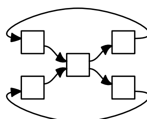
Автоматный серпентарий

© 2006 Дмитрий Павлов
SMTP: rain.ifmo.ru, pavlov

1. Введение. Рассмотрим сильно связную сеть синхронизированных конечных автоматов, по рёбрам которой можно передавать символы некоторого конечного алфавита. Количество состояний каждого автомата и размер алфавита ограничены некоторым заранее заданным многочленом от максимума суммы полустепени исхода и захода среди всех вершин и не зависит от количества вершин и рёбер в графе. Автоматы не знают, какие вершины являются их соседями, и могут лишь передавать им информацию без обратной связи. В такой модели можно за линейное или квадратичное время решить следующие задачи:

- передать заданное сообщение всем узлам;
- построить ориентированное остовное дерево, в котором рёбра идут к корню или от корня;
- выполнить на получившемся дереве поиск в глубину;
- передать сообщение от одного узла к другому;
- синхронизировать узлы (заставить их одновременно выполнить некоторое действие).

Пример сети:



Основная проблема при решении этих задач состоит в том, что два узла в сети могут быть сколь угодно далеки друг от друга, а передавать по сети произвольные числа нельзя, так как размер алфавита не зависит от количества вершин в графе, поэтому использование традиционных алгоритмов вроде поиска в глубину становится невозможным.

Отметим, что на практике конечность числа состояний является нереалистичным ограничением, ибо передача объёма данных, пропорциональных по своему размеру логарифму количества узлов, вполне возможна. Поэтому данная программа не имеет практического значения в силу того, что для практических задач такого рода существуют гораздо более простые алгоритмы.

Данная программа реализует эти алгоритмы, эмулируя их работу на виртуальной машине. Все алгоритмы взяты из статьи Even, Litman, Winkler, *Journal of Algorithms* **24** (1997), 158–170.

2. Программа написана на языке `CWEB`, реализующем концепцию грамотного программирования, которую предложил Donald E. Knuth. Описание языка можно найти в книге Donald E. Knuth и Silvio Levy, The `CWEB` System of Structured Documentation (Reading, Massachusetts: Addison-Wesley, 1993). Программа на языке `CWEB` состоит из секций. Каждая секция имеет номер и состоит из описания, за которым следует определение модуля — куска программы на языке C, который может внутри себя использовать другие модули.

Имена модулей заключаются в угловые скобки, перед закрывающейся угловой скобкой ставится номер секции, в который этот модуль впервые определяется. Один и тот же модуль может определяться в нескольких секциях, при этом соответствующие куски программ соединяются в один. Если модуль определяется в первый раз, то после его имени идёт знак '≡', а иначе '+≡'.

3. Общая структура программы. Заданы n одинаковых вычислительных узлов, каждый из которых является детерминированным конечным автоматом. Также заданы каналы связи, которые являются дугами, соединяющими пары узлов. Каналы являются односторонними, передача данных по ним может осуществляться только от первого узла ко второму. При этом граф, вершины которого — вычислительные узлы, а рёбра — каналы связи, должен быть сильно связным — от любой вершины можно пройти к любой другой вершине по дугам, соблюдая ориентацию дуг. Каждый вычислительный узел имеет несколько входных и выходных портов, количество которых ограничено заранее заданным числом. Каждый канал связи реализуется путём соединения одного из выходных портов начального узла с одним из входных портов конечного узла. При этом один и тот же порт не может использоваться несколькими каналами связи. Порты, к которым ничего не подсоединено, называются *мёртвыми*, а все остальные порты называются *живыми*. Конфигурация сети остаётся неизменной в течении всего времени работы.

Каждый узел знает, какие из его *входных* портов являются мёртвыми.

За один такт по каждому каналу можно передать символ из заранее заданного конечного алфавита, размер которого зависит от ограничения на число портов. При этом один из этих символов, который называется *нейтральным*, подаётся на вход всем мёртвым портам. На каждом такте вычислительный узел считывает данные, которые приняли все входные порты, и, основываясь на этих данных, изменяет своё состояние и подаёт некоторый набор символов на свои выходные порты. Число состояний каждого автомата конечно и зависит от ограничения на число портов. Все узлы работают одновременно, при этом их работа синхронизирована глобальным тактовым генератором. За один такт все узлы принимают данные, обрабатывают их и подают новые данные на свои выходные порты.

Отметим, что функциональность глобального тактового генератора легко эмулируется, если каналы позволяют передавать информацию с задержкой. Для этого на очередном шаге узел ждёт, пока на каждый входной канал не поступит символ, считывает его, и освобождает канал. Затем узел обрабатывает данные, после этого начинает выводить символы на выходные каналы, при этом, если выходной канал занят, узел ждёт, пока он освободится.

В начале каждый автомат находится в *нейтральном* состоянии — состоянии, в котором он ничего не делает и остаётся в нём до тех пор, пока на все входы поступает нейтральный символ. При этом на все выходные порты также подаётся нейтральный символ. Для того, чтобы запустить какое-то вычисление в сети, следует перевести один из узлов, который называется *корнем* или *корневым узлом*, в состояние, отличное от нейтрального.

Выпишем каркас программы.

```
#include <vector>
#include <cstdio>
#include <cassert>
  using namespace std;
  ⟨ Функции рисования 79 ⟩
  ⟨ Структуры данных 4 ⟩
  struct net {
    struct arc {
      int begn, begp, endn, endp;
    };
    vector⟨arc⟩ arcs;
    vector⟨node⟩ nodes;
    void step( )
    {
      for (int k = 0; k < arcs.size( ); k++) {
        arc a = arcs[k];
        nodes[a.endn].input[a.endp] = nodes[a.begn].output[a.begp];
      }
      for (int k = 0; k < nodes.size( ); k++) {
```

```
      nodes[k].output.assign(nodes[k].out, symbol());
      nodes[k].step();
    }
  }
  ⟨ Ввести сеть 68 ⟩
  int root;
  ⟨ Инициализировать сеть 70 ⟩
  ⟨ Нарисовать постоянную часть сети 71 ⟩
  ⟨ Обновить изображение сети 72 ⟩
};
```

4. ⟨ Структуры данных 4 ⟩ ≡

```
typedef int port;    /* тип данных для номера порта */
const int nil = −1;    /* обозначает несуществующий порт или отсутствие сигнала */
⟨ Вспомогательные структуры данных 6 ⟩
struct symbol {
  ⟨ Переменные символа 8 ⟩
  symbol()
  {
    ⟨ Инициализировать символ 31 ⟩
  }
  ⟨ Нарисовать символ 74 ⟩
};
```

Смотри также секцию 5.

Этот код используется в секции 3.

5. ⟨ Структуры данных 4 ⟩ +≡

```
struct node {
  typedef vector⟨bool⟩ vbool;
  int in, out;    /* количество входных и выходных портов */
  vbool is_alive;    /* является ли данный входной порт живым? */
  bool is_root;    /* является ли данный узел корневым? */
  vector⟨symbol⟩ input, output;    /* на каждом такте массив input содержит данные входных
      портов, массив output инициализирован нейтральными символами */
  enum {
    ⟨ Состояния узла 7 ⟩
  } state;    /* текущее состояние узла */
  enum r_state {
    neutral,    /* ничего не делаем */
    ⟨ Состояния корня 16 ⟩
  } root_state;    /* текущее состояние корня */
  bool start;    /* в момент запуска очередного алгоритма устанавливается в true для корня */
  ⟨ Переменные узла 9 ⟩
  void init(bool Is_root)
  {
    input.assign(in, symbol());
    output.assign(out, symbol());
    is_root = Is_root;
    state = transfer;
    root_state = neutral;
    start = false;
    ⟨ Инициализировать узел 10 ⟩
  }
  void step()
  {
    if (start) {
      start = false;
      switch (root_state) {
        ⟨ Запустить алгоритм 13 ⟩
      }
    }
    ⟨ Построить цикл 46 ⟩    /* важно, чтобы этот фрагмент кода стоял в начале */
    ⟨ Передать змей 17 ⟩
    ⟨ Построить дерево 14 ⟩
    ⟨ Выполнить обход 37 ⟩
    ⟨ Синхронизироваться 64 ⟩
  }
  ⟨ Обновить изображение узла 73 ⟩
};
```

6. Змеи. Основным ограничением рассматриваемой модели является ограниченная память у каждого узла. Это ограничение преодолевается путём использования *змей* — последовательностей символов, которые путешествуют по сети как единое целое. Каждый символ является номером некоторого выходного порта. Первый и последний символ последовательности снабжены специальными пометками для того, чтобы можно было выделить голову и хвост змеи из общего потока символов. За один такт каждый сегмент змеи продвигается к следующему узлу. Алгоритмы используют три вида змей: питоны, гадюки и кобры. Во время своего существования гадюки постоянно растут, питоны остаются неизменными, а кобры непрерывно укорачиваются. Змеи используются в качестве средства транспортировки сообщений во всех алгоритмах, кроме синхронизации.

Выпишем структуры данных для змей.

```
⟨ Вспомогательные структуры данных 6 ⟩ ≡
  struct snake     /* структура для хранения сегмента змеи */
  {
    port body;     /* очередной сегмент тела змеи; nil в случае отсутствия такового */
    bool is_head, is_tail;
    snake(port Body, bool Is_head, bool Is_tail)
    {
      body = Body;
      is_head = Is_head;
      is_tail = Is_tail;
    }
    snake()
    {
      body = nil;
      is_head = is_tail = false;
    }
    ⟨ Нарисовать сегмент змеи 75 ⟩
  };
```

Смотри также секции 61 и 62.

Этот код используется в секции 4.

7.
```
⟨ Состояния узла 7 ⟩ ≡
  transfer,     /* передаём змей */
```

Смотри также секции 38, 48, и 66.

Этот код используется в секции 5.

8.
```
⟨ Переменные символа 8 ⟩ ≡
  snake adder, python, cobra;     /* переменные для хранения сегментов змей */
```

Смотри также секции 30, 41, 50, 54, и 67.

Этот код используется в секции 4.

9. ⟨ Переменные узла 9 ⟩ ≡

```
enum    /* состояние передачи данного типа змеи */
{
   gap,     /* ничего не передаём */
   head,    /* передаём голову кобры */
   body,    /* передаём тело змеи */
   tail,    /* передаём хвост гадюки */
   ignore   /* игнорируем питона или кобру */
} adder, python, cobra;
```

Смотри также секции 11, 19, 22, 26, 32, 56, 63, и 69.

Этот код используется в секции 5.

10. ⟨ Инициализировать узел 10 ⟩ ≡

```
adder = python = cobra = gap;
```

Смотри также секции 12, 20, 23, 27, 33, и 57.

Этот код используется в секции 5.

11. Остовные деревья. Первый алгоритм строит на сети два вида деревьев. *Дерево истока* представляет собой набор рёбер, такой, что до любой вершины можно дойти из корня по рёбрам дерева ровно одним способом. Если в дереве истока есть ребро $u \to v$, то вершина u называется *отцом* вершины v, а вершина v называется *сыном* вершины u. У корня нет отца, а у любой другой вершины есть ровно один отец. У каждой вершины может быть произвольное число сыновей. Когда построение дерева истока завершено, каждая вершина знает, к какому из её входных портов подсоединено ребро, приходящее от её отца и какие из её выходных портов соответствуют рёбрам, ведущим к её сыновьям.

Остовные деревья управляют потоками информации во всех алгоритмах, в частности, определяют пути следования змей.

Дерево стока получается из дерева истока, если в определении последнего обратить направления рёбер. Таким образом, от каждой вершины можно пройти по рёбрам дерева стока до корня ровно одним способом. Для дерева стока вместо терминов «отец» и «сын» будем использовать термины «мать» и «дочь». Каждое из деревьев вначале содержит только корень. Постепенно к каждому из деревьев подсоединяются вершины, пока все вершины не войдут в дерево. Рёбра, которые уже добавлены в дерево, остаются в нём все оставшееся время.

Алгоритм состоит из нескольких одновременно работающих этапов. На первом этапе строится дерево истока, на втором корню отсылается вспомогательная информация, необходимая для построения дерева стока, на третьем этапе происходит построение дерева стока, а на четвёртом — проверяется готовность узлов. Этапы будут описаны последовательно.

Выпишем структуры данных для остовных деревьев.

⟨ Переменные узла 9 ⟩ +≡
port *father*;
/* порт, в который приходит ребро от отца или *nil*, если узел корневой или не входит в дерево */
vbool *is_son*; /* ведёт ли данный порт к сыну? */
port *mother*;
vbool *is_daughter*;

12. ⟨ Инициализировать узел 10 ⟩ +≡
father = *mother* = *nil*;
is_son.*assign*(*out*, *false*);
is_daughter.*assign*(*in*, *false*);

13. Алгоритм начинается с того, что корневой узел на каждом шаге рассылает односимвольных гадюк по всем своим выходным портам, при этом единственный символ гадюки соответствует номеру порта, по которому она уползла от корня. Гадюки, приползающие в корневой узел, игнорируются, а остальные узлы ведут себя по отношению к гадюкам следующим образом: если данный узел ещё не входит в дерево истока, то он включает себя в это дерево, запоминая при этом, что порт, через который к нему приползла гадюка, соответствует ребру, приходящему от его отца в дереве. Если при этом в узел вползло несколько гадюк, то выбирается любая из них. После присоединения узла к дереву алгоритм поступает также, как если бы узел получил гадюку, уже будучи включёнными в дерево. Поведение узла, включённого в дерево, описано ниже.

⟨ Запустить алгоритм 13 ⟩ ≡
case *tree*:
for (**int** $k = 0$; $k < out$; $k{+}{+}$) /* говорим соседям, что узел готов, смотри ниже */
output[*k*].*ready* = *true*;
break;

Смотри также секции 36, 45, и 58.

Этот код используется в секции 5.

14. ⟨ Построить дерево 14 ⟩ ≡

```
if (root_state ≡ tree) ⟨ Отослать волну гадюк 15 ⟩
```

Смотри также секцию 29.

Этот код используется в секции 5.

15. ⟨ Отослать волну гадюк 15 ⟩ ≡

```
for (int k = 0; k < out; k++) output[k].adder = snake(k, true, true);
```

Этот код используется в секциях 14, 44, и 45.

16. ⟨ Состояния корня 16 ⟩ ≡

```
tree,     /* строим дерево */
```

Смотри также секции 39, 49, и 65.

Этот код используется в секции 5.

17. ⟨ Передать змей 17 ⟩ ≡

```
if (father ≡ nil ∧ ¬is_root)
  for (int k = 0; k < in; k++)
    if (input[k].adder.is_head) {
      father = k;
      mutant = true;     /* смотри ниже */
      break;
    }
```

Смотри также секции 18, 21, и 24.

Этот код используется в секции 5.

18. Гадюки, которые приползают не от отца, игнорируются. Если же узел получил гадюку от своего отца, то он начинает ретранслировать эту гадюку на все свои выходные порты в неизменном виде, за исключением того, что в хвост этой гадюки он добавляет символ, соответствующий номеру выходного порта, через который он выпустил данную гадюку. Поскольку для выполнения последнего действия нам необходим один дополнительный такт времени, нельзя начать обработку новой гадюки сразу после того, как целиком получена старая гадюка, ибо в противном случае нам пришлось бы задействовать память неограниченного размера для хранения тех символов гадюк, которые ещё не отправлены через выходной порт. Поэтому, если голова новой гадюки получена на следующем такте после получения хвоста старой гадюки, новая гадюка игнорируется.

Отметим, что построенное дерево истока является деревом поиска в ширину. Каждый узел уже знает своего отца, но не своих сыновей. Информация о сыновьях будет получена позже.

Для построения дерева стока нам необходимо задействовать второй тип змей — питонов. Если узел, который ещё не входит в дерево стока, занимается ретрансляцией гадюки, то одновременно с рассылкой гадюк он высылает по всем портам питонов, состоящих из тех же символов, что и ретранслируемая гадюка. Каждый узел, не входящий в дерево стока, игнорирует всех вползающих в него питонов.

```
⟨ Передать змей 17 ⟩ +≡
  switch (adder) {
  case tail: adder = gap;
    for (int k = 0; k < out; k++) {
      output[k].adder = snake(k, false, true);
      if (mutate) output[k].python = snake(k, false, true);
    }
    mutate = false;
    break;
  case gap:
    if (father ≡ nil ∨ ¬input[father].adder.is_head) break;
    if (mutant) mutate = true;
    adder = body;
  case body: snake s = input[father].adder;
    if (s.is_tail) {
      s.is_tail = false;
      adder = tail;
    }
    for (int k = 0; k < out; k++) {
      output[k].adder = s;
      if (mutate) output[k].python = s;
    }
    break;
  }
  if (state ≡ search)      /∗ требуется при построении цикла ∗/
  {
    if (mother ≠ nil) output[mother].python = snake( );
    if (unready ≡ 0) {
      state = transfer;
      mutant = false;
      python = cobra = gap;
      unready = nil;
    }
  }
```

19. ⟨ Переменные узла 9 ⟩ +≡

```
bool mutant,    /* превращаем гадюк в питонов? */
mutate;    /* создаём в данный момент питона из гадюки? */
```

20. ⟨ Инициализировать узел 10 ⟩ +≡

```
mutant = mutate = false;
```

21. Каждый некорневой узел, который уже вошёл в дерево стока, ретранслирует вползающих в него питонов своей матери. Если узел уже передаёт питона, а в это время в него вползает другой питон, то последний игнорируется. Также, если несколько питонов вползают в узел одновременно, то для ретрансляции выбирается любой из них, а остальные игнорируются.

⟨ Передать змей 17 ⟩ +≡

```
if (python ≡ gap ∧ mother ≠ nil ∧ output[mother].python.body ≡ nil)
  for (int k = 0; k < in; k++)
    if (input[k].python.is_head) {
      python = body;
      python_port = k;
      break;
    }
if (python ≡ body ∧ (output[mother].python = input[python_port].python).is_tail) {
  python = gap;
  python_port = nil;
}
```

22. ⟨ Переменные узла 9 ⟩ +≡

```
port python_port;    /* порт, через который вползает питон */
```

23. ⟨ Инициализировать узел 10 ⟩ +≡

```
python_port = nil;
```

24. Если корневой узел получает питона, он трансформирует его в кобру. Эта кобра проделает тот же путь, что когда-то проделала гадюка, превратившаяся потом в питона, который дополз до корневого узла. Направить кобру по этому пути несложно, так как её первый символ указывает на выходной порт, по которому она должна покинуть корневой узел, второй символ — на порт, по которому она должна покинуть следующий узел, и так далее. Когда кобра достигает очередного узла, от её головы отделяется первый символ, соответствующий порту, через который подаётся оставшаяся часть кобры.

Каждый выходной порт, через который проходит кобра, помечается как порт, ведущий к одному из сыновей данного узла в дереве истока. Каждый некорневой узел в какой-то момент включится в дерево стока, а кобра, инициировавшая это включение, пройдёт предварительно по ребру, соединяющему отца данного узла и сам узел, что позволяет каждому узлу узнать всех его сыновей, в результате чего будет получена вся информация о дереве истока.

```
⟨ Передать змей 17 ⟩ +≡
  snake s;
  switch (cobra) {
  case gap:
    if (father ≠ nil) s = input[father].cobra;
    else if (is_root)
      for (int k = 0; k < in; k++)
        if ((s = input[k].python).is_head) {
          python_port = k;
          break;
        }
    if (s.is_head) {
      ⟨ Инициировать выползание кобры 25 ⟩
    }
    break;
  case head: case body: s = is_root ? input[python_port].python : input[father].cobra;
    if (cobra ≡ head) {
      cobra = body;
      s.is_head = true;
    }
    output[cobra_port].cobra = s;
    if (root_state ≡ cycle ∨ (father ≠ nil ∧ input[father].mark_source))      /* смотри ниже */
      output[cobra_port].mark_source = true;
    if (s.is_tail) {
      cobra = gap;
      if (is_root) python_port = nil;
      cobra_port = nil;
    }
    break;
  }
```

25. ⟨ Инициировать выползание кобры 25 ⟩ ≡

```
if (s.is_tail) {
  ⟨ Обработать кобру длины 1 28 ⟩
  if (is_root) python_port = nil;
}
else {
  cobra_port = s.body;
  cobra = head;
  is_son[cobra_port] = true;
}
```

Этот код используется в секции 24.

26. ⟨ Переменные узла 9 ⟩ +≡

```
port cobra_port;   /* порт, через который выползает кобра */
```

27. ⟨ Инициализировать узел 10 ⟩ +≡

```
cobra_port = nil;
```

28. Когда кобра укоротится до длины 1, она приползёт в тот самый узел, в котором её предшественница гадюка превратилась в питона. При этом единственный символ, из которого состоит кобра, указывает на выходной порт, который ведёт к узлу v дерева стока. Если к этому моменту узел u, в который приползла кобра, уже присоединился к дереву стока, то кобра игнорируется, а в противном случае к дереву стока добавляется ребро $u \to v$. Для этого узел u посылает узлу v сообщение, говорящее ему о том, что у него появилась новая дочка, соответствующая входному порту, из которого он получил сообщение.

На все остальные выходные порты узел u посылает специальный сигнал. Каждый узел запоминает, от каких из своих входных портов он уже получил этот сигнал. Если на очередном такте оказалось, что сигналы уже получены со всех живых портов, и при этом сам узел входит в дерево стока, то такой же сигнал посылается матери узла. Отметим, что именно в этом месте алгоритма используется тот факт, что нам известны мёртвые входные порты. Корневой узел, вместо того, чтобы посылать сигнал своей матери (которой у него нет, так как он круглый сирота), завершает алгоритм.

⟨ Обработать кобру длины 1 28 ⟩ ≡

```
if (mother ≡ nil ∧ ¬is_root) {
  mutant = false;
  output[mother = s.body].daughter = true;
  for (int k = 0; k < out; k++)
    if (k ≠ mother) output[k].ready = true;
}
```

Смотри также секции 43 и 53.

Этот код используется в секции 25.

29. ⟨ Построить дерево 14 ⟩ +≡

```
for (int k = 0; k < in; k++) {
  if (input[k].ready) unready−−;
  if (input[k].daughter) is_daughter[k] = true;
}
if (unready ≡ 0 ∧ (is_root ∨ mother ≠ nil)) {
  unready = nil;
  if (mother ≠ nil) output[mother].ready = true;
  else root_state = neutral;
}
```

30. ⟨ Переменные символа 8 ⟩ +≡

```
bool daughter, ready;
```

31. ⟨ Инициализировать символ 31 ⟩ ≡

```
daughter = ready = false;
```

Смотри также секции 42, 51, и 55.

Этот код используется в секции 4.

32. ⟨ Переменные узла 9 ⟩ +≡

```
int unready;    /* количество неготовых узлов */
```

33. ⟨ Инициализировать узел 10 ⟩ +≡

⟨ Вычислить количество живых входных портов 34 ⟩

34. ⟨ Вычислить количество живых входных портов 34 ⟩ ≡

```
unready = 0;
for (int k = 0; k < in; k++)
  if (is_alive[k]) unready++;
```

Этот код используется в секциях 33 и 47.

35. Так как граф узлов является сильно связным, по завершении алгоритма будут полностью построены деревья истока и стока. Время работы алгоритма квадратично по числу узлов, так как дерево истока строится за линейное время, а размер временных промежутков между последовательными присоединениями вершин к дереву стока линейно ограничен по числу узлов.

36. Обход графа. Второй алгоритм выполняет обход графа. Цель обхода состоит в том, чтобы побывать в каждой вершине графа ровно один раз. Процесс посещения вершины заключается в том, что она переходит в специальное состояние, выполняет некоторое время в этом состоянии произвольные действия, а затем выходит из него. В каждый момент времени не больше одной вершины может находиться в специальном состоянии. Наш алгоритм посещает вершины в прямом порядке обхода дерева истока. Первым посещается корень.

Пусть пройдена очередная вершина, как узнать, какая вершина должна быть следующей? Если у текущей вершины есть сыновья, то первый из них становится следующей вершиной, а текущая вершина просто сообщает ему об этом. Если же у текущей вершины нет сыновей, то тогда следующей вершиной должен стать её ближайший брат справа, если он существует. Для того, чтобы он узнал это, необходимо произвести откат. Для этого текущая вершина при помощи дерева стока посылает корню специальный сигнал, который заставляет корень выслать одну волну гадюк всем своим сыновьям. Каждый узел дерева истока, получив гадюку от своего отца, ретранслирует её удлинённую версию своим сыновьям. Когда текущая вершина получит гадюку, она трансформирует её в питона, которого отправляет корню при помощи дерева стока. Корень, получив питона, трансформирует его в кобру, которую он отправляет по дереву истока. Когда кобра укоротится до длины 1, она приползёт в узел, являющийся отцом текущей вершины. Этот узел сообщает следующему брату посещённой вершины, что он является следующей вершиной. Если же сыновей больше не осталось, то узел выполняет вышеописанную процедуру отката. Алгоритм завершается при попытке отката из корня. Отметим, что внеся небольшие изменения, можно передавать ограниченный объём информации от текущего узла к следующему.

⟨ Запустить алгоритм 13 ⟩ +≡
case *walk*: *state* = *visit*;
  break;

37. ⟨ Выполнить обход 37 ⟩ ≡
  ⟨ Обработать сигнал отката 44 ⟩
  if $(state \equiv backtrack \land (output[mother].python = input[father].adder).is_tail)$ $state = transfer$;
  if $(state \equiv visit)$ { /* здесь можно выполнить действия по посещению вершины */
    $state = active$;
    int $k = 0$;
    ⟨ Найти следующую вершину 40 ⟩
  }
  if $(father \neq nil \land input[father].activate)$ $state = visit$;

Этот код используется в секции 5.

38. ⟨ Состояния узла 7 ⟩ +≡
  visit, /* посещаем данную вершину */
  active, /* являемся активной вершиной */
  backtrack, /* выполняем откат */

39. ⟨ Состояния корня 16 ⟩ +≡
  walk, /* выполняем обход */

40. ⟨ Найти следующую вершину 40 ⟩ ≡

```
for ( ; k < out; k++)
  if (is_son[k]) {
    output[k].activate = true;
    break;
  }
if (k ≡ out)
  if (is_root) {
    state = transfer;
    root_state = neutral;
  }
  else {
    state = backtrack;
    output[mother].backtrack = true;
  }
```

Этот код используется в секциях 37 и 43.

41. ⟨ Переменные символа 8 ⟩ +≡

```
bool backtrack, activate;
```

42. ⟨ Инициализировать символ 31 ⟩ +≡

```
backtrack = activate = false;
```

43. ⟨ Обработать кобру длины 1 28 ⟩ +≡

```
if (state ≡ active) {
  int k = s.body + 1;
  ⟨ Найти следующую вершину 40 ⟩
}
```

44. ⟨ Обработать сигнал отката 44 ⟩ ≡

```
for (int k = 0; k < in; k++)
  if (input[k].backtrack) {
    if (is_root) ⟨ Отослать волну гадюк 15 ⟩
    else output[mother].backtrack = true;
    break;
  }
```

Этот код используется в секции 37.

45. Длинный цикл. Третий алгоритм строит цикл, который проходит от некоторой вершины по дереву стока до корня, после этого идёт по дереву истока от корня до другой вершины, и в конце проходит ровно по одному ребру. Из всех таких циклов будет найден тот, который имеет максимальную длину, а если таких несколько — любой из них.

Сначала корень рассылает своим сыновьям одну волну гадюк, вместе с которыми также отсылаются идентичные им питоны. Каждый узел, который получил гадюку, ретранслирует её всем своим сыновьям с соответствующими изменениями. На каждый выходной порт, кроме порта, ведущего к матери узла, отправляется также питон, идентичный ползущей вместе с ним гадюке.

```
⟨ Запустить алгоритм 13 ⟩ +≡
case cycle: ⟨ Отослать волну гадюк 15 ⟩
  for (int k = 0; k < out; k++) output[k].python = snake(k, true, true);
  ⟨ Инициализировать поиск цикла 47 ⟩
  break;
```

46. ⟨ Построить цикл 46 ⟩ ≡

```
  if (father ≠ nil ∧ input[father].cycle) {
    ⟨ Инициализировать поиск цикла 47 ⟩
  }
```

Смотри также секцию 52.

Этот код используется в секции 5.

47. ⟨ Инициализировать поиск цикла 47 ⟩ ≡

```
  state = search;
  mutant = true;
  python = cobra = ignore;
  for (int k = 0; k < out; k++) output[k].cycle = true;
  ⟨ Вычислить количество живых входных портов 34 ⟩
```

Этот код используется в секциях 45 и 46.

48. ⟨ Состояния узла 7 ⟩ +≡

```
  search,     /* ищем цикл */
```

49. ⟨ Состояния корня 16 ⟩ +≡

```
  cycle,     /* строим цикл */
```

50. ⟨ Переменные символа 8 ⟩ +≡

```
  bool cycle;     /* требуется ли строить цикл? */
```

51. ⟨ Инициализировать символ 31 ⟩ +≡

```
  cycle = false;
```

52. Первое время каждый узел игнорирует поступающих к нему питонов, помечая лишь входные порты, с которых они к нему поступили. Когда в последний непомеченный живой входной порт приползает питон, он ретранслируется матери узла. Неоднозначности разрешаются произвольным образом.

Корневой узел вместо ретрансляции питона своей матери преобразует его в кобру, которая пройдёт вторую половину длинного цикла. Когда кобра достигает узла, в котором её предшественница-гадюка превратилась в питона, этот узел высылает по соответствующему порту специальный сигнал. Этот сигнал проделает первую часть длинного цикла, спускаясь вниз по дереву стока.

По построению следует, что получившийся путь будет иметь максимальную длину. Отметим, что длина получившегося пути меньше, чем $2n$, и больше, чем половина длины кратчайшего пути между любой парой вершин. Это свойство используется при синхронизации.

```
⟨ Построить цикл 46 ⟩ +≡
  if (state ≡ search)
    for (int k = 0; k < in; k++)
      if (input[k].python.is_head) unready −−;
  for (int k = 0; k < in; k++)
    if (input[k].mark_sink) {
      if (mother ≠ nil) output[mother].mark_sink = true;
      else root_state = neutral;
      prev = k;
      break;
    }
```

53.
```
⟨ Обработать кобру длины 1 28 ⟩ +≡
  if (root_state ≡ cycle ∨ (father ≠ nil ∧ input[father].mark_source)) {
    next = s.body;
    output[s.body].mark_sink = true;
  }
```

54. Пометки *mark_source* и *mark_sink* указывает на то, что данная вершина включается в длинный цикл как часть пути в дереве истока или стока.

```
⟨ Переменные символа 8 ⟩ +≡
  bool mark_source, mark_sink;
```

55.
```
⟨ Инициализировать символ 31 ⟩ +≡
  mark_source = mark_sink = false;
```

56. Для каждой вершины той части цикла, которая идёт по дереву стока (в частности, для корня), номер порта, соответствующего предыдущей вершине цикла, хранится в переменной *prev*. Для остальных вершин эта переменная принимает значение *nil*. Для вершины цикла, из которой выходит промежуточное ребро, номер порта, соответствующего следующей вершине цикла, хранится в переменной *next*. Для остальных вершин эта переменная принимает значение *nil*.

```
⟨ Переменные узла 9 ⟩ +≡
  port prev, next;
```

57.
```
⟨ Инициализировать узел 10 ⟩ +≡
  prev = next = nil;
```

58. Синхронизация. Четвёртый алгоритм позволяет *синхронизировать* узлы — одновременно перевести их в специальное состояние, так, чтобы каждый узел оказался в этом состоянии в первый раз.

Прежде всего заметим, что за один такт один символ может перейти от одного узла к другому. Если заставить узлы задерживать символы, то можно добиться того, что один символ переходит от одного узла к другому за два или три такта.

Сначала опишем решение этой задачи для случая, когда граф является ориентированным циклом, длина которого — степень 2. Будем называть узлы солдатами, задача заключается в том, чтобы все солдаты выстрелили одновременно. Во время работы алгоритма некоторые солдаты становятся генералами. В начале работы корневой узел становится генералом. Когда солдат становится генералом, он порождает четыре сигнала A, B, C, D, которые движутся со скоростями 1/2, 1/3, 1/6, 1/2 соответственно. Если солдат получает сигналы A и C одновременно, то он становится генералом. Если генерал получает сигнал B, то он заново порождает все четыре сигнала. Если солдат получает сигнал D, то он его игнорирует, а если генерал получает сигнал D, то он стреляет.

Через $3n$ тактов два противоположных узла, один из которых корневой, становятся генералами. Через $3n/2$ тактов четыре одинаково расположенных узла становятся генералами, и так далее. Через $6n-6$ тактов все солдаты станут генералами, а ещё через один такт все генералы выстрелят.

Теперь адаптируем решение для случая, когда длина цикла не является степенью 2. Если солдат получил сигнал A, а за такт до этого он получил сигнал C, то он становится генералом и размещает перед собой дополнительного виртуального солдата. Тем самым, на очередном этапе алгоритма каждый сегмент делится на две одинаковые части, в одной из которых может стоять виртуальный солдат.

Наконец, решим задачу в общем случае. Для этого построим длинный цикл, на котором запустим алгоритм. Заметим, что некоторые вершины могут входить в цикл дважды; для таких вершин необходимо продублировать солдат. Кроме этого, каждая вершина графа ведёт себя также, как и вершины цикла, и передаёт сигналы всем своим сыновьям. Свойство максимальности длины цикла гарантирует, что алгоритм работает корректно.

```
⟨ Запустить алгоритм 13 ⟩ +≡
case synchronization: primary.primary.w[0] = 0;
  primary.primary.w[2] = 0;
  break;
```

59.

```
⟨ Передать сигналы 59 ⟩ ≡
  if (fire) fire = false;
  else
    for (int k = 0; k < 4; k++)      /* A = 0, B = 1, C = 2, D = 3 */
      if (s[k]) w[k] = 0;
  bool v = false;      /* надо ли создавать виртуального солдата? */
  ⟨ Обработать сигналы 60 ⟩
  for (int k = 0; k < 4; k++) {
    t[k] = false;
    if (w[k] ≠ nil) {
      w[k]++;
      if (w[k] ≡ speed[k]) {
        w[k] = nil;
        t[k] = true;
      }
    }
  }
  return v;
```

Этот код используется в секции 62.

60. ⟨ Обработать сигналы 60 ⟩ ≡

```
if (w[3] ≡ 0)
  if (is_general) {
    w[0] = w[1] = w[2] = w[3] = nil;
    is_general = false;
    fire = true;
  }
  else w[3] = nil;
if (w[0] ≡ 0 ∧ w[2] ≡ 0) {
  is_general = true;
  w[1] = 0;
}
if (w[0] ≡ 1 ∧ w[2] ≡ 3) {
  is_general = true;
  w[1] = 0;
  v = true;
}
if (is_general ∧ w[1] ≡ 0) {
  w[0] = 0;
  w[1] = 0;
  w[2] = 0;
  w[3] = 0;
}
```

Этот код используется в секции 59.

61. ⟨ Вспомогательные структуры данных 6 ⟩ +≡

```
struct set     /* пакет сигналов */
{
  bool data[4];
  set()
  {
    data[0] = data[1] = data[2] = data[3] = false;
  }
  bool &operator[](int k)
  {
    return data[k];
  }
  ⟨ Нарисовать пакет сигналов 76 ⟩
};
```

62. Выпишем структуры данных для солдат.

```
⟨ Вспомогательные структуры данных 6 ⟩ +≡
  struct soldier {
    static const int speed[4];
    int w[4];
    bool is_general, fire;
    soldier()
    {
      for (int k = 0; k < 4; k++) w[k] = nil;
      is_general = fire = false;
    }
    bool pass(set s, set &t)
    {
      ⟨ Передать сигналы 59 ⟩
    }
    ⟨ Нарисовать солдата 77 ⟩
  };
  const int soldier::speed[4] = {2, 3, 6, 2};
  struct soldier_pair    /* один солдат и необязательный виртуальный солдат */
  {
    soldier primary, secondary;    /* secondary — это виртуальный солдат */
    bool has_secondary;
    set intermediate;
    soldier_pair()
    {
      has_secondary = false;
    }
    bool pass(set s, set &t)
    {
      if (has_secondary) {
        primary.pass(intermediate, t);
        secondary.pass(s, intermediate);
      }
      else has_secondary = primary.pass(s, t);
      return primary.fire;
    }
    ⟨ Нарисовать пару солдат 78 ⟩
  };
```

63. ⟨ Переменные узла 9 ⟩ +≡

```
  soldier_pair primary, secondary;    /* secondary — пара солдатов, стоящая на части цикла,
      проходящей по дереву стока, исключая корень */
```

64. ⟨ Синхронизироваться 64 ⟩ ≡

```
if (state ≡ fire) {
  state = transfer;
  root_state = neutral;
}
set u, v;
if (is_root) {
  if (prev ≠ nil) u = input[prev].secondary;
  else if (root_state ≡ synchronization) u[3] = true;
}
else if (father ≠ nil) u = input[father].primary;
if (primary.pass(u, v)) state = fire;
for (int k = 0; k < out; k++)
  if (is_son[k]) output[k].primary = v;
if (next ≠ nil) output[next].secondary = v;
if (prev ≠ nil ∧ ¬is_root) secondary.pass(input[prev].secondary, output[mother].secondary);
```

Этот код используется в секции 5.

65. ⟨ Состояния корня 16 ⟩ +≡

```
synchronization,    /* синхронизируемся */
```

66. ⟨ Состояния узла 7 ⟩ +≡

```
fire,    /* огонь! */
```

67. ⟨ Переменные символа 8 ⟩ +≡

```
set primary, secondary;
  /* secondary используется для передачи данных в части цикла, проходящей по дереву стока */
```

68. Ввод и вывод. Файл, задающий сеть автоматов, состоит из двух секций, разделённых пустой строкой. Каждая строка первой секции задаёт узел. Первый символ строки — это идентификатор узла, далее следуют два числа — номер строки и столбца, в которых будет изображаться идентификатор узла, а затем ещё два числа — количество входных и выходных портов. Каждая строка второй секции задаёт канал связи и состоит из описания начального и конечного портов, разделённых пробелом. Описание порта состоит из идентификатора его узла и его номера среди других портов данного узла, разделённых пробелом.

```
⟨ Ввести сеть 68 ⟩ ≡
  void input( )
  {
    char c, e;
    int d, f;
    while (isgraph(c = getchar( ))) {
      node n;
      n.id = c;
      scanf("%d%d%d%d", &n.a, &n.b, &n.in, &n.out);
      n.is_alive.resize(n.in);
      n.input.resize(n.in);
      n.output.resize(n.out);
      nodes.push_back(n);
      while (getchar( ) ≠ '\n') ;
    }
    while (scanf("␣%c%d␣%c%d", &c, &d, &e, &f) ≡ 4) {
      arc a;
      for (int k = 0; k < nodes.size( ); k++) {
        if (nodes[k].id ≡ c) a.begn = k;
        if (nodes[k].id ≡ e) a.endn = k;
      }
      a.begp = d;
      a.endp = f;
      arcs.push_back(a);
      nodes[a.endn].is_alive[a.endp] = true;
    }
    root = nil;
  }
```

Этот код используется в секции 3.

69. ⟨ Переменные узла 9 ⟩ +≡

```
  char id;    /* идентификатор */
  int a, b;   /* координаты места изображения идентификатора */
```

70. ⟨ Инициализировать сеть 70 ⟩ ≡

```
  void init(int Root)
  {
    root = Root;
    for (int k = 0; k < nodes.size( ); k++) nodes[k].init(k ≡ root);
  }
```

Этот код используется в секции 3.

71. Каждому узлу выделяется пространство в 18 столбцов. Четыре столбца посередине служат для изображения состояния узла. По бокам от этих столбцов расположены два блока по три столбца, которые служат для отображения флагов портов. По краям расположены два блока по четыре столбца, используемые для вывода данных, передаваемых по каналам связи. Слева расположены входные порты, а справа — выходные. Каждому входному порту соответствует блок размера 2 на 7 в двух левых группах столбцов. Первая часть блока содержит символ, поступивший по входному каналу, а вторая — флаги, как описано ниже.

В первой строке:

- вползает ли через данный порт питон;
- идентификатор узла, из которого вышел канал связи;
- номер порта, из которого вышел канал связи.

Во второй строке:

- является ли данный канал ребром второй части длинного цикла;
- идёт ли данный канал от отца;
- идёт ли данный канал от дочери.

Если входной порт является мёртвым, то вместо идентификатора узла и номера порта, из которого вышел канал связи, ставятся два знака '-'. Блок, соответствующий выходному порту изображается аналогично.

Опишем содержимое центрального блока столбцов. Слева вверху находится идентификатор узла, координаты которого заданы извне, далее идёт буква 'M', если переменная *mutant* истинна, после неё выводится значение переменной *unready*. Если происходит синхронизация, то на следующих строках изображаются пары солдат, отвечающие за передачу сигналов по дереву истока и стока, при этом вторая пара может отсутствовать. Иначе на следующей строке выводятся первые буквы названий состояний передачи гадюки, питона, кобры, как указано в §9.

⟨ Нарисовать постоянную часть сети 71 ⟩ ≡

```
void draw()
{
  for (int k = 0; k < nodes.size(); k++) {
    draw_char(nodes[k].a, nodes[k].b, nodes[k].id, white);
    for (int l = 0; l < nodes[k].in; l++) {
      draw_char(nodes[k].a + 2 * l, nodes[k].b - 2, '-', white);
      draw_char(nodes[k].a + 2 * l, nodes[k].b - 1, '-', white);
    }
    for (int l = 0; l < nodes[k].out; l++) {
      draw_char(nodes[k].a + 2 * l, nodes[k].b + 4, '-', white);
      draw_char(nodes[k].a + 2 * l, nodes[k].b + 5, '-', white);
    }
  }
  for (int k = 0; k < arcs.size(); k++) {
    arc a = arcs[k];
    draw_char(nodes[a.begn].a + a.begp * 2, nodes[a.begn].b + 4, nodes[a.endn].id, white);
    draw_char(nodes[a.begn].a + a.begp * 2, nodes[a.begn].b + 5, '0' + a.endp, white);
    draw_char(nodes[a.endn].a + a.endp * 2, nodes[a.endn].b - 2, nodes[a.begn].id, white);
    draw_char(nodes[a.endn].a + a.endp * 2, nodes[a.endn].b - 1, '0' + a.begp, white);
  }
}
```

Этот код используется в секции 3.

72. ⟨ Обновить изображение сети 72 ⟩ ≡

```
void update()
{
  int rs = nodes[root].root_state;
  const char s[5] = {'N', 'T', 'W', 'C', 'S'};
  draw_char(0, 0, s[rs], white);
  bool ds = rs ≡ node::synchronization;
  for (int k = 0; k < nodes.size(); k++) nodes[k].draw(ds);
}
```

Этот код используется в секции 3.

73. ⟨ Обновить изображение узла 73 ⟩ ≡

```
void draw(bool draw_soldiers)
{
  for (int k = 0; k < 8; k++) wipe(a + k, b, 4);
  draw_char(a, b, id, is_root ? red : green);
  const char c[6] = {'T', 'V', 'A', 'B', 'S', 'F'};
  draw_char(a, b + 1, c[state], white);
  draw_char(a, b + 2, unready ≡ nil ? '␣' : '0' + unready, white);
  draw_char(a, b + 3, mutant ? 'M' : '␣', white);
  draw_char(a + 1, b + 3, mutate ? 'm' : '␣', white);
  if (draw_soldiers) {
    primary.draw(a + 1, b);
    if (prev ≠ nil ∧ ¬is_root) secondary.draw(a + 5, b);
  }
  else {
    const char d[5] = {'G', 'H', 'B', 'T', 'I'};
    draw_char(a + 1, b, d[adder], white);
    draw_char(a + 1, b + 1, d[python], white);
    draw_char(a + 1, b + 2, d[cobra], white);
  }
  for (int k = 0; k < in; k++) {
    draw_char(a + 2 * k + 1, b − 2, k ≡ father ? 'f' : '␣', white);
    draw_char(a + 2 * k + 1, b − 1, is_daughter[k] ? 'd' : '␣', white);
    draw_char(a + 2 * k, b − 3, k ≡ python_port ? 'p' : '␣', white);
    draw_char(a + 2 * k + 1, b − 3, k ≡ prev ? 'P' : '␣', white);
    input[k].draw(a + 2 * k, b − 7, draw_soldiers, k ≡ prev);
  }
  for (int k = 0; k < out; k++) {
    draw_char(a + 2 * k + 1, b + 4, k ≡ mother ? 'm' : '␣', white);
    draw_char(a + 2 * k + 1, b + 5, is_son[k] ? 's' : '␣', white);
    draw_char(a + 2 * k, b + 6, k ≡ cobra_port ? 'c' : '␣', white);
    draw_char(a + 2 * k + 1, b + 6, k ≡ next ? 'N' : '␣', white);
    output[k].draw(a + 2 * k, b + 7, draw_soldiers, k ≡ next ∨ (k ≡ mother ∧ prev ≠ nil));
  }
}
```

Этот код используется в секции 5.

74. Поскольку в различных алгоритмах используются различные типы символов, изображение символа зависит от выполняемого алгоритма. При синхронизации содержимое занимает две строки, состоящие из пакета сигналов, переданного по дереву истока и стока соответственно. Второй пакет может отсутствовать. Для всех остальных алгоритмов первая строка состоит из трёх символов, изображающие переданный сегмент гадюки, питона, кобры. Вторая строка содержит набор событий, передающийся символом. Каждому событию соответствует своя буква, которая выводится только в том случае, если событие произошло.

```
⟨ Нарисовать символ 74 ⟩ ≡
  void draw(int a, int b, bool draw_sets, bool draw_secondary)
  {
    wipe(a, b, 4);
    wipe(a + 1, b, 4);
    if (draw_sets) {
      primary.draw(a, b);
      if (draw_secondary) secondary.draw(a + 1, b);
    }
    else {
      adder.draw(a, b);
      python.draw(a, b + 1);
      cobra.draw(a, b + 2);
      bool f[7] = {daughter, ready, backtrack, activate, cycle, mark_source, mark_sink};
      const char c[7] = {'D', 'R', 'B', 'A', 'C', 'O', '1'};
      for (int k = 0; k < 7; k++)
        if (f[k]) draw_char(a + 1, b++, c[k], white);
    }
  }
```

Этот код используется в секции 4.

75. Сегмент змеи изображается номером порта. При этом тело змеи выводится стандартным цветом, голова змеи — зелёным цветом, хвост змеи — синим цветом, а односегментная змея — красным цветом. При отсутствии сегмента выводится символ '-'.

```
⟨ Нарисовать сегмент змеи 75 ⟩ ≡
  void draw(int a, int b)
  {
    const int color[2][2] = {{white, blue}, {green, red}};
    draw_char(a, b, body ≡ nil ? '-' : '0' + body, color[is_head][is_tail]);
  }
```

Этот код используется в секции 6.

76. Пакет сигналов изображается строкой из четырёх символов, соответствующих сигналам A, B, C, D. Символ '-' выводится при отсутствии сигнала, а при наличии сигнала выводится его обозначение.

```
⟨ Нарисовать пакет сигналов 76 ⟩ ≡
  void draw(int a, int b)
  {
    for (int k = 0; k < 4; k++) draw_char(a, b + k, data[k] ? 'A' + k : '-', white);
  }
```

Этот код используется в секции 61.

77. Состояние солдата изображается строкой из двух символов, соответствующих сигналам B и C. Если солдат задерживает сигнал B, то на первое место выводится символ '0', а иначе '-'. Если солдат задерживает сигнал C, то на второе место выводится символ '0' или '1' в зависимости от стадии задержки, а иначе выводится символ '-'.

⟨ Нарисовать солдата 77 ⟩ ≡
```
void draw(int a, int b)
{
  for (int k = 0; k < 4; k++)
    draw_char(a, b + k, w[k] ≡ nil ? '-' : '0' + w[k], fire ? red : is_general ? green : white);
}
```
Этот код используется в секции 62.

78. Состояние пары солдат изображается тремя строками. На первой строке выводится виртуальный солдат, на третьей строке — невиртуальный солдат, а во второй строке содержится пакет сигналов, посланный на данном такте виртуальным солдатом.

⟨ Нарисовать пару солдат 78 ⟩ ≡
```
void draw(int a, int b)
{
  primary.draw(a + 2, b);
  if (has_secondary) {
    secondary.draw(a, b);
    intermediate.draw(a + 1, b);
  }
  else {
    wipe(a, b, 4);
    wipe(a + 1, b, 4);
  }
}
```
Этот код используется в секции 62.

79. ⟨ Функции рисования 79 ⟩ ≡
```
#include <curses.h>
  const int white = 1, red = 2, green = 3, blue = 4;
  void draw_char(int a, int b, int c, int color)
  {
    move(a, b);
    color_set(color, 0);
    addch(c);
  }
  void wipe(int a, int b, int n)
  {
    while (n-- > 0) draw_char(a, b++, '␣', white);
  }
```
Этот код используется в секции 3.

80. Основная функция программы обрабатывает нажатия клавиш. Клавиша 'x' завершает программу. Клавиша 'r', за которой идёт идентификатор узла, делает данный узел корневым и инициализирует сеть. Клавиши 't', 'w', 'c', 's' запускают алгоритмы построения деревьев, обхода, построения цикла, синхронизации. Клавиша '␣' выполняет один такт.

```
int main()
{
  net n;
  n.input();
  freopen("/dev/tty", "r", stdin);
  initscr();
  noecho();
  cbreak();
  curs_set(0);
  start_color();
  init_pair(white, COLOR_WHITE, COLOR_BLACK);
  init_pair(red, COLOR_RED, COLOR_BLACK);
  init_pair(green, COLOR_GREEN, COLOR_BLACK);
  init_pair(blue, COLOR_BLUE, COLOR_BLACK);
  n.draw();
  while (1)
    switch (int c = getch()) {
    case 'x': endwin();
      return 0;
    case 'r': c = getch();
      for (int k = 0; k < n.nodes.size(); k++)
        if (n.nodes[k].id ≡ c) {
          n.init(k);
          n.update();
        }
      break;
    case '␣':
      if (n.root ≡ nil) break;
      n.step();
      n.update();
      break;
    case 'f':
      if (n.root ≡ nil) break;
      while (n.nodes[n.root].root_state ≠ node::neutral) {
        n.step();
        n.update();
        refresh();
      }
      break;
    default:
      if (n.root ≡ nil) break;
      char d[4] = {'t', 'w', 'c', 's'};
      node::r_state rs[4] = {node::tree, node::walk, node::cycle, node::synchronization};
      int k;
      for (k = 0; k < 4; k++)
        if (d[k] ≡ c) break;
      if (k ≡ 4) break;
```

```
      n.nodes[n.root].root_state = rs[k];
      n.nodes[n.root].start = true;
      n.update( );
      break;
    }
}
```

81. Заключение. По адресу `http://rain.ifmo.ru/ pavlov/serp/` находится исходный текст, программы для генерации примеров, откомпилированная программа в форматах C++, TeX, DVI, PDF и PostScript.

82. Предметный указатель.

a: 3, 68, 69, 71, 74, 75, 76, 77, 78, 79.
activate: 37, 40, 41, 42, 74.
active: 37, 38, 43.
addch: 79.
adder: 8, 9, 10, 15, 17, 18, 37, 73, 74.
arc: 3, 68, 71.
arcs: 3, 68, 71.
assign: 3, 5, 12.
b: 69, 74, 75, 76, 77, 78, 79.
backtrack: 37, 38, 40, 41, 42, 44, 74.
begn: 3, 68, 71.
begp: 3, 68, 71.
blue: 75, 79, 80.
Body: 6.
body: 6, 9, 18, 21, 24, 25, 28, 43, 53, 75.
c: 68, 73, 74, 79.
cbreak: 80.
cobra: 8, 9, 10, 18, 24, 25, 47, 73, 74.
cobra_port: 24, 25, 26, 27, 73.
color: 75, 79.
`COLOR_BLACK`: 80.
`COLOR_BLUE`: 80.
`COLOR_GREEN`: 80.
`COLOR_RED`: 80.
color_set: 79.
`COLOR_WHITE`: 80.
curs_set: 80.
cycle: 24, 45, 46, 47, 49, 50, 51, 53, 74, 80.
d: 68, 73, 80.
data: 61, 76.
daughter: 28, 29, 30, 31, 74.
draw: 71, 72, 73, 74, 75, 76, 77, 78, 80.
draw_char: 71, 72, 73, 74, 75, 76, 77, 79.
draw_secondary: 74.
draw_sets: 74.
draw_soldiers: 73.
ds: 72.
e: 68.
endn: 3, 68, 71.
endp: 3, 68, 71.
endwin: 80.
f: 68, 74.
false: 5, 6, 12, 18, 20, 28, 31, 42, 51, 55, 59, 60, 61, 62.
father: 11, 12, 17, 18, 24, 37, 46, 53, 64, 73.
fire: 59, 60, 62, 64, 66, 77.
freopen: 80.
gap: 9, 10, 18, 21, 24.
getch: 80.
getchar: 68.
green: 73, 75, 77, 79, 80.
has_secondary: 62, 78.
head: 9, 24, 25.
id: 68, 69, 71, 73, 80.
ignore: 9, 47.
in: 5, 12, 17, 21, 24, 29, 34, 44, 52, 68, 71, 73.
init: 5, 70, 80.
init_pair: 80.
initscr: 80.
input: 3, 5, 17, 18, 21, 24, 29, 37, 44, 46, 52, 53, 64, 68, 73, 80.
intermediate: 62, 78.
is_alive: 5, 34, 68.
is_daughter: 11, 12, 29, 73.
is_general: 60, 62, 77.
is_head: 6, 17, 18, 21, 24, 52, 75.
Is_head: 6.
is_root: 5, 17, 24, 25, 28, 29, 40, 44, 64, 73.
Is_root: 5.
is_son: 11, 12, 25, 40, 64, 73.
is_tail: 6, 18, 21, 24, 25, 37, 75.
Is_tail: 6.
isgraph: 68.
k: 3, 13, 15, 17, 18, 21, 24, 28, 29, 34, 37, 43, 44, 45, 47, 52, 59, 61, 62, 64, 68, 70, 71, 72, 73, 74, 76, 77, 80.
l: 71.
main: 80.
mark_sink: 52, 53, 54, 55, 74.
mark_source: 24, 53, 54, 55, 74.
mother: 11, 12, 18, 21, 28, 29, 37, 40, 44, 52, 64, 73.
move: 79.
mutant: 17, 18, 19, 20, 28, 47, 71, 73.
mutate: 18, 19, 20, 73.
n: 68, 79, 80.
net: 3, 80.
neutral: 5, 29, 40, 52, 64, 80.
next: 53, 56, 57, 64, 73.
nil: 4, 6, 11, 12, 17, 18, 21, 23, 24, 25, 27, 28, 29, 37, 46, 52, 53, 56, 57, 59, 60, 62, 64, 68, 73, 75, 77, 80.
node: 3, 5, 68, 72, 80.
nodes: 3, 68, 70, 71, 72, 80.
noecho: 80.
out: 3, 5, 12, 13, 15, 18, 28, 40, 45, 47, 64, 68, 71, 73.
output: 3, 5, 13, 15, 18, 21, 24, 28, 29, 37, 40, 44, 45, 47, 52, 53, 64, 68, 73.
pass: 62, 64.
port: 4, 6, 11, 22, 26, 56.
prev: 52, 56, 57, 64, 73.

primary: 58, 62, 63, 64, 67, 73, 74, 78.
push_back: 68.
python: 8, 9, 10, 18, 21, 24, 37, 45, 47, 52, 73, 74.
python_port: 21, 22, 23, 24, 25, 73.
r_state: 5, 80.
ready: 13, 28, 29, 30, 31, 74.
red: 73, 75, 77, 79, 80.
refresh: 80.
resize: 68.
root: 3, 68, 70, 72, 80.
Root: 70.
root_state: 5, 14, 24, 29, 40, 52, 53, 64, 72, 80.
rs: 72, 80.
s: 18, 24, 62, 72.
scanf: 68.
search: 18, 47, 48, 52.
secondary: 62, 63, 64, 67, 73, 74, 78.
set: 61, 62, 64, 67.
size: 3, 68, 70, 71, 72, 80.
snake: 6, 8, 15, 18, 24, 45.
soldier: 62.
soldier_pair: 62, 63.
speed: 59, 62.
start: 5, 80.
start_color: 80.
state: 5, 18, 36, 37, 40, 43, 47, 52, 64, 73.
std: 3.
stdin: 80.
step: 3, 5, 80.
symbol: 3, 4, 5.
synchronization: 58, 64, 65, 72, 80.
t: 62.
tail: 9, 18.
transfer: 5, 7, 18, 37, 40, 64.
tree: 13, 14, 16, 80.
true: 5, 13, 15, 17, 18, 24, 25, 28, 29, 40, 44, 45, 47, 52, 53, 59, 60, 64, 68, 80.
u: 64.
unready: 18, 29, 32, 34, 52, 71, 73.
update: 72, 80.
v: 59, 64.
vbool: 5, 11.
vector: 3, 5.
visit: 36, 37, 38.
w: 62.
walk: 36, 39, 80.
white: 71, 72, 73, 74, 75, 76, 77, 79, 80.
wipe: 73, 74, 78, 79.

⟨ Функции рисования 79 ⟩ Используется в секции 3.
⟨ Инициализировать поиск цикла 47 ⟩ Используется в секциях 45 и 46.
⟨ Инициализировать сеть 70 ⟩ Используется в секции 3.
⟨ Инициализировать символ 31, 42, 51, 55 ⟩ Используется в секции 4.
⟨ Инициализировать узел 10, 12, 20, 23, 27, 33, 57 ⟩ Используется в секции 5.
⟨ Инициировать выползание кобры 25 ⟩ Используется в секции 24.
⟨ Найти следующую вершину 40 ⟩ Используется в секциях 37 и 43.
⟨ Нарисовать пакет сигналов 76 ⟩ Используется в секции 61.
⟨ Нарисовать пару солдат 78 ⟩ Используется в секции 62.
⟨ Нарисовать постоянную часть сети 71 ⟩ Используется в секции 3.
⟨ Нарисовать сегмент змеи 75 ⟩ Используется в секции 6.
⟨ Нарисовать символ 74 ⟩ Используется в секции 4.
⟨ Нарисовать солдата 77 ⟩ Используется в секции 62.
⟨ Обновить изображение сети 72 ⟩ Используется в секции 3.
⟨ Обновить изображение узла 73 ⟩ Используется в секции 5.
⟨ Обработать кобру длины 1 28, 43, 53 ⟩ Используется в секции 25.
⟨ Обработать сигналы 60 ⟩ Используется в секции 59.
⟨ Обработать сигнал отката 44 ⟩ Используется в секции 37.
⟨ Отослать волну гадюк 15 ⟩ Используется в секциях 14, 44, и 45.
⟨ Передать сигналы 59 ⟩ Используется в секции 62.
⟨ Передать змей 17, 18, 21, 24 ⟩ Используется в секции 5.
⟨ Переменные символа 8, 30, 41, 50, 54, 67 ⟩ Используется в секции 4.
⟨ Переменные узла 9, 11, 19, 22, 26, 32, 56, 63, 69 ⟩ Используется в секции 5.
⟨ Построить цикл 46, 52 ⟩ Используется в секции 5.
⟨ Построить дерево 14, 29 ⟩ Используется в секции 5.
⟨ Синхронизироваться 64 ⟩ Используется в секции 5.
⟨ Состояния корня 16, 39, 49, 65 ⟩ Используется в секции 5.
⟨ Состояния узла 7, 38, 48, 66 ⟩ Используется в секции 5.
⟨ Структуры данных 4, 5 ⟩ Используется в секции 3.
⟨ Вспомогательные структуры данных 6, 61, 62 ⟩ Используется в секции 4.
⟨ Ввести сеть 68 ⟩ Используется в секции 3.
⟨ Выполнить обход 37 ⟩ Используется в секции 5.
⟨ Вычислить количество живых входных портов 34 ⟩ Используется в секциях 33 и 47.
⟨ Запустить алгоритм 13, 36, 45, 58 ⟩ Используется в секции 5.

Автоматный серпентарий

	Секция	Страница
Введение	1	1
Общая структура программы	3	2
Змеи	6	5
Остовные деревья	11	7
Обход графа	36	14
Длинный цикл	45	16
Синхронизация	58	18
Ввод и вывод	68	22
Заключение	81	29
Предметный указатель	82	30