    bool ifSorted();
    // Проверить список внешних воздействий на упорядоченность

    bool transformToDeterminate(C2MapBitSet* numerate, int countStart);
    // Преобразовать состояние в детерминированное
    void goneRound(CBitSet* visited);
    // Посетить всех соседей и отметить их в множестве visited

    bool m_ELSEEVENTlink;
    // Существует ли переход ELSE
    void refreshLinks(vector<int> &newids);
```

```
    // Изменить идентификаторы состояний, в которые делаются переходы.
   // Число "-1" означает, что состояние удаляется

public:
    CState(CAutomaton *parent, int idthat);
    // Создать вершину заданного автомата и заданным индексом
    ~CState(void);

    bool checkStateDeterminate();
    // Проверить состояние на “детерминированность”

    int getThisId();
    // Узнать id состояния
    int getCountLinks() const;
    // Получить количество переходов
    void addCrossing(int event, int nextstate);
    // Добавить переход
    void unionWith(CState* right);
    // Объединить два состояния
    int crossByEvent(int eventid);
    // Номер состояния, куда совершен переход по внешнему воздействию

    void setFinalProperty(bool p_final);
    // Установить свойство состояния быть конечным
    void setStartProperty(bool p_start);
    // Установить свойство состояния быть начальным
    bool checkStartProperty();
    // Проверить свойство состояния быть начальным
    bool checkFinalProperty();
    // Проверить свойство состояния быть конечным
    bool checkELSEEVENTLink();
    // Проверить есть ли переход по внешнему воздействию ELSE

    ostream& operator<< (ostream&);
    // Сохранить состояние в выходной поток
};

ostream& operator<< (ostream& out, CState* state);
ostream& operator<< (ostream& out, CState& state);
```

7.3. Класс “CLexems”

```
/* CLexems - класс предназначен для разбора файлов на лексемы. Он используется для чтения входного потока и его разбора */

enum TLexem {TL_ENDFILE, TL_OPENBRACE, TL_CLOSEBRACE, TL_IDENTY,
            TL_OPENFILE, TL_SEMICOLON, TL_KEYSTATE, TL_KEYFINAL,
            TL_KEYSTART, TL_COMMA, TL_UNKNOWN};
// Стандартные лексемы

// Класс, разбивающий входной поток на лексемы
class CLexems
{

private:
    istream* m_instream; // Входной поток

    char m_curchar; // Текущий символ
    char* m_buffer; // Текущее слово

    TLexem m_curLexem; // Текущая лексема

    void nextChar();
// Считать следующий символ
    bool ifSkipChar(char ch);
// Проверить символ на принадлежность к "белым" пробелам
    void scanWord();
// Считать слово

    void skipToLineEnd();
// Пропустить все символы до конца строки
    void skipToCommentEnd();
// Пропустить все символы до символов */
    int m_countread, m_countprev;
// Общее количество прочитанных символов, и прочитанных к моменту начала разбора предыдущей лексемы

public:
    CLexems(void);
    CLexems(istream& s_in);
// Создать класс, и сразу подать ему на вход поток
    void openStream(istream& s_in);
// Открыть поток
```

```
    TLexem getCurrentLexem();
// Узнать текущую лексему
    TLexem nextLexem();
// Прочитать следующую лексему и узнать её

    void compare(TLexem now, TLexem exp);
// Сравнить две лексемы - одна текущая, другая ожидаемая
    TLexem throwBad(CString st);
// Вызвать исключение, обругавшись на вход
    const char* getString();
// Получить текущее слово

    ~CLexems(void);
};
```

7.4. Класс “C2MapBitSet”

```
/* C2MapBitSet - класс предназначен для работы с парами, каждая из которых
состоит из двух элементов - множества чисел и  числа.  При этом допустимо искать
пару, как по первому элементу, так и по второму. Пары хранятся в виде АВЛ -
дерева */

class C2MapBitSet {

private:
    map<CBitSet, int> m_first_sort;
    // Дерево, отсортированное по первому ключу
    map<int, CBitSet> m_second_sort;
    // Дерево, отсортированное по второму ключу

public:
    C2MapBitSet(void);
    ~C2MapBitSet(void);

    void add(const CBitSet& fs, const int sec);
    // Добавить пару
    int findByFirst(const CBitSet& fs);
    // Найти по первому ключу
    CBitSet const* findBySecond(const int sec);
    // Найти по второму ключу
    void clear();
    // Очистить от всех ключей

};
```

7.5. Класс “CBitSet”

```
/* CBitSet - класс предназначен для работы с множествами (объединение, включение
и исключение элементов) */

class CBitSet
{
private:
    int m_length; // Элемент  уже не допускаемый множеством
    bool* m_set;  // Булевый вектор - представление множества

public:
    CBitSet(int nlength);
        // Создать множество, допускающее элементы от 0..nlength-1
    CBitSet(const CBitSet& right);
        // Конструктор присваивания
    CBitSet& operator= (const CBitSet& right);
        // Конструктор копирования

    int getLength() const;
        // Узнать размерность множества

    bool check(int index) const;
        // Проверить наличие в множестве элемента index
    void set(int index);
        // Включить в множество элемент index
    void reset(int index);
        // Исключить элемент index из множества
    void reset();
        // Исключить все элементы

    bool operator< (const CBitSet &right)  const;
        // Сравнить два множества
    operator += (const CBitSet &right);
        // Объединить два множества

    ~CBitSet(void);
};
```

7.6. Класс “CListWord”

```
/* CListWord - класс предназначен для хранения в АВЛ - дереве  пары - строка и число */

typedef map<CString, int> MAPSTRING;

class CListWord
{
private:
    vector<char*> m_names; // Список имен
    MAPSTRING m_idkeys;
    // АВЛ-дерево, отсортированное по первому ключу

public:
    CListWord(void);
    ~CListWord(void);

    CString getListNames();
    // Получить список всех имен, записанный в строку, с символами конца строки

    void getVectorNames(vector<const char*>& list);
    // Получить список всех имен

    int getCount() const;
    // Узнать количество пар

    int addWord(const char *name);
    // Получить id по имени, если имени нет, то оно добавляется

    int findWord(const char *name);
    // Найти id по имени, если имени нет, то возвращает -1

    const char* findById(int id);
     // Найти имя по индексу

    void clearList();
    // Очистить пары

    bool deleteByMask(const CBitSet* set);
    // Удалить пары, в которых индексу не из множества set
};
```

7.7. Файл “vecutils.h”

```
// Это вспомогательный файл, содержащий в себе одну функцию
// Шаблон, позволяющий удалять из вектора по маске

template <class TVecElem>
bool deleteVectorByMask(vector<TVecElem*> *vec, const CBitSet* mask)
{
    if (vec->size() != mask->getLength()) return false;

    vector<TVecElem*> old;
    old.clear();
    old.swap(*vec);

    for (int i = 0; i < mask->getLength(); i++)
        if (mask->check(i))
            vec->push_back(old[i]);
        else
            delete (old[i]);

    return true;
}
```

8. Листинги

Ниже приведены листинги описанных выше классов.

8.1. Класс “CAutomaton”

```
/* CAutomaton - класс, который хранит автомат. Он позволяет считывать из потока данных описание НКА, проверять детерминированность, приводить к ней и удалять недостижимые (лишние) состояния, а также выводить полученное описание ДКА в поток */


char* NAMEELSEEVENT = "ELSE";
char* NAMEANYEVENT = "ANY";

CAutomaton::CAutomaton(void)
{
    m_eventsnames = new CListWord();
    m_statesnames = new CListWord();
    m_vstates.clear();

    m_automateName = NULL;
    int IDANYEVENT = getEventIndexByName(NAMEANYEVENT);
    int IDELSEEVENT = getEventIndexByName(NAMEELSEEVENT);
}

CAutomaton::CAutomaton(const char* name)
{
    m_eventsnames = new CListWord();
    m_statesnames = new CListWord();
    m_vstates.clear();

    m_automateName = strdup(name);
    int IDANYEVENT = getEventIndexByName(NAMEANYEVENT);
    int IDELSEEVENT = getEventIndexByName(NAMEELSEEVENT);
}

CAutomaton::~CAutomaton(void)
{
    clearAutomaton();
    delete m_eventsnames;
    delete m_statesnames;
}
```

```
int CAutomaton::countStates()
    // Количество состояний
{
    return m_vstates.size();
}

int CAutomaton::countEvents()
    // Количество событий
{
    return m_eventsnames->getCount();
}

int CAutomaton::countUnachievable()
    // Количество недостижимых состояний
{
    CBitSet* reach = findAchievable();

    int cnt = 0;
    for (int i = 0; i < countStates(); i++)
        if (!reach->check(i)) cnt++;

    delete reach;

    return cnt;
}

CState* CAutomaton::getStateByIndex(int index)
    // Получить состояние по его id (индексу)
{
    if ((index >= 0) && (index < countStates())) return m_vstates[index];
    else return NULL;
}

void CAutomaton::createState()
    // Создать состояние
{
    m_vstates.push_back(new CState(this, countStates()));
}
```

```
CState* CAutomaton::getStateByName(const char* name)
    // Получить состояние по его имени
{
    int index = m_statesnames->addWord(name); // !?
    if (index >= countStates()) createState();

    return getStateByIndex(index);
}

int CAutomaton::getEventIndexByName(const char* name)
    // Получить индекс события  по имени
{
    return m_eventsnames->addWord(name);
}

const char* CAutomaton::getEventNameByIndex(int idevent)
    // Получить имя события по индексу
{
    return m_eventsnames->findById(idevent);
}

const char* CAutomaton::getStateNameByIndex(int index)
    // Получить имя состояния по его индексу
{
    return m_statesnames->findById(index);
}

CState* CAutomaton::createStateRandom()
    // Создать состояние со "случайным" именем
{
    CString ss;
    ss.Format("CState%i", countStates());
    return getStateByName(ss);
}

void CAutomaton::scanLink(CLexems *lex, CState *state)
    // Считывание переходов из потока lex и запись их в состояние state
{
    lex->compare(lex -> getCurrentLexem(), TL_IDENTY);

    char *eventname = strdup(lex->getString());
    lex->compare(lex -> nextLexem(), TL_IDENTY);
```

```
    CState* idstate = getStateByName(lex->getString());

    state->addCrossing(getEventIndexByName(eventname), idstate->getThisId());
    lex->compare(lex -> nextLexem(), TL_SEMICOLON);

    lex -> nextLexem();
    delete eventname;
}

void CAutomaton::scanState(CLexems *lex)
    // Считывание состояния из потока lex
{
    bool w_start = false, w_final = false, w_state = false, w_def = false;

    while (!w_def) {
        switch (lex -> getCurrentLexem()) {
            case TL_KEYSTART:
                    if (w_start) lex->throwBad("Double start definition");
                    w_start = true; break;
            case TL_KEYFINAL:
                    if (w_final) lex->throwBad("Double final definition");
                    w_final = true; break;
            case TL_KEYSTATE:
                    if (w_state) lex->throwBad("Double state definition");
                    w_state = true; break;
            case TL_IDENTY: w_def = true; break;
            default: lex->throwBad("Strange symbol defined");; break;
        };
        lex->nextLexem();
    };

    if (!w_state) lex->throwBad("State not defined");

    CState *state;
    state = getStateByName(lex->getString());

    state->setFinalProperty(w_final);
    state->setStartProperty(w_start);

    lex->compare(lex -> getCurrentLexem(), TL_OPENBRACE);

    lex->nextLexem();
    while (lex -> getCurrentLexem() != TL_CLOSEBRACE) scanLink(lex, state);
```

```
    state->processAnyState();
    extern IDELSEEVENT;
    if (!state->checkELSEEVENTLink())
            state->addCrossing(IDELSEEVENT, state->getThisId());

    lex -> nextLexem();

}

void CAutomaton::scanAutomaton(CLexems *lex)
    // Считывание автомата из потока lex
{
    clearAutomaton();

    lex->compare(lex->getCurrentLexem(), TL_IDENTY);

    m_automateName = strdup(lex->getString());

    lex->compare(lex->nextLexem(), TL_OPENBRACE);

    lex->nextLexem();
    while (lex->getCurrentLexem() != TL_CLOSEBRACE) scanState(lex) ;
    lex->compare(lex->nextLexem(), TL_ENDFILE);

}

void CAutomaton::clearAutomaton()
    // Очистить автомат
{
    for (int i = 0 ; i < countStates(); i++) delete m_vstates[i];
    m_vstates.clear();

    int IDANYEVENT = getEventIndexByName(NAMEANYEVENT);
    int IDELSEEVENT = getEventIndexByName(NAMEELSEEVENT);

    delete m_automateName;
    m_automateName = NULL;
}

bool CAutomaton::loadFromFile(const char* name)
    // Считать описание автомата из файла
{
```

```
    try {
        ifstream fin(name);
        fin >> (*this);
        fin.close();
    }
    catch (...) {
        clearAutomaton();
        return false;
    };
    return true;
}

bool CAutomaton::saveToFile(const char *name)
    // Сохранить описание автомата в файл
{
    try {
        ofstream output(name);
        if (!output) throw "File is not accessible";
        output << (*this);
        output.close();
    }
    catch (...) { return false; };
    return true;
}

bool CAutomaton::checkDeterminate()
    // Проверить детерминированность
{
    for (int i = 0; i < countStates(); i++)
    if (!(getStateByIndex(i)->checkStateDeterminate())) return false;

    return true;
}

bool CAutomaton::turnToDeterminate()
    // Привести к детерминированному
{
    C2MapBitSet listenum;

    CBitSet a = CBitSet(countStates());
    a.reset();

    for (int i = 0; i < countStates(); i++) {
        a.set(i);
```

```
        listenum.add(a, i);
        a.reset(i);
    };

    int cnt = countStates();

    i = 0;
    while (i < countStates()) {
        CState* state;
        state = getStateByIndex(i);
        if (!(state->checkStateDeterminate())) {
                if (!state->transformToDeterminate(&listenum, cnt)) return false;
        };
        i++;
    }
    return true;
}

CBitSet* CAutomaton::findAchievable()
    // Найти множество всех достижимых вершин
{
    bool existanystart = false;

    CBitSet* reach = new CBitSet(countStates());
    reach->reset();

    for (int i = 0; i < countStates(); i++)
    {
        CState* state = getStateByIndex(i);
        if (state->checkStartProperty()) {
            state->goneRound(reach);
            existanystart = true;
        };
    }
    if (!existanystart) {
        for (int i = 0; i < countStates(); i++) reach->set(i);
    };
    return reach;
}

void CAutomaton::deleteStatesByMask(CBitSet* set)
    // Удалить состояния по маске
{
    if (set->getLength() != countStates()) return;
```

```
    vector<int> newids(countStates());
    int newid = 0;
    for (int i = 0; i < countStates(); i++)
        if (set->check(i)) {
            newids[i] = newid;
            newid ++;
        } else newids[i] = -1;

    m_statesnames->deleteByMask(set);
    deleteVectorByMask<CState> (&m_vstates, set);

    for (i = 0; i<countStates(); i++) {
        m_vstates[i]->m_myOwnId = i;
        m_vstates[i]->refreshLinks(newids);
    };
}

bool CAutomaton::deleteUnachievable()
    // Удалить недостижимые вершины
{
    CBitSet* reach = findAchievable();
    deleteStatesByMask(reach);
    delete reach;
    return true;
}

ostream& operator<< (ostream& out, CAutomaton* aut)
    // Вывести описание автомата в поток
{
    return (aut->operator <<(out));
}

ostream& operator<< (ostream& out, CAutomaton& aut)
    // Вывести описание автомата в поток
{
    return (aut.operator <<(out));
}

ostream& CAutomaton::operator<< (ostream& out)
    // Вывести описание автомата в поток
{
    out << m_automateName << " {\n\n";
```

```
    for (int i = 0; i < countStates(); i++) {
        CState *state = getStateByIndex(i);
        out << state << "\n";
    };
    out << "}  // End of " << m_automateName;
    return out;
}

istream& operator>> (istream& sin, CAutomaton* aut)
    // Считать описание автомата из потока
{
    return (aut->operator >>(sin));
}

istream& operator>> (istream& sin, CAutomaton& aut)
    // Считать описание автомата из потока
{
    return (aut.operator >>(sin));
}

istream& CAutomaton::operator>> (istream& sin)
    // Считать описание автомата из потока
{
    CLexems* lex;
    try {
        lex = new CLexems(sin);
        scanAutomaton(lex);
        delete lex;
    }
    catch (CString err) {
        if (lex != NULL) delete lex;
        clearAutomaton();
        throw err;
    };
    return sin;
}

void CAutomaton::getNamesStates(vector<const char*>& list)
    // Получить список имен состояний
{
    m_statesnames->getVectorNames(list);
}
```

```
void CAutomaton::getNamesEvents(vector<const char*>& list)
    // Получить список имен событий
{
    m_eventsnames->getVectorNames(list);
}

void CAutomaton::getNamesUnachievable(vector<const char*>& list)
    // Получить список имен недостижимых вершин
{
    CBitSet* reach = findAchievable();
    list.clear();

    for (int i = 0; i < countStates(); i++)
        if (!reach->check(i)) list.push_back(m_statesnames->findById(i));
    delete reach;
}
```

8.2. Класс “CState”

```
/* CState - класс хранит состояние автомата. Он содержит данные о переходах из
этих состояний, а также различные методы, вспомогательные по отношению к методам
класса CAutomaton */

int IDELSEEVENT = 1;
int IDANYEVENT = 0;

bool POINTless(POINT p1, POINT p2)
{
    return ((p1.x < p2.x) || ((p1.x == p2.x) && (p1.y < p2.y)));
}

CState::CState(CAutomaton *par, int idthat)
// Создать вершину заданного автомата и заданным indexом
{
    if (par == NULL) throw "Null pointer assignment";

    m_liststates = par;
    m_myOwnId = idthat;

    m_property_final = false;
    m_property_start = false;
    m_idevents.clear();
    m_ELSEEVENTlink = false;
}

CState::~CState(void)
{
    m_idevents.clear();
}

int CState::getThisId()
// Узнать индекс состояния
{
    return m_myOwnId;
}

void CState::addCrossing(int event, int nextstate)
// Добавить переход
{
    POINT c = {event, nextstate};
    if (event == IDELSEEVENT) m_ELSEEVENTlink = true;
    m_idevents.push_back(c);
}

bool CState::checkELSEEVENTLink()
// Проверить есть ли переход по событию ELSEEVENT
{
    return m_ELSEEVENTlink;
}

bool CState::checkStateDeterminate()
// Проверить состояние на детерминированность
{
    sortLinks();

    for (int i = 0; i < getCountLinks() - 1; i++)
        if (m_idevents[i].x == m_idevents[i + 1].x) return false;

    return true;
}
```

```
bool CState::transformToDeterminate(C2MapBitSet* numerate, int countStart)
// Преобразовать состояние в детерминированное
{
    sortLinks();

    int i = 0, j = 0;

    while (i < getCountLinks()) {
        j = i;
        while ((j < getCountLinks() - 1) &&
                (m_idevents[j].x == m_idevents[j + 1].x)) j++;

        if (j != i) {
            CBitSet set = CBitSet(countStart);
            for (int k = i; k <= j; k++) {
                set += *(numerate->findBySecond(m_idevents[k].y));
            };

            int idnew = (numerate->findByFirst(set));
            CState* newstate;

            if (idnew < 0) {
                newstate = m_liststates->createStateRandom();
                idnew = newstate->getThisId();
                numerate->add(set, idnew);

                for (int k = i; k <= j; k++)
                    newstate->unionWith(
                        m_liststates->getStateByIndex(m_idevents[k].y));
            };


            while (i <= j) {
                m_idevents[i].y = idnew;
                i++;
            };
        }
        else i++;
    };

    sortLinks();

    return true;
}

void CState::processAnyState()
// Обработать ANY переходы
{
    vector <POINT> idnew;
    idnew.clear();
    idnew.swap(m_idevents);

    for (int i = 0; i < (int) idnew.size(); i++)
        if (idnew[i].x == IDANYEVENT) {
            for (int j = 0; j < (int) idnew.size(); j++)
            if (idnew[j].x != IDANYEVENT) addCrossing(idnew[j].x, idnew[i].y);
            addCrossing(IDELSEEVENT, idnew[i].y);
        } else addCrossing(idnew[i].x, idnew[i].y);
}

void CState::sortLinks()
// Отсортировать список внешних воздействий
{
    std::sort(m_idevents.begin(), m_idevents.end(), POINTless);

    vector <POINT> idnew;
    idnew.clear();
    idnew.swap(m_idevents);
```

```
    for (int i = 0; i < (int) idnew.size(); i++) {
        POINT c = idnew[i];
        if (i == 0) m_idevents.push_back(idnew[i]);
        else
        if (POINTless(idnew[i-1], idnew[i])) m_idevents.push_back(idnew[i]);
    };

    idnew.clear();
}

typedef vector<POINT> intvector;
typedef intvector::iterator iterint;

int compare(iterint& lf, iterint& rg, const iterint& lfend,
                    const iterint& rgend)
// Сравнить данные, записанные в векторе. За концом вектора - плюс бесконечность
{
    if (lf == lfend) return 1;
    if (rg == rgend) return -1;

    int li = lf->x, ri = rg->x;

    if (li < ri) return -1;
    if (ri > li) return 1;
    return 0;
}

void addaoth(iterint& itf, intvector& oth, intvector& addon)
// Добавить все ссылки из второго вектора, которые пропущены в первом
{
    iterint mv = oth.begin();

    POINT c;
    while (mv != oth.end()) {

        if (mv->x != IDELSEEVENT) break ;

        c.x = itf->x;
        c.y = mv->y;
        addon.push_back(c);

        mv++;
    };
}

void skipeq(iterint& itf, intvector& itwhere)
// Пропустить одинаковые числа в векторах
{
    if (itf == itwhere.end()) return;

    int sp = itf->x;

    while (itf != itwhere.end()) {
        int nw = itf->x;
        if (nw != sp) return;
        itf++;
    };
}

void CState::unionWith(CState* right)
// Объединить два состояния
{
    if (right == NULL) return ;

    m_property_final |= right->m_property_final;
    m_ELSEEVENTlink |= right->m_ELSEEVENTlink;
    sortLinks();
```

```
    right->sortLinks();

    intvector addon;
    addon.clear();

    iterint itf, its;

    itf = m_idevents.begin();
    its = right->m_idevents.begin();

    if ((itf != m_idevents.end()) && (itf->x == IDELSEEVENT))
                skipeq(itf, m_idevents);
    if ((its != right->m_idevents.end()) && (its->x == IDELSEEVENT))
                 skipeq(its, right->m_idevents);

    while ((itf != m_idevents.end()) || (its != right->m_idevents.end())) {

        switch (compare(itf, its, m_idevents.end(), right->m_idevents.end())) {
            case -1: addaoth(itf, right->m_idevents, addon); itf++; break;
            case  0: skipeq(itf, m_idevents);
                        skipeq(its, right->m_idevents); break;
            case  1: addaoth(its, m_idevents, addon); its++; break;
        };

    };

    m_idevents.insert(m_idevents.end(),
        right->m_idevents.begin(),
        right->m_idevents.end());

    m_idevents.insert(m_idevents.end(), addon.begin(), addon.end());

}

void CState::setFinalProperty(bool p_final)
// Установить свойство состояния быть конечным
{
    m_property_final = p_final;
}

void CState::setStartProperty(bool p_start)
// Установить свойство состояния быть начальным
{
    m_property_start = p_start;
}

ostream& operator<< (ostream& out, CState* state)
// Сохранить состояние в выходной поток
{
    return (state->operator <<(out));
}

ostream& operator<< (ostream& out, CState& state)
// Сохранить состояние в выходной поток
{
    return (state.operator <<(out));
}

ostream& CState::operator<< (ostream& out)
// Сохранить состояние в выходной поток
{
    out << "\t";

    sortLinks();

    if (m_property_start) out << "start ";
    if (m_property_final) out << "final ";
```

```
    out << "state " << m_liststates->getStateNameByIndex(m_myOwnId)
               << " {\n";

    for (int j = 0; j < 2; j++)
    for (int i = 0; i < getCountLinks(); i++)
    if ((m_idevents[i].x != IDELSEEVENT) ^ j) {
        out << "\t\t";
        out << m_liststates->getEventNameByIndex(m_idevents[i].x);
        out << '\t';
        out << m_liststates->getStateNameByIndex(m_idevents[i].y);
        out << ";\n";
    };

    out << "\t} // End of state " <<
        m_liststates->getStateNameByIndex(m_myOwnId) << "\n";


    return out;
}

bool CState::checkStartProperty()
// Проверить свойство состояния быть начальным
{
    return m_property_start;
}

bool CState::checkFinalProperty()
// Проверить свойство состояния быть конечным
{
    return m_property_final;
}


void CState::goneRound(CBitSet* visited)
// Посетить всех соседей и отметить их в множестве visited
{
    int id = getThisId();
    if (visited->check(id)) return;
    visited->set(id);

    for (int i = 0; i < getCountLinks(); i++) {
        CState *state = m_liststates->getStateByIndex(m_idevents[i].y);
        state->goneRound(visited);
    }
}

int CState::getCountLinks() const
// Получить количество переходов
{
    return m_idevents.size();
}

void CState::refreshLinks(vector<int> &newids)
// Изменить идентификаторы состояний, в которые делаются переходы. Число "-1"
означает, что состояние удаляется
{
    for (int i = 0; i < getCountLinks(); i++)
        m_idevents[i].y = newids[m_idevents[i].y];
}

bool assignment(int& lastval, int newval)
// Возможно ли присваивание
{
    if (newval == -1) newval = lastval;
    if (lastval == -1) lastval = newval;
    return (lastval == newval);
}
```

```
int CState::crossByEvent(int eventid)
// Номер состояния, куда совершен переход по внешнему воздействию
{
    int onAny = -1, onElse = -1, onThis = -1;

    for (int i = 0; i < getCountLinks(); i++) {
        if (m_idevents[i].x == eventid)
                if (!assignment(onThis, m_idevents[i].y)) return -1;
        if (m_idevents[i].x == 1)
                if (!assignment(onElse, m_idevents[i].y)) return -1;
        if (m_idevents[i].x == 0)
                if (!assignment(onAny, m_idevents[i].y)) return -1;
    };

    if (!assignment(onThis, onAny)) return -1;
    if (onThis != -1) return onThis;
        else return onElse;

}
```

8.3. Класс “CLexems”

```
/* CLexems - класс предназначен для разбора файлов на лексемы. Он используется
для чтения входного потока и его разбора */

const int maxbuffer = 10000;
const int CH_ENDFILE = 0;


TLexem CLexems::getCurrentLexem()
// Узнать текущую лексему
{
    return m_curLexem;
}

bool CLexems::ifSkipChar(char ch)
// Проверить символ на принадлежность к "белым" пробелам
{
    if (m_curLexem == TL_ENDFILE) return false;

    switch (ch) {
        case ' ': return true; break;
        case '\n': return true; break;
        case 13: return true; break;
        case '\t': return true; break;
        default: return false;
    };
}

void CLexems::nextChar()
// Считать следующий символ
{
    if (m_curLexem == TL_ENDFILE) return;


    if (m_instream->eof()) m_curchar = CH_ENDFILE;
    else {
        try {
            m_instream->read(&m_curchar, 1);
            m_countread++;
        }
        catch (...) {
            m_curchar = CH_ENDFILE;
        };
    };
}

void CLexems::scanWord()
// Считать слово
{
    int i = 0;
    while (isalpha(m_curchar) || isdigit(m_curchar) || (m_curchar == '_')) {
         m_buffer[i++] = m_curchar;
         nextChar();
    };
    m_buffer[i] = 0;

    m_curLexem = TL_IDENTY;
    if (strcmp(m_buffer, "state") == 0) m_curLexem = TL_KEYSTATE;
    if (strcmp(m_buffer, "final") == 0) m_curLexem = TL_KEYFINAL;
    if (strcmp(m_buffer, "start") == 0) m_curLexem = TL_KEYSTART;
};

void CLexems::skipToLineEnd()
// Пропустить все символы до конца строки
{
```

```
    while ((m_curchar != '\n') && (m_curchar != CH_ENDFILE)) nextChar();
    nextChar();
}

void CLexems::skipToCommentEnd()
// Пропустить все символы до символов */
{
    int cur = 0;
    while ((m_curchar != CH_ENDFILE) && (cur != 2)) {
        nextChar();
        if (m_curchar == '*') cur = 1;
        if ((cur == 1) && (m_curchar == '/')) cur = 2;
    };

    nextChar();
}

TLexem CLexems::nextLexem()
// Прочитать следующую лексему и узнать её
{
    if (m_curLexem == TL_ENDFILE) return m_curLexem;
    m_countprev = m_countread;

    while (ifSkipChar(m_curchar)) nextChar();

    if (isalpha(m_curchar)) scanWord();
    else
    switch (m_curchar) {
        case '_': scanWord(); break;
        case '{': m_curLexem = TL_OPENBRACE; nextChar(); break;
        case '}': m_curLexem = TL_CLOSEBRACE; nextChar(); break;
        case ';': m_curLexem = TL_SEMICOLON; nextChar(); break;
        case ',': m_curLexem = TL_COMMA; nextChar(); break;
        case '/': nextChar();
                switch(m_curchar) {
                    case '/': skipToLineEnd(); m_curLexem = nextLexem(); break;
                    case '*': skipToCommentEnd();
                        m_curLexem = nextLexem(); break;
                    default: m_curLexem = TL_UNKNOWN; break;
                };
                break;
        case CH_ENDFILE: m_curLexem = TL_ENDFILE; break;
        default: m_curLexem = TL_UNKNOWN; nextChar(); break;
    }

    return m_curLexem;
}

const char* CLexems::getString()
// Получить текущее слово
{
    return m_buffer;
}

CLexems::CLexems(void)
{
    m_buffer = new char[maxbuffer];
    m_buffer[0] = 0;
    m_curchar = ' ';
    m_curLexem = TL_OPENFILE;
    m_countread = 0;
    m_countprev = 0;
    m_instream = NULL;
}

CLexems::CLexems(istream& s_in)
// Создать класс, и сразу подать ему на вход поток
{
```

```
    m_buffer = new char[maxbuffer];
    m_buffer[0] = 0;
    m_curchar = ' ';
    m_instream = &s_in;
    m_curLexem = TL_OPENFILE;
    m_countread = 0;
    m_countprev = 0;
    nextLexem();
}

void CLexems::openStream(istream& s_in)
// Открыть поток
{
    m_instream = &s_in;
    m_curLexem = TL_OPENFILE;
    m_countread = 0;
    m_countprev = 0;
    nextLexem();
}

CLexems::~CLexems(void)
{
    delete m_buffer;
}


void CLexems::compare(TLexem now, TLexem exp)
// Сравнить две лексемы - одна текущая, другая ожидаемая
{
    if (now != exp)
        switch (exp) {
            case TL_ENDFILE: throwBad("Expected end of description"); break;
            case TL_OPENBRACE: throwBad("Expected open brace"); break;
            case TL_CLOSEBRACE: throwBad("Expected close brace"); break;
            case TL_IDENTY: throwBad("Expected identifier"); break;
            case TL_SEMICOLON: throwBad("Expected semicolon"); break;
            case TL_KEYSTATE: throwBad("Expected keyword state"); break;
            case TL_KEYFINAL: throwBad("Expected keywrod final"); break;
            case TL_KEYSTART: throwBad("Expected keyword start"); break;
            case TL_COMMA: throwBad("Expected comma"); break;
        };
}

TLexem CLexems::throwBad(CString st)
// Вызвать исключение, обругавшись на вход
{
    CString num;
    num.Format("%i ", m_countprev);
    throw num + st;
}
```

8.4. Класс “C2MapBitSet”

```
/* C2MapBitSet - класс предназначен для работы с парами, каждая из которых состоит из двух элементов - множества чисел и  числа.  При этом допустимо искать пару, как по первому элементу, так и по второму. Пары хранятся в виде АВЛ - дерева */

typedef map<CBitSet, int> MAPTOFS;
typedef map<int, CBitSet> MAPTOSEC;

C2MapBitSet::C2MapBitSet(void)
{
}

C2MapBitSet::~C2MapBitSet(void)
{
}

void C2MapBitSet::clear()
// Очистить от всех ключей
{
    m_first_sort.clear();
    m_second_sort.clear();
}

int C2MapBitSet::findByFirst(const CBitSet& fs)
// Найти по первому ключу
{
    MAPTOFS::iterator it = m_first_sort.find(fs);
    if (it == m_first_sort.end()) return -1;

    return it->second;
}


CBitSet const* C2MapBitSet::findBySecond(const int sec)
// Найти по второму ключу
{
    MAPTOSEC::iterator it = m_second_sort.find(sec);
    if (it == m_second_sort.end()) return NULL;

    return &(it->second);
}

void C2MapBitSet::add(const CBitSet& fs, const int sec)
// Добавить пару
{
    m_first_sort.insert(MAPTOFS::value_type(fs, sec));
    m_second_sort.insert(MAPTOSEC::value_type(sec, fs));
}
```

8.5. Класс “CBitSet”

```
/* CBitSet - класс предназначен для работы с множествами (объединение, включение
и исключение элементов) */

CBitSet::CBitSet(int nlength)
// Создать множество, допускающее элементы от 0..nlength-1
{
    m_length = nlength;
    m_set = new bool[nlength];
    for (int i = 0; i < nlength; i++) m_set[i] = false;
}

bool CBitSet::check(int index) const
// Проверить наличие в множестве элемента index
{
    if ((index < 0) || (index >= m_length)) return false;
    return m_set[index];
}

int CBitSet::getLength() const
// Узнать размерность множества
{
    return m_length;
}

CBitSet::~CBitSet(void)
{
    delete m_set;
}

void CBitSet::set(int index)
// Включить в множество элемент index
{
    m_set[index] = true;
}

void CBitSet::reset(int index)
// Исключить элемент index из множества
{
    m_set[index] = false;
}

void CBitSet::reset()
// Исключить все элементы
{
    for (int i=0; i < m_length; i++)
        m_set[i] = false;
}

bool CBitSet::operator< ( const CBitSet &right) const
// Сравнить два множества
{
    if (m_length < right.m_length) return true;
    if (m_length > right.m_length) return false;

    for (int i = 0; i < m_length; i++)
    if  (m_set[i] ^ right.m_set[i]) {
        if (m_set[i]) return true;
        else return false;
    };

    return false;
}
```

```
CBitSet::operator += (const CBitSet &right)
// Объединить два множества
{

    int a = m_length;
    if (right.m_length < a)
        a = right.m_length;

    for (int i = 0; i < a; i++)
        m_set[i] |= right.m_set[i];

    return 0;
}

CBitSet::CBitSet(const CBitSet& right)
// Конструктор присваивания
{
    m_length = right.m_length;

    m_set = new bool[m_length];
    memcpy(m_set, right.m_set, m_length*sizeof(bool));
}

CBitSet& CBitSet::operator =(const CBitSet& right)
// Конструктор копирования
{
    if (this != &right) {
        m_length = right.m_length;
        delete m_set;

        m_set = new bool[m_length];
        memcpy(m_set, right.m_set, m_length*sizeof(bool));
    };

    return *this;
}
```

8.6. Класс “CListWord”

```
/* CListWord - класс предназначен для хранения в АВЛ - дереве пары - строка и
число */

CListWord::CListWord(void)
{
    m_idkeys.clear();
    m_names.clear();
}

CListWord::~CListWord(void)
{
    clearList();
}

int CListWord::addWord(const char *name)
// Получить индекс по имени, если имени нет, то оно добавляется
{
    MAPSTRING::iterator it = m_idkeys.find(name);
    if ((it != m_idkeys.end()) && (strcmp(it->first, name) == 0))
                  return it->second;

    char* sname = strdup(name);

    int id = getCount();
    m_names.push_back(sname);
    m_idkeys.insert(MAPSTRING::value_type(sname, id));

    return id;
}

int CListWord::findWord(const char *name)
// Найти индекс по имени, если имени нет, то возвращает -1
{
    MAPSTRING::iterator it = m_idkeys.find(name);
    if ((it != NULL) && (strcmp(it->first, name) == 0)) return it->second;

    return -1;
}

const char* CListWord::findById(int id)
// Найти имя по id
{
    if ((id < 0 ) || (id >= getCount())) return NULL;
    return m_names[id];
}

void CListWord::clearList()
// Очистить пары
{

    for (int i = 0; i < getCount(); i++) {
        char* name = m_names[i];
        delete name;
    }

    m_names.clear();
    m_idkeys.clear();
}


bool CListWord::deleteByMask(const CBitSet* set)
// Удалить пары, в которых id не из множества set
{
```

```
    if (!deleteVectorByMask<char> (&m_names, set)) return false;

    m_idkeys.clear();
    for (int i = 0; i < getCount(); i++)
           m_idkeys.insert(MAPSTRING::value_type(m_names[i], i));

    return true;
}

int CListWord::getCount() const
// Узнать количество пар
{
    return m_names.size();
}

void CListWord::getVectorNames(vector<const char*>& list)
// Получить список всех имен
{
    list.clear();
    for (int i = 0; i < getCount(); i++) list.push_back(m_names[i]);
}

CString CListWord::getListNames()
// Получить список всех имен, записанные в строку, разделенные концом строки
{
    CString res;

    int len = 0;
    for (int i = 0; i < getCount(); i++) len += 3 + strlen(m_names[i]);

    res.GetBuffer(len);

    res.Empty();
    for (i = 0; i < getCount(); i++) {
        res += CString(m_names[i]);
        res += CString('\n');
    };

    return res;

}
```

Литература

1. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
2. *Фридл Дж.* Регулярные выражения. СПб.: Питер, 2002.
3. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.
4. *Страуструп Б.* Язык программирования C++. М.: БИНОМ, СПб.: Невский диалект, 2001.
5. *Гасфилд Д.* Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, 2003.