И.Б.Бурдонов.
ОБХОД НЕИЗВЕСТНОГО ОРИЕНТИРОВАННОГО ГРАФА КОНЕЧНЫМ РОБОТОМ.
"Программирование". 2004. No. 4.

Обход неизвестного ориентированного графа конечным роботом

И.Б.Бурдонов
Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
igor@ispras.ru
http://www.ispras.ru/~RedVerst/

Abstract. Обход ориентированного графа – это маршрут, проходящий через все вершины и дуги графа, причем дуга проходится только в направлении ее ориентации. Обход, начинающийся с любой начальной вершины, существует только для сильно-связных графов, в которых из каждой вершины можно попасть в каждую вершину по некоторому маршруту. Сильная связность – это единственное ограничение на рассматриваемый класс графов. Как известно, на классе таких графов длина обхода $\Theta(nm)$, где n – число вершин, а m – число дуг графа. Для любого графа существует обход длиной $O(nm)$ и существуют графы с минимальной длиной обхода $\Omega(nm)$. Обход неизвестного графа означает, что топология графа заранее неизвестна и мы узнаем ее только в процессе движения по графу. В каждой вершине видно, какие дуги из нее исходят, но в какую вершину ведет дуга можно узнать, только пройдя по ней. Это аналогично задаче обхода лабиринта роботом, находящимся внутри него и не имеющем плана лабиринта. Если робот – это «компьютер общего вида» без ограничения на число его состояний, то известны алгоритмы обхода с той же оценкой $O(nm)$.

Если число состояний ограничено, то робот – это конечный автомат. Такой робот является аналогом машины Тьюринга: лента заменена графом, а ее ячейки привязаны к вершинам и дугам графа. В настоящее время неизвестна нижняя оценка длины обхода конечным роботом. В 1971 г. автор статьи предложил робот с длиной обхода $O(nm+n^2 \log n)$. Алгоритм робота основан на построении остова графа, ориентированного от корня, и метода поиска в ширину (BFS) на этом остове. В 1993 г. Y.Afek и E.Gafni [1] описали алгоритм с той же оценкой длины обхода, также основанный на построении остова графа, но использующий метод поиска в глубину (DFS). В настоящей статье предлагается алгоритм, совмещающий поиск в ширину (BFS) с методом отката по остову графа (backtracking), предложенным Y.Afek и E.Gafni, за счет чего достигается оценка $O(nm+n^2 \log\log n)$. Робот использует константное число битов памяти для каждой вершины и дуги графа.

1. Введение

Задача обхода графа, то есть, построения маршрута, проходящего по всем ребрам графа, хорошо известна. Для ориентированного графа задача усложняется, поскольку маршрут должен проходить каждое ориентированное ребро (дугу) только в направлении его ориентации. Для существования обхода ориентированного графа с любой начальной

вершины, граф должен быть сильно-связным, то есть, каждая его вершина достижима из каждой вершины по некоторому маршруту.

В большинстве работ предполагается, что граф задан явно до построения его обхода [12,13]. Более сложным является случай, когда до начала работы о графе ничего неизвестно и мы получаем информацию об устройстве графа в процессе его обхода [2,9]. Это известная задача обхода лабиринта [14] человеком или устройством, находящимся внутри него и не имеющим плана лабиринта. Дуге графа соответствует коридор лабиринта, а вершине – перекресток. Находясь на перекрестке, мы видим исходящие из него коридоры, но мы не знаем, куда ведет тот или иной коридор, до тех пор, пока не прошли по нему до следующего перекрестка. Для выполнения нашей задачи мы, во-первых, имеем некоторую внутреннюю память (блокнот в руках человека), куда можем записывать полученную информацию о пройденной части лабиринта, и, во-вторых, делать пометки в пройденных перекрестках и коридорах. Ориентированному графу соответствует лабиринт, в котором все коридоры «с односторонним движением».
Если внутренняя память ограничена конечным числом состояний, мы имеем робот (конечный автомат) на графе как разновидность машины Тьюринга [1,3,4,10,11,15]. Вместо ленты у нас есть граф, ячейке ленты соответствует вершина графа, а движение влево или вправо заменяется на переход по одной из дуг, исходящих из текущей вершины графа. Робот должен каким-то образом указывать, по какой исходящей дуге он перемещается. Если дуги, исходящие из вершины v перенумерованы $1..d_{out}(v)$, где $d_{out}(v)$ – полустепень исхода вершины v, робот может указывать номер исходящей дуги. Однако, чтобы выходной алфавит робота оставался конечным, граф должен иметь ограничение сверху на полустепень исхода d_{out}.

Это ограничение легко снимается, если в каждой вершине v добавить столько ячеек памяти, сколько дуг исходит из v (полустепень исхода d_{out}), и связать эти ячейки в цикл, который будем называть v-циклом. Для робота добавляется *внутреннее* перемещение на ячейку следующей по v-циклу дуги. При *внешнем* перемещении по дуге $(v,v`)$ робот попадает в ячейку первой дуги в $v`$-цикле. Тем самым, роботу не нужно идентифицировать дугу, а достаточно указать, какой переход он делает: внешний (o) или внутренний (i) (рис.1).

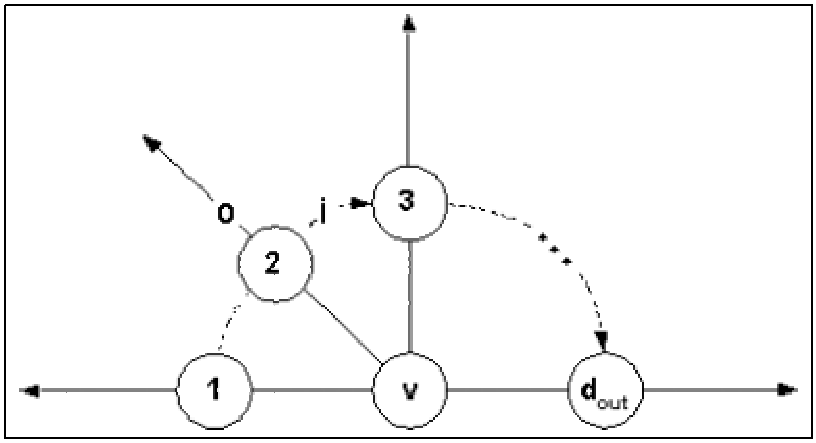


Рис.1: Вершина v и v-цикл исходящих дуг

Мы будем считать, что роботу одновременно доступны для чтения и записи две ячейки: ячейка вершины и ячейка текущей дуги, исходящей из этой вершины. Заметим, что это не является ограничением. Если ячейка вершины v отсутствует, то вместо нее всегда может использоваться ячейка первой дуги v-цикла. Попадая в вершину, робот считывает информацию о вершине из ячейки первой дуги и запоминает ее в своем состоянии. Для

изменения информации о вершине, робот по *v*-циклу перемещается в ячейку первой дуги и делает запись в нее. Предполагается, что первоначально все ячейки пусты (содержат стандартный пустой символ).

Задача обхода графа конечным роботом была поставлена М.О. Рабином в 1967 г. [15].

Автор настоящей статьи занимался этой проблемой в 1969-1971 гг., во время обучения на механико-математическом факультете Московского Государственного Университета. В то время было известно, что на классе сильно связных ориентированных графов длина (нероботного) обхода $\Theta(nm)$, где n – число вершин, а m – число дуг графа. Для любого графа существует обход длиной $O(nm)$ и существуют графы с минимальной длиной обхода $\Omega(nm)$. Также был известен робот R_0 с одним состоянием, который мог обходить любой сильно-связный ориентированный граф за экспоненциальное время, но не умел останавливаться после завершения обхода. Этот робот проходит дуги, исходящие из вершины, все время по одному и тому же циклу: $1,2,...,d_{out},1,2,....$

Автору удалось показать [5], что экспонециальная длина маршрута и невозможность остановки имеют место для любого робота с одним состоянием. Также были предложены два робота для графов с ограниченной полустепенью исхода. Оба алгоритма были основаны на построении в процессе обхода графа *out*-дерева – остова графа, ориентированного от корня, и леса *in*-деревьев, ориентированных к корням. Эти деревья позволяли роботу найти путь из конца пройденной дуги в ее начало. Алгоритмы обходили *out*-дерево, используя различные методы. Первый робот R_1 использовал поиск в глубину (DFS) и строил обход длиной $O(n^3)$, второй робот R_2 использовал поиск в ширину (BFS) и строил обход длиной $O(n^2 logn)$. В переложении для графа без ограничения на полустепень исхода, но с *v*-циклами дуг, эти оценки имеют вид $O(nm+n^3)$ и $O(nm+n^2 logn)$.

В обеих оценках второе слагаемое возникает из-за *проблемы отката* (*backtracking*): необходимости в процессе поиска по *out*-дереву $n-1$ раз возвращаться в предыдущую вершину дерева (ближе к корню). Хотя для отката по одной дуге нужно пройти простой путь длиной $O(n)$, робот, не умеющий «заглядывать вперед», вынужден многократно проходить такой путь для поиска нужной вершины. Робот R_1 строил простой *out*-путь от начальной вершины и лес *in*-деревьев, позволявший ему из любой пройденной вершины попасть на *out*-путь. Когда все дуги, исходящие из конца *out*-пути оказывались пройденными, роботу требовалось вернуться из конца *out*-пути в предыдущую вершину. Для этого использовался контур из простого *in*-пути и простого *out*-пути. Робот помечал первую вершину на *out*-части контура и, проходя одну дугу, запоминал, не является ли ее конец концом *out*-пути. Если нет, то на следующем проходе метка смещалась на следующую вершину контура. В результате требовалось $O(n)$ проходов и суммарно на все возвращения тратилось $O(n^3)$ проходов по дугам. Робот R_2 использовал тот же прием, но вместо *out*-пути у него было *out*-дерево и метка смещалась не на следующую врешину, а на ближайшую *развилку* этого *out*-дерева. За счет этого суммарная оценка уменьшалась до $O(n^2 logn)$.

В 1978 г. появилась статья К. Кобаяши [17], где он представил останавливающийся алгоритм обхода экспоненциальной сложности, основанный на идее DFS. В 1988 г. С. Куттен [18] предложил алгоритм обхода минимальной сложности $O(nm)$, но его робот не был конечным, так как использовал ячейки графа с логарифмическим (от числа вершин) количеством битов памяти. Наконец, в 1993 г. Y.Afek и E.Gafni [1] предложили конечный робот *A* (который они назвали *Traversal-3*), основанный на поиске в глубину (DFS), с оценкой длины обхода $O(nm+n^2 logn)$. Этот робот похож на робот R_1, но делает меньшее число проходов по каждому контуру при откате. Для этого метка ставится на всех

вершинах *out*-пути, кроме его конца, и определяется четность числа помеченных вершин. На следующем проходе удаляются метки с вершин противоположной четности и заново определяется четность числа оставшихся помеченными вершин, и так далее до тех пор, пока не останется одна помеченная вершина – предпоследняя вершина *out*-пути. За счет этого «*метода четности*» вместо $O(n)$ проходов достаточным оказывается $O(\log n)$ проходов.

Как практическая задача обхода графов возникает в сетях передачи данных. Граф интерпретируется как сеть, вершины графа – это узлы сети, а дуги – соединения. Изучение распределенных сетей обычно фокусируется на двунаправленных сетях, соответствующих неориентированным графам. Однако, с однонаправленными сетями (ориентированные графы) приходится иметь дело чаще, чем это можно было бы ожидать. Во-первых, однонаправленные сети возникают как результат сбоев или разрушениях в соединениях. Например, модемная БИС на одном конце соединения может перестать принимать (или посылать) данные, но продолжать работать в другом направлении. Кроме того, односторонние соединения можно обнаружить в радиосетях с асимметричными матрицами передачи как результат различий в пропускной способности станций, в оптоволоконных сетях, и в СБИС [11].

В терминах сети обход графа можно интерпретировать двумя способами. В одном случае единственный мобильный робот – это единственный процесс, который перемещается по сети от узла к узлу, читая пометки в узлах и записывая новые пометки. Узлы сети пассивны и представляют собой лишь память для хранения пометок робота. В другом случае, наоборот, перемещается в сети единственное пассивное сообщение, пересылаемое от узла к узлу. Активными являются узлы сети, которые срабатывают как автоматы только при получении сообщения, отправляя, в свою очередь, новое сообщение по одному из исходящих соединений. С математической точки зрения, эти интерпретации различаются тем, что понимается под состоянием автомата (сообщение в сети или информация в узле) и входными/выходными символами (пометки в узлах или принимаемое и посылаемое сообщение).

В некоторых приложениях, таких как СБИС, размер памяти в узлах и длина сообщения ограничены. Именно в этом случае возникает проблема алгоритмического обхода с константным числом битов в каждом узле и с процессом обхода, несущим конечное количество информации (во второй интерпретации, в однонаправленной сети конечных автоматов, в которой циркулирует единственное сообщение конечной длины).

Интерес автора к этой проблеме вновь возник в 90-ые годы во время работы в группе RedVerst [16], занимающейся задачей тестирования на основе имплицитных формальных спецификаций программных объектов, моделируемых конечными автоматами [6,7,8,19,20]. Здесь также возникает проблема обхода неизвестного графа состояний тестируемого автомата. Правда, как правило, не требуется реализация алгоритма обхода конечным роботом. Некоторым «побочным» результатом этой работы стал предлагаемый в настоящей статье робот R_3, совмещающий поиск в ширину робота R_2 и «метод четности» робота A. В результате достигается оценка длины обхода $O(nm+n^2 \log\log n)$.

2. Постановка задачи

Ориентированный граф, на котором работает робот, можно определить как $G=(V,E,\alpha,\beta,\gamma,\delta,X,\chi)$, где:

- V – множество вершины;
- E – множество дуг (для удобства будем считать, что $E \cap V = \varnothing$);

- $\alpha: E \to V$ – функция, определяющая начальную вершину (начало) дуги;
- $\beta: E \to V$ – функция, определяющая конечную вершину (конец) дуги;
- $\gamma: V \to E$ – функция, определяющая первую дугу в цикле исходящих дуг, с условием:
  - $\forall v \in V\ d_{out}(v) > 0 \Rightarrow \alpha(\gamma(v)) = v$;
- $\delta: E \to E$ – функция, определяющая следующую дугу в цикле исходящих дуг, с условием:
  - $\forall e \in E\ \exists k = 0..d_{out}(\alpha(e))-1\ \delta^k(e) = \gamma(\alpha(e))$, где $\delta^k = \delta \circ \delta \circ \ldots \circ \delta$ и знак суперпозиции $\circ$ применяется $k-1$ раз;
- X – множество символов, которые могут храниться в ячейках вершин и дуг;
- $\chi: V \cup E \to X$ – функция, определяющая символы, хранящиеся в ячейках вершин и дуг.

Граф *конечен*, если конечны множества V и E. Число вершин конечного графа обозначим $n = |V|$. Две дуги e и $e`$ *смежны*, если $\beta(e) = \alpha(e`)$. *Маршрут или путь* (*path*) – это последовательность смежных дуг. Маршрут проходит через вершину, если она является началом или концом некоторой дуги маршрута. Начало первой дуги маршрута называют началом маршрута, а конец последней дуги – концом маршрута. *Простой путь* (*simple path*) – маршрут, не проходящий через одну вершину более одного раза. *Контур* (*cycle*) – маршрут, начало и конце которого совпадают; в *простом контуре* (*simple cycle*) начало и конец – единственные совпадающие вершины. *Обход* – маршрут, содержащий все дуги графа. Граф *сильно связен*, если любая пара вершин связана некоторым маршрутом.

Робот на графе G определяется как $R = (Q, X, T)$, где:

- Q – множество состояний;
- X – множество входных символов (совпадает с множеством символов в ячейках графа);
- $T \subseteq Q \times X \times X \times Q \times X \times X \times \{i, o\}$ – множество переходов.

В каждый момент времени робот располагается в текущей вершине $v \in V$ на текущей дуге $e \in E$, исходящей из v, то есть, $\alpha(e) = v$. Робот находится в состоянии $q \in Q$, читает символ вершины $x_v = \chi(v)$ и символ дуги $x_e = \chi(e)$. Переход $(q, x_v, x_e, q`, x`_v, x`_e, i/o) \in T$ означает, что робот переходит в состояние $q`$, записывает символы в ячейку вершины $\chi(v) = x`_v$ и в ячейку дуги $\chi(e) = x`_e$. При внутреннем переходе (i) робот остается в той же вершине v, но переходит на следующую дугу в v-цикле $\delta(e)$. При внешнем переходе (o) робот переходит по дуге e в ее конечную вершину $\beta(v)$ на первую исходящую из нее дугу $\gamma(\beta(v))$.

Робот *конечен*, если множества Q и X конечны. Робот *детерминирован*, если для каждой тройки $(q, x_v, x_e) \in Q \times X \times X$ существует не более одного перехода $(q, x_v, x_e, q`, x`_v, x`_e, i/o) \in T$. Если для некоторой такой тройки (q, x_v, x_e) нет ни одного перехода, будем говорить, что робот *останавливается* в состоянии q в вершине с символом x_v на дуге с символом x_e.

Будем считать, что один символ $\varepsilon \in X$ выделен как *начальный* символ, который находится во всех ячейках вершин и дуг в начале работы робота. Вершина, с которой робот начинает работать, будем называть *начальной* и обозначать v_0, начальной дугой является первая исходящая дуга $\gamma(v_0)$.

Последовательность внешних переходов, которые делает робот R на графе G с начала работы, очевидно, определяет маршрут в G, который мы будем называть пройденным маршрутом. Если робот останавливается, этот маршрут конечен. Робот обходит граф, если он останавливается на этом графе и пройденный маршрут является обходом. Если каждая дуга, исходящая из вершины v, принадлежит маршруту P, эту вершину будем называть

полностью пройденной (в маршруте *P*). Очевидно, для сильно связных графов пройденный маршрут является обходом тогда и только тогда, когда все его вершины полностью пройдены.

Задача, решаемая в настоящей статье, – построить конечный робот, который обходит любой конечный сильно связный ориентированный граф с любой начальной вершиной.

Произвольный ориентированный граф разбивается на компоненты сильно-связности. Для вершины *v* через *K(v)* будем обозначать компонент, которому она принадлежит. Дуга, начало и конец которой принадлежат разным компонентам, будем называть *связующей*. *Графом 1-го рода* будем называть граф с линейным порядком компонентов, в котором из каждого непоследнего компонента исходит ровно одна дуга, ведущая в следующий компонент (рис.2). *Пройденным графом* маршрута будем называть подграф, состоящий из дуг маршрута и инцидентных им вершин, и обозначать G_t.

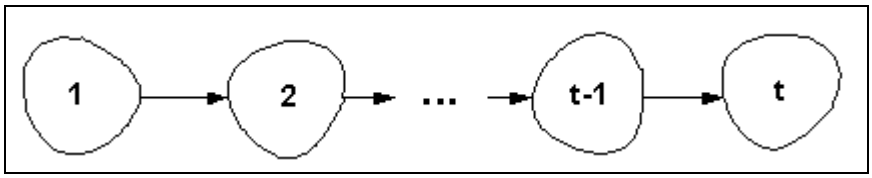


Рис.2: Граф 1-го рода

Лемма 1: *Пройденный граф маршрута является графом 1-го рода. Начало и конец маршрута принадлежат, соответственно, его первому и последнему компонентам.*

Доказательство этой Леммы очевидно. □

3. Описание робота R_3

Предлагаемый робот R_3 удовлетворяет следующему ограничению: повторный проход по дуге разрешается делать только из полностью пройденной вершины. Фактически, в неполностью пройденной вершине *v* робот R_3 ведет себя так же, как робот R_0: проходит все дуги в порядке *v*-цикла, который не зависит от робота и задан самим устройством графа. В частности, это означает, что поведение робота R_3 в вершине не зависит от числа исходящих из нее непройденных дуг.

Хотя робот предназначен для обхода сильно-связных графов, он может работать на любом ориентированном графе и на конечном графе останавливается через конечное число шагов.

Алгоритм робота мы будем записывать на языке Си. Ячейки вершин и дуг представляются структурами, состоящие из нескольких *полей*, в каждом из которых может храниться конечное число значений – *меток*. Тем самым, множество символов *X* – это множество значений таких структур. Начальным символом ε будем считать структуру с нулевыми значениями всех полей. Текущую вершину будем обозначать через *v*, а текущую исходящую дугу через *p*; таким образом, доступ в полю *field* ячейки вершины или дуги осуществляется конструкцией, соответственно, *v.field* или *p.field*. Состояние робота – это набор значений переменных состояния, которыми являются обычные переменные языка Си (кроме *v* и *p*). Для внутреннего и внешнего перемещений будем использовать внешние процедуры *Next* и *Traverse*, изменяющие *v* и *p*. Внешняя процедура *Sink* позволяет определить, является ли текущая вершина конечной (вершиной без исходящих дуг) (рис.3).

```
/* Внутреннее перемещение: p=δ(p) */
Next ();

/* Внешнее перемещение. v=β(p), p=γ(β(p)) */
Traverse ();

/* Возвращает 1, если текущая вершина конечная d_out(v)=0, и 0, в противном случае */
unsigned Sink ();
```

Рис.3: Внешние процедуры робота

a. Структура на графе

Опишем структуру данных на графе, которую создает и использует робот. Эта структура состоит из ячеек вершин и ячеек дуг, связанных в циклы исходящих дуг для каждой вершины. Содержание ячеек показано на рис.4; инициализирующие значения указаны по состоянию до начала работы робота.

```
struct vertex {
        unsigned close: 1 = 0;
        unsigned root: 1 = 0;
        unsigned end: 1 = 0;
        unsigned new: 1 = 0;
        unsigned crotch: 1 = 0;
};
vertex v;
```

```
struct arc {
        unsigned status: 2 = 0;
        unsigned in: 1 = 0;
        unsigned out: 1 = 0;
};
arc p;
```

Рис.4: Структура ячеек вершины и дуги

Close-вершина *v* (*v.close == 1*) – это вершина, про которую робот знает, что она полностью пройдена. Это происходит тогда, когда робот повторно заходит в вершину после того, как вышел из нее по последней непройденной дуге. Вершину, не являющуюся *close*-вершиной, будем называть *open*-вершиной (v.close == 0).

Для дуги поле *status* может принимать четыре значения (рис.5):

- *0*-дуги – непройденные дуги; ячейка *0*-дуги содержит пустой символ.
- *1*-дуги – пройденные дуги, образующие *out*-дерево T_1, ориентированное от корня – начальной вершины v_0, содержащее все *open*-вершины, причем все листья T_1 *open*-вершины.
- *2*-дуги – бывшие *1*-дуги. *1*-дуги и *2*-дуги вместе образуют *out*-дерево T_{12} – остов пройденного графа G_t, ориентированный от корня – начальной вершины v_0. Очевидно, T_1 – поддерево T_{12} с тем же корнем v_0.
- *3*-дуги – хорды дерева T_{12}, то есть, дуги, начало и конец которых принадлежит T_{12}, а сами они не принадлежат T_{12}.

В каждой вершине дерева T_1 метку *out* имеет одна исходящая дуга, по которой робот последний раз выходил из вершины, когда двигался по ранее непройденным дугам или по дугам T_1 в поиске непройденных дуг. В *open*-вершине *v out*-дугой может быть пройденная дуга любого статуса (1,2 или 3), за которой в *v*-цикле находится либо 0-дуга, либо, если

все дуги, исходящие из *v*, пройдены, первая дуга *v*-цикла. Если *v close*-вершина, то исходящая из нее *out*-дуга – это всегда *1*-дуга (рис.5)

In-дуги образуют *лес in*-деревьев, покрывающий все пройденные вершины (рис.6). Каждое *in*-дерево – это остов компонента пройденного графа G_t, ориентированный к своему корню, который мы будем называть также *корнем компонента*. Корень начального компонента совпадает с начальной вершиной v_0, а корень другого компонента – с концом связующей дуги пройденного графа G_t, ведущей в этот компонент. Согласно Лемме 1, все корни располагаются на одном пути *out*-дерева T_1 от корня до листа.

Метка вершины *root* показывает, что вершина является корнем компонента. Остальные метки вершины носят вспомогательный характер и мы объясним их в процессе описания алгоритма.

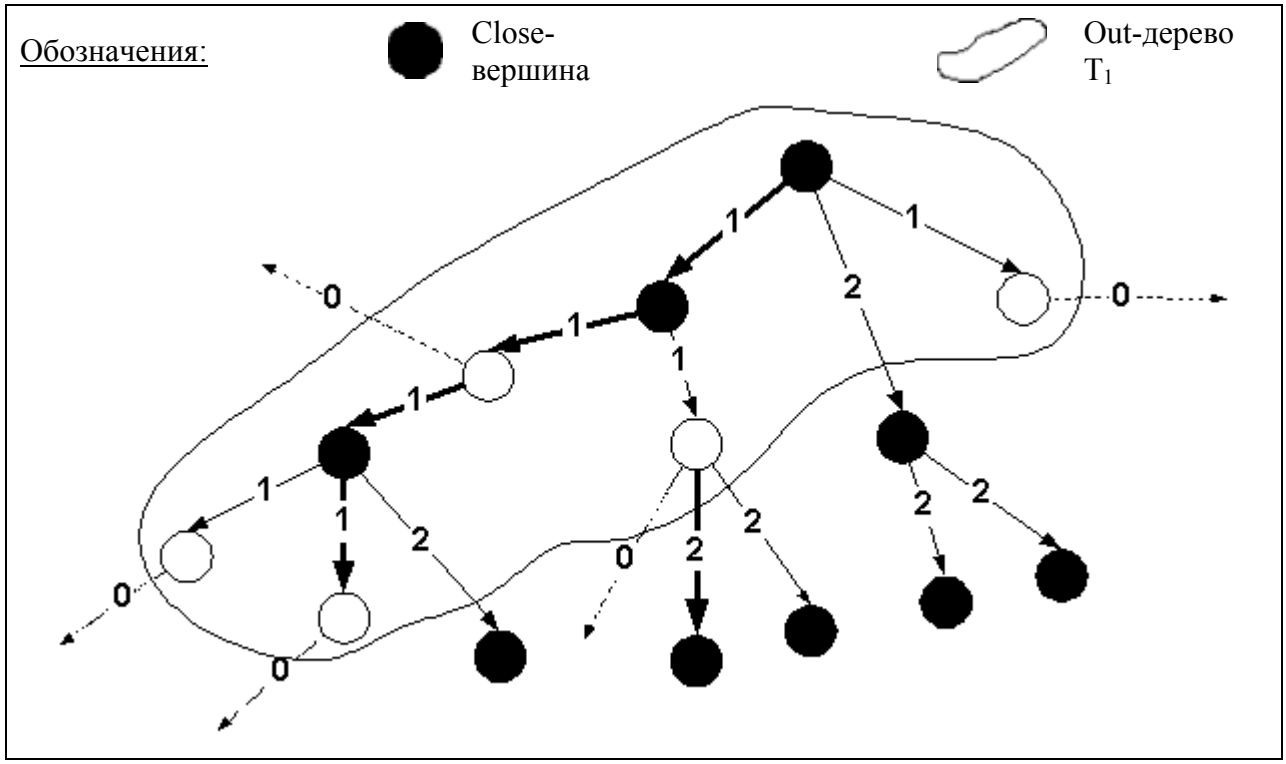


Рис.5: Out-деревья T_1 и T_{12}

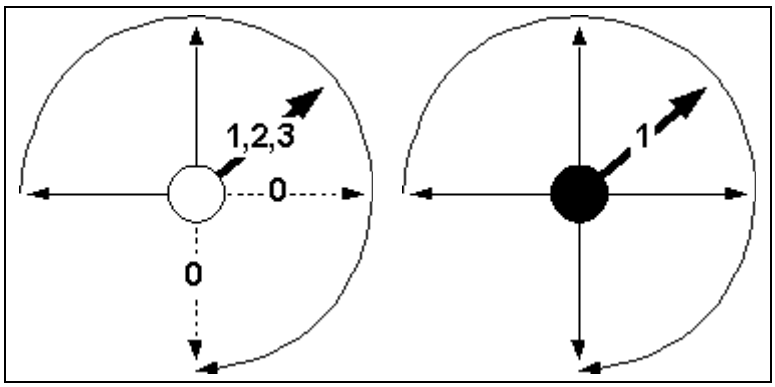


Рис.6: Out-дуга в цикле дуг, исходящих из вершины

Обозначения:

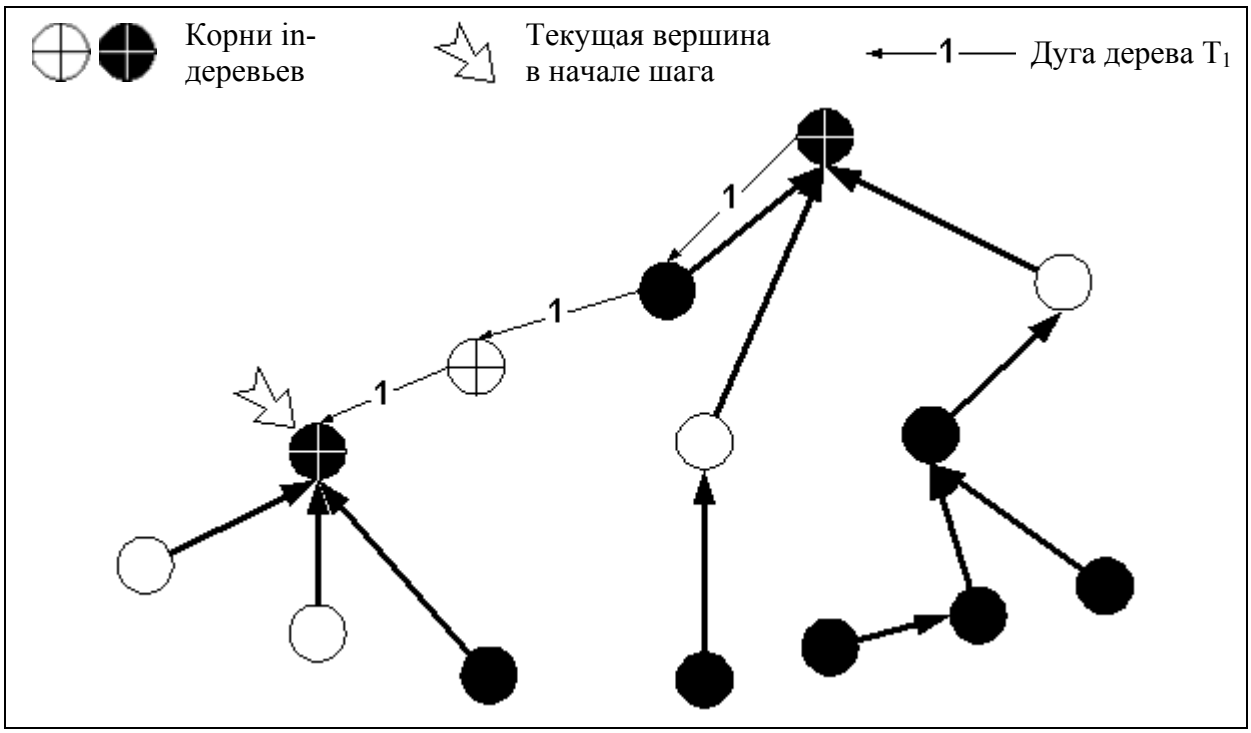


Рис.7: Лес in-деревьев

b. Алгоритм робота

Работа робота представляет собой последовательность однотипных шагов, описываемых с помощью четырех процедур:

1. Поиск непройденной дуги
2. Проход по непройденным дугам
3. Возвращение по хорде
4. Сокращение дерева T_1

В начале каждого шага текущей вершиной является корень *r* последнего компонента пройденного графа. Каждый шаг, кроме первого и последнего, состоит в последовательном выполнении процедур 1,2,3 или 1,4. Первый шаг отличается от остальных тем, что начинается не с процедуры 1, а с процедуры 2. Последний шаг может закончиться после выполнения процедуры 2 или 4.

Описанный выше смысл разметки вершин и дуг относится к состоянию на начало шага. В середине шага разметка может локально отличаться от стандартной.

Out-дерево используется как средство достижения неполностью пройденной вершины или полностью пройденной, но еще не помеченной как *close*, листовой вершины *out*-дерева T_1 (процедура «Поиск непройденной дуги»). При каждом таком поиске в дереве T_1 выделяется *out*-путь таким образом, чтобы последовательность *out*-путей, соответствующая последовательности шагов робота, имитировала поиск в ширину (BFS) остова графа. Возможны два случая:

- *Случай 1*. Если достигается неполностью пройденная вершина, робот двигается по непройденным дугам до тех пор, пока не попадет в конечную вершину ($d_{out}=0$) и тогда останавливается, или пока не пройдет хорду дерева T_1 (процедура «Проход по непройденным дугам») и тогда возвращается в корень последнего компонента пройденного графа (процедура «Возвращение по хорде»).

- *Случай 2*. Если достигается полностью пройденный лист дерева T_1, робот помечает его как *close* и удаляет конечный отрезок ведущего к нему пути дерева T_1, содержащий только *close*-вершины (процедура «Сокращение дерева T_1»).

В последних двух процедурах робот использует также лес *in*-деревьев: *out*- и *in*-дуги образуют контур, по которому робот многократно двигается для достижения своей цели. Более подробно этот процесс описывается при описании процедур шага.

Общая схема алгоритма дана на рис.8.

```
Robot() {

   /* Первый шаг */
      v.root = 1;
      if ( Проход по непройденным дугам () == "пройдена хорда" )
         Возвращение по хорде ();
      else /* "пришли в конечную вершину" */
         return;

   /* Остальные шаги */
      while (1) {
         if ( Поиск непройденной дуги () == "найдена непройденная дуга" )
         /*  Случай 1 */
            if ( Проход по непройденным дугам () == "пройдена хорда" )
               Возвращение по хорде ();
            else /* "пришли в конечную вершину" */
               return;
         else /* "лист дерева T1 стал close-вершиной" */
         /*  Случай 2 */
            if ( Сокращение дерева T1 () == "последний компонент полностью пройден" )
               return;
            /* else "дерево успешно сокращено" */
      }
}
```

Рис.8: Общий алгоритм робота

Поиск непройденной дуги

Процедура формально описана на рис. 9 и проиллюстрирована рис.10.

Мы находимся в корне *r* последнего компонента, не являющегося листом дерева T_1. Поэтому из него исходит хотя бы одна *1*-дуга. Наша задача – пройти из корня *r* по пути дерева T_1 до начала непройденной дуги или до полностью пройденного (но еще не *close*) листа дерева. При проходе пути дерева в каждой текущей вершине *v* выбираем *1*-дугу, следующую после *out*-дуги *q* в *v*-цикле дуг. Для этого просматриваем *v*-цикл до *out*-дуги *q*, снимаем с нее метку *out* и смотрим следующую по *v*-циклу дугу q_{next}. Здесь возможны три случая:

- *Случай 1*: Вершина *v open*, а q_{next} непройдена (*0*-дуга). Возвращаемся из процедуры с ответом "найдена непройденная дуга".
- *Случай 2*: Вершина *v open*, а q_{next} пройдена (не *0*-дуга). Заключаем, что все исходящие дуги пройдены. Вершину *v* помечаем как *close* и проверяем, является ли *v* листовой вершиной дерева T_1, то есть, исходят ли из нее *1*-дуги. Если вершина *v* листовая,

возвращаемся из процедуры с ответом “лист дерева T_1 стал *close*-вершиной”. Если вершина *v* не листовая, то выполняем те же действия, что в случае 3.

- *Случай 3*: Вершина *v* внутренняя *close*-вершина. В этом случае из *v* исходит хотя бы одна *1*-дуга. Просматриваем *v*-цикл дуг до ближайшей *1*-дуги и ставим на нее метку *out*. Идем по *out*-дуге и повторяем действия для новой текущей вершины.

```
char* Поиск непройденной дуги () {
    while (1) {
        while ( !p.out  ) Next (); /* поиск out-дуги q */
        p.out = 0;
        Next (); /* переход на дугу q_next */
        if ( !v.close  )
            if ( p.status == 0 )

/* Случай 1: Вершина v open, а дуга q_next непройдена */
                return (“найдена непройденная дуга”);

/* Случай 2: Вершина v open, а дуга q_next пройдена */
            v.close = 1;
            /* проверяем, является ли v листом дерева T_1 */
            p.out = 1;
            do Next (); while ( p.status != 1 && !p.out );
            if ( p.status != 1 ) return (“лист дерева T1 стал close-вершиной”);
            while ( !p.out ) Next ();
            p.out = 0;
        }

/* Случай 3: Вершина v внутренняя close-вершина */
        while ( p.status != 1 ) Next (); /* поиск 1-дуги */
        p.out = 1;
        Traverse ();
    }
}
```

Рис.9: Процедура «Поиск непройденной дуги»

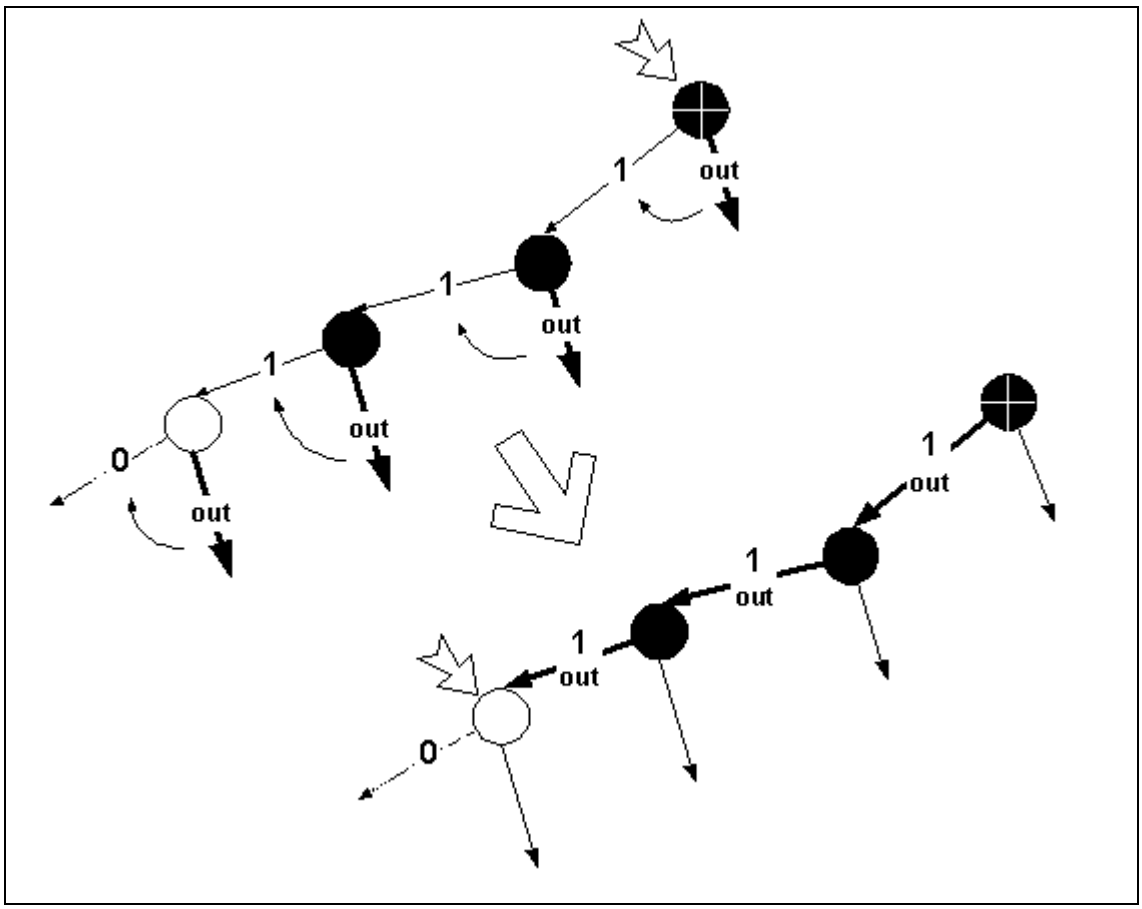


Рис.10: Поиск непройденной дуги

Проход по непройденным дугам

Процедура формально описана на рис. 11 и проиллюстрирована рис.12.

Мы находимся в *open*-вершине *v* и, если из нее исходит хотя бы одна дуга, то текущая дуга *p* – первая по *v*-циклу непройденная дуга. Двигаемся по непройденным дугам до пройденной вершины или до конечной вершины. Для этого сначала проверяем, не является ли *v* конечной вершиной. Если вершина *v* конечная, то возвращаемся из процедуры с ответом “пришли в конечную вершину”. Заметим, что в сильно-связном графе конечная вершина может быть только в том случае, если она единственная вершина графа. В этом случае, очевидно, робот обошел граф. Если вершина *v* не конечная, из нее исходит непройденная дуга *p=(v,w)*. Помечаем вершину *v* меткой *new*, *status* дуги *p* меняем с *0* на *1*, помечаем ее как *out*-дугу и идем по ней в ее конец *w*. Если вершина *w* пройдена, возвращаемся из процедуры с ответом “пройдена хорда”. В противном случае, помечаем ее как *root*-вершину и повторяем действия для вершины *w*.

```
char* Проход по непройденным дугам () {
    while (1) {
        if ( Sink() )

    /* Нет исходящих дуг * /

            return (“пришли в конечную вершину”);
```

```
    /* Есть исходящие дуги * /

        v.new = 1;
        p.status = 1;
        p.out = 1;
        Traverse ();

        if ( p.status != 0 )

        /* Конец дуги пройден */
                return ("пройдена хорда");

        /* Конец дуги не пройден */
        v.root = 1;
    }
}
```

Рис.11: Процедура «Проход по непройденным дугам»

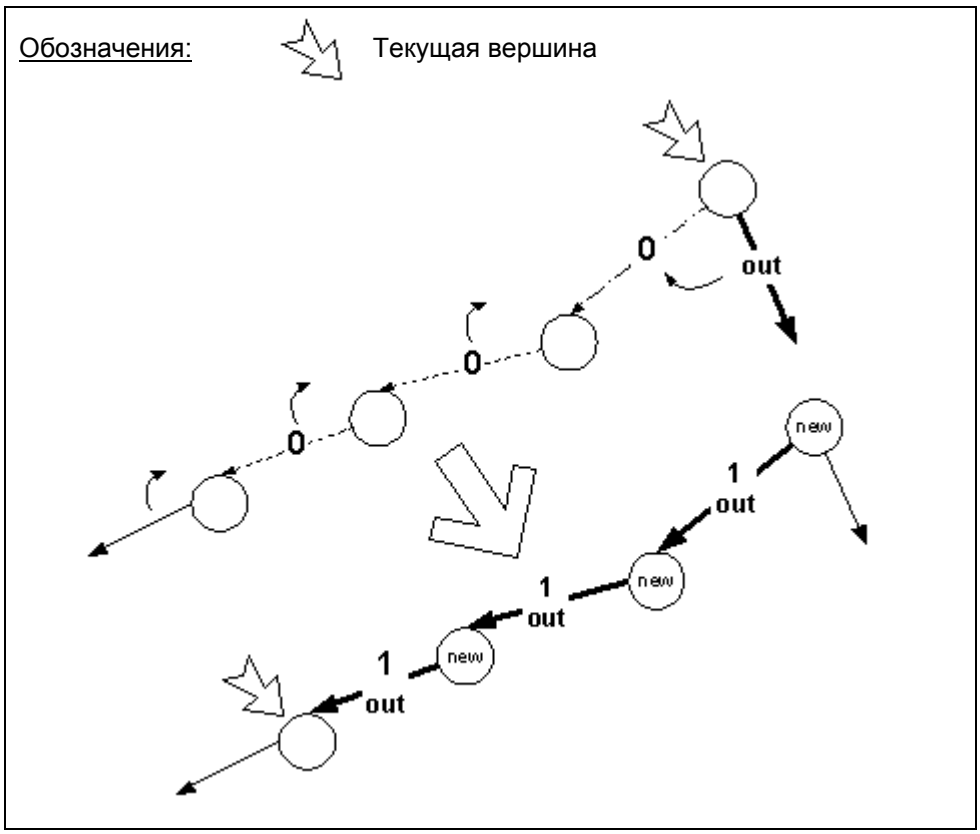


Рис.12: Проход по непройденным дугам

Возвращение по хорде

Процедура формально описана на рис. 13 и проиллюстрирована рис.14.

Мы прошли хорду, которую обозначим *(a,b)*, и находимся в ее конце *v=b*. Наша задача: 1) скорректировать, при необходимости, *in*-деревья; 2) найти вершину *a* и поменять статус

дуги *(a,b)* с *1* на *3*; 3) удалить ненужные метки new; 4) перейти в корень *r* последнего компонента.

Прежде всего, маркируем вершину *b* меткой *end*, чтобы ее можно было опознать, когда мы в нее будем попадать.

Пусть *r* – корень компонента *K(b)*. Мы имеем контур *P* из простого *[b,r]-in*-пути и *[r,b]-out*-маршрута, состоящего из простого *[r,a]-out*-пути и *out*-дуги *(a,b)*. Если *b=r*, *in*-путь нулевой длины; если *a=r*, *out*-путь нулевой длины. Нашу задачу мы будем решать, двигаясь по этому контуру *P*. Проход *in*-маршрута выполняется так: в каждой вершине *v* просматриваем дуги по *v*-циклу до первой *in*-дуги и идем по ней до тех пор, пока не попадем в *root*-вершину. Проход *out*-маршрута выполняется так: в каждой вершине *v* просматриваем дуги по *v*-циклу до первой *out*-дуги и идем по ней до тех пор, пока не попадем в вершину с меткой *end*.

Сначала переходим из вершины *b* в корень *r* по *in*-маршруту.

Затем двигаемся из корня *r* обратно в вершину *b* по *out*-маршруту. При этом движении мы во всех *root*-вершинах *после r* удаляем метку *root* и, начиная с первой встреченной *root*-вершины *после r*, устанавливаем в каждой вершине исходящую из нее *in*-дугу совпадающей с *out*-дугой. В результате нужные компоненты склеются с компонентом *K(b)*, и из всех их вершин по *in*-дугам мы сможем попадать в корень *r*. Кроме этого, устанавливаем переменную *new_counter > 1*, если на *out*-маршруте встретилось больше одной *new*-вершины. Заметим, что вершина *a* является *new*-вершиной.

Теперь нам нужно найти вершину *a*. Для этого роботу придется проходить контур P столько раз, сколько *new*-вершин на его *out*-части: при каждом проходе, кроме последнего, удаляется одна, самая первая встреченная метка *new*.

Мы находимся в вершине *b*, и знаем, совпадает самая первая на *out*-маршруте *new*-вершина с вершиной *a*, или нет. Если нет, двигаемся по котнтуру *P* до корня *r* и далее до ближайшей *new*-вершины, снимаем с нее метку *new* и двигаемся дальше, устанавливая переменную *new_counter > 1*, если на *out*-маршруте осталось больше одной *new*-вершины. Повторяем этот процесс, пока не останется только одна *new*-вершина (*new_counter = 1*), которая, очевидно, совпадает с вершиной *a*.

Теперь мы находимся в вершине *b* и снимаем с нее метку *end*. Затем двигаемся по контуру *P* до *new*-вершины *a* и снимаем с нее метку *new*, а статус *out*-дуги *(a,b)* меняем с 1 на 3. Переходим по *in*-дугам в корень *r* и возвращаемся из процедуры.

```
void Out-Traverse () { /* переход по out-дуге */
    while ( !p.out ) Next ();
    Traverse ();
}


void In-Traverse () { /* переход по in-дуге */
    while ( !p.in) Next ();
    Traverse ();
}
```

```
void Возвращение по хорде () {
    unsigned new_counter  = 0;
    unsigned root_flag = 0;

    v.end = 1; /* вершина b */



          while ( !v.root  ) In-Traverse (); /* [b,r]-in-маршрут */

    while ( !v.end ) { /* [r,b]-out-маршрут */

        if ( v.new && new_counter < 2 ) new_counter += 1; /* подсчет числа  new-вершин */

        Out-Traverse ();

        /* склеивание компонентов */

        if ( v.root ) {
            v.root = 0;
            root_flag = 1;
            while ( !p.out ) Next ();
            p.in = 1;
        }
        else
            if ( root_flag ) {
                while ( !p.in ) Next ();
                p.in = 0;
                while ( !p.out ) Next ();
                p.in = 1;
            }
    }

    /* поиск вершины a */

    while ( new_counter != 1 ) {
        new_counter = 0;

              while ( !v.root  ) In-Traverse (); /* [b,r]-in-маршрут */
        while ( !v.new ) Out-Traverse (); /* ищем первую new-вершину */
        v.new = 0; /* удаление первой new-вершины */
        while ( !v.end ) { /* подсчет числа оставшихся new-вершин */
            if ( v.new && new_counter < 2 ) new_counter += 1;
            Out-Traverse ();
        }
    }
    v.end = 0; /* вершина b */

    /* идем в вершину a */
```

```
        while ( !v.root  ) In-Traverse (); /* [b,r]-in-маршрут */
    while ( !v.new )  Out-Traverse (); /* [r,a]-out-маршрут */

    v.new = 0; /* вершина a */
    while ( !p.out ) Next ();
    p.status = 3; ; /* дуга (a,b) */

    while ( !v.root  ) In-Traverse (); /* [a,r]-in-маршрут */
    return;
}
```

Рис.13: Процедура «Возвращение по хорде»

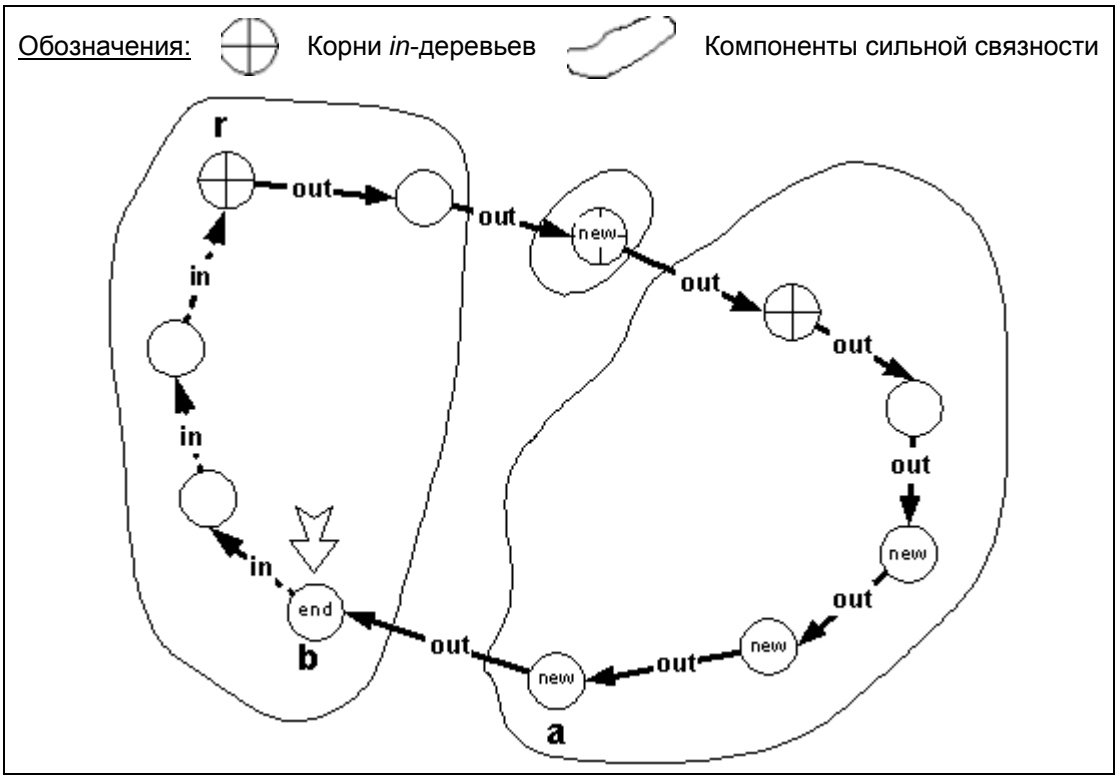


Рис.14: Возвращение по хорде

Сокращение дерева

Процедура формально описана на рис. 15 и проиллюстрирована рис.16.

Текущая вершина a – лист дерева T_1, ставший *close*-вершиной, текущая дуга – *out*-дуга. Наша задача: удалить из дерева T_1 вершину a вместе с заходящей в нее дугой (a_1,a). Если теперь вершина a_1 тоже листовая close-вершина, то мы должны удалить и ее вместе с заходящей дугой (a_2,a_1). И так далее.

Назовем *развилкой close*-вершину, из которой исходит более одной *1*-дуги. Тогда нужно удалить из дерева T_1 максимальный конечный отрезок $[c,a]$ пути $[v_0,a]$, не содержащий *open*-вершин и развилок. Если в T_1 остались *open*-вершины или развилки, то вершина c, очевидно, должна быть последней на пути $[v_0,a]$ *open*-вершиной и/или развилкой. Мы будем искать такую вершину c на пути $[r,a]$, где r – корень последнего компонента. Если окажется, что r не развилка и *close*, робот остановится. Заметим, что в сильно-связном

графе это может быть только в том случае, когда $r=v_0$, и тогда, очевидно, робот обошел граф.

Вершину *c* можно искать аналогично тому, как мы искали вершину *a* в процедуре «Возвращение по хорде». При этом нам нужно будет столько раз пройти путь *[r,a]*, сколько на нем *open*-вершин и развилок. Мы применим здесь «метод четности», предложенный Y.Afek и E.Gafni: пометим все *open*-вершины и развилки пути *[r,a]* меткой *crotch*, а потом на каждом проходе пути будем уменьшать число *crotch*-меток вдвое так, чтобы искомая вершина *c* всегда оставалась помеченной.

Прежде всего, проверяем, не является ли вершина *a root*-вершиной. Если да, то возвращаемся из процедуры с ответом “последний компонент полностью пройден”. В противном случае, маркируем вершину *a* меткой *end*, чтобы чтобы ее можно было опознать, когда мы в нее будем попадать.

Мы имеем контур *P* из простого *[a,r]-in*-пути и простого *[r,a]-out*-пути, причем $a \neq r$. Двигаясь по P маркируем меткой *crotch* корень *r* и далее все *open*-вершины и развилки на *[r,a]-out*-пути. При этом запоминаем четность числа *crotch*-вершин в переменной *crotch_parity* и устанавливаем переменную *crotch_counter>1*, если число *crotch*-вершин больше одной. Заметим, что четность искомой вершины *c* равна *crotch_parity*. Если *crotch_counter>1*, снова проходим маршрут *P*, снимаем метку *crotch* со всех *crotch*-вершин четности |*crotch_parity -1*|, заново запоминаем в *crotch_parity* четность числа оставшихся *crotch*-вершин и снова устанавливаем переменную *crotch_counter>1*, если число оставшихся *crotch*-вершин больше одной. Заметим, что вершина *c* осталась *crotch*-вершиной и ее четность снова равна *crotch_parity*. Так продолжаем до тех пор, пока не останется только одна *crotch*-вершина (*crotch_counter = 1)*, которая, очевидно, совпадает с вершиной *c*.

Двигаемся по контуру *P* до *crotch*-вершины *c* и снимаем с нее метку *crotch*. Далее продолжаем движение от *c* до *a* и меняем статус всех дуг *[c,a]-out*-пути с 1 на 2, и снимаем метку *out* со всех этих дуг, кроме первой дуги, исходящей из *c*. С вершины *a* снимаем ее *end*-метку. Переходим по *in*-дугам в корень *r*.

Теперь нам нужно проверить, что корень *r open*-вершина или из него выходит хотя бы одна *1*-дуга. Если это так, то возвращаемся из процедуры с ответом “дерево успешно сокращено”. В противном случае, возвращаемся из процедуры с ответом “последний компонент полностью пройден”.

```
void Сокращение дерева () {
    unsigned crotch_counter = 0;
    unsigned crotch_parity = 0;

    v.end = 1; /* вершина a */



          while ( !v.root  ) In-Traverse (); /* [a,r]-in-маршрут */

    /* рассстановка меток crotch по [r,a]-out-маршруту */

    while ( !v.end ) {
```

```
    if ( v.root || !v.close ) v.crotch = 1; /* корень r или open-вершина */
    else { /* проверка развилки */
        while ( p.status != 1 ) Next ();
        if ( p.out ) do Next (); while ( p.status != 1 );
        if ( !p.out ) v.crotch = 1; /* развилка */
    }
    if ( v.crotch ) {
        crotch_parity = ( crotch_parity == 1 ) ? 0: 1;
        if ( crotch_counter < 2 ) crotch_counter += 1; /* подсчет числа  crotch -вершин */
    }
    Out-Traverse ();
}

/* метод четности */

while (crotch _counter != 1 ) {
    unsigned v_crotch_parity: 1 = 0;
    unsigned new_crotch_parity: 1 = 0;
    crotch _counter = 0;

        while ( !v.root  ) In-Traverse (); /* [a,r]-in-маршрут */
    while ( !v.end ) { /* [r,a]-out-маршрут */
        if ( v.crotch ) {
            v_crotch_parity = (v_crotch_parity == 1 ) ? 0: 1;
            if  ( v_crotch_parity != crotch_parity )
                v.crotch = 0; /* удаляем метку crotch */
            else {  /* оставшаяся crotch-вершина */
                new_crotch_parity = (new_crotch_parity == 1 ) ? 0: 1;
                if ( crotch _counter < 2 )
                    crotch _counter += 1; /* подсчет числа оставшихся  crotch -вершин */
            }
        }
        Out-Traverse ();
    }
    crotch_parity = new_crotch_parity;
}

/* идем в вершину c */


      while ( !v.root  ) In-Traverse (); /* [a,r]-in-маршрут */
while ( !v.crotch ) Out-Traverse ();  /* [r,c]-out-маршрут */

v.crotch = 0; /* вершина c */

/* удаляем [c,a]-путь */

while ( !p.out ) Next ();
p.status == 2;
while ( !v.end ) {
    Out-Traverse ();
    while ( !p.out ) Next ();
```

```
        p.status == 2;
        p.out == 0;
    }

    v.end = 0; /* вершина a */

    /* идем в корень r */

    while ( !v.root  ) In-Traverse ();

    /* проверяем, что корень r open-вершина или из него выходит хотя бы одна 1-дуга */

    if ( v.close ) {
        while ( !p.out ) Next ();
        do Next (); while ( p.status != 1 && !p.out );
        if ( p.out ) return ("последний компонент полностью пройден");
    }
    return ("дерево успешно сокращено");
}
```

Рис.15: Процедура «Сокращение дерева»

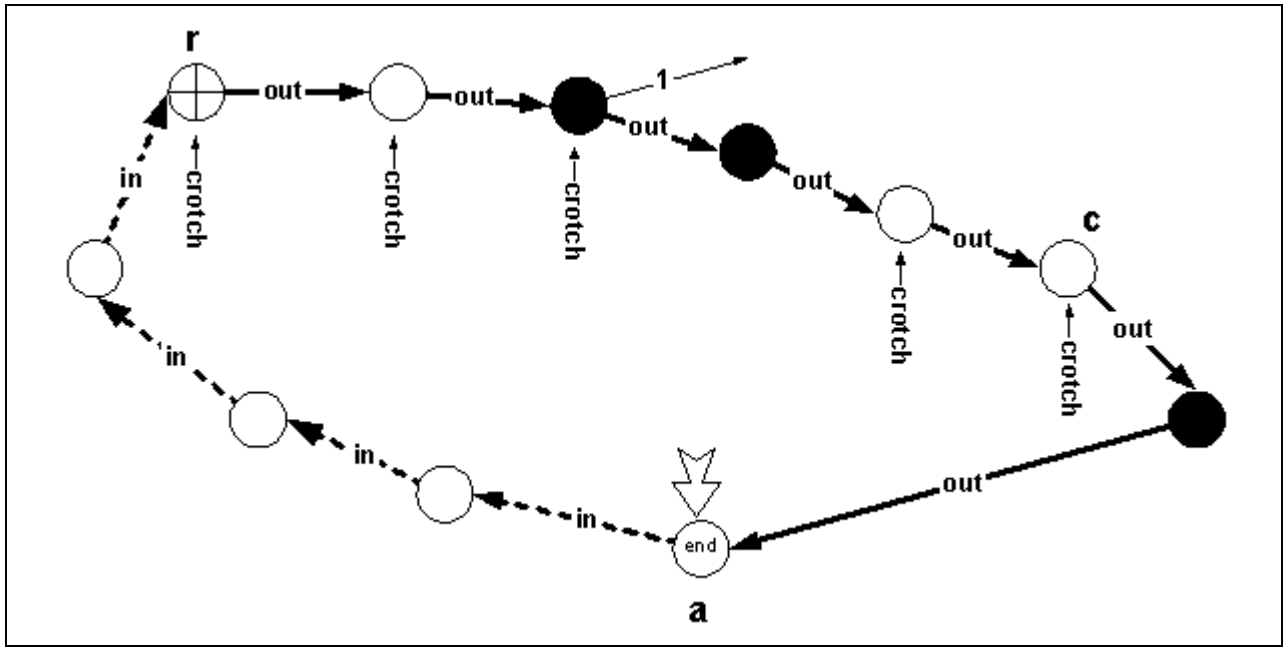


Рис.16: Сокращение дерева

4. Теорема о роботе R_3

Теорема: Робот R_3 останавливается на любом конечном ориентированном графе, и обходит все конечные сильно-связные графы. Длина обхода $O(nm+n^2 loglogn)$, требуемая память $O(n+m)$. Для любых n и m существует граф с n вершинами и $m` \geq m$ дугами, на котором длина обхода роботом $R_3$ равна $\Omega(nm`+n^2 loglogn)$,

Граф, на котором работает робот, обозначим G, VG – множество его вершин.

Робот останавливается на любом конечном графе. Выполнение условий шага перед начальным шагом, а также в конце каждого шага, если они выполнены в его начале, проверяется непосредственно по описанию алгоритма робота. Также легко видеть, что в каждом шаге каждая из 4-х процедур выполняется конечное время. Поэтому, чтобы показать, что через конечное время робот остановится, достаточно показать, что он выполнит конечное число шагов. Действительно, начальный шаг начинается с процедуры 2, «*Проход по непройденным дугам*», где проходится хотя бы одна непройденная дуга, если из начальной вершины v_0 вообще исходит хотя бы одна дуга. В любом последующем шаге в процедуре 1, «*Поиск непройденной дуги*», либо одна *open*-вершина становится *close*-вершиной, либо мы переходим к процедуре 2, где проходим хотя бы одну непройденную ранее дугу. Поскольку число вершин и дуг в графе конечно, число шагов конечно и робот остановится через конечное время.

Робот обходит все конечные сильно-связные графы. Условие остановки: корень последнего компонента пройденного графа G_t становится *close*-вершиной и, значит, полностью пройден, и из него не исходят 1-дуги. Отсюда следует отсутствие *open*-вершин в последнем компоненте и, тем самым, отсутствие непройденных дуг, начинающихся в вершинах последнего компонента, то есть, последний компонент полностью пройден. В сильно-связном графе G это может быть только в том случае, когда пройденный граф G_t состоит из одного компонента и совпадает со всем графом G.

Объем используемой памяти $O(n+m)$. Поскольку ячейки вершин и дуг могут содержать конечное число меток из конечного алфавита, общий объем памяти $O(n+m)$.

Верхняя оценка длины пройденного маршрута $O(nm+n^2 loglogn)$.

В процедуре 1, «*Поиск непройденной дуги*», мы двигаемся по пути дерева T_1 и поэтому проходим простой путь длиной не более $n-1$. При этом либо одна *open*-вершина становится *close*-вершиной, либо мы переходим к процедуре 2, «*Проход по непройденным дугам*», где проходим хотя бы одну непройденную ранее дугу. Тем самым, суммарно по всем шагам в процедуре 1 мы проходим не более $(n-1)(n+m) = O(nm)$ дуг.

В процедуре 2, «*Проход по непройденным дугам*», мы двигаемся по непройденным дугам и, тем самым, суммарно по всем шагам в этой процедуре проходим не более m дуг.

В процедуре 3, «*Возвращение по хорде*», мы проходим несколько раз контур P, состоящий из двух простых путей и одной дуги, то есть, длиной не более $n-1 + n-1 + 1 = 2n-1$, после чего проходится один раз простой *in*-путь длиной не более $n-1$. Число проходов контура P в i-ом шаге равно k_i+1, где k_i - число *new*-вершин на контуре P как начал дуг, впервые пройденных в процедуре 2, «*Проход по непройденным дугам*» (в этом шаге и предыдущих шагах). Поэтому в i-ом шаге мы в процедуре 3 проходим не более $k_i(2n-1)+n-1$ дуг. Поскольку дуга только один раз меняет свой статус с непройденной на пройденную, сумма k_i по всем шагам $\Sigma\{k_i | i=1..\} \le m$. Тем самым, суммарно по всем шагам в процедуре 3 мы проходим не более $m(2n-1)+n-1 = O(nm)$ дуг.

Наибольший интерес представляет процедура 4, «*Сокращение дерева*». Мы покажем, что суммарно по всем шагам робот в процедуре 4 проходит не более $O(n^2+n^2 loglogn)$ дуг. Тем самым, поскольку $n-1 \le m$, общая оценка сверху получится $O(nm+n^2 loglogn)$.

Мы будем рассматривать только те шаги робота, на которых он выполняет процедуру 4, и использовать для обозначения шага индекс $i=1,2,..,I$. На i-ом шаге в процедуре 4 мы несколько раз проходим контур P_i, состоящий из двух простых путей, то есть, длиной не

более $n-1 + n-1 = 2n$, после чего проходится один раз простой *in*-путь длиной не более $n-1$. Два прохода контура P_i мы делаем всегда: 1) в начале для расстановки меток *crotch* и 2) в конце для снятия одной оставшейся метки *crotch*, изменения статуса всех дуг *[c,a]*-пути с 1 на 2, снятия метки *out* со всех дуг *[c,a]*-пути, кроме первой дуги с началом в *c*, и для снятия метки *end* с вершины *a*. Число остальных проходов контура P_i, которые и составляют основную часть *сокращения дерева* (*backtracking*), обозначим b_i. Таким образом, на *i*-ом шаге в процедуре 4 число проходимых дуг не превосходит $2n(2+b_i) + n-1 = 2nb_i+5n-1$.

Обозначим через c_i число *open*-вершин и развилок на *out*-части контура P_i. После первого прохода контура P_i все эти вершины, а также корень *r*, получают метку *crotch*, то есть, не более c_i+1 вершин. Поскольку каждый из b_i проходов приводит к уменьшению числа *crotch*-вершин в два раза, имеем $b_i \le log_2(c_i+1)$. Таким образом, на *i*-ом шаге в процедуре 4 робот проходит не более $2nlog_2(c_i+1) + 5n-1$ дуг. Поскольку из $b_i>0$ следует $c_i \ge 1$, имеем $2nlog_2(c_i+1) + 5n-1 \le 2nlog_2 2c_i + 5n-1 = 2nlog_2 c_i + 7n-1$. Нам достаточно показать, что сумма $log_2 c_i$ по всем шагам $\Sigma log_2 c_i \le O(nloglogn)$.

Заметим, что сокращение дерева происходит только при $n \ge 2$, причем при $n=2$ имеет место $\Sigma log_2 c_i = 0 = nlog_2 log_2 n$. Для $n \ge 3$ воспользуемся неравенством Коши: $(\Pi c_i)^{1/n} \le n^{-1} \Sigma c_i$, из которого следует $\Sigma log_2 c_i \le nlog_2(n^{-1} \Sigma c_i)$. Если мы покажем, что $\Sigma c_i \le Cnlog_2 n$, для некоторой константы *C*, то получим $\Sigma log_2 c_i \le nlog_2(n^{-1} Cnlog_2 n) \le C` nlog_2 log_2 n$, где $C` = 1 + log_2 C / log_2 log_2 3$. Таким образом, достаточно показать, что $\Sigma c_i = O(nlogn)$. Для вершины *v* обозначим через *c(v)* число шагов, в которых она получала метку *crotch*. Очевидно, $\Sigma c_i = \Sigma\{c(v) | v \in VG\}$, где *VG* – множество вершин графа *G*. Для графа *G* мы будем обозначать $C(G) = \Sigma\{c(v) | v \in VG\}$.

Дуга может менять свой статус только в порядке $0 \to 3$ (хорда) или $0 \to 1 \to 2$ (дуга дерева T_{12}). Поэтому дерево T_{12} от шага к шагу может только расти: $T_{12}(i) \subseteq T_{12}(i+1)$. Оно становится максимальным в момент остановки робота $T_{12}(I)$. Дерево T_1 может как расти (процедура 2, «*Проход по непройденным дугам*»), так и сокращаться (процедура 4, «*Сокращение дерева*»). Рассмотрим граф $T(i) = (T_{12}[I] \setminus T_{12}(i)) \cup T_1(i)$, составленный в начале *i*-го шага из 1-дуг и тех 0-дуг, которые в будущем станут 1- или 2-дугами, то есть, из всех дуг дерева $T_{12}[I]$, кроме дуг, уже ставших 2-дугами к началу *i*-го шага. Из вершины дерева $T_{12}(i)$, не принадлежащей дереву $T_1(i)$, то есть, из конца 2-дуги, очевидно, могут исходить только 2-дуги дерева $T_{12}(i)$. Поэтому $T(i)$ также является деревом с корнем v_0 (рис.17). На каждом шаге (напомним, что мы рассматриваем только те шаги, в которых выполняется процедура 4) дерево $T(i)$ теряет ровно одну свою листовую вершину, поскольку на *out*-пути, ведущим в нее, все дуги меняют статус с 1 на 2. Максимальным дерево $T(i)$ является в самом начале, перед первым шагом, $T=T(1)$.

Обозначения:  Дерево $T[i]$  дерево $T_1(i)$

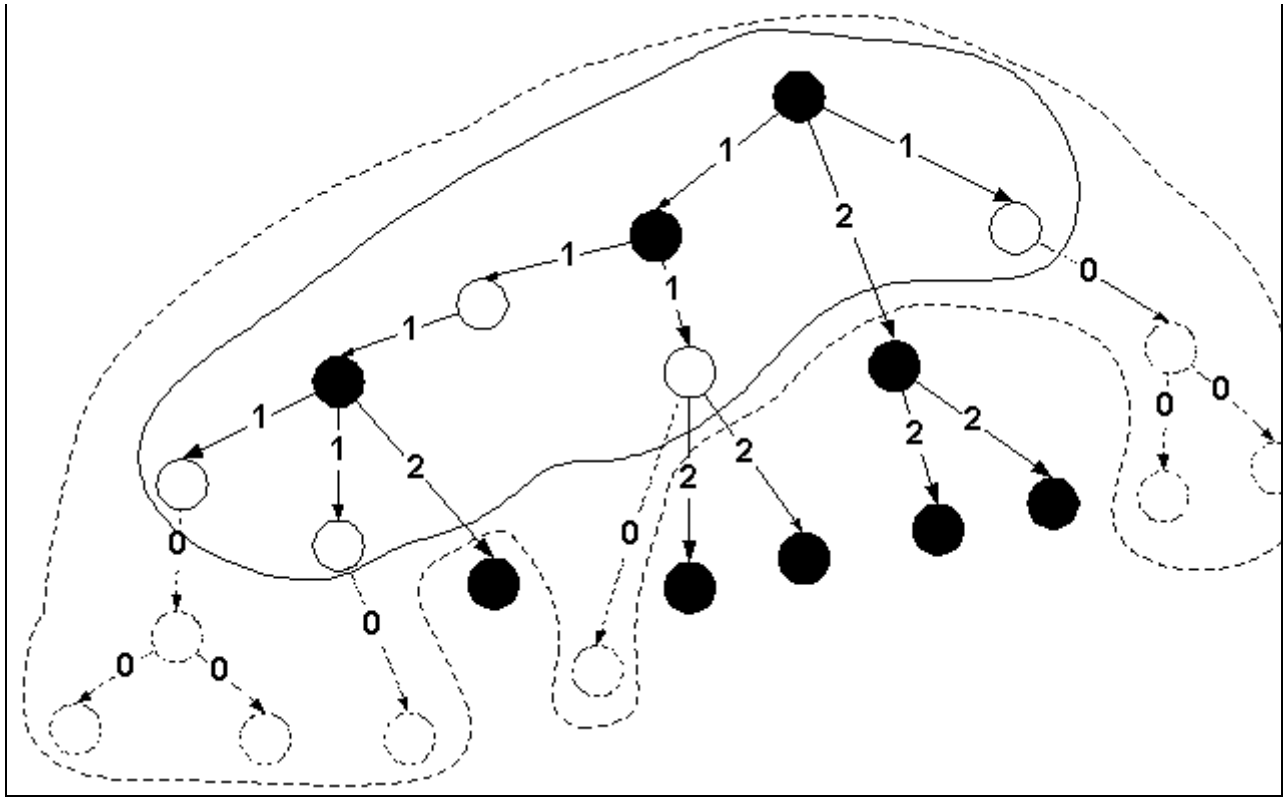


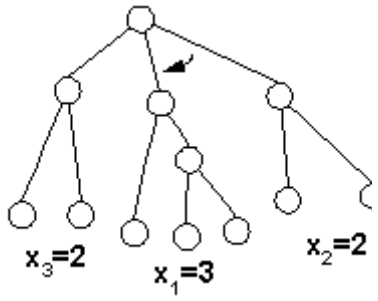
Рис.17: Деревья $T_1(i)$, $T_{12}(i)$, $T_{12}[I]$ и $T[i]$

Для вершины v рассмотрим поддерево $T(v,i)\subseteq T(i)$, порожденное своим корнем v. Очевидно, если вершина v не является развилкой в дереве T, то $c(v)=0$. Теперь рассмотрим случай, когда вершина v развилка дерева T. На каждом шаге, на котором вершина v получает в процедуре 4 метку *crotch*, она является *open*-вершиной или развилкой дерева T_1 и, следовательно, развилкой дерева $T(v,i)$, и это дерево на этом шаге теряет ровно одну листовую вершину. Нас интересует, через сколько шагов вершина v перестает быть развилкой дерева $T(v,i)$; эта величина, очевидно, и есть $c(v)$.

Перенумеруем дуги дерева T, исходящие из v, в порядке v-цикла: $j=1,2,\ldots$. Конец j-ой дуги обозначим v_j, а число листовых вершин в поддереве i-го шага $T(v_j,i)\subseteq T(i)$, порожденном корнем v_j, обозначим через $x_j(i)$, причем, если из v_j не исходят дуги дерева $T(i)$, то будем считать $x_j(i)=1$ (одна листовая вершина, совпадающая с корнем v_j), а если сама вершина v_j не входит в $T(i)$, то $x_j(i)=0$. Очевидно, числа $x_j(i)$ с ростом i не возрастают и имеют максимальные значения перед первым шагом $x_1(1),x_2(1),\ldots$. На каждом шаге, на котором вершина v получает в процедуре 4 метку *crotch*, ровно одно из ненулевых чисел $x_1(i),x_2(i),\ldots$ уменьшается на 1. Перенумеруем эти начальные значения $x_1(1),x_2(1),\ldots$ в порядке невозрастания, используя для нумерации нижний индекс: $x_1 \geq x_2 \geq x_3 \geq \ldots$. Их сумму обозначим $x = x_1 + x_2 + x_3 + \ldots$. Вершина v перестает быть развилкой, когда остается только одно ненулевое число $x_j(i)$. Поскольку числа $x_1(i),x_2(i),\ldots$ уменьшаются на 1 поочередно в порядке v-цикла, возможны три случая, в которых мы вводим для $c(v)$ оценку сверху $s(v)$ (рис.18):

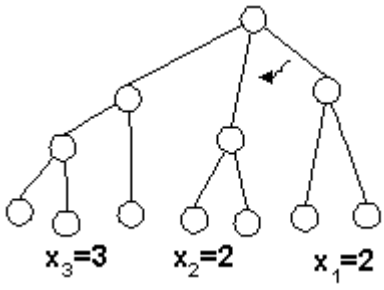
a. $x_1>x_2$ и в v-цикле дуг число $x_1=x_{j1}(1)$ соответствует более *ранней* дуге, чем *хотя бы одна* дуга с числом $x_2=x_{j2}(1)$, то есть, $j_1<j_2$. Тогда $c(v) \leq x_2 + x_2 + x_3 + \ldots = x - x_1 + x_2$
b. $x_1>x_2$ и в v-цикле дуг число $x_1=x_{j1}(1)$ соответствует более *поздней* дуге, чем *любая* дуга с числом $x_2=x_{j2}(1)$, то есть, $j_1>j_2$. Тогда $c(v) \leq (x_2 - 1) + x_2 + x_3 + \ldots = x - x_1 + x_2 - 1$
c. $x_1=x_2$. Тогда $c(v) \leq x_1 + x_2 + x_3 + \ldots - 1 = x - 1$

1) $c(v) \le x - x_1 + x_2 = 7 - 3 + 2 = 6$



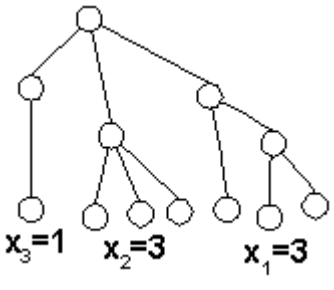
x_3	x_1	x_2	NN
2	3	2	1
2	3	1	2
2	2	1	3
1	2	1	4
1	2	0	5
1	1	0	6
0	1	0	

2) $c(v) \le x - x_1 + x_2 - 1 = 7 - 3 + 2 - 1 = 5$



x_3	x_2	x_1	NN
3	2	2	1
3	2	1	2
3	1	1	3
2	1	1	4
2	1	0	5
2	0	0	

3) $c(v) \le x - 1 = 7 - 1 = 6$



x_3	x_2	x_1	NN
1	3	3	1
1	3	2	2
1	2	2	3
0	2	2	4
0	2	1	5
0	1	1	6
0	1	0	

Рис.18: Оценка c(v)

Равенство "$c(v)=\ldots$" достигается тогда, когда никакой лист дерева $T(v,i)$ не удаляется «без участия» вершины v, то есть, пока вершину v саму не «сократили» из дерева T_1, она в процедуре 4 проходится по *out*-пути для сокращения дерева. Иными словами, выше v по дереву T в процессе работы не образуются корни *in*-деревьев. В противном случае, имеет место строгое неравенство "$c(v)<\ldots$".

Определив $s(v) = x - x_1 + x_2$, мы имеем оценку сверху $c(v) \le s(v)$.

Величина $s(v)$ определяется $s(v)=0$ для вершин, не являющихся развилками дерева T, в частности для листовых вершин, а для остальных вершин – по рекуррентной формуле $s(v) = x - x_1 + x_2$, распространяющей вычисления по дереву T от листьев к корню. Заметим, что, если $c(v)$ зависит от упорядочивания дуг в v-цикле, то $s(v)$ от этого не зависит.

Обозначая, $S = S(G) = \Sigma\{s(v)|v \in VG\}$, имеем $C(G) \le S(G)$. Остов T графа G с заданными v-циклами дуг для каждой вершины v будем обозначать T^v. Очевидно, что $C(G)$ фактически зависит от T^v, а $S(G)$ – от T, и мы будем обозначать $C(T^v)=C(G)$ и $S(T)=S(G)$. Обозначим теперь через X число листовых вершин дерева T. Мы покажем, что $S(T) \le X\log_2 X$.

Прежде всего отметим, что вершину a, не являющуюся развилкой в дереве T, можно не учитывать: если дуги (b,a) и (a,c) слить в одну дугу (b,c), удалив промежуточную вершину a, (если $a=v_0$ корень T, то просто удаляем дугу (a,c) и корнем становится c) то ни одна из рассматриваемых величин $c(v)$ и $s(v)$ не изменится ни для какой развилки v. Повторим эту процедуру до тех пор, пока все нелистовые вершины дерева T не станут развилками. В дальнейшем мы будем рассматривать только такие *гомеоморфно несводимые* деревья.

Вершину дерева будем называть *бинарной*, если из нее исходят ровно две дуги. Дерево будем называть *бинарным*, если в нем все нелистовые вершины бинарные.

Лемма 2: Для любого небинарного дерева T с X листовыми вершинами существует бинарное дерево T^* с тем же числом листовых вершин и $S(T) \le S(T^*)$.

Поскольку в дереве T все нелистовые вершины являются развилками, общее число дуг не меньше удвоенного числа нелистовых вершин $m \ge 2(n-X)$. Обозначим число «лишних» дуг $L = m-2(n-X)$. Мы определим элементарное преобразование дерева T, которое сохраняет число X, уменьшает число L и не уменьшает сумму $S(T)$. Пусть в T есть небинарная нелистовая вершина a, из которой исходят, по крайней мере, три дуги и, следовательно, $x_1 \ge x_2 \ge x_3 \ge 1$. Концы дуг, исходящих из a, обозначим соответственно $a_1, a_2, a_3, \ldots$, где a_i – вершина, чье поддерево $T(a_i)$ имеет x_i листьев. Добавим вершину $a`$ и заменим дуги $(a,a_1)$ и $(a,a_2)$ на три дуги $(a,a`)$, $(a`,a_1)$ и $(a`,a_2)$ (рис.19). Полученное дерево обозначим $T`$.

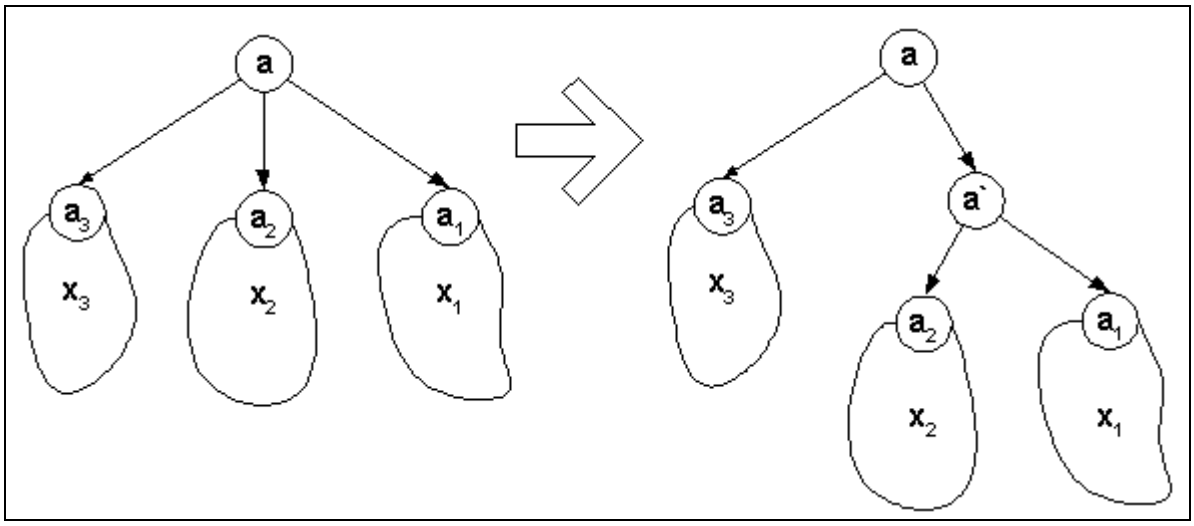


Рис.19: Преобразование небинарного дерева в бинарное

Очевидно, величина $s(v)$ изменилась только в вершине a и добавилась в новой вершине $a`$ (эти измененные значения будем снабжать штрихом $s`(v)$).

(1) $S(T) = \Sigma\{s(v) | v \in VG \& v \ne a\} + s(a)$
(2) $S(T`) = \Sigma\{s(v) | v \in VG \& v \ne a\} + s`(a) + s`(a`)$
(3) $s(a) = (x_1+x_2+x_3+\ldots)-x_1+x_2 = x-x_1+x_2$, где $x = x_1+x_2+x_3+\ldots$
(4) $s`(a) = ((x_1+x_2)+x_3+\ldots)-(x_1+x_2)+x_3 = x-x_1-x_2+x_3$, поскольку теперь деревья $T(a_1)$ и $T(a_2)$ слились в дерево $T(a`)$ с x_1+x_2 листьями, и $x_1+x_2 > x_2 \ge x_3$
(5) $s`(a`) = x_1+x_2-x_1+x_2 = 2x_2$
(6) $S(T`) - S(T) = s`(a) + s`(a`) - s(a) = (x-x_1-x_2+x_3) + 2x_2 - (x-x_1+x_2) = x_3 \ge 1$

Повторяя эту процедуру, пока возможно, получим искомое бинарное дерево T^*. □

Лемма 3: Для $y \ge x \ge 1$ верно следующее неравенство: $y\log_2 y + x\log_2 x + 2x \le (x+y)\log_2(x+y)$.

Рассмотрим разность $f(x,y) = (x+y)\log_2(x+y) - y\log_2 y - x\log_2 x - 2x$. Нам надо доказать, что $f(x,y) \ge 0$. Возьмем вторую производную по x:

$f_x`(x,y) = \log_2(x+y) + \log_2 e - \log_2 x - \log_2 e - 2$
$f_x``(x,y) = (\log_2 e) / (x+y) - (\log_2 e) / x = (\log_2 e)(-y) / (x+y) < 0$ для $y \ge x \ge 1$.

Поэтому нам достаточно проверить неравенство $f(x,y) \geq 0$ на граничных точках 1) $x=1$ и 2) $x=y$.

1) $f(1,y) = (1+y)\log_2(1+y) - 1\log_2 1 - y\log_2 y - 2$. Возьмем первую производную по y:
$f_1`(y) = \log_2(1+y) + \log_2 e - \log_2 y - \log_2 e = \log_2(1 + 1/y) > 0$ для $y \geq 1$.

Следовательно $f(1,y)$ не убывает, и нам достаточно проверить неравенство $f(1,y) \geq 0$ на минимальном значении $y=1$:
$f(1,1) = 2\log_2 2 - 1\log_2 1 - 2 = 0$.

2) $f(y,y) = 2y\log_2 2y - y\log_2 y - y\log_2 y - 2y = 2y(\log_2 2y - \log_2 y - 1) = 2y(\log_2 2 - 1) = 0$. □

Лемма 4: Для бинарных деревьев с X листовыми вершинами оценка $S(T) \leq X\log_2 X$ верна.

Доказательство будем вести индукцией по числу листьев X. Для $X=1$, очевидно, $S(T) = 0 = 1\log_2 1$. Предположим, что утверждение верно для всех бинарных деревьев с числом листьев меньшим X, где $X>1$. Рассмотрим бинарное дерево T с X листьями. Из его корня v_0 ведут две дуги в вершины v_1 и v_2, и поддеревья $T(v_1)$ и $T(v_2)$ имеют, соответственно, X_1 и X_2 листьев; $X = X_1 + X_2$. Пусть для определенности $X_1 \geq X_2$. Тогда:

(1) $S(T) = S(T(v_1)) + S(T(v_2)) + s(v_0)$
(2) По предположению индукции $S(T(v_1)) \leq X_1\log_2 X_1$ и $S(T(v_2)) \leq X_2\log_2 X_2$
(3) $s(v_0) = (X_1 + X_2) - X_1 + X_2 = 2X_2$
(4) По Лемме 3, $S(T) = S(T(v_1)) + S(T(v_2)) + s(v_0) \leq X_1\log_2 X_1 + X_2\log_2 X_2 + 2X_2 \leq (X_1 + X_2)\log_2(X_1 + X_2) = X\log_2 X$ □

Из Лемм 2 и 4 следует, что для любого дерева T с X листовыми вершинами верна оценка $S(T) \leq X\log_2 X$.

Нижняя оценка длины пройденного маршрута $\Omega(nm`+n^2 loglogn)$.

Сначала покажем, что $X\log_2 X$ является точной по порядку оценкой для $C(T^v)$.

Лемма 5: Для любой константы $0<A<1$ существует такая бесконечная последовательность деревьев $T_1, T_2, \ldots$ с бесконечно возрастающим числом листьев $X_1, X_2, \ldots$, где X_i – число листьев в дереве T_i, что $C(T_i^v) \geq X_i\log_2 X_i$.

Рассмотрим граф, остов которого – бинарное выровненное дерево T с расстоянием h от корня до каждого листа. Для сильно-связности графа достаточно из каждой листовой вершины дерева провести хорду в его корень. На уровне h находится $X=2^h$ листьев, а на уровне $i=0,1,\ldots,h-1$ – 2^i нелистовых вершин, и из каждой вершины ведут две дуги в вершины уровня $i+1$. Поскольку для каждой нелистовой вершины v поддеревья T_1 и T_2 с корнями в концах исходящих из v двух дуг имеют равное число листьев $X_1 = X_2$, то $c(v) = (X_1+X_2)-1$. Если v находится на уровне $i<h$, то $X_1 = X_2 = 2^{h-i-1}$. Поэтому:

$C(T^v) = 1(2^h - 1) + 2(2^{h-1} - 1) + 2^2(2^{h-2} - 1) + \ldots + 2^{h-1}(2^1 - 1) + 2^h(2^0 - 1) = h2^h - (1 + 2 + \ldots + 2^{h-1}) = h2^h - 2^h + 1 = X\log_2 X - X + 1$.

Для $0<A<1$ и любого $h > 1/(1-A)$ имеем:

(1) $X = 2^h > 2^{1/(1-A)}$
(2) $\log_2 X > 1/(1-A) > [\ 1/(1-A)\ ]\ (1 - 1/X) = (X-1) / [\ (1-A)X\ ] = (1 - 1/X) / (1-A)$

(3) $(1\text{-}A)X \log_2 X > X–1$
(4) $C(T^v) = X \log_2 X – X + 1 > AX\log_2 X$ □

Подберем число листьев X дерева T из Леммы 5 так, чтобы в T было число вершин n_T ближайшее снизу к $n/2–1$: $n_T=2^{h+1}–1$, $\log_2 n–3<h\le\log_2 n–2$. Для каждой листовой вершины дерева T добавим хорду, ведущую в начальную вершину v_0; число хорд равно 2^h. Для оставшихся $n\text{-}n_T$ вершин добавим простой путь длиной $n\text{-}n_T$ из начальной вершины в корень дерева. Для полученного графа достигается оценка $\Omega(n^2 \log\log n)$ для суммарного *backtracking`a*. В этом графе число дуг $m_T = (n_T–1) + 2^h + (n–n_T)$. Определим $m`=max\{m,m_T\}$. Если $m_T \ge m$, то получим длину обхода $\Omega(nm`+n^2\log\log n)$. Если же $m_T<m$, достаточно добавить недостающие $m\text{-}m_T$ дуг, ведущие из листовых вершин дерева T в начальную вершину (рис.20).

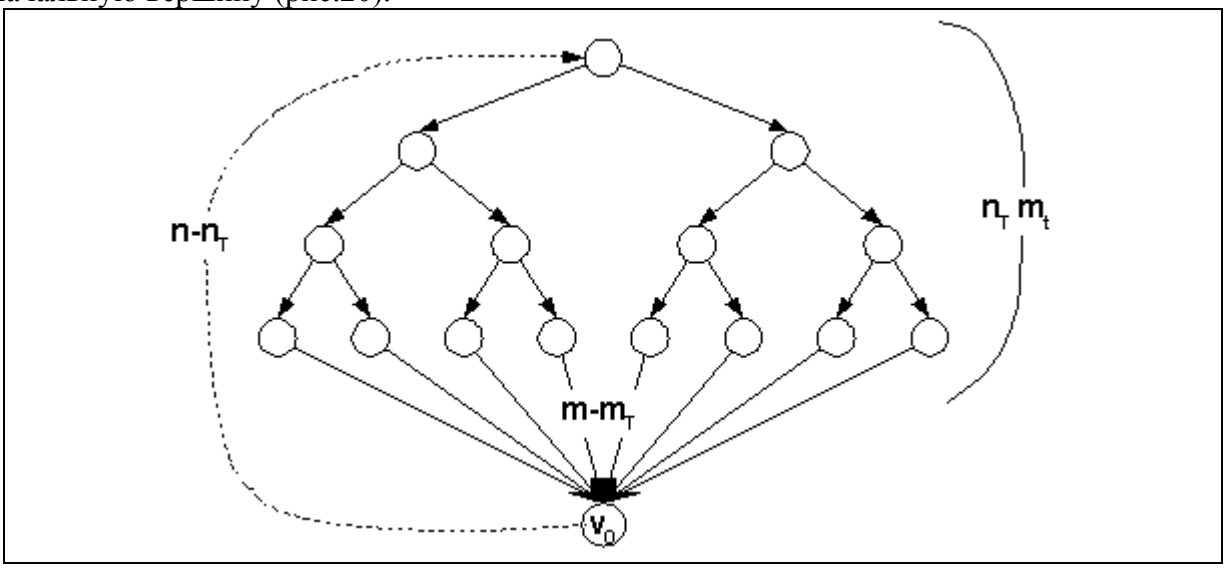


Рис.20: Пример графа, на котором достигается оценка длины обхода $\Omega(nm`+n^2\log\log n)$

На этом доказательство Теоремы о роботе R_3 заканчивается. □

5. Заключение

К сожалению, не известна точная оценка длины обхода конечного ориентированного сильно-связного графа конечным роботом (минимум из верхних оценок алгоритмов по всем возможным алгоритмам обхода). Более того, хотя представляется сомнительным, чтобы конечный робот мог обходить граф за $\Omega(nm)$, тем не менее, это также не доказано.

Литература

1. Y. Afek and E. Gafni, Distributed Algorithms for Unidirectional Networks, SIAM J. Comput., Vol. 23, No. 6, 1994, pp. 1152-1178.
2. S. Albers and M.R. Henzinger, Exploring Unknown Environments, SIAM J. Comput., Vol. 29, No. 4, 2000, pp. 1164-1188.
3. Bhatt, S., Even S., Greenberg, D., and Tayar, R., "Traversing directed eulerian mazes", Graph Theoretic Concepts in Computer Science, Proceedings of WG'2000, U. Brandes and D. Wagner (eds), Lecture Notes in Computer Science No. 1928, pp. 35-46, Springer, 2000.

4. M. Blum and W.J. Sakoda, On the Capability of Finite Automata in 2 and 3 Dimensional Space. In Proceeding of the Eighteenth Annual Symposium on Foundations of Computer Science, 1977. pp. 147-161.
5. И.Б.Бурдонов. Изучение поведения автоматов на графах. Дипломная работа, МГУ им. М.В.Ломоносова, механико-математический факультет, 1971 г.
IB Bourdonov. Обход ориентированных графов автоматами на графе. Degree work, Moscow State University, Faculty of mechanics and mathematics, 1971 (in Russian).
6. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS, vol. 1708, pp. 608621, Springer-Verlag, 1999.
7. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Использование конечных автоматов для тестирования программ. "Программирование", 2000, №2.
IB Bourdonov, AS Kossatchev, VV Kuliamin. Using Finite State Machines in Program Testing. "Programmirovanije", 2000, No. 2 (in Russian).
8. I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proceedings of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.
9. X. Deng and C.H. Papadimitriou, Exploring an Unknown Graph, J.of Graph Th., Vol. 32, No. 3, 1999, pp. 265-297.
10. S. Even, Graph Algorithms, Computer Science press, 1979.
11. S. Even, A. Litman and P. Winkler, Computing with Snakes in Directed Networks of Automata. J. of Algorithms, Vol. 24, 1997, pp. 158-170.
12. D. Hoffman, P.Strooper. ClassBench: a Framework for Automated Class Testing. *Software Maintenance: Practice and Experience*, Vol. 27, No. 5, May 1997, pp. 573-579.
13. L. Murray, D. Carrington. I. MacColl, J. McDonald, P. Strooper. Formal Derivation of Finite State Machines for Class Testing. In J.P. Bowen, A.Fett, M.G.Hinchey, editors, *ZUM'98: The Z Formal Specification Notation.*, *11-th Int. Conf. of Z Users*, Vol. 1493 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998, pp. 42-59.
14. О.Оре. Теория графов. Пер. с англ., М., «Наука», 1968
O.Ore. Theory of Graphs. AMS Colloquium Publications 38. AMS 1962
15. M.O. Rabin, Maze Threading Automata. An unpublished lecture presented at MIT and UC Berkeley, 1967.
16. http://www.ispras.ru/~RedVerst/
17. K. Kobayashi. The firing squad synchronization problem for a class of polyautomata networks. Journal of Computer and System Science, 17:300-318, 1978.
18. S. Kutten. Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed networks. In J. Raviv, editor, Proceedings of the Ninth Internationak Conference on Computer Communication, pages 446-452, October 1988.
19. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Детерминированнный случай. "Программирование", 2003, №5.
20. И.Б.Бурдонов, А.С.Косачев, В.В.Кулямин. Неизбыточные алгоритмы обхода ориентированных графов. Недетерминированнный случай. "Программирование", 2004, №1.