Programming and Computer Software, Vol. 30, No. 4, 2004, pp. 188–203. Translated from Programmirovanie, Vol. 30, No. 4, 2004.
Original Russian Text Copyright © 2004 by Bourdonov.

Traversal of an Unknown Directed Graph by a Finite Robot

I. B. Bourdonov

Institute for System Programming, Russian Academy of Sciences,
Bol'shaya Kommunisticheskaya ul. 25, Moscow, 109004 Russia
e-mail: igor@ispras.ru

Received January 20, 2004

Abstract—A covering path in a directed graph is a path passing through all vertices and arcs of the graph, with each arc being traversed only in the direction of its orientation. A covering path exists for any initial vertex only if the graph is strongly connected, i.e., any of its vertices can be reached from any other vertex by some path. The strong connectivity is the only restriction on the considered class of graphs. As is known, on the class of such graphs, the covering path length is $\Theta(nm)$, where n is the number of vertices and m is the number of arcs. For any graph, there exists a covering path of length $O(nm)$, and there exist graphs with covering paths of the minimum length $\Omega(nm)$. The traversal of an unknown graph implies that the topology of the graph is not a priori known, and we learn it only in the course of traversing the graph. At each vertex, one can see which arcs originate from the vertex, but one can learn to which vertex a given arc leads only after traversing this arc. This is similar to the problem of traversing a maze by a robot in the case where the plan of the maze is not available. If the robot is a "general-purpose computer" without any limitations on the number of its states, then traversal algorithms with the same estimate $O(nm)$ are known. If the number of states is bounded, then this robot is a finite automaton. Such a robot is an analogue of the Turing machine, where the tape is replaced by a graph and the cells are assigned to the graph vertices and arcs. Currently, the lower estimate of the length of the traversal by a finite robot is not known. In 1971, the author of this paper suggested a robot with the traversal length $O(nm + n^2\log n)$. The algorithm of the robot is based on the construction of the output directed spanning tree of the graph and on the breadth-first search (BFS) on this tree. In 1993, Afek and Gafni [1] suggested an algorithm with the same estimate of the covering path length, which was also based on constructing a spanning tree but used the depth-first search (DFS) method. In this paper, an algorithm is suggested that combines the breadth-first search with the backtracking (suggested by Afek and Gafni), which made it possible to reach the estimate $O(nm + n^2\log\log n)$. The robot uses a constant number of memory bits for each vertex and arc of the graph.

1. INTRODUCTION

The graph traversal problem, i.e., the construction of a path passing through all graph edges, is well known. In the case of a directed graph, the problem is more complicated, since the path must traverse any directed edge (arc) only in one direction. A directed graph can be traversed starting from any initial vertex only if it is strongly connected, i.e., any vertex of the graph can be reached from any other vertex by a certain path.

In the majority of studies, the graph is assumed to be a priori known in an explicit form [2, 3] before a covering path is constructed. The case where, before starting the traversal, nothing is known about the graph and the information is obtained in the course of the traversal is more complicated [4, 5]. This is the well-known problem of traversing a maze [6] by a man (or device) in the case where the plan of the maze is unknown. The maze passages and junctions correspond to the graph edges and vertices, respectively. From a junction, we can see passages that form this junction but do not know where any passage leads to until we traverse it and reach another junction. To solve our problem, we, first, are provided with a certain internal memory (e.g., a notepad), where we can write down information obtained in the course of the maze traversal, and, second, may mark up passages and junctions traversed. A directed graph corresponds to a maze with one-way passages.

If the internal memory is limited to a finite number of states, we obtain a robot (finite automaton) on a graph, which is a kind of the Turing machine [1, 7–11]. Instead of the tape, we have the graph; a cell of the tape corresponds to a graph vertex; and the tape motions to the left or to the right correspond to traversing one of the arcs originating from the current vertex. The robot must indicate somehow which outgoing arc has been selected. If the arcs originating from a vertex v are numbered by means of indices 1, …, $d_{out}(v)$, where $d_{out}(v)$ is the outdegree of the vertex v, then the robot may simply indicate the arc number. However, to ensure the finiteness of the output robot alphabet, the outdegree d_{out} is to be bounded from above.

This restriction can easily be removed if we add $d_{out}(v)$ memory cells to each vertex v and combine these cells into a loop, which is further referred to as the v-loop. The robot is supplemented by the *inner* transition to the cell of the next arc in the v-loop. In the case of the *outer* transition along the arc (v, v'), the robot occurs in the cell of the first arc in the v'-loop. Thus, there is no need to identify the arc; the robot may just

indicate what—outer (o) or inner (i)—transition it implements (Fig. 1).

We assume that two cells are simultaneously accessible to robot for reading and writing: the cell of a vertex and the cell of the current arc originating from this vertex. Note that this is not a restriction: if the cell of a vertex v is not available, then the cell of the first arc of the v-loop can always be used instead. When the robot occurs at a vertex, it reads information about the vertex from the cell of the first arc and stores it in its state. To update the information about the vertex, the robot moves along the v-loop to the cell of the first arc and writes to it. All cells are assumed to be initially empty (contain a standard empty symbol).

The problem of the traversal of a graph by a finite robot was set by Rabin [11] in 1967.

The author of this paper studied this problem in 1969–1971, when he was a student of the Department of Mechanics and Mathematics of Moscow State University. At that time, it was known that, in the class of strongly connected directed graphs, the traversal (nonrobot) length was $\Theta(nm)$, where n is the number of vertices and m is the number of arcs in the graph. For any graph, there exists a covering path of length $O(nm)$. There also exist graphs with the covering paths of minimal length $\Omega(nm)$. The robot R_0 having one state was known to be able to traverse any strongly connected directed graph for exponential time but could not stop after completing the traversal. This robot traverses all arcs originating from a vertex always following one and the same loop: 1, 2, …, d_{out}, 1, 2, ….

The author proved [12] that any robot with one state had an exponential path length and could not stop. In that work, two robots for graphs with a bounded outdegree have been suggested. Both algorithms were based on constructing an *out*-tree (output directed spanning tree) and a forest of *in*-trees (input directed spanning trees) in the course of the graph traversal. These trees allowed the robot to find a path from the end of a traversed arc to its beginning. The algorithms used different methods for traversing the *out*-tree. The first robot R_1 applied depth-first search (DFS) and constructed a covering path of length $O(n^3)$, and the second robot R_2 applied breadth-first search (BFS) and constructed a covering path of length $O(n^2\log n)$. Being applied to graphs without restrictions on the outdegree (with v-loops of arcs), these estimates take the form $O(nm + n^3)$ and $O(nm + n^2\log n)$, respectively.

In both estimates, the second addend appears because of the *backtracking*, i.e., returning $n - 1$ times to the previous (closer to the root) vertex in the course of searching for the *out*-tree. Although the backtracking along one arc involves a simple path of length $O(n)$, the robot is not capable of reading the future and has to repeatedly pass along this path when searching for the desired vertex. The robot R_1 constructed a simple *out*-path beginning at the initial vertex and a forest of *in*-trees, which allowed it to reach the *out*-path from any

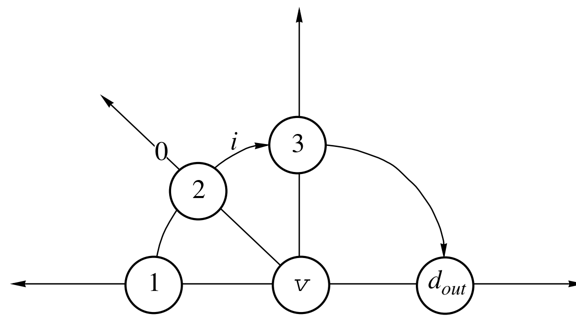

Fig. 1. Vertex v and v-loop of the outgoing arcs.

traversed vertex. When all arcs originating from the end of the *out*-path turned out traversed, the robot had to return from the end of the *out*-path to the previous vertex. For this purpose, a cycle consisting of a simple *in*-path and a simple *out*-path was used. The robot marked the first vertex on the *out*-part of the cycle and, having traversed one arc, remembered whether its end was the end of the *out*-path. If it was not, then, after the next passage, the label was shifted to the next vertex of the cycle. As a result, $O(n)$ passages were required, so that all returns required totally $O(n^3)$ passages along the arcs. The robot R_2 used the same expedient; however, instead of the *out*-path, the *out*-tree was used, and the label was shifted to the nearest crotch of this *out*-tree rather than to the next vertex. Owing to this, the total estimate reduced to $O(n^2\log n)$.

In 1978, Kobayashi [13] suggested an algorithm based on the depth-first search. The algorithm has an exponential complexity and stops after completing the traversal. In 1988, Kutten [14] suggested a traversal algorithm of minimal complexity $O(nm)$; however, his robot was not finite since it used graph cells with the numbers of memory bits logarithmically depending on the number of vertices. Finally, in 1993, Afek and Gafni [1] suggested a finite robot A (called *Traversal*-3) based on the depth-first search with the traversal length estimate $O(nm + n^2\log n)$. This robot is similar to R_1 but requires fewer passages in each cycle in the course of the backtracking. This was achieved through labeling all vertices of the *out*-path, except for its end, and determining the parity of the number of vertices marked. At the next passage, labels of the vertices having the opposite parity are deleted, and the parity of the number of the remaining marked vertices is determined anew, and so on, until there remains one marked vertex, which is the next-to-last vertex of the *out*-path. Owing to this "parity method," the number of passages reduces from $O(n)$ to $O(\log n)$.

The problem of the graph traversal has practical applications in data networks. The graph is interpreted as a network, with the graph vertices and arcs being the network nodes and connections, respectively. Studies in distributed networks focus usually on bidirectional networks, which correspond to undirected graphs.

However, one encounters unidirectional networks (directed graphs) more often than one might expect. First of all, unidirectional networks appear as a result of failures or breaks in connections. For example, a modem LSI can stop receiving (or sending) data at one end of a connection while continuing its operation in the opposite direction. In addition, unidirectional connections can be found in radio networks with asymmetric transmission matrices, which are due to differences in station capacities, in fiber-optic networks, and in VLSI [10].

In the network terms, the traversal of a graph can be interpreted in two ways. In one interpretation, the only mobile robot is one process moving from one node to another, reading labels in the nodes, and writing new labels. The network nodes are passive devises designed only for storing the labels made by the robot. In another interpretation, vice versa, there is one passive message moving from one node to another. Active devices are network nodes, which actuate as automata only when they receive the message, sending, in turn, a new message through one of the outgoing connections. From a mathematical standpoint, these interpretations differ in that they treat differently the notions of the states (messages in the network versus information in the nodes) and input/output symbols (labels in the nodes versus received/transmitted messages).

In some applications, such as VLSI, the size of memory in nodes and the message length are bounded. It is in this case that we face the problem of an algorithmic traversal with a constant number of bits at each node and the traversal process carrying a finite amount of information (in the second interpretation, in a unidirectional network of finite automata with the only circulating message of finite length).

The author became interested in this problem again in the 1990s during his work in the RedVerst [15] group in the course of the execution of a project on software testing based on implicit formal specifications of program objects modeled by finite automata [16–20]. Here, one also faces the problem of traversing an unknown graph of states of the automaton being tested. True, in this case, the finiteness of the robot implementing the traversal algorithm, as a rule, is not required. One side result of that work was the robot R_3 discussed in this paper, which combines the breadth-first search used by R_2 and the “parity method” used by the robot A. As a result, we get the estimate $O(nm + n^2 \log\log n)$ for the covering path length.

2. PROBLEM STATEMENT

A *directed graph* (on which the robot works) can be defined as $G = (V, E, \alpha, \beta, \gamma, \delta, X, \chi)$, where

- V is a set of vertices;
- E is a set of arcs (for convenience, we assume that $E \cap V = \varnothing$);
- α: $E \longrightarrow V$ is a function determining the initial vertex (beginning) of an arc;
- β: $E \longrightarrow V$ is a function determining the terminal vertex (endpoint) of an arc;
- γ: $V \longrightarrow E$ is a function determining the first arc in the loop of the outgoing arcs with the condition

$$\forall v \in V\ d_{out}(v) > 0 \Rightarrow \alpha(\gamma(v)) = v;$$

- δ: $E \longrightarrow E$ is a function determining the next arc in the loop of the outgoing arcs with the condition

$$\forall e \in E\ \exists k = 0, \dots, d_{out}(\alpha(e)) - 1: \delta^k(e) = \gamma(\alpha(e)),$$

where $\delta^k = \delta \circ \delta \circ \dots \circ \delta$ and the superposition sign is applied $k - 1$ times;

- X is a set of symbols that can be stored in the cells of vertices and arcs; and
- $\chi : V \cup E \longrightarrow X$ is a function determining symbols stored in the cells of vertices and arcs.

A graph is said to be *finite* if the sets V and E are finite. The number of vertices of a finite graph is denoted by $n = |V|$. Two arcs e and e' are said to be *adjacent* if $\beta(e) = \alpha(e')$. A *path* is a sequence of adjacent arcs. A path is said to pass through a vertex if this vertex is either the beginning or the endpoint of some arc of the path. The beginning point of the first arc of a path is called the beginning of the path, and the endpoint of the last arc of the path is called its end. A *simple path* is a path that does not pass through one vertex more than once. A *cycle* is a path the beginning and the endpoint of which coincide; in a *simple cycle* the beginning and the endpoint are the only coinciding vertices. A path is referred to as a *covering path* if it contains all arcs of the graph. A graph is said to be *strongly connected* if any two vertices of the graph are connected by some path.

A *robot on a graph G* is defined as $R = (Q, X, T)$, where

- Q is a set of states,
- X is a set of input symbols (coincides with a set of symbols in the cells of the graph), and

$T \subseteq Q \times X \times X \times Q \times X \times X \times \{i, o\}$ is a set of transitions.

At any time, the robot is located at a current vertex $v \in V$ on a current arc $e \in E$ originating from v; i.e., $\alpha(e) = v$. The robot is in a state $q \in Q$ and reads the symbol of the vertex $x_v = \chi(v)$ and the symbol of the arc $x_e = \chi(e)$. The transition $(q, x_v, x_e, q', x'_v, x'_e, i/o) \in T$ means that the robot transfers to the state q' and writes symbols to the cell of the vertex $\chi(v) = x'_v$ and to the cell of the arc $\chi(e) = x'_e$. In the case of an inner transition (i), the robot remains in the same vertex v but transfers to the next arc in the v-loop $\delta(e)$. In the case of an outer transition (o), the robot passes along the arc e to its terminal vertex $\beta(v)$ on the first arc $\gamma(\beta(v))$ originating from it.

The robot is said to be *finite* if the sets Q and X are finite. The robot is *deterministic* if, for any triple $(q, x_v, x_e) \in Q \times X \times X$, there does not exist more than one transition $(q, x_v, x_e, q', x'_v, x'_e, i/o) \in T$. If, for some triple (q, x_v, x_e), no transition exists, the robot is said to *stop* in the state q at a vertex with the symbol x_v on an arc with the symbol x_e.

One symbol $\varepsilon \in X$ is assumed to be *initial*; it is contained in all cells of the graph vertices and arcs at the beginning of the robot operation. The vertex at which the robot starts its operation is called *initial* and denoted by v_0; the initial arc is the first outgoing arc $\gamma(v_0)$.

The sequence of all outer transitions of the robot R in a graph G determines a path in G, which is referred to as a traversed path. If the robot stops, this path is finite. The robot is said to traverse a graph if it stops on this graph and the traversed path is a covering path. If each arc originating from a vertex v belongs to a path P, this vertex is said to be *completely traversed* (in the path P). Clearly, in the case of strongly connected graphs, the traversed path is a covering path if and only if all vertices of the graph are completely traversed.

The problem to be solved in this paper is to construct a finite robot that traverses any finite strongly connected directed graph starting from any initial vertex.

An arbitrary directed graph is partitioned into strongly connected components. Given a vertex v, $K(v)$ denotes the component to which this vertex belongs. A *connecting* arc is an arc the beginning and endpoint of which belong to different components of the graph. A *graph of the first kind* is a graph with a linear order of the components in which each (but the last) component has exactly one outgoing arc leading to the next component (Fig. 2). A *traversed graph* of a path is a subgraph consisting of the arcs belonging to the path and the incidental vertices. We use the notation G_t for this graph.

Lemma 1. A traversed graph of a path is a graph of the first kind. The beginning and the end of the path belong to the first and last components, respectively.

The **proof** of this lemma is obvious.

3. DESCRIPTION OF THE ROBOT R_3

The suggested robot R_3 satisfies the following restriction: an arc can repeatedly be traversed only from a completely traversed vertex. Actually, in an incompletely traversed vertex v, the robot R_3 behaves like the robot R_0: it traverses all arcs in the order they are arranged in the v-loop, which does not depend on the robot and is determined by the graph itself. In particular, this implies that the behavior of the robot R_3 at a vertex does not depend on the number of untraversed arcs originating from this vertex.

Although the robot is designed for traversing strongly connected graphs, it can work in any directed graph and, in the case of a finite graph, stops in a finite number of steps.

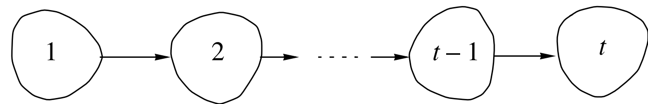

Fig. 2. A graph of the first kind.

We will use C to write an algorithm of the robot. Cells of the vertices and arcs are represented by structures consisting of several *fields*, each of which can store a finite number of values called *labels*. Thus, the set of symbols X is a set of values of such structures. An initial symbol ε is considered to be a structure with zero values in all fields. The current vertex is denoted by v, and the current outgoing arc, by p; thus, to access the field *field* of the cell of the vertex or arc, the constructs *v.field* or *p.field*, respectively, are used. A state of the robot is a set of values of state variables, which are ordinary variables of C (except for v and p). For inner and outer transitions, external procedures *Next* and *Traverse* modifying v and p are used. The external procedure *Sink* is used for determining whether a current vertex is terminal (a vertex without outgoing arcs) (Fig. 3).

3.1. Structure in a Graph

Let us describe a data structure in a graph, which is created and used by the robot. This structure consists of vertex cells and arc cells connected into loops of the outgoing arcs for each vertex. The cell content is shown in Fig. 4; the initializing values depicted correspond to the state existing before the beginning of the robot operation.

The *close*-vertex v (*v.close* == 1) is a vertex that is known to be completely traversed. This happens when the robot repeatedly comes to the vertex after it has left this vertex along the last untraversed arc. A vertex that is not a *close*-vertex is called an *open*-vertex (*v.close* == 0).

For an arc, the field *status* may take the four following values (Fig. 5):

• 0-arcs are untraversed arcs (the cell of a 0-arc contains an empty symbol);

• 1-arcs are traversed arcs forming an output directed *out*-tree T_1 (with the root coinciding with the initial vertex v_0), which contains all *open*-vertices, such that all leaves of T_1 are *open*-vertices;

• 2-arcs are former 1-arcs; 1-arcs and 2-arcs altogether form the *out*-tree T_{12}, which is the output directed spanning tree of the traversed graph G_t (with the root coinciding with the initial vertex v_0); clearly, T_1 is a subtree of T_{12} with the same root v_0;

• 3-arcs are chords of the tree T_{12}, i.e., arcs the beginnings and ends of which belong to T_{12} but the arcs themselves do not belong to T_{12}.

At each vertex of the tree T_1, one outgoing arc has the label *out*. This is the last of the traversed arcs originating from this vertex traversed by the robot when it moved along untraversed arcs or arcs of T_1 searching

```
/* Inner transition: p=((p) */
Next ();

/* Outer transition: v=((p), p=((((p)) */
Traverse ();

/* Returns 1 if the current vertex is terminal dout(v)=0 or 0 otherwise */
unsigned Sink ();
```

Fig. 3. Robot's external procedures.

```
struct vertex {                      struct arc {
      unsigned close: 1 = 0;               unsigned status: 2 =
      unsigned root: 1 = 0;                0;
      unsigned end: 1 = 0;                 unsigned in: 1 = 0;
      unsigned new: 1 = 0;                 unsigned out: 1 = 0;
      unsigned crotch: 1 =           };
      0;                             arc p;
};
vertex v;
```

Fig. 4. Structure of the vertex and arc cells.

for untraversed arcs. At an *open*-vertex v, an *out*-arc may be any traversed arc with an arbitrary status (1, 2, or 3) that is followed in the v-loop either by a 0-arc or, if all arcs originating from v have already been traversed, by the first arc of the v-loop. If v is a *close*-vertex, then the *out*-arc originating from it is always a 1-arc (Fig. 5).

The *in*-arcs form a forest of *in*-trees covering all traversed vertices (Fig. 6). Each *in*-tree is an input directed spanning tree of a component of the traversed graph G_t; the root of this tree is also called the *root of the component.* The root of the initial component coincides with the initial vertex v_0, and the root of another component coincides with the end of the connecting arc of the traversed graph G_t leading to this component. According to Lemma 1, all roots are located on one path in the *out*-tree T_1 leading from the root to a leaf.

The label *root* of a vertex indicates that this vertex is the root of a component. Other labels of a vertex play an auxiliary role and will be explained later in the course of the algorithm description.

3.2. Robot's Algorithm

The operation of the robot can be represented as a sequence of steps including the following four procedures: (1) search for an untraversed arc, (2) traversal of an untraversed arc, (3) returning along a chord, and (4) reduction of the tree T_1.

At the beginning of each step, the current vertex is a root r of the last component of the traversed graph. Each step, except for the first and last ones, consists in the successive execution of procedures 1, 2, and 3 or 1 and 4. The first step differs from the others in that it begins with

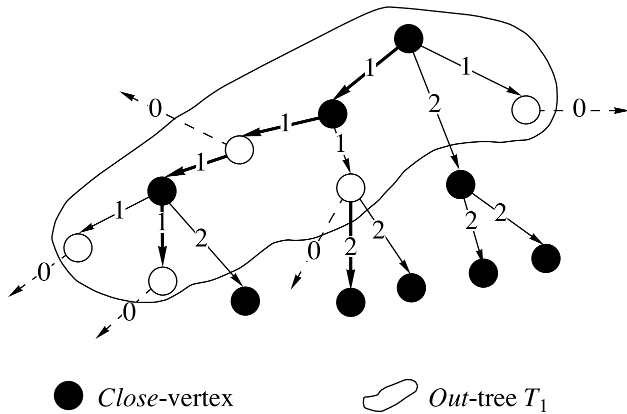

Fig. 5. *Out*-trees T_1 and T_{12}.

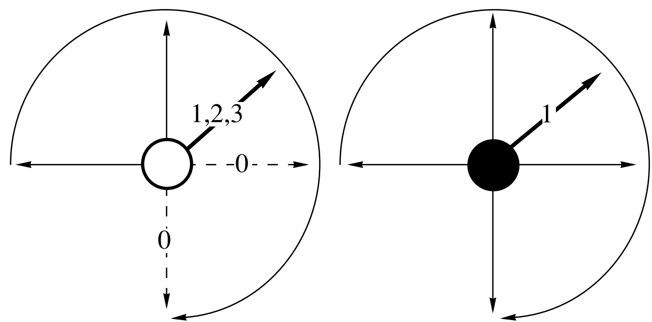

Fig. 6. An *out*-arc in the loop of arcs originating from a vertex.

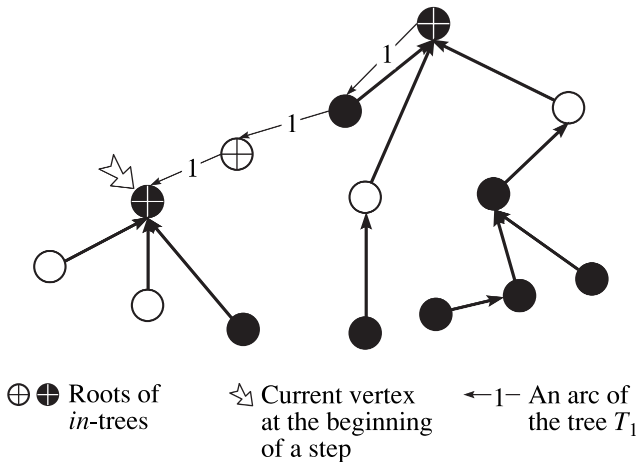

Fig. 7. Forest of *in*-trees.

procedure 2 rather than with procedure 1. The last step may stop after the execution of procedures 2 or 4.

The meaning of the vertex and arc marking described above refers to the state existing before the beginning of the step. In the course of the step execution, the marking may differ from the standard one.

The *out*-tree is used as a means for reaching an incompletely traversed vertex or a completely traversed leaf vertex of the *out*-tree T_1 that has not been marked as *close* yet (the procedure "Search for an untraversed arc"). In the course of each search in the tree T_1, an *out*-path is selected such that the sequence of the *out*-paths corresponding to the sequence of robot's steps simulates the breadth-first search in the spanning tree of the graph. The two following cases are possible:

• *Case 1.* If an incompletely traversed vertex is reached, the robot moves along untraversed arcs until it occurs in a terminal vertex ($d_{out} = 0$) and, then, stops, or until it traverses a chord of the tree T_1 (the procedure "Traversal of an untraversed arc"); in the latter case, it returns to the root of the last component of the traversed graph (the procedure "Returning along a chord").

• *Case 2.* If a completely traversed leaf of the tree T_1 is reached, the robot marks it as *close* and deletes the last segment of the path (containing only *close*-vertices) in the tree T_1 leading to it (the procedure "Reduction of the tree T_1").

In the last two procedures, the robot uses also a forest of *in*-trees: *out*- and *in*-arcs form a cycle along which the robot repeatedly moves trying to attain its goal. This process is discussed in more detail in the descriptions of the procedures constituting the steps.

The general scheme of the algorithm is shown in Fig. 8.

3.2.1. Search for an untraversed arc. The procedure is formally described in Fig. 9 and is illustrated by an example shown in Fig. 10.

We are situated in a root r of the last component, which is not a leaf of the tree T_1. Hence, it has at least one outgoing 1-arc. Our goal is to move from the root r along a path in the tree T_1 to the beginning of an untraversed arc or to a completely traversed (but not marked as *close* yet) leaf of the tree. During the passage, at each current vertex v, we select a 1-arc that follows an *out*-arc q in the v-loop of arcs. To do this, we look through

```
Robot() {
  /* First step */
     v.root = 1;
     if (Traversal of untraversed arcs () == "chord traversed")
         Returning along a chord ();
     else /* "came to a terminal vertex" */
         return;

  /* Other steps */
     while (1) {
       if (Search for untraversed arcs () == "untraversed arc is found")
       /* Case 1 */
          if (Traversal of untraversed arcs () == "chord traversed")
              Returning along a chord ();
          else /* "came to a terminal vertex" */
              return;
       else /* "leaf of tree T1 became a close-vertex" */
       /* Case 2 */
          if (Reduction of tree T1 () == "last component is completely traversed")
              return;
          /* else "tree is successfully reduced" */
    }
}
```

Fig. 8. General scheme of robot's algorithm.

```
char* Search for an untraversed arc () {
  while (1) {
    while ( !p.out ) Next (); /* search for an out-arc q */
    p.out = 0;
    Next (); /* transition to the arc qnext */
    if ( !v.close )
        if ( p.status == 0)

/* Case 1: Vertex v is open, and arc qnext is not traversed */
         return ("an untraversed arc is found");

/* Case 2: Vertex v is open, and arc qnext is traversed */
        v.close = 1;
        /* check whether v is a leaf of the tree T1 */
        p.out = 1;
        do Next (); while ( p.status != 1 && !p.out );
        if (p.status != 1) return ("leaf of the tree T1 has become a close-vertex");
        while ( !p.out ) Next ();
        p.out = 0;
    }

/* Case 3: Vertex v is an internal close-vertex */
    while (p.status != 1) Next (); /* search for a 1-arc */
    p.out = 1;
    Traverse ();
  }
}
```

Fig. 9. Procedure "Search for untraversed arcs."

the v-loop until the *out*-arc q is found, remove the label *out* from it, and inspect the next arc q_{next} in the v-loop. Here, the following three cases are possible:

• *Case 1*: v is an *open*-vertex, and q_{next} is an untraversed arc (0-arc). Return from the procedure with the verdict "an untraversed arc is found."

• *Case 2*: v is an *open*-vertex, and q_{next} is a traversed arc (not a 0-arc). Conclude that all outgoing arcs are traversed. Mark v as *close* and check whether v is a leaf vertex of the tree T_1, i.e., whether it has outgoing 1-arcs. If v is a leaf vertex, return from the procedure with the verdict "the leaf of the tree T_1 became a *close*-vertex." If v is not a leaf vertex, perform the same actions as in Case 3.

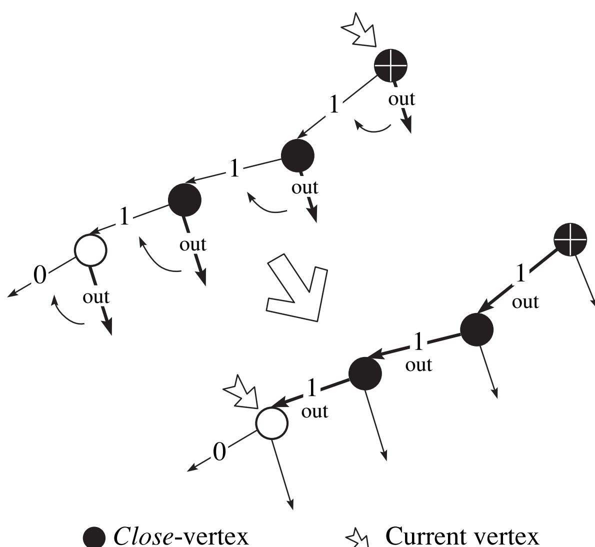

Fig. 10. Search for untraversed arcs.

• *Case 3*: v is an inner *close*-vertex. In this case, at least one 1-arc originates from v. Look through the v-loop until the closest 1-arc is found and place the label *out* on it. Go along the *out*-arc and repeat these actions for a new current vertex.

3.2.2. Traversal of untraversed arcs. The procedure is formally described in Fig. 11 and is exemplified in Fig. 12.

We are situated in an *open*-vertex v. If at least one arc originates from it, then the current arc p is the first untraversed arc in the v-loop. Move along untraversed arcs to a traversed or terminal vertex. To this end, we first check whether v is a terminal vertex. If v is terminal, then return from the procedure with the verdict "reached a terminal vertex." Note that, in a strongly connected graph, a vertex may be terminal only if it is the only vertex of the graph. In this case, the graph has obviously been traversed by the robot. If v is not a terminal vertex, there exists an untraversed arc $p = (v, w)$ originating from it. Mark the vertex v by the label *new*, modify *status* of the arc p from 0 to 1, mark it as an *out*-arc, and move along it to its end w. If w is traversed, return from the procedure with the verdict "the chord is traversed." Otherwise, mark it as a *root*-vertex and repeat the actions for the vertex w.

```
char* Traversal of untraversed arcs () {
  while (1) {
    if ( Sink() )

  /* No outgoing arcs */

      return ("came to a terminal vertex");

  /* Outgoing arcs are available */

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

    if ( p.status != 0 )

    /* The end of the arc has been traversed */
         return ("the chord has been traversed");

    /* The end of the arc has not been traversed */
    v.root = 1;
  }
}
```

Fig. 11. Procedure "Traversal of untraversed arcs."

3.2.3. Returning along a chord. The procedure is formally described in Fig. 13 and illustrated in Fig. 14.

We have traversed a chord (a, b) and are situated at its end $v = b$. Our goal is as follows: (1) to modify, if necessary, the *in*-trees, (2) to find the vertex a and change the status of the arc (a, b) from 1 to 3, (3) delete all redundant labels *new*, and (4) go to the root r of the last component.

First of all, we mark the vertex b by the label *end* in order to recognize it when we occur in it again.

Let r be the root of the component $K(b)$. We have a cycle P consisting of a simple [b, r]-*in*-path and an [r, b]-*out*-path, which, in turn, consists of a simple [r, a]-*out*-path and an *out*-arc (a, b). If $b = r$, then the *in*-path is of zero length; if $a = r$, then the *out*-path is of zero length. We will solve our problem traversing this cycle P. The *in*-path is traversed as follows: at each vertex v, we look through arcs in the v-loop until the first *in*-arc is found and move along it until we occur in a *root*-vertex. The *out*-path is traversed as follows: at each vertex v, we look through arcs in the v-loop until the first *out*-arc is found and move along it until we occur at a vertex with the label *end*.

First, we move from the vertex b to the root r along the *in*-path.

Then, we return from the root r to the vertex b along the *out*-path. In the course of this motion, in all *root*-vertices after r, we delete the label *root* and, starting from the first *root*-vertex we met after r, we make the *in*-arc originating from it coincide with the *out*-arc. As a result, all required components are glued with the component $K(b)$, and we will be able to reach the root r from all their vertices by the *in*-arcs. In addition, we set the variable *new_counter* > 1 if there are more than one *new*-vertex on the *out*-path. Note that a is a *new*-vertex.

Now, we need to find the vertex a. To do this, the robot will have to traverse the cycle P as many times as there are *new*-vertices on its *out*-path; in each passage (but the last one), the label *new* that was met first is removed.

We are situated at the vertex b and know whether the *new*-vertex that was met first on the *out*-path coincides with a. If it does not coincide, we move along the cycle P to the root r and, further, to the closest *new*-vertex, remove the label *new* from it, and move further setting

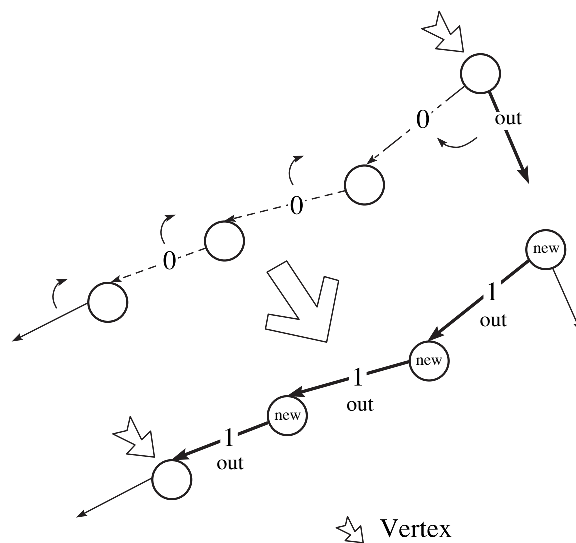

Fig. 12. Traversal of untraversed arcs.

```
void Out-Traverse () { /* traversing an out-arc */
   while ( !p.out ) Next ();
   Traverse ();
}

void In-Traverse () { /* traversing an in-arc */
   while ( !p.in ) Next ();
   Traverse ();
}

void Returning along a chord () {
   unsigned new_counter = 0;
   unsigned root_flag = 0;

   v.end = 1; /* vertex b */

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

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

     if ( v.new && new_counter < 2 ) new_counter += 1;
     /* counting the number of new-vertices */

     Out_Traverse ();

     /* gluing components */

     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;
        }
}

   /* search for a vertex a */

   while ( new_counter != 1) {
     new_counter = 0;
     while ( !v.root ) In-Traverse (); /* [b, r]-in-path */
     while ( !v.new ) Out-Traverse (); /* searching for the first new-vertex */
     v.new = 0; /* removing the first new-vertex */
     while ( !v.end ) { /* counting the number of the remaining new-vertices */
       if ( v.new && new_counter < 2 ) new_counter += 1;
       Out-Traverse ();
     }
   }
   v.end = 0; /* vertex b */

   /* moving to vertex a */

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

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

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

Fig. 13. Procedure “Returning along a chord.”

the variable *new_counter* > 1 if there remained more than one *new*-vertices on the *out*-path. This process is repeated until there remains only one *new*-vertex (*new_counter* = 1), which, obviously, coincides with the vertex *a*.

Now, we are situated at the vertex *b* and remove the label *new* from it. Then, we move along the cycle *P* to the *new*-vertex *a*, remove the label *new* from it, change the status of the *out*-arc (*a*, *b*) from 1 to 3, move along the *in*-arcs to the root *r*, and return from the procedure.

3.2.4. Reduction of the tree. The procedure is formally described in Fig. 15 and illustrated in Fig. 16.

The current vertex *a* is a leaf of the tree T_1 that turned to a *close*-vertex, and the current arc is an *out*-arc. Our goal is to remove the vertex *a* together with the incoming arc (a_1, *a*) from the tree T_1. Now, if the vertex a_1 is also a leaf *close*-vertex, we have to remove it together with the incoming arc (a_2, a_1), and so on.

A *close*-vertex with more than one outgoing 1-arcs is called a *crotch*. Hence, we have to remove from the tree T_1 a maximal finite segment [*c*, *a*] of the path [v_0, *a*] that does not contain *open*-vertices and crotches. If there remained *open*-vertices or crotches in T_1, then, clearly, the vertex *c* is to be the last *open*-vertex and/or crotch on the path [v_0, *a*]. We will seek such a vertex *c* on the path [*r*, *a*], where *r* is the root of the last component. If it turns out that *r* is not a crotch but is a *close*-vertex, then the robot stops. Note that, in a strongly connected graph, this may happen only if $r = v_0$, and, in this case, the robot, obviously, has traversed the graph.

The vertex *c* can be sought in the same way as the vertex *a* was sought in the procedure "Returning along a chord." In this case, the path [*r*, *a*] is to be traversed as many times as there are *open*-vertices and crotches on it. We, however, will use the "parity method" suggested by Afek and Gafni. Namely, we mark all *open*-vertices and crotches on the path [*r*, *a*] by the label *crotch* and, then, on each passage of the path, reduce the number of the *crotch* labels by two times, such that the desired vertex *c* always remains marked.

First of all, we check whether the vertex *a* is a *root*-vertex. If it is, we return from the procedure with the verdict "the last component has been completely traversed." Otherwise, we mark *a* by the label *end* in order to recognize it when we occur in it again.

We have a cycle *P* consisting of a simple [*a*, *r*]-*in*-path and a simple [*r*, *a*]-*out*-path, such that $a \neq r$. Moving along *P*, we mark the root *r* and all subsequent *open*-vertices and crotches on the [*r*, *a*]-*out*-path by the label *crotch*. In doing so, we store the parity of the number of the *crotch*-vertices in the variable *crotch_parity* and set the variable *crotch_counter* > 1 if the number of the *crotch*-vertices is greater than one. Note that the parity of the desired vertex *c* is equal to *crotch_parity*. If *crotch_counter* > 1, then we pass along the path *P* again, remove the label *crotch* from all *crotch*-vertices of the parity |*crotch_parity* – 1|, store again the parity of the number of the remaining *crotch*-vertices in the variable *crotch_parity*, and set anew the variable *crotch_counter* > 1 if the number of the remaining *crotch*-vertices is greater than one. Note that the vertex *c* remained a *crotch*-vertex, and its parity is again equal to *crotch_parity*. This process continues until there remains only one *crotch*-vertex (*crotch_counter* = 1), which, obviously, coincides with *c*.

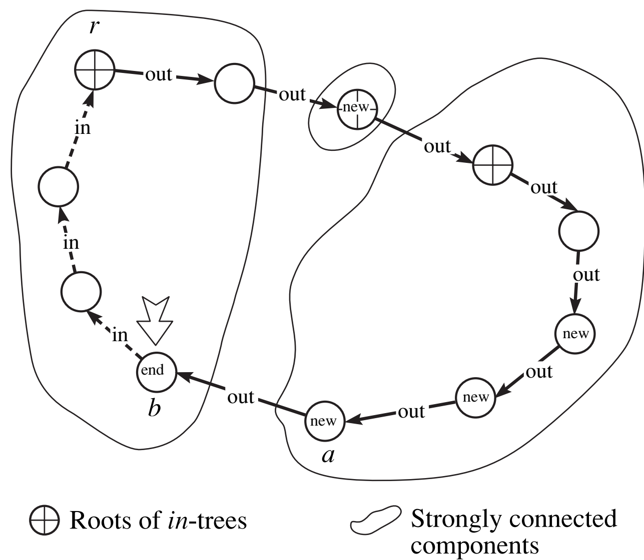

Fig. 14. Returning along a chord.

Further, we move along the cycle *P* to the *crotch*-vertex *c* and remove the label *crotch* from it. Then, we continue moving from *c* to *a*, change the status of all arcs belonging to the [*c*, *a*]-*out*-path from 1 to 2, remove the label *out* from all these arcs (except for the first arc originating from *c*), remove the label *end* from the vertex *a*, and move to the root *r* along the *in*-arcs.

Now, it is required to check whether the root *r* is an *open*-vertex or has at least one outgoing 1-arc. If this is true, we return from the procedure with the verdict "the tree has successfully been reduced." Otherwise, we return from the procedure with the verdict "the last component has successfully been traversed."

4. THEOREM ABOUT THE ROBOT R_3

Theorem. The robot R_3 stops on any finite directed graph and traverses all finite strongly connected graphs. The length of the covering path is $O(nm + n^2\log\log n)$, and the required memory is $O(n + m)$. For any *n* and *m*, there exists a graph with *n* vertices and $m' \geq m$ arcs, for which the length of the covering path constructed by the robot R_3 is equal to $\Omega(nm' + n^2\log\log n)$.

Proof. Denote the graph processed by the robot by *G*, and let *VG* be a set of its vertices.

The robot stops on any finite graph. The fulfillment of the step conditions before the initial step, as well as at the end of each step if these conditions are fulfilled in the beginning of the step, is checked immediately by

```
void Reduction of the tree () {
  unsigned crotch_counter = 0;
  unsigned crotch_parity = 0;

  v.end = 1; /* vertex a */

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

  /* placing labels crotch on [r, a]-out-path */

  while ( !v.end ) {
    if ( !v.root || !v.close ) v.crotch = 1; /* root r or an open-vertex */
    else {/* checking a crotch */
       while ( p.status != 1 ) Next ();
       if ( p.out ) do Next (); while ( p.status != 1);
       if ( !p.out ) v.crotch = 1; /* crotch */
    }
    if ( v.crotch ) {
      crotch_parity = ( crotch_parity == 1 ) ? 0: 1;
      if ( crotch_counter < 2 ) crotch_counter += 1; /* counting the number of crotch-vertices */
    }
    Out-Traverse ();
  }

  /* parity method */

  while ( croth _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-path */
    while ( !v.end ) { /* [r, a]-out-path */
      if ( v.crotch ) {
          v_crotch_parity = (V_crotch_parity == 1 ) ? 0: 1;
          if ( v_crotch_parity != crotch_parity )
               v.crotch = 0; /* removing label crotch */
          else { /* remaining crotch-vertex */
             new_crotch_parity = (new_crotch_parity == 1 ) ? 0: 1;
             if ( croth _counter < 2 )
                 croth _counter += 1; /* counting the number of the remaining crotch-vertices */
          }
      }
      Out-Traverse ();
    }
    crotch_parity = new_crotch_parity;
  }
  /* moving to vertex c */
  while ( !v.root ) In-Traverse (); /* [a, r]-in-path */
  while ( !v.crotch ) Out-Traverse (); [r, c]-out-path */

  v.crotch = 0; /* vertex c */

  /* remove [c, a]-path */

  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; /* vertex a */

  /* moving to root r */

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

  /* checking whether the root r is an open-vertex or has at least one outgoing 1-arc */

  if ( v.close ) {
      while ( !p.out ) Next ();
      do Next (); while ( p.status != 1 && !p.out );
      if (p.out) return ("the last component is completely traversed");
  }
  return ("the tree has successfully been reduced");
}
```

Fig. 15. Procedure “Reduction of the tree.”

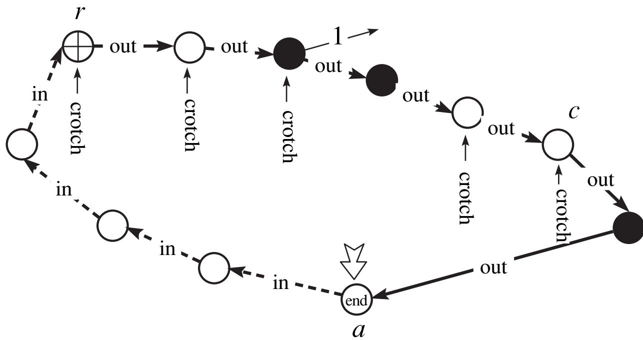

Fig. 16. Reduction of the tree.

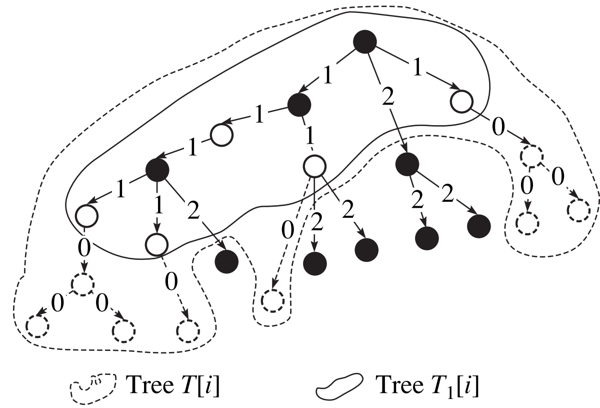

Fig. 17. Trees $T_1(i)$, $T_{12}(i)$, $T_{12}[I]$, and $T[i]$.

using the description of robot's algorithm. It is easy to see that, at each step, all four procedures are executed in a finite time. Therefore, in order to prove that the robot stops in a finite time, it is sufficient to show that it performs a finite number of steps. Indeed, the first step begins with the second procedure "Traversal of untraversed arcs," where, at least, one untraversed arc is traversed if the initial vertex v_0 has outgoing arcs at all. During any subsequent step, in the first procedure "Search for an untraversed arc," either one *open*-vertex becomes a *close*-vertex or we turn to the second procedure, where at least one untraversed arc is traversed. Since the number of vertices and arcs in the graph is finite, the number of steps is finite as well, and the robot stops in a finite time.

The robot traverses all finite strongly connected graphs. The termination condition is as follows: the root of the last component of the traversed graph G_t becomes a *close*-vertex and, hence, is completely traversed and has no outgoing 1-arcs. Hence, it follows that there are no *open*-vertices in the last component and, thus, there are no untraversed arcs beginning in vertices of the last component; i.e., the last component is completely traversed. In a strongly connected graph, this may happen only in the case where the traversed graph consists of one component and coincides with the entire graph G.

The amount of the required memory is $O(n + m)$. Since the cells of the vertices and arcs may contain a finite number of labels from a finite alphabet, the total amount of memory is $O(n + m)$.

The upper bound of the covering path length is $O(nm + n^2 \log\log n)$. In the procedure "*Search for an untraversed arc*," we follow a path in the tree T_1 and, therefore, pass along a simple path of length not greater than $n - 1$. In so doing, either one *open*-vertex becomes a *close*-vertex or we turn to the procedure "Traversal of untraversed arcs," where we traverse at least one untraversed arc. Thus, the total number of arcs traversed in the first procedure at all steps is not greater than $(n - 1)(n + m) = O(nm)$.

In the procedure "*Traversal of untraversed arcs*," we move along untraversed arcs and, thus, the total number of arcs traversed in this procedure at all steps is not greater than m.

In procedure 3, "*Returning along a chord*," we follow several times the cycle P consisting of two simple paths and one arc, whose length is not greater than $n - 1 + n - 1 + 1 = 2n - 1$, and, then, traverse once a simple *in*-path of length not greater than $n - 1$. The number of passages of the cycle P during the ith step is $k_i + 1$, where k_i is the number of *new*-vertices in the cycle P as the beginnings of arcs traversed first time in the procedure 2 "*Traversal of untraversed arcs*" (at this and previous steps). Therefore, at the ith step, in procedure 3, we traverse not more than $k_i(2n - 1) + n - 1$ arcs. Since an arc changes its status (from an untraversed arc to a traversed one) only once, the sum of k_i over all steps can be estimated as $\sum\{k_i | i = 1\ldots\} \le m$. Thus, the total number of arcs traversed in procedure 3 over all steps is not greater than $m(2n - 1) + n - 1 = O(nm)$.

The fourth procedure "*Reduction of the tree*" is of most interest for us. We will show that the total number of arcs traversed by the robot in this procedure over all steps is not greater than $O(n^2 + n^2 \log\log n)$. Thus, since $n - 1 \le m$, the total upper estimate is $O(nm + n^2 \log\log n)$.

We will consider only those steps of the robot at which it performs the fourth procedure and use the index $i = 1, 2, \ldots, I$ to number the steps. At the ith step in procedure 4, we several times traverse the cycle P_i consisting of two simple paths whose length, thus, is not greater than $n - 1 + n - 1 = 2n - 2$. Then, we traverse once a simple *in*-path of length not greater than $n - 1$. The cycle P_i is traversed at least twice: (1) in the very beginning to place labels *crotch* and (2) in the very end to remove labels *out* from all arcs of the $[c, a]$-path, except for the first arc with the beginning at c, and to remove label *end* from the vertex a.

The number of the remaining passages of the cycle P_i, which constitute the basic part of the *backtracking*, is denoted by b_i. Thus, the number of the traversed arcs

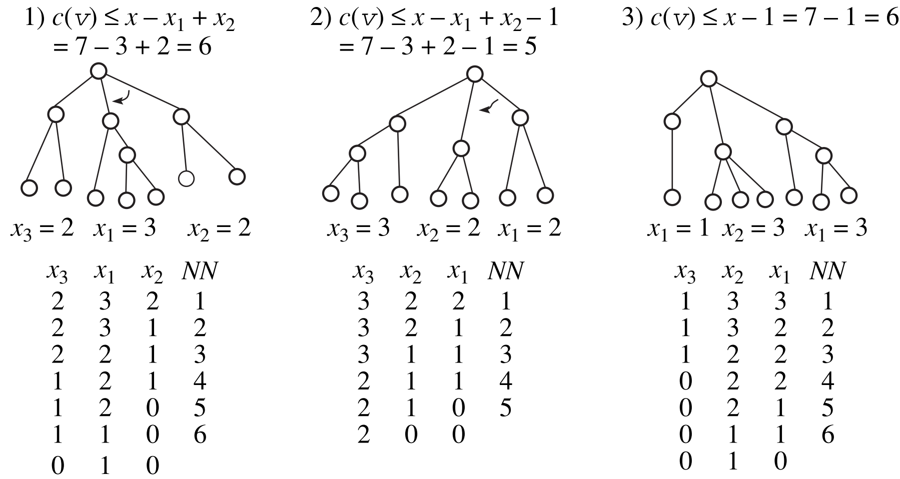

Fig. 18. Estimate for $c(v)$.

in procedure 4 at the ith step does not exceed $(2n - 2)(2 + b_i) + n - 1 = (2n - 2)b_i + 5n - 5$.

Let us denote by c_i the number of *open*-vertices and crotches in the *out*-part of the cycle P_i. After the first passage of the cycle P_i, all these vertices and the root r receive the label *crotch*; hence, the number of such vertices does not exceed $c_i + 1$. Since each of these b_i passages reduces the number of the *crotch*-vertices by two times, we have $b_i \leq \log_2(c_i + 1)$. Thus, at the ith step in procedure 4, the robot traverses not more than $(2n - 2)\log_2(c_i + 1) + 5n - 5$ arcs. Since the inequality $b_i > 0$ implies that $c_i \geq 1$, we have $(2n - 2)\log_2(c_i + 1) + 5n - 5 \leq 2n\log_2 2c_i + 5n - 5 = 2n\log_2 c_i + 7n - 5$. Now, it suffices to show that the sum of $\log_2 c_i$ over all steps satisfies the inequality $\sum \log_2 c_i \leq O(n\log\log n)$.

Note that the tree is reduced only if $n \geq 2$, and, in the case of $n = 2$, we have $\sum \log_2 c_i = 0 = n\log_2\log_2 n$. If $n \geq 3$, we take advantage of the Cauchy inequality $(\prod c_i)^{1/n} \leq n^{-1}\sum c_i$, from which it follows that $\sum \log_2 c_i \leq n\log_2(n^{-1}\sum c_i)$. If we prove that $\sum c_i \leq Cn\log_2 n$ for some constant C, we will have $\sum \log_2 c_i \leq n\log_2(n^{-1}Cn\log_2 n) \leq C'n\log_2\log_2 n$, where $C' = 1 + \log_2 C/\log_2\log_2 3$. Thus, it is sufficient to show that $\sum c_i = O(n\log n)$. Let $c(v)$ denote the number of steps at which the vertex v received the label *crotch*. Clearly, $\sum c_i = \sum\{c(v) | v \in VG\}$, where VG is the set of vertices of the graph G. For a graph G, we introduce the notation $C(G) = \sum\{c(v) | v \in VG\}$.

The status of an arc can be changed only as follows: $0 \longrightarrow 3$ (a chord) or $0 \longrightarrow 1 \longrightarrow 2$ (an arc of the tree T_{12}). Therefore, the tree T_{12} may only grow: $T_{12}(i) \subseteq T_{12}(i + 1)$. It is maximal ($T_{12}$) when the robot stops $T_{12}(I)$. The tree T_1 may both grow (in the procedure 2 "Traversal of untraversed arcs") and reduce (in the procedure 4 "Reduction of the tree"). Let us consider the graph $T(i) = (T_{12}[I] \backslash T_{12}(i)) \cap T_1(i)$ at the beginning of the ith step; it is composed of 1-arcs and the 0-arcs that are going to turn to 1- or 2-arcs, i.e., of all arcs of the tree $T_{12}[I]$ but those that have already become 2-arcs by the beginning of the ith step. It is evident that, from a vertex of the tree $T_{12}(i)$ that does not belong to the tree $T_1(i)$, i.e., from the end of a 2-arc, there may originate only 2-arcs of the tree $T_{12}(i)$. Therefore, $T(i)$ is also a tree with the root v_0 (Fig. 17). At each step (recall that we consider only the steps that include procedure 4), the tree $T(i)$ looses exactly one leaf vertex, since all arcs belonging to the *out*-path leading to it change their status from 1 to 2. The tree $T(i)$ is maximal at the very beginning, $T = T(1)$.

For a given vertex v, we consider a subtree $T(v, i) \subseteq T(i)$ generated by its root v. Clearly, if v is not a crotch in the tree T, then $c(v) = 0$. Now, we consider the case where v is a crotch in the tree T. At each step when the vertex v receives the label *crotch* in procedure 4, it is an *open*-vertex or a crotch of the tree T_1 and, hence, a crotch of the tree $T(v, i)$, and the latter tree looses exactly one leaf vertex at each step. We are interested to know in how many steps the vertex v stops being a crotch of the tree $T(v, i)$, since this number is, obviously, just $c(v)$.

Let us number the arcs of the tree T originating from v in the order they are arranged in the v-loop by the index $j = 1, 2, \ldots$. Let v_j denote the end of the jth arc and $x_j(i)$ denote the number of leaf vertices in the subtree $T(v_j, i) \subseteq T(i)$ of the ith step generated by the root v_j. (If no arcs of the tree $T(i)$ originate from v_j, then we assume that $x_j(i) = 1$ (one leaf vertex coinciding with the root v_j); if the vertex v_j itself does not belong to $T(i)$, then we assume that $x_j(i) = 0$.) Clearly, the num-

bers $x_j(i)$ do not increase with the growth of i and achieve their maximal values $x_1(1)$, $x_2(1)$, … before the first step. At each step when the vertex v receives the label *crotch* in procedure 4, one of the nonzero numbers $x_1(i)$, $x_2(i)$, … reduces by 1. Let us number their initial values $x_1(1)$, $x_2(1)$, … in nondecreasing order of their magnitudes, using a subscript for the numbering: $x_1 \geq x_2 \geq x_3 \geq \ldots$. Let us introduce the notation for their sum $x = x_1 + x_2 + x_3 + \ldots$. The vertex v stops being a crotch when there remains only one nonzero number $x_j(i)$. Since the numbers $x_1(i)$, $x_2(i)$, … reduce by 1 in turn, in the order determined by the v-loop, the following three cases where we introduce an upper estimate $s(v)$ for $c(v)$ are possible (Fig. 18):

(1) $x_1 > x_2$ and, in the v-loop, the number $x_1 = x_{j1}(1)$ corresponds to an earlier arc than at least one arc with the number $x_2 = x_{j2}(1)$; i.e., $j_1 < j_2$. Then, $c(v) \leq x_2 + x_2 + x_3 + \ldots = x - x_1 + x_2$.

(2) $x_1 > x_2$ and, in the v-loop, the number $x_1 = x_{j1}(1)$ corresponds to a later arc than any arc with the number $x_2 = x_{j2}(1)$; i.e., $j_1 > j_2$. Then, $c(v) \leq (x_2 - 1) + x_2 + x_3 + \ldots = x - x_1 + x_2 - 1$.

(3) $x_1 = x_2$. Then, $c(v) \leq x_1 + x_2 + x_3 + \ldots - 1 = x - 1$.

The equality "$c(v) = \ldots$" takes place when none of the leaves of the tree $T(v, i)$ can be removed "without participation" of the vertex v; i.e., until the vertex v itself "is reduced" from the tree T_1, it is traversed in procedure 4 by an *out*-path for reducing the tree. In other words, in the course of the robot operation, no roots of *in*-trees are formed in the tree T above v. Otherwise, the strict inequality "$c(v) < \ldots$" takes place.

Defining $s(v) = x - x_1 + x_2$, we obtain the upper estimate $c(v) \leq s(v)$.

For vertices that are not crotches of the tree T and, in particular, for leaf vertices, $s(v)$ is defined as $s(v) = 0$; for the other vertices, it is defined by the recurrent formula $s(v) = x - x_1 + x_2$, which extends the computations along the tree T from the leaves to the root. Note that, although $c(v)$ depends on the order of arcs in the v-loop, $s(v)$ does not.

Introducing the notation $S = S(G) = \sum\{s(v) | v \in VG\}$, we have $C(G) \leq S(G)$. The spanning tree T of the graph G with given v-loops of arcs for each vertex v is denoted as T^v. It is evident that $C(G)$ actually depends on T^v and $S(G)$ depends on T, so that we use the notation $C(T^v) = C(G)$ and $S(T) = S(G)$. Let X be the number of leaf vertices of the tree T. We will show that $S(T) \leq X\log_2 X$.

First of all, we note that a vertex a that is not a crotch in the tree T can be not taken into account: if the arcs (b, a) and (a, c) merge together in one arc (b, c) with the deletion of the intermediate vertex a (if $a = v_0$ is the root of T, we simply delete the arc (a, c) such that c becomes the root), then neither $c(v)$ nor $s(v)$ is changed whatever the crotch v. Let us repeat this procedure until all nonleaf vertices of T become crotches. In what follows, we consider only such, *homeomorphically irreducible*, trees.

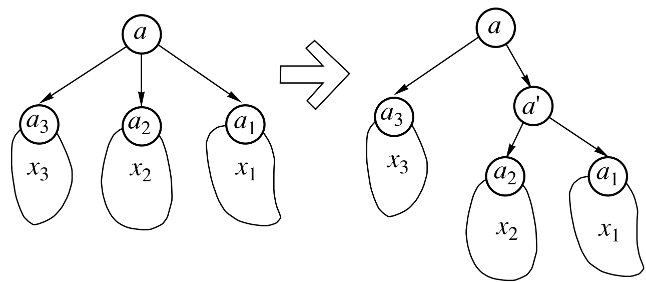

Fig. 19. Transformation of a nonbinary tree to a binary one.

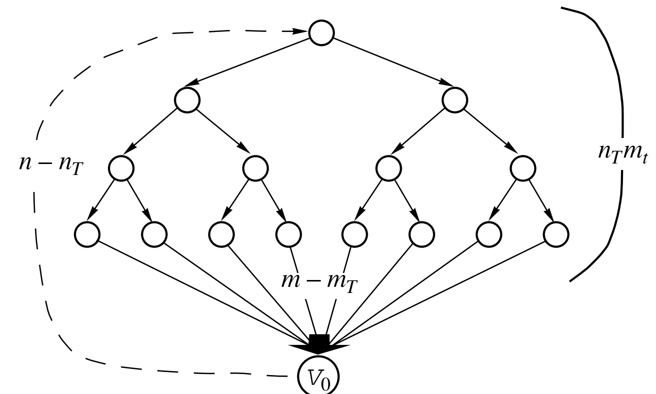

Fig. 20. An example of a graph for which estimate $\Omega(nm' + n^2\log\log n)$ of the covering path length is achieved.

A tree vertex is called *binary* if it has exactly two outgoing arcs. A tree is said to be *binary* if all of its nonleaf vertices are binary.

Lemma 2. For any nonbinary tree T with X leaf vertices, there exists a binary tree T^* with the same number of leaf vertices such that $S(T) \leq S(T^*)$.

Proof. Since all nonleaf vertices in the tree T are crotches, the total number of arcs is not less than twice the number of the nonleaf vertices, i.e., $m \geq 2(n - X)$. Denote by $L = m - 2(n - X)$ the number of "redundant" arcs. We introduce an elementary transformation of the tree T that conserves the number X, reduces L, and does not reduce the sum $S(T)$. Let T contain a nonbinary nonleaf vertex a with at least three outgoing arcs and, hence, $x_1 \geq x_2 \geq x_3 \geq 1$. Let the ends of the arcs originating from a be denoted as $a_1, a_2, a_3, \ldots$, where a_i is a vertex whose subtree $T(a_i)$ has x_i leaves. Let us add a vertex a' and replace the arcs (a, a_1) and (a, a_2) by the three arcs (a, a'), (a', a_1), and (a', a_2) (Fig. 19). The tree obtained is denoted by T'.

Clearly, the number $s(v)$ changed only at the vertex a and was added to the new vertex a' (the modified values are equipped with the prime, $s'(v)$).

(1) $S(T) = \sum\{ s(v) | v \in VG \& v \neq a\} + s(a)$;

(2) $S(T') = \sum\{ s(v) | v \in VG \& v \neq a\} + s'(a) + s'(a')$;

(3) $s(a) = (x_1 + x_2 + x_3 + \ldots) - x_1 + x_2 = x - x_1 + x_2$, where $x = x_1 + x_2 + x_3 + \ldots$;

(4) $s'(a) = ((x_1 + x_2) + x_3 + \ldots) - (x_1 + x_2) + x_3 = x - x_1 - x_2 + x_3$, since the trees $T(a_1)$ and $T(a_2)$ merged into one tree $T(a')$ with $x_1 + x_2$ leaves and $x_1 + x_2 > x_2 \geq x_3$;

(5) $s'(a') = x_1 + x_2 - x_1 + x_2 = 2x_2$;

(6) $S(T') - S(T) = s'(a) + s'(a') - s(a) = (x - x_1 - x_2 + x_3) + 2x_2 - (x - x_1 + x_2) = x_3 \geq 1$.

Repeating this procedure as many times as possible, we obtain the desired binary tree T^*. □

Lemma 3. For $y \geq x \geq 1$, the inequality $y\log_2 y + x\log_2 x + 2x \leq (x + y)\log_2(x + y)$ holds.

Proof. Consider the difference $f(x, y) = (x + y)\log_2(x + y) - y\log_2 y - x\log_2 x - 2x$. We need to show that $f(x, y) \geq 0$. Let us take the second derivative with respect to x:

$$f'_x(x, y) = \log_2(x + y) + \log_2 e - \log_2 x - \log_2 e - 2,$$

$$f''_x(x, y) = (\log_2 e)/(x + y) - (\log_2 e)/x = (\log_2 e)(-y)/(x + y) < 0$$

for $y \geq x \geq 1$. Therefore, it suffices to check whether the inequality $f(x, y) \geq 0$ holds at the boundaries (1) $x = 1$ and (2) $x = y$.

(1) $f(1, y) = (1 + y)\log_2(1 + y) - 1\log_2 1 - y\log_2 y - 2$. Let us take the first derivative with respect to y:

$$f'_1(y) = \log_2(1 + y) + \log_2 e - \log_2 y - \log_2 e = \log_2(1 + 1/y) > 0$$

for $y \geq 1$. As can be seen, $f(1, y)$ does not decrease, so that it is sufficient to check whether the inequality $f(1, y) \geq 0$ holds for the minimal value $y = 1$. We have

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

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

Lemma 4. For binary trees with X leaf vertices, the estimate $S(T) \leq X\log_2 X$ holds.

Proof. We prove this lemma by induction on the number of leaves X. For $X = 1$, we have $S(T) = 0 = 1\log_2 1$. Let us assume that the assertion of the lemma is true for all binary trees with the number of leaves less than X, where $X > 1$. Consider a binary tree T with X leaves. Two arcs originating from its root v_0 lead to vertices v_1 and v_2; the subtrees $T(v_1)$ and $T(v_2)$ have X_1 and X_2 leaves, respectively; and $X = X_1 + X_2$. Let, for definiteness, $X_1 \geq X_2$. Then,

(1) $S(T) = S(T(v_1)) + S(T(v_2)) + s(v_0)$;

(2) by the induction hypothesis, $S(T(v_1)) \leq X_1\log_2 X_1$ and $S(T(v_2)) \leq X_2\log_2 X_2$;

(3) $s(v_0) = (X_1 + X_2) - X_1 + X_2 = 2X_2$;

(4) by Lemma 3, $S(T) = S(T(v_1)) + S(T(v_2)) + s(v_0) \leq X_1\log_2 X_1 + X_2\log_2 X_2 + 2X_2 \leq (X_1 + X_2)\log_2(X_1 + X_2) = X\log_2 X$. □

From Lemmas 2 and 4, it follows that, for any tree T with X leaf vertices, the estimate $S(T) \leq X\log_2 X$ holds.

The lower bound for the covering path length is $\Omega(nm' + n^2\log\log n)$. First, we show that $X\log_2 X$ is an exact (in terms of the order) estimate for $C(T^v)$.

Lemma 5. For any constant $0 < A < 1$, there exists an infinite sequence of trees $T_1, T_2, \ldots$ with an infinitely growing number of leaves $X_1, X_2, \ldots$, where X_i is the number of leaves in the tree T_i, such that $C(T_i^v) \geq X_i\log_2 X_i$.

Proof. Consider a graph the spanning tree of which is a binary justified tree T such that the distance from the root to each leaf is h. In order to make the graph strongly connected, it is sufficient to draw a chord from each leaf vertex to the root of the tree. The level h contains $X = 2^h$ leaves, and the level $i = 0, 1, \ldots, h - 1$ contains 2^i nonleaf vertices, such that each vertex has two outgoing arcs leading to vertices belonging to level $i + 1$. Since, for each nonleaf vertex v, the subtrees T_1 and T_2 with the roots at the ends of two arcs originating from v have equal numbers of leaves, $X_1 = X_2$, we have $c(v) = (X_1 + X_2) - 1$. If v is located at a level $i < h$, then $X_1 = X_2 = 2^{h-i-1}$. Therefore,

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

For $0 < A < 1$ and any $h > 1/(1 - A)$, we have

(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) $(1 - A)X\log_2 X > X - 1$;

(4) $C(T^v) = X\log_2 X - X + 1 > AX\log_2 X$. □

Let us select the number of leaves X of the tree T from Lemma 5 such that the number of vertices n_T in T would approximate the number $n/2 - 1$ from below as close as possible: $n_T = 2^{h+1} - 1$, $\log_2 n - 3 < h \leq \log_2 n - 2$. For each leaf vertex of the tree T, we add a chord leading to the initial vertex v_0; the number of the chords is equal to 2^h. For the remaining $n - n_T$ vertices, we add a simple path of length $n - n_T$ leading from the initial vertex to the root of the tree. For the graph obtained, the estimate for the total backtracking is $\Omega(n^2\log\log n)$. The number of arcs in this graph is $m_T = (n_T - 1) + 2^h + (n - n_T)$. Let us define $m' = \max\{m, m_T\}$. If $m_T \geq m$, the covering path length is $\Omega(nm' + n^2\log\log n)$. If $m_T < m$, it is sufficient to add $m - m_T$ arcs leading from the leaf vertices of the tree T to the initial vertex (Fig. 20).

This completes the proof of the theorem on the robot R_3. □

5. CONCLUSIONS

Unfortunately, an exact estimate of the length of the traversal of a finite directed strongly connected graph by a finite robot (minimum of the upper bounds of the algorithms over all possible traversal algorithms) is not known. Moreover, although it seems unlikely that a finite robot could traverse a graph for $\Omega(nm)$, this fact has not been proved yet.

REFERENCES

1. Afek, Y. and Gafni, E., Distributed Algorithms for Unidirectional Networks, *SIAM J. Comput.*, 1994, vol. 23, no. 6, pp. 1152–1178.
2. Hoffman, D. and Strooper, P., ClassBench: A Framework for Automated Class Testing, *Software Maintenance: Practice Experience*, 1997, vol. 27, no. 5, pp. 573–579.
3. Murray, L., Carrington, D., MacColl, I., McDonald, J., and Strooper, P., Formal Derivation of Finite State Machines for Class Testing, *Lecture Notes Comput. Sci.* (Proc. of the 11th Int. Conf. of Z Users), Berlin: Springer, 1998, vol. 1493, pp. 42–59.
4. Albers, S. and Henzinger, M.R., Exploring Unknown Environments, *SIAM J. Comput.*, 2000, vol. 29, no. 4, pp. 1164–1188.
5. Deng, X. and Papadimitriou, C.H., Exploring an Unknown Graph, *J. Graph Theory*, 1999, vol. 32, no. 3, pp. 265–297.
6. Ore, O., *Theory of Graphs*, Providence: AMS, 1962. Translated under the title *Teoriya grafov*, Moscow: Nauka, 1968.
7. Bhatt, S., Even, S., Greenberg, D., and Tayar, R., Traversing Directed Eulerian Mazes, *Proc. of WG'2000*, Brandes, U. and Wagner, D., Eds., *Lecture Notes in Computer Science,* vol. 1928, pp. 35–46, Berlin: Springer, 2000.
8. Blum, M. and Sakoda, W.J., On the Capability of Finite Automata in 2 and 3 Dimensional Space, *Proc. of the Eighteenth Annu. Symp. on Foundations of Comput. Sci.*, 1977, pp. 147–161.
9. Even, S., *Graph Algorithms*, Comput. Sci., 1979.
10. Even, S., Litman, A., and Winkler, P., Computing with Snakes in Directed Networks of Automata, *J. Algorithms*, 1997, vol. 24, pp. 158–170.
11. Rabin, M.O., Maze Threading Automata. Lecture Presented at MIT and UC Berkley, 1967.
12. Bourdonov, I.B., Study of the Automaton Behavior on Graphs, *MS Dissertation*, Moscow: Moscow State University, 1971.
13. Kobayashi, K., The Firing Squad Synchronization Problem for a Class of Polyautomata Networks, *J. Comput. System Sci.*, 1978, vol. 17, pp. 300–318.
14. Kutten, S., Stepwise Construction of an Efficient Distributed Traversing Algorithm for General Strongly Connected Directed Networks, *Proc. of the Ninth Int. Conf. on Comput. Commun.*, 1988, pp. 446–452.
15. http://www.ispras.ru/RedVerst/.
16. Bourdonov, I., Kossatchev, A., Petrenko, A., and Gatter, D., KVEST: Automated Generation of Test Suites from Formal Specifications, *Proc. of FM'99*, Lecture Notes in Computer Science, vol. 1708, pp. 608–621, Berlin: Springer, 1999.
17. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Use of Finite Automata for Program Testing, *Programmirovanie*, 2000, no. 2, pp. 12–28.
18. Bourdonov, I., Kossatchev, A., Kuliamin, V., and Petrenko, A., UniTesK Test Suite Architecture, *Proc. of FME 2002*, Lecture Notes in Computer Science, Berlin: Springer, 2002, vol. 2391, pp. 77–88.
19. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case, *Programmirovanie*, 2003, no. 5, pp. 11–30.
20. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case, *Programmirovanie*, 2004, no. 1, pp. 4–24.