Programming and Computer Software, Vol. 30, No. 6, 2004, pp. 305–322. Translated from Programmirovanie, Vol. 30, No. 6, 2004.

Backtracking Problem in the 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. 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. 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 estimate $O(nm)$ are known, where n is the number of vertices and m is the number of arcs. If the number of states is finite, 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. The selection of the arc that has not been traversed yet among those originating from the current vertex is determined by the order of the outgoing arcs, which is a priori specified for each vertex. The best known traversal algorithms for a finite robot are based on constructing the output directed spanning tree of the graph with the root at the initial vertex and traversing it with the aim to find all untraversed arcs. In doing so, we face the *backtracking* problem, which consists in searching for all vertices of the tree in the order inverse to their natural partial ordering, i.e., from the leaves to the root. Therefore, the upper estimate of the algorithms is different from the optimal estimate $O(nm)$ by the number of steps required for the backtracking along the outgoing tree. The best known estimate $O(nm + n^2\log\log n)$ has been suggested by the author in the previous paper [1]. In this paper, a finite robot is suggested that performs a backtracking with the estimate $O(n^2\log^*(n))$. The function log* is defined as an integer solution of the inequality $1 \le \log_2^{\log^*}(n) < 2$, where $\log^t = \log \circ \log \circ ... \circ \log$ (the superposition $\circ$ is applied $t-1$ times) is the tth compositional degree of the logarithm. The estimate $O(nm + n^2\log^*(n))$ for the covering path length is valid for any strongly connected graph for a certain (unfortunately, not arbitrary) order of the outgoing arcs. Interestingly, such an order of the arcs can be marked by symbols of the finite robot traversing the graph. Hence, there exists a robot that traverses the graph twice: first traversal with the estimate $O(nm + n^2\log\log n)$ and the second traversal with the estimate $O(nm + n^2\log^*(n))$.

1. INTRODUCTION

This paper is devoted to the same problem as that discussed in the previous paper [1] of the author, i.e., to the problem of traversing an unknown graph by a finite robot. This problem was first formulated by Rabin [2] in 1967. A covering path is a path beginning at a given starting vertex and passing through all edges of the graph. In the case of a directed graph, each directed edge (arc) can be traversed only in the direction of its orientation. The graph is assumed to be a priori unknown, and its topology is learnt only in the course of traversing the graph. 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. The finite robot is an analogue of the Turing machine, where the tape is replaced by a graph: the cell storing a symbol of the external robot alphabet corresponds to a vertex or arc of the graph, and the robot moves along an arc in the direction of its orientation. The robot solves the traversal problem if it stops in a finite number of steps on any strongly connected graph with any initial vertex, and the path traversed by the robot is a covering path.

The robot must indicate somehow which arc originating from the current vertex has been selected. If the arcs originating from a vertex v are numbered from 1 through $d_{\text{out}}(v)$, where $d_{\text{out}}(v)$ is the outdegree of the vertex v, then the robot may simply indicate the arc number. However, to ensure the finiteness of the robot in this case, the outdegree d_{out} is to be bounded from above. This restriction can easily be removed if we add a memory cell for each arc originating from the vertex v and combine these cells into a loop, which is further referred to as the v-loop (Fig. 1). The graph with specified v-loops of the outgoing arcs for all vertices v is called an *ordered* graph. The robot is supplemented by the *inner* transition (denoted by the letter i) to the cell of the next arc in the v-loop. In the case of the *outer*

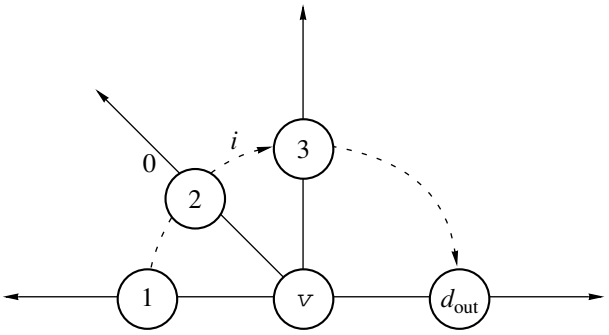

Fig. 1. Vertex v and v-loop of the outgoing arcs.

transition (denoted by the letter o) along the arc (v, v'), the robot occurs in the cell of the first arc in the v'-loop. Thus, the robot indicates what—outer (o) or inner (i)—transition it implements and does not need to identify the outgoing arc (v, v') along which it wants to move, since this is always the current arc in the cell of which the robot occurs.

We assume that two cells are simultaneously accessible to robot for reading and writing: the cell of the current 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 marks up the cell of the first arc of the v-loop, reads the information about the vertex from this cell, and stores it in its state. To update the information about the vertex, the robot moves along the v-loop to the marked cell of the first arc and writes to it.

The length of the minimal covering path for a strongly connected graph has the estimate $\Theta(nm)$, where n is the number of vertices and m is the number of arcs of the graph. The best known algorithms for traversing graphs by a finite robot are based on constructing two spanning trees for the graph: the output and input spanning trees with the root located at the initial vertex of the graph. The output tree is used by the robot for moving from the initial vertex to any other vertex and, thus, to the beginning of any graph arc that has not been traversed yet. The input tree is used for returning to the initial vertex after traversing a chord of the output tree.

After all arcs originating from a leaf vertex of the output tree have been traversed, this vertex and the tree arc leading to it are removed (marked by special terminal labels). Thus, the output tree always terminates only in the leaf vertices with untraversed outgoing arcs. The vertices are terminated in the order reverse to their natural partial order in the output tree: from the leaves to the root. This process is referred to as a *backtracking* along the tree. The robot stops when the tree root (the initial vertex of the graph) is terminated.

Depending on the way the output tree is traversed, we distinguish between the depth-first search (DFS) and breadth-first search (BFS) algorithms. In both variants, the upper estimate differs from the optimal estimate $O(nm)$ by the quantity equal to the number of steps required for the complete backtracking along the outgoing tree. In 1971, the author of this paper suggested a BFS algorithm with the backtracking that requires $O(n^2\log n)$ passages [3]. In 1993, Afek and Gafni [4] suggested a DFS algorithm with the same estimate of the backtracking length. In the previous paper [1], the author of this paper has obtained the improved estimate $O(n^2\log\log n)$.

This paper is devoted to the problem of backtracking along a tree. For returning to the tree root (initial vertex), a chord leading to the root is added to each leaf vertex. We suggest a finite robot that performs a DFS backtracking with the estimate $O(n^2\log^*(n))$. The function log* defined as the number of the logarithm operations is an integer solution of the inequality $1 \le \log_2^{\log^*}(n) < 2$, where $\log^t = \log \circ \log \circ \ldots \circ \log$ (the superposition is applied $t-1$ times) is the tth compositional degree of the logarithm.

Unfortunately, this result does not mean that it is possible to construct a robot for traversing any strongly connected graph with the estimate $O(nm + n^2\log^*(n))$. The point is that, before the traversal is completed, the robot may occur in a vertex for which there does not exist a path leading from this vertex to the initial vertex and consisting of only the traversed arcs. In this case, in the traversed part of the graph, instead of the input tree, a forest of input trees is constructed, and the robot returns not to the root of the input tree (initial vertex) but to the root of the last of such input trees. The backtracking problem in such a situation becomes more complicated, and the estimate for its solution may exceed $O(n^2\log^*(n))$. What output tree and what forest of input trees are separated by the robot in a given graph depends on the order in the graph, i.e., on the order of arcs in the v-loops for all vertices v. This order is introduced *ab extra* and, essentially, determines the selection of a current untraversed arc by the robot from the set of the untraversed arcs originating from the current vertex. In conclusion, it is shown that, for any strongly connected (unordered) graph, there exists an ordering in the loops of the outgoing arcs such that the forest of input trees in the course of the robot operation always consists of only one tree with the root at the initial vertex. Such an ordering of the arcs may be marked by symbols of the finite robot traversing the graph. Hence, it is possible to construct a robot that traverses the graph twice, first time, with the estimate $O(nm + n^2\log\log n)$ and, second time, with the estimate $O(nm + n^2\log^*(n))$.

2. DEFINITION OF ROBOT ON A GRAPH

We define the graph and the robot on the graph in a formal way. We will use the algebraic notation for functions, i.e., write xf instead of $f(x)$.

A *directed graph* (in 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

$$\forall v \in V \; v\gamma\alpha = v;$$

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

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

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

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

A graph is said to be finite if the sets V and E are finite. A vertex v and an arc e are *incidental* if $v = e\alpha$ or $v = e\beta$. Two arcs e and e' are said to be *adjacent* if $e\beta = \alpha e'$. A *path* is a sequence of adjacent arcs. 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*. An $[a, b]$-*path* is a finite path beginning at the vertex a and ending at b. In this case, the vertex b is said to be *reachable* from the vertex a. An $[a, b]$-*path* is *closed* if $a = b$. A *simple path* is an open path in which each vertex is incidental to not more than two arcs of the path. If the path in this definition is closed (the beginning and the endpoint of the path coincide), it is called a *simple cycle*. A graph is said to be *connected* if, for any pair of vertices, at least one of the vertices is reachable from another and *strongly connected* if any vertex is reachable from any other vertex. In what follows, we consider only finite strongly connected graphs.

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 the set of symbols 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., $e\alpha = v$. The robot is in a state $q \in Q$ and reads the symbol of the vertex vx and the symbol of the arc ex. The transition $(q, vx, ex, q', x'_v, x'_e, i/o) \in T$ means that the robot transfers to the state q' and writes symbols to the cell of the vertex $vx = x'_v$ and to the cell of the arc $ex = x'_e$. 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 $e\delta$. In the case of an outer transition (o), the robot passes along the arc e to its terminal vertex $v\beta$ on the first arc $v\beta\gamma$ originating from it.

```
/* Inner transition: e = eδ */

void Next();
/* Outer transition: v = eβ, e = eβγ */
void Traverse();
```

Fig. 2. Robot's external procedures.

The robot is said to be *finite* if the sets Q and X are finite. The robot is *deterministic* if, for any triple $(q, vx, ex) \in Q \times X \times X$, there does not exist more than one transition $(q, vx, ex, q', x'_v, x'_e, i/o) \in T$. If, a robot occurs in a state q at the current vertex v on the current arc e and, for the triple (q, vx, ex), no transition exists, the robot is said to *stop*. One symbol from X is assumed to be *initial* and 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_1; the initial arc is the first outgoing arc $v_1\gamma$. The sequence of all outer transitions of the robot R in a graph G determines a path in G beginning at the initial vertex, which is referred to as a traversed path. If the robot stops, this path is finite. In this paper, we consider finite deterministic robots.

We will use the C language to write an algorithm of the robot. Return from the main function of the robot is interpreted as its stop. Cells of the vertices and arcs are represented by structures consisting of several *fields*. Thus, the set of symbols X is the set of values of such structures. As a rule, we use one-bit fields, the unit values of which are called *labels*. An initial symbol 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 e; thus, to access the field *field* of the cell of the vertex or arc, the constructs v.*field* or e.*field*, respectively, are used. For the inner and outer transitions, external functions *Next* and *Traverse* modifying v and e are used (Fig. 2). We will construct a series of robots such that each next robot is more complicated than the previous one and uses completely or partially less complicated robots from this series. To this end, the C programs are decomposed into functions with regard to their usage in the programs of the subsequent robots.

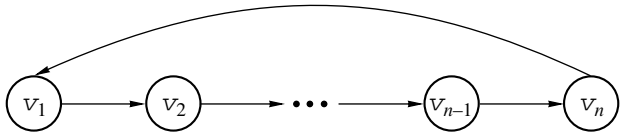

Fig. 3. A linear graph.

$v1$	$v2$	$v3$	$v4$	$v5$	$v6$	$v7$	$v8$	$v9$	$v10$	$v11$	$v12$
1	0	1	0	1	0	1	0	1	0	1	0
	1		0		1		0		1		0
			1				0				1
			1								0
											1

Candidate vertex Terminated vertex

Fig. 4. Illustration of the search for the last vertex for $O(\log n)$ passages.

3. LINEAR BACKTRACKING PROBLEM

If G is a simple cycle, it is called a linear graph (Fig. 3). Let $v_1, \ldots, v_n$ be a sequence of vertices of a cycle such that $d_{\text{out}}(v_i) = 1$ for $i = 1, \ldots, n$ and, for $i = 1, \ldots, n-1$, $v_i\gamma\beta = v_{i+1}$ and $v_n\gamma\beta = v_1$. Denote by $G[i]$ the ith arc of the graph. For $i = 1, \ldots, n$, $G[i]\alpha = v_i$, and, for $i = 1, \ldots, n-1$, $G[i]\beta = v_{i+1}$ and $G[n]\beta = v_1$.

Since the outdegree of any vertex of a linear graph is equal to 1, the function *Next* is not used for a robot on a linear graph. We assume that, in the set of robot's symbols, a subset of *terminal* symbols is separated. In a C code of an algorithm, a terminal symbol is a symbol containing the label *terminal*. The vertex marked by the label *terminal* is called a *terminal* vertex, and the marking of a vertex by the label *terminal* is referred to as the *termination* of the vertex.

A robot is said to perform a linear backtracking if it stops on any linear graph and, before the stop, terminates some vertices of the graph in the order reverse to their natural ordering: $v_{i[m]}, \ldots, v_{i[1]}$, where $i[m] > i[m-1] > \ldots > i[2] > i[1]$. Such a robot is called a *linear robot*. A *passage* of a robot along a linear graph is its displacement from the initial vertex until it comes again to the initial vertex or stops. The complexity of a linear robot is measured in terms of the number of its passages. Since, in the course of each passage (except for, perhaps, the last one), the robot passes n arcs, the length of the traversed path is bounded from above by the number of the passages multiplied by n.

Eventually, we are interested in the *complete linear backtracking* problem. In the course of such a backtracking, the robot terminates all vertices of the graph starting from the last vertex v_n and ending with the first vertex v_1. The robot that solves this problem is called a *complete linear robot*.

3.1. Search for the Last Vertex for $\Theta(\log n)$ Passages

The problem of searching for the last vertex consists in the following: starting from the first vertex of a linear graph, the robot has to find the last vertex and stop. This problem is the simplest case of the linear backtracking, where only the last vertex is terminated. Afek and Gafni showed that the problem of searching for the last vertex (the so-called "last in the ring" problem) is solved by a finite robot for $\Theta(\log n)$ passages. In this paper, we give a different variant of the proof.

Theorem 1. The problem of searching for the last vertex is solved by a finite robot for $\Theta(\log n)$ passages.

Proof. *Upper estimate.* The formal description of robot $R1$ that solves the problem of searching for the last vertex for $O(\log n)$ passages is presented in Fig. 5, and its operation is illustrated in Fig. 4. First, all vertices are candidates for the last place and, in the course of the first passage, are marked by the label *candidate*. In the next passage, the robot excludes the half of all candidates removing each alternate label. As a result, there remain candidates the parity of which in the sequence of the candidates coincides with the parity of the number of the candidates and, hence, with the parity of the last vertex (Fig. 5). To do this, in the course of each passage, the robot remembers the parity of the number of candidates and checks whether the number of the candidates remained is greater than one. If only one candidate—the last vertex—remained, the robot performs the passage to the only candidate and stops. Thus, the number of the passages is $O(\log n)$.

Lower estimate. Suppose that there exists a robot that solves the problem of the last vertex. First, we consider the behavior of the robot on an infinite simple path consisting of vertices $v_1, v_2, \ldots$. We assume that, in each passage, the robot is placed into an initial vertex in a certain arbitrarily selected initial state, and, then, it moves along the path placing new symbols to the vertices. Let us show that, before the ith passage, the sequence S_i of symbols in the vertices is periodic and can be represented as $S_i = A_i \wedge B_i^{\omega}$, where the sum of lengths of the pre-period and period satisfies the inequality $|A_i| + |B_i| \le |Q|^{i-1}$. Indeed, before the first passage, all vertices had the initial symbol, and $|A_1| = 0$, $|B_1| = 1$, $|Q|^{1-1} = 1$. Applying the induction, we assume that, before the ith passage, the assertion is true. Consider the first $|Q| + 1$ periods B_i. At least in two of them, the robot reads the first symbol of the period $B_i[1]$ in one and the same state. Suppose that the numbers of the first and second of these two periods are j and k, respectively, $1 \le j < k \le |Q| + 1$. Then, $|A_{i+1}| = |A_i| + (j-1)|B_i|$ and $|B_{i+1}| = (k-j)|B_i|$. Adding these equalities together, we find that, before the $(i+1)$th passage, $|A_{i+1}| + |B_{i+1}| = |A_i| + (k-1)|B_i| \le |A_i| + |Q||B_i| \le |Q|(|A_i| + |B_i|) \le |Q|^i$.

Let us select the sequence of the initial states of the robot on an infinite path that coincides with that on a finite linear graph. Then, in the beginning of the ith pas-

```
struct vertex { /* structure of the vertex symbol */
    unsugned start: 1 = 0; /* initial vertex */
    unsigned candidate: 1 = 0;
};
struct vertex v; /* current vertex */

void R1() {
    unsigned counter = 0; /* = 0, 1 or 2 */
    unsigned parity = 0;

    /* first passage */

    v.start = 1;
    do {
    v.candidate = 1; parity ^= 1; (counter < 2) counter++;
    Traverse();
    } while (!v.start);

    /* intermediate passages */

    while (counter > 1) {
        unsigned candidate_parity = 0;
        unsigned new_parity = 0;
        counter = 0;
        do {
            if (v.candidate){
                candidate_parity ^= 1;
                if (candidate_parity != parity) v.candidate = 0;
                else {new_parity ^= 1; if (counter < 2) counter++; }
           }
           Traverse();
        } while (!v.start);
        parity = new_parity;
    }


    /* last passage */

    while (!v.candidate) Traverse();
    v.candidate = 0;
}
```

Fig. 5. Robot $R1$. Search for the last vertex for $O(\log n)$ passages.

sage, the sequence S_i^* of symbols in the vertices of the linear graph is an initial segment of the sequence S_i on the infinite path. If, in the course of the kth passage, the robot stops its operation at the last vertex, then the sum of the pre-period and period for the sequence S_{k+1} is not less than n. Hence, we obtain $n \le |A_{k+1}| + |B_{k+1}| \le |Q|^k$; i.e., $k = \Omega(\log n)$. □

3.2. Filtering Robot

Let $U \subseteq V$ be a subset of vertices of a linear graph G containing the initial vertex. Then, for an arbitrary robot R on this graph, one can construct a *filtering robot* $R\langle U\rangle$ that behaves like robot R on the U-vertices and ignores the $V\backslash U$-vertices. To do this, it is sufficient to replace the function *Traverse* in the program of robot R by the function *Traverse*$\langle U\rangle$, which filters the vertices checking their membership in the set U:

```
void Traverse<U>()
{ Traverse(); while (v ∉ U) Traverse(); }
```

Clearly, the behavior of robot $R\langle U\rangle$ on the graph G is equivalent to that of robot R on the graph $G\langle U\rangle$, which is obtained from G by removing all vertices belonging to $V\backslash U$ and merging two arcs incidental to each vertex being removed. If robot R stops on any linear graph and accomplishes $T(|V|)$ passages, then the filtering robot $R\langle U\rangle$ also stops and accomplishes $T(|U|)$ passages. If R finds the last vertex in the graph $G\langle U\rangle$, robot $R\langle U\rangle$ finds the last vertex in the set U by using the same number of passages. If R executes the complete linear backtracking, then robot $R\langle U\rangle$ executes a partial (incomplete) linear backtracking terminating all vertices of the set U, which is equivalent to the complete linear backtracking of robot R in the graph $G\langle U\rangle$.

```
struct vertex { /* structure of the vertex symbol */
    unsugned terminal: 1 = 0; /* label of terminal symbol */
    unsugned start: 1 = 0; /* initial vertex */
    unsigned candidate: 1 = 0;
};
struct vertex v; /* current vertex */

void Traverse_range() /* filtering function Traverse for robot R1 */
{ Traverse(); if (v.terminal) while (!v.start) Traverse(); }

void R2() {
    do {
        R1<Traverse_range>(); v.terminal = 1;
        while (!v.start) Traverse(); /* return to initial vertex */
    } while (!v.terminal);
}
```

Fig. 6. Robot $R2$ on the basis of robot $R1$. Complete linear backtracking for $O(\log n)$ passages.

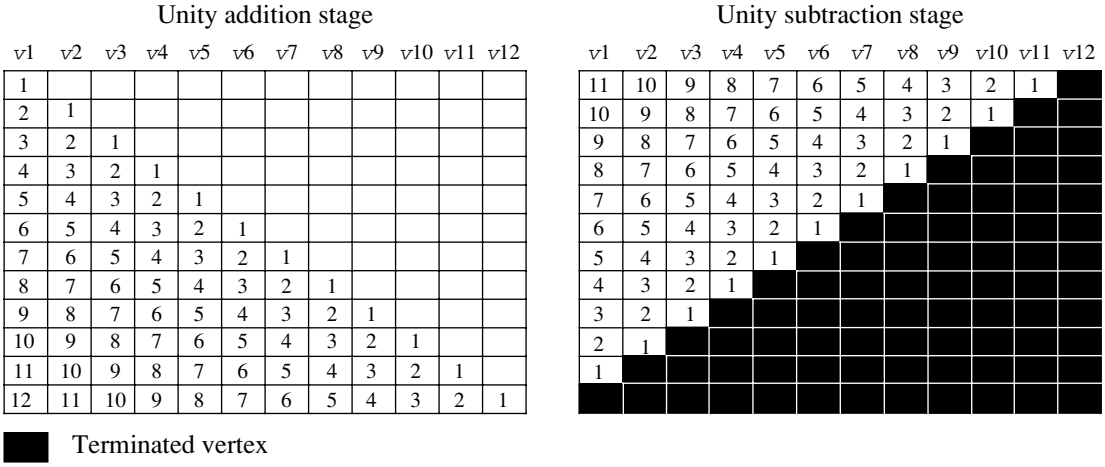

Fig. 7. Illustration of the complete linear backtracking for $O(n)$ passages in the case of an infinite number of symbols.

We will use the following two methods for specifying the set U.

The first method. In the structure of a vertex symbol, we define the label *filter*. The set U is the set of the *filter* vertices. The filtering is implemented as follows:

```
void Traverse_filter()
{ Traverse(); while (!v.filter) Trave-
rse(); }
```

The second method. In the structure of a vertex symbol, we define the label *range* of the beginning of a range. The set U is the set of all vertices belonging to the interval between the beginning of the range (inclusive) and the first terminated or initial vertex (not inclusive). The filtering is implemented as follows:

```
void Traverse_range()
{ Traverse(); if (v.terminal || v.start)
while (!v.range) Traverse(); }
```

A filtering robot with a filtering function *Traverse_…* is denoted as $R\langle Traverse\_\ldots\rangle$.

3.3. Complete Linear Backtracking for $O(n\log n)$ Passages

Now, using robot $R1$, we construct the first complete linear robot $R2$.

Theorem 2. There exists a finite robot $R2$ that solves the complete linear backtracking problem for $O(n\log n)$ passages.

Proof. The formal description of robot $R2$ is presented in Fig. 6. To terminate the last nonterminated vertex, the filtering robot $R1\langle Traverse\_range\rangle$ is invoked, and the label *start* is used for the label *range*. Robot $R2$ stops when the initial vertex is terminated. Let us denote by $k(i)$ the number of passages of robot $R1$ in a linear graph containing i vertices. Then, $k(i) \le C\log_2(i)$ for $i > N$, where C and N are constants. If $n > N$, the number of passages of robot $R2$ does not exceed $\sum\{k(i)|i = 1, \ldots, N\} + \sum\{C\log_2(i)|i = N+1, \ldots, n\} \le \sum\{k(i)|i = 1, \ldots, N\} + Cn\log_2 n = O(n\log n)$. □

```
struct vertex { /* structure of the vertex symbol */
    unsigned terminal: 1 = 0; /* label of terminal symbol */
    unsigned logstart: 1 = 0; /* initial vertex */
    unsigned number = 0; /* infinite set of possible values */
};
struct vertex v; /* current vertex */

#define GO_TO_LOGSTART while (!v.logstart) Traverse();

void Plus_pass() { /* unity addition stage */
    while (v.number){ v.number++; Traverse(); }
    v.number = 1; Traverse();
}

void Minus_pass() { /* unity subtraction stage */
    while (v.number > 1) { v.number--; Traverse(); }
    v.number = 0;
}

void R3() {
    unsigned counter = 1;
    v.logstart = 1;
    do { /* unity addition stage */
        Plus_pass(); if (v.logstart) counter = 0; GO_TO_LOGSTART
    } while(counter);
    do { /* unity subtraction stage */
        Minus)pass(); v.terminal = 1; GO_TO_LOGSTART
    } while (!v.terminal);
}
```

Fig. 8. Robot *R*3. Complete linear backtracking for $O(n)$ passages in the case of an infinite number of symbols.

3.4. Complete Linear Backtracking for O(n) Passages in the Case of an Infinite Number of Robot Symbols

Theorem 3. There exists a robot *R*3 with an infinite number of symbols and a finite number of states that solves the complete linear backtracking problem for $O(n)$ passages.

Proof. The operation of robot *R*3 consists of the following two stages (the formal description of robot *R*3 and an illustration of its operation are presented in Figs. 8 and 7, respectively).

Unity addition stage. In the first passage, the first vertex gets number one. Each next passage increases numbers of the vertices by one, puts number one to the first unnumbered vertex, and checks whether there remained unnumbered vertices. This stage is over when all vertices got numbers $n, n-1, \ldots, 2, 1$.

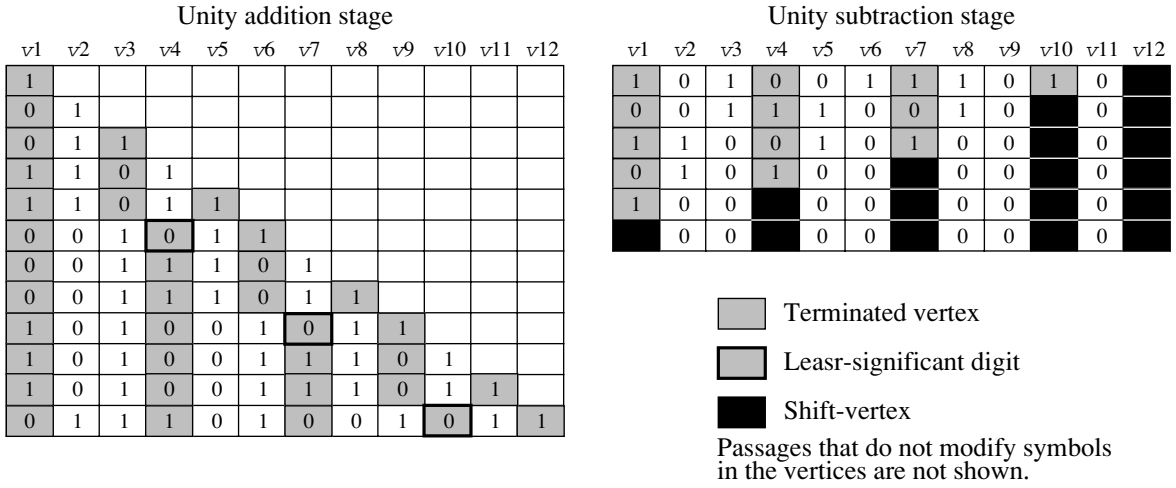

Fig. 9. Illustration of the logarithmic linear backtracking for $O(n)$ passages.

```
struct vertex {/* structure of the vertex symbol */
   unsigned terminal: 1 = 0; /* label of terminal symbol */
   unsigned logstart: 1 = 0; /* initial vertex */
   unsigned number: 1 = 0; /* indicator of number digit */
   unsigned low: 1 = 0; /* indicator of least-significant digit of number */
   unsigned bit: 1 = 0; /* content of number digit */
   unsigned shift: 1 = 0; /* from this point,
   unity addition stage continues after shift */
};
struct vertex v; /* current vertex */

#define END_OF_NUMBER v.low || !v.number || v.logstart || v.terminal
#define END_OF_LAST_NUMBER !v.number || v.logstart || v.terminal

unsigned vertex_counter = 0; /* = 1, 2, 3 number of vertices */

void Number_calculation() { /* calculation of number and the number of vertices */
   unsigned bit_position = 0; /* = 0,1,2 */
   unsigned number = 0; /* = 0,1,2,3,4 */
   do {
        switch (bit_position) { /* calculation of number */
             case 0: bit_position++; number = v.bit; break;
             case 1: bit_position++; number += 2*v.bit; break;
             default: if (v.bit) number = 4;
        }
        if (vertex_coumter < 3) vertex_coumter++;
        Traverse();
   } while (!END_OF_NUMBER);
   return number;
}

void Shift_pass(low) unsigned low; { /* shift of numbers by one vertex */
    struct vertex prev = {0,0,1,0,1,0}; /* position containing 1 */
    struct vertex curr;
    prev.low = low;
    while (v.number)
        { curr = v; v = prev; prev = curr; Traverse(); }
    v = prev; Traverse();
}

char * Plus_pass() { /* 1-2 passages of unity addition */
    unsigned carry = 1;
    unsigned last;
    unsigned shift = 0; /* =1 in the carry from the most significant digit on
    a nonlast number */
    unsigned end; /* = 1 if all vertices are numbered */

    if (!v.number) {/* there are no numbered vertices */
        Shift_pass(1); /* new number 1 */
    else { /* numbered vertices are available */
        do { /* calculation of the last number */
            last = Number_calculation();
        } while (!END_OF_LAST_NUMBER);
        if (last == 2) { /* the last number 2 */
            Shift_pass(1); /* new number 1 */
        }
        else { /* the last number 1 */
            GO_TO_LOGSTART
            while (!v.shift) Traverse(); /* search for shift vertices */
            v.shift = 0;
            while (1){ /* loop of unity addition */
                do { /* addition of unity to one number */
                    v.bit ^= carry; carry &= !v.bit; Traverse();
                } while (!END_OF_NUMBER);
                if (carry) { /* carry from the most significant number */
                    if (!END_OF_LAST_NUMBER) { shift = 1; v.shift = 1; }
                    Shift_pass(0); break;

                }
            }
        }
    }
    while (!v.logstart) /* GO_TO_LOGSTART with searching for unnumbered ver-
    tices */
    { if (!v.number) end = 0; Traverse(); }
    if (!shift) v.shift = 1; /* no carry from the most significant digit of a
    nonlast number */
    if (end) return ''all vertices are numbered'';
    else return ''there remained unnumbered vertices'';
}
```

Fig. 10. Robot *R*4 on the basis of robot *R*3. Logarithmic linear backtracking for $O(n)$ passages (end).

```
char * Minus_pass() { /* from one through four passages of unity subtrac-
tion */
     unsigned num_counter = 0; /* = 1, 2 the number of numbers */
     unsigned last = 0; /* = 1, 2, 3, 4 the last number */
     unsigned next_to_last; /* = 1, 2, 3, 4 the next-to-last number */
     unsigned steal = 1;
     do { /* Passage for computing the number of vertices,
     the number of numbers, and the last two numbers */
          next_to_last = last; /* the next-to-last number */
          last = Number_calculation();
          if (num_counter < 2) num_counter++; /* the number of numbers */
     } while (!END_OF_LAST_NUMBER);
     GO_TO_LOGSTART
     if (vertex_counter == 1) return ''one vertex'';
     if (vertex_counter == 2) /* two vertices, number 2 */
     { Traverse(); return ''more than one vertex''; }
     if (num_counter == 1) /* least-significant digit of the only number */
          return ''more than one vertex'';
     next_to_last -= last;
     while (last > 0) { /* From one to two passages of unity subtraction */
          do {
               do { /* subtraction of unity from one number */
                    v.bit ^= steal; steal &= v.bit; Traverse();
               } while (!END_OF_NUMBER);
          } while (!END_OF_LAST_NUMBER);
          GO_TO_LOGSTART
          last--;
     }
     /* One passage of searching for next-to-last number */
     while (Number_calculation() != next_to_last);
     return ''more than one vertex''; /* least-significant digit of the last
     number */
}

void Log_Terminal() { v.terminal = 1; }

void R4() {
     v.logstart = 1;
     /* unity addition stage */
     while (Plus_pass() == ''there remained unnumbered vertices'');
     do { /* unity subtraction stage */
          Minus_pass(); Log_Terminal(); GO_TO_LOGSTART
     while (!v.terminal);
}
```

Fig. 10. (Contd.)

Unity subtraction stage. In the course of each passage, the robot decreases numbers of the vertices by one. As soon as the number of a vertex becomes equal to zero, this vertex is terminated. This stage is over when the initial vertex is terminated.

In each stage, the robot performs not more than n passages; therefore, the total number of the passages is not greater than $2n$. □

3.5. Logarithmic Linear Backtracking for O(n) Passages

A linear backtracking is said to be *logarithmic* if, in a sequence of vertices $v_{i[m]}, \ldots, v_{i[1]}$ being terminated, the distance between two adjacent terminated vertices, as well as the distance between the extreme terminated vertices and extreme vertices of the graph, is bounded from above by the logarithm of the number of the graph vertices, i.e., $n - i[m] < [\log_2 n] + 1$, $i[j] - i[j-1] < [\log_2 n] + 1$, for $m \geq j > 1$, $i[1] - 1 < [\log_2 n] + 1$. If a robot solves this problem, it is referred to as a *logarithmic robot.*

Theorem 4. There exists a finite robot $R4$ that solves the logarithmic linear backtracking problem for $O(n)$ passages.

Proof. This robot emulates the operation of robot $R3$. The idea is as follows. In order to write a number i, its representation B_i in the form of a binary positional code, which is written bitwise in a sequence of vertices, is used. This code is a sequence of binary digits 0 and 1

```
truct vertex { /* structure of the vertex symbol */
   unsigned terminal: 1 = 0; /* common field of robots R4 and R */
   unsigned logstart: 1 = 0; /* field of robot R4 */
   unsigned number: 1 = 0; /* field of robot R4 */
   unsigned low: 1 = 0; /* field of robot R4 */
   unsigned bit: 1 = 0; /* field of robot R4 */
   unsigned shift: 1 = 0; /* field of robot R4 */
   unsigned range: 1 = 0; /* label of range beginning */
   ... /* field of robot R, except field terminal */
};
struct vertex v; /* current vertex */

void Traverse_range() { /* filtering function Traverse for robot R */
    Traverse(); if (v.terminal || v.logstart) while (!v.range) Traverse();
}

void Log_Terminal(){ /* backtracking inside number */
    v.range = 1;
    R<Traverse_range>(); /* call of robot R in the range */
    v.range = 0;
}
```

Fig. 11. Robot $R5$ (R). Combination of logarithmic ($R4$) and complete (R) linear robots.

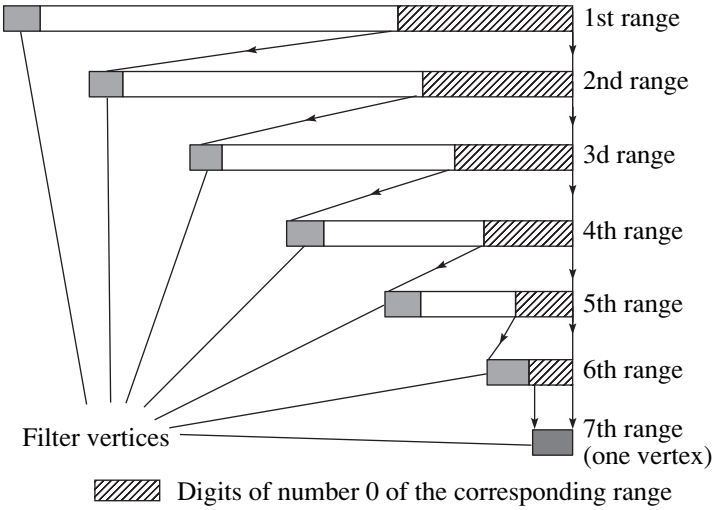

Fig. 12. Illustration of the complete linear backtracking for $O(n\log^*(n))$ passages.

of length $[\log_2 i] + 1$. Note that the least-significant bit corresponds to the beginning of the sequence and the most significant bit contains 1 and corresponds to the end of the sequence. For example, the binary representations for numbers $i = 7$ and 8 are $B_7 = 111$ and $B_8 = 0001$. The vertex containing the least-significant bit is marked by the label *low*. For the logarithmic backtracking, only the first vertex (containing the least-significant bit), rather than all vertices B_i, is terminated (the exception is a code consisting of two digits; in this case both vertices are terminated). The formal description of this robot and an illustration of its operation are presented in Figs. 10 and 9, respectively.

The number i and unity are added together bitwise by means of the standard algorithm for the binary addition: $0 + 1 = 1$, $1 + 1 = 0$ and the carry of 1 to the next position. The carry is stored in the state of the robot. This procedure starts when the robot is in the vertex of the least-significant digit of the number. If there is no carry from the most significant digit of the number, the length of the binary code is not changed, and the robot adds one to the next number. Otherwise, for the number of the form $i = 2^s - 1$, the addition of one causes the carry from the most significant digit of the number and increases the length of the binary code by one. The robot marks the next vertex by the label *shift* and carries over all bits of the subsequent numbers without any changes by one vertex in the forward direction. In the next passage, the addition of unity starts from the label *shift*. If the carry from the most significant digit of the last number occurs, the label *shift* marks the initial vertex. On the whole, the sequence of numbers has the form $k, k-1, \ldots, r+1, [r,] r-1, \ldots, 2, 1$, where one number r, $k > r \geq 1$, may be lacking (clearly, the lacking

```
struct vertex { /* structure of the vertex symbol */
    unsigned filter: 1 = 0; /* filtering field for the function Traverse_filter */
    /* fields of robot R4 */
    unsigned terminal: 1 = 0; /* label of terminal symbol */
    unsigned logstart: 1 = 0; /* label of the range beginning */
    unsigned number: 1 = 0; /* indicator of number digit */
    unsigned low: 1 = 0; /* indicator of least-significant digit of number */
    unsigned bit: 1 = 0; /* content of number digit */
    unsigned shift: 1 = 0; /* from this point, unity addition stage continues
    after shift */
    /* fields of robot R1 */
    unsigned start: 1 = 0; /* initial vertex */
    unsigned candidate: 1 = 0;
};
struct vertex v; /* current vertex */
void Traverse_filter() { /* filtering field for the function Traverse for
robot R1 */
{ Traverse(); while (!v.filter) Traverse(); }

void Traverse_range() /* filtering field for the function Traverse for
robot R4 */
{ Traverse(); if (v.terminal || v.start) GO_TO_LOGSTART }

char * Minus_pass7() { /* unity subtraction */
    unsigned new_range_start;
    if (Minus_pass<Traverse_range>() == ''more than one vertex'') {
        /* opening new nested range */
        if (!v.logstart) /* new beginning of nested range */
        { new_range_start = 1; v.logstart = 1; v.filter = 1; }
        else new_range_start = 0; /* old beginning of nested range */
        do { /* placement of initial symbol in the range */
            v.number = 0; v.low = 0; v.bit = 0; v.shift = 0;
            Traverse();
        } while (!v.terminal && !v.start);
        GO_TO_LOGSTART
        if (new_range_start) /* transition to the new beginning of nested range */
            { v.logstart = 0; GO_TO_LOGSTART }
        return ''more than one vertex'';
    }
    else return ''one vertex''
}

void R7() {
    v. start = 1; v.logstart = 1;
    while (1) {
        /* unity addition stage in the range*/
        while (Plus_pass<Traverse_range>()
        == ''there remained unnumbered vertices'');
        while (Minus_pass7() == ''one vertex'') { /* unity subtraction */
            v.terminal = 1; /* vertex termination */
            v.logstart = 0; v.filter = 0; /* removal of range from one vertex */
            if (v.start) return; /* initial vertex is terminated */
            R1<Traverse_filter>(); /* searching for beginning
            of the enclosing range */
            v.logstart = 1; /* transition to the enclosing range */
        } /* more than one vertex in the range */
    }
}
```

Fig. 13. Robot $R7$ on the basis of logarithmic robot ($R4$) and robot searching for the last vertex ($R1$). Complete linear backtracking for $O(n\log^*(n))$ passages.

number has the form $r = 2^s - 1$). The last number is always 1 ($r > 1$) or 2 ($r = 1$). In the function *Plus_pass*, the robot performs one or two passages. If none of the vertices is numbered yet, the initial vertex becomes the least-significant and only digit of the only number 1. Otherwise, the last number is determined first. If it is equal to 1, unity is added to all numbers until the first carry from the most significant digit. If the last number

is 2, the robot finds it and places number 1 behind it. Each call of the function *Plus_pass* adds one, and only one, vertex to the list of numbered vertices.

The subtraction of unity from the number is also implemented bitwise by means of the standard algorithm for the binary subtraction: $1 - 1 = 0$, $0 - 1 = 1$ and the borrow of 1 from the next position. The borrow is stored in the state of the robot. Note that, at this stage, the binary code of number i can contain more than $[\log_2 i] + 1$ digits; i.e., the higher digits may be zero. At this stage, the robot terminates vertices of the lower digits of the number (except for the case of two vertices, when the last vertex is terminated). In the function *Minus_pass*, in the course of one passage, the robot, first, computes the number of vertices (1, 2, or greater) counting only nonterminated vertices, the number of numbers (1 or greater), and the last and next-to-last numbers. If there are one or two vertices, the last vertex is terminated. If there are more than two vertices and only one number, then the initial vertex is terminated. In all other cases, the robot performs one or two passages subtracting 1 until the last number vanishes. Then, in the course of one passage, the robot finds the next-to-last number and terminates the next vertex corresponding to the least-significant digit of the last number. The stage is over, when the initial vertex is terminated.

In the course of each call of the function *Plus_pass* (one or two passages), one vertex is numbered; therefore, the unity addition stage requires not more than $2n$ passages. At the second stage, one vertex is terminated for not more than four passages. Since the number of numbers is obviously not greater than n, this stage requires not more than $4n$ passages. Totally, we have $O(n)$ passages. It is evident that the length of the binary code for each number $k, \ldots, 1$ does not exceed the length of the binary code of number $k \le n$, and the length of the binary code of number n is equal to $[\log_2 n] + 1$. Thus, the robot accomplishes the logarithmic linear backtracking. □

3.6. Combination of the Logarithmic and Complete Linear Robots

Lemma 1. Let $S(k) = \sum\{ L(i) | i = 1, \ldots, k\} - L(r)$, where $[\log_2 k] + 1 \ge L(k) \ge 1$ for $i < k$, $L(i) = [\log_2 i] + 1$, and $k > r \ge 1$. If $S(k) = n$, then, for sufficiently large $n > N_2$, $L(i) \le \log_2 n$ for $i = 1, \ldots, k$.

Proof. Indeed, since $L(k) \ge 1$, $L(i) = [\log_2 i] + 1 \ge \log_2 i$, and $\log_2(k-1) + 2 \ge [\log_2(k-1)] + 1 \ge L(k-1) \ge L(r)$, we have $n = S(k) \ge 1 + \sum\{ \log_2 i | i = 1, \ldots, k-1\} - \log_2(k-1) - 2 = \log_2(k-1)! - \log_2(k-1) - 1 = \log_2((k-2)!/2)$. Since logarithm of the factorial grows faster than a linear function, we have $\log_2((k-2)!/2) \ge 4k$ for sufficiently large k ($k \ge 49$). Hence, $\log_2 n \ge \log_2 k + 2$. For $i = 1, \ldots, k$, we have $L(i) \le [\log_2 k] + 1 \le \log_2 k + 2$. Thus, for sufficiently large k, $\log_2 n \ge \log_2 k + 2 \ge L(i)$ for $i = 1, \ldots, k$.

On the other hand, since $L(k) \le [\log_2 k] + 1$, $L(i) = [\log_2 i] + 1 \le \log_2 i + 1$, and $L(r) \ge L(1) = 1$, we have $n = S(k) \le \log_2 k + 1 + \sum\{ \log_2 i + 1 | i = 1, \ldots, k-1\} - 1 = k - 1 + \log_2 k!$. Since the function $k - 1 + \log_2 k!$ grows monotonically, for sufficiently large $n > N_2$ ($N_2 = 256$), k is also large ($k > 49$), and, hence, $\log_2 n \ge L(i)$ for $i = 1, \ldots, k$. □

Lemma 2. Let $T(n)$ be a monotone nondecreasing function, and let, starting from some $n > N$, $L(i) \le \log_2 n$, $\sum\{ L(i)\} \le n$. Then, $\sum\{ L(i)T(L(i))\} \le nT(\log_2 n)$ for $n > N$.

Proof. Indeed, since T is a monotone nondecreasing function, we have $\sum\{ L(i)T(L(i))\} \le \sum\{ L(i)\}T(\log_2 n) = \sum\{ L(i)T(\log_2 n)\} \le nT(\log_2 n)$. □

Theorem 5. Let a finite robot R solve the complete linear backtracking problem for $O(nT(n))$ passages,

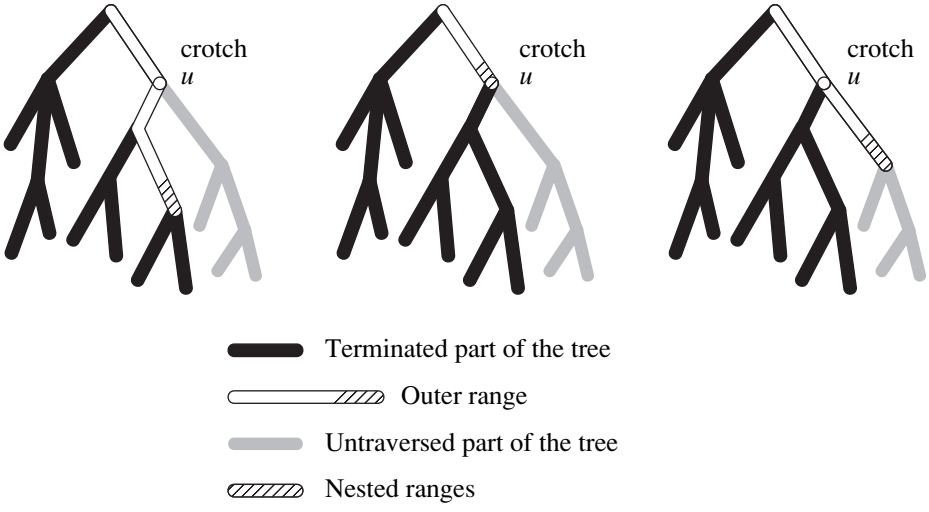

Fig. 14. Illustration of the backtracking along a tree for $O(n\log^*(n))$ passages.

where $T(n)$ is a monotone nondecreasing positive function. Then, there exists a finite robot $R5(R)$ that solves the complete linear backtracking problem for $O(nT(\log_2 n))$ passages.

Proof. Robot $R5$ (R) is constructed on the basis of robot $R4$, which performs the logarithmic linear backtracking for $O(n)$ passages. The latter robot is modified by modifying the function *Log_Terminal*. Each time when robot $R4$ terminates the vertex corresponding to the least-significant digit of number i, robot $R5$ places the label of the range beginning *range* and calls the filtering robot $R\langle Traverse\_range\rangle$, which performs the complete linear backtracking in the range corresponding to all digits of number i. After this, the label *range* is deleted.

The code of this robot in Fig. 11 presents only the vertex symbol structure; the function *Log_Terminal*, which makes the modified robot $R4$ different from the original one; and the filtering function *Traverse_range*, which is used by the filtering robot as described above (see Section 3.2). Note that, in the course of constructing $R5(R)$, the fields of robots $R4$ and R are placed into the symbol structure. Possible collisions of names are resolved by renaming the fields.

The number of passages when robot $R5(R)$ works like robot $R4$ is obviously the same as that for robot $R4$; i.e., $P_{R4}(n) = O(n)$.

At the end of the unity addition stage performed by robot $R4$, the sequence of numbers has the form $k, k-1, \ldots, r+1, [r,] r-1, \ldots, 2, 1$, where one number r, $k > r \geq 1$, may be lacking. The range length is $L(i) = [\log_2 i] + 1$. The sum of lengths of the range is $S(k) = \sum\{ L(i) | i = 1, \ldots, k\} - L(r) = n$. Thus, the assumptions of Lemma 1 are fulfilled, and, hence, for $n > N_2$, the lengths of all ranges satisfy the inequality $L(i) \leq \log_2 n$. The filtering robot $R(Traverse\_range)$ works in all ranges corresponding to the digits of number i. The number of passages of the robot in the ith range satisfies the inequality $P_R(i) \leq CL(i)T(L(i))$ for $i > N$, where C and N are some constants. Since T is a monotonically nondecreasing function, the assumptions of Lemma 2 for the case of $i > N$ are also fulfilled. Therefore, the total number of passages of the filtering robot satisfies the inequality $\sum P_R(i) \leq \sum\{ P_R(i) | i = 1, \ldots, N\} + \sum\{ CL(i)T(L(i)) | i = N, \ldots, k\} \leq \sum\{ P_R(i) | i = 1, \ldots, N\} + CnT(\log_2 n)$ for $n > N_2$. Since T is a monotone nondecreasing and positive function, we obtain the estimate $\sum P_R(i) = O(nT(\log_2 n))$.

Thus, the total number of passages of robot $R5(R)$ is $P_{R5}(n) = P_{R4}(n) + \sum P_R(i) = O(n) + O(nT(\log_2 n))$. Since T is a monotone nondecreasing and positive function, we have the estimate $P_{R5}(n) = O(nT(\log_2 n))$. □

3.7. Compositional Degree of Logarithmic Robot and Complete Linear Backtracking for $O(n\log_2^t n)$ Passages for Any Fixed $t \geq 1$

A *compositional degree* of logarithm is the function $\log_2^t = \log_2 \circ \log_2 \circ \ldots \circ \log_2$, where the superposition sign is applied $t-1$ times.

Theorem 6. For any integer $t \geq 1$, there exists a finite robot $R6(t)$ that solves the complete linear backtracking problem for $O(n\log_2^t n)$ passages.

The theorem is *proved* by the induction on t. In accordance with Theorem 2, for $t = 1$, we have robot $R6(1) = R2$ with the number of passages $O(n\log_2 n)$. Suppose that the assertion of the theorem is true for t; i.e., there exists a robot $R6(t)$ with the number of passages $O(n\log_2^t n)$. Let us prove the assertion for $t+1$. Denoting $T(n) = \log_2^t n$, we apply Theorem 5 using robot $R6(t)$ for robot R in robot $R5(R)$. Then, we obtain robot $R6(t+1)$ with the number of passages $O(nT(\log_2 n)) = O(n\log_2^t \log_2 n) = O(n\log_2^{t+1} n)$, which was required to prove. □

3.8. Complete Linear Backtracking for $O(n\log^*(n))$ Passages

The number of passages of robot $R6(t)$ is bounded from above by $O(n\log_2^t n)$. For a sufficiently large t, we have $\log_2^t n = O(1)$; i.e., we can apply the composition of the logarithmic robots until the length of the intervals obtained becomes less than a certain constant. In the small intervals obtained, we can now apply the backtracking algorithm with the number of passages bounded from above by some constant. As a result, we obtain a robot with the number of passages of the order of $n\log^*(n)$, where the function log* yields the required number of logarithm operations and is defined as an integer solution of the inequality $1 \leq \log_2^{\log^*}(n) < 2$ (the logarithm base for log* is equal to 2).

However, there remains a difficulty associated with the number of states and symbols of the robot. Each robot composition increases the number of states and symbols, which become functions of $\log^*(n)$ and, thus, n. This can be avoided if we get rid of the recursion in the robot composition replacing it by iteration. Note that this can be done without increasing the order of the number of passages.

Theorem 7. There exists a finite robot $R7$ that solves the complete linear backtracking problem for $O(n\log^*(n))$ passages.

Proof. The formal description of robot $R7$ is presented in Fig. 13, and an illustration of its operation is shown in Fig. 12. Like robot $R6(t)$, robot $R7$ is constructed as a system of logarithmic robots $R4$ filtering

```
struct vertex { /* structure of the vertex symbol */
    unsigned filter: 1 = 0; /* filtering field for the function Traverse_filter */
    /* fields of robot R4 */
    unsigned terminal: 1 = 0; /* label of terminal symbol */
    unsigned logstart: 1 = 0; /* label of the range beginning */
    unsigned lastlogstart: 1 = 0; /* label of the beginning of the last range */
    unsigned number: 1 = 0; /* indicator of number digit */
    unsigned low: 1 = 0; /* indicator of least-significant digit of number */
    unsigned bit: 1 = 0; /* content of number digit */
    unsigned shift: 1 = 0; /* from this point, unity addition stage continues
    after shift */
    /* fields of robot R1 */
    unsigned start: 1 = 0; /* initial vertex */
    unsigned candidate: 1 = 0;
};
struct vertex v; /* current vertex */

struct arc { /* structure of the arc symbol */
    unsigned first: 1 = 0; /* indicator of arc vβ, the first arc in the v-loop
    of arcs */
    unsigned active: 1 = 0; /* active arc indicator */
};
struct arc e; /* current arc */

void Traverse_active() { /* function Traverse for traversing active cycle */
    if (!e.first) { e.first = 1; e.active = 1; }
    while (!e.active) Next(); Traverse();
}

void Traverse_filter() /* filtering field for the function Traverse for
robot R1 */
{ Traverse_active(); while (!v.filter) Traverse_active(); }

#define GO_TO_LOGSTART while (!v.logstart) Traverse_active();

void Traverse_range() { /* filtering field for the function Traverse for
robot R4 */
    Traverse_active(); if (v.terminal || v.start) GO_TO_LOGSTART
}

#define END_OF_NUMBER v.low || !v.number || v.filter || v.terminal
#define END_OF_LAST_NUMBER !v.numver || v.filter || v.terminal

char * Minus_pass8() { /* unity subtraction */
    unsigned new_range_start;
    if (Minus_pass<Traverse_range>() == ''more than one vertex'') {
        /* opening new nested range */
        if (!v.logstart) /* new beginning of nested range */
        { new_range_start = 1; v.logstart = 1; v.filter = 1; }
        else new_range_start = 0; /* old beginning of nested range */
        do { /* placement of initial symbol in the range */
            v.number = 0; v.low = 0; v.bit = 0; v.shift = 0;
            Traverse_active();
        } while (!v.terminal && c.start);
        GO_TO_LOGSTART
        if (new_range_start) /* transition to the new beginning of nested range */
            { v.logstart = 0; GO_TO_LOGSTART }
        return ''more than one vertex'';
    }
    else return ''one vertex''
}
```

Fig. 15. Robot $R8$. Backtracking along a tree for $O(n\log^*(n))$ passages.

```
void R8() {
    v.start = 1; v.logstart = 1;
    while (1) {
        /* unity addition stage in the range*/
        while (Plus_pass<Traverse_range>()
        == ''there remained unnumbered vertices'') ;
        while (Minus_pass8() == ''one vertex'') {
            /* changing active arc */
            while (!e.active) Next(); e.active = 0; Next(); e.active = 1;
            if (!e.filter) { /* new active arc */
                while (!e.filter) Next();
                /* transition to the first range*/
                v.logstart = 0; v.lastlogstart = 1;
                while (!v.start) Traverse_active();
                v.logstart = 1;
                /* unity addition stage in the first range */
                while (Plus_pass<Traverse_range>()
                == ''there remained unnumbered vertices'') ;
                /* transition to the last range */
                v.logstart = 0;
                while (!lastlogstart) Traverse_active();
                v.logstart = 1; v.lastlogstart = 0;
            }
            else { /* all outgoing arcs have already been activated */
                v.terminal = 1; /* vertex termination */
                v.logstart = 0; v.filter = 0; /* removal of unit range */
                if (v.start) return; /* initial vertex is terminated */
                R1<Traverse_filter>(); /* searching for beginning
                of the enclosing range */
                v.logstart = 1; /* transition to the enclosing range */
        } /* more than one vertex in the range,
        transition to unity addition stage */
    }
}
```

Fig. 15. (Contd.)

by the nested ranges. The outermost range (the range of level 1) is the entire path from the first to the last vertex. After opening the range of level i, the unity addition stage is executed. Then, the function *Minus_pass* is invoked (from one through four passages). This function gives us information on the number of vertices in the range (one or greater). If the range has more than one vertex, the next nested range of level $i + 1$ is opened, which contains vertices of all digits of the last number of level i. (In the range of length 2, the nested unit range containing the last vertex is opened.) The range begins with the label *logstart*, and the label *logstart* of the enclosing range is deleted (excluding the case where both ranges begin with one vertex). In addition, the beginnings of all acting ranges are marked by the label *filter*. When opening a nested range, an initial symbol is placed to its vertices (except for the labels *logstart* and *filter*). If the innermost range contains only one vertex (the information provided by the function *Minus_pass*), it is deleted, the vertex is terminated, and the labels *logstart* and *filter* are removed. Then, the robot returns to the enclosing range. To this end, by means of the filtering robot $R1\langle Traverse\_filter\rangle$, the last *filter* vertex (i.e., the beginning of the enclosing range) is sought and marked by the label *logstart*.

Let us estimate the number of passages of the robot. First, we consider the passages executed by robots $R4$ (viewed as an aggregate of the functions *Plus_pass* and *Minus_pass*). By Theorem 4, the number of passages of this robot in a range of length m satisfies the inequality $P_{R4}(m) \le Cm$ for $m > N$, where C and N are some constants. Let C be sufficiently large such that the inequality $P_{R4}(m) \le Cm$ holds for any positive integer m. The outermost robot $R4_1$ performs not more than Cn passages and places numbers $k, k-1, \ldots, r+1, [r,] r-1, \ldots, 2, 1$, where one number r, $k > r \ge 1$, may be lacking, to the vertices. If $n > 1$, for any number $k_j > 1$ from this sequence, a range containing $L(k_j) = [\log_2 k_j] + 1$ vertices is created, in which the nested robot $R4_2$ works performing not more than $CL(i)$ passages. In view of the additivity of the linear function, robot $R4_2$ also performs totally not more than Cn passages. The same is valid for all nested robots $R4_j$. Thus, for all nested robots $R4$, the total number of passages satisfies the inequality $P^{\Sigma}_{R4}(n) \le Cnt(n)$, where $t(n)$ is the number of the nesting levels. According to Lemma 1, each nesting level reduces the size of range $m > N_2$ to $\log_2 m$. Hence,

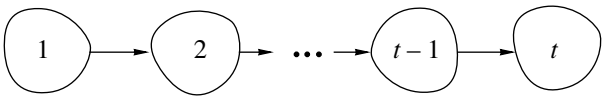

Fig. 16. A graph of the first kind.

$t(n) \le N_2 + \log^*(n) = O(\log^*(n))$ and $P^{\Sigma}_{R4}(n) = O(n\log^*(n))$.

When opening each range (except for the outermost one), one passage for marking the range by the initial symbol is executed. After the unity addition stage, in the end of each range containing m vertices, we have number 1 or 2. In the range consisting of two vertices, we open a nested unit range that contains the last vertex. The unit ranges are deadlocks (do not contain nested ranges). Thus, a vertex may be the end of not more than three ranges (containing m, 2, or 1 vertices). Therefore, the total number of ranges of all levels is equal to $O(n)$; hence, for the marking of all ranges by the initial symbol, the total number of $P_{\text{mark}} = O(n)$ passages is required.

Now, it remains to estimate the total number of passages $P^{\Sigma}_{R1}(n)$ of the filtering robot $R1\langle Traverse\_filter\rangle$. The search for the last *filter* vertex requires $P_{R1}(t) = O(\log_2 t)$ passages, where t is the number of the *filter* vertices. Each search of this kind is executed after deleting the unit range. Hence, the number of these searches does not exceed the total number of ranges $O(n)$. Since $t = O(\log^*(n))$, we have $P^{\Sigma}_{R1}(n) = O(n)O(\log_2\log^*(n)) = O(n\log_2\log^*(n)) = O(n(\log^*(n) - 1)) = O(n\log^*(n))$.

Totally, robot $R7$ performs $P_7(n) = P^{\Sigma}_{R4}(n) + P_{\text{mark}}(n) + P^{\Sigma}_{R1}(n) = O(n\log^*(n)) + O(n) + O(n\log^*(n)) = O(n\log^*(n))$ passages. □

4. BACKTRACKING ALONG A CYCLIC TREE

A *forest* is a graph not containing closed paths. A *tree* is a connected forest; any forest is a union of trees. A *rooted tree* is a tree with a distinguished vertex called a root. An *output tree* is a rooted tree in which any vertex is reachable from the root. Correspondingly, an *input tree* is a rooted tree in which the root is reachable from any vertex. A *spanning subgraph* is a subgraph of the graph with the same set of vertices. If a spanning subgraph is a tree, it is referred to as a *spanning tree*. A *chord* is a graph arc that does not belong to a spanning tree separated in the graph.

A *cyclic tree* is a graph T^* obtained from an input tree T by adding chords that lead from the leaves to the root r. We assume that the initial vertex is the tree root. The number of vertices of the cyclic tree is denoted as n. A vertex of a tree with the outdegree greater than one is called a *crotch*. Clearly, leaves of a tree cannot be crotches.

A robot on a cyclic tree uses not only the external function *Traverse* but also the function *Next*. The robot is said to perform a *backtracking along the tree* if, for any input tree T, it stops on the cyclic tree T^* and, before it stops, terminates all vertices of the tree in the order inverse to their natural partial ordering, i.e., from the leaves to the root. A *passage* of a robot along the tree is its displacement from the root to the leaves and, further, along a chord back to the root or a displacement from the root to a stop. The complexity of the backtracking is measured in terms of the number of the passages. In the course of each passage (except for, perhaps, the last one), the robot passes a simple cycle consisting of a simple path from the root to a leaf and the chord leading to the root. The length of such a cycle does not exceed n; therefore, the length of the traversed path is bounded from above by the number of the passages multiplied by n.

Theorem 8. There exists a finite robot $R8$ that solves the problem of a backtracking along the tree for $O(n\log^*(n))$ passages.

Proof. The formal description of robot $R8$ is presented in Fig. 15, and an illustration of its operation is shown in Fig. 14. Robot $R8$ is a modification of robot $R7$, which is obtained as follows. First of all, the robot selects one cycle (a linear subgraph) in the tree, which is referred to as an *active cycle*. Its arcs are marked by the label *active* and are called *active arcs*. In addition, for each traversed vertex v, the first outgoing arc of the v-cycle is marked by the label *first*. The first active cycle P_1 will consist of such first arcs.

On an active cycle P_i, the robot works similarly to robot $R7$. To this end, instead of the external function *Traverse*, the function *Traverse_active* of the traversing of an active cycle is used. The backtracking is performed to the crotch u that is closest to the leaf rather than to the initial vertex (tree root). The robot does not terminate the vertex u at once but checks whether the active arc originating from it is the last one in the u-loop. If it is the last one, the vertex is terminated, and the backtracking continues. Otherwise, the next arc e becomes active. The active cycle after the vertex u is prolonged along the arc e and, further, along the first outgoing arcs $e\beta\gamma$, $e\beta\gamma\beta\gamma$, $e\beta\gamma\beta\gamma\beta\gamma$, … to a new leaf vertex and is terminated by a chord. The cycle P_i is replaced by the cycle P_{i+1}; these cycles has a common simple $[r, u]$-path P^1_{i+1}. Thus, the robot searches through the active cycles P_1, P_2, … (all simple paths from the root to the leaf vertices), implementing thus the depth-first search of the tree, and stops when the tree root is terminated.

The change of the active cycle in the crotch u interrupts the unity subtraction stage in the last nested range, for which u is the endpoint, and continues the unity addition stage in the first (outermost) range. All ranges,

but the first one, are located in a finite range of numbered vertices of the active path, in the vertices of the zeroth number of the external range. The addition of unity shifts this finite interval until no unnumbered vertices remain in the new active path. Then, the unity subtraction stage starts again in the last range.

Now, let us estimate the number of passages of the robot. To this end, we note that robot $R8$ differs from robot $R7$ in that the addition and subtraction of unity in the first range are performed alternately rather than successively (when the subtraction is performed after the addition). The addition of unity is repeated until no unnumbered vertices remain in the current active path. Then, the subtraction of unity is performed until the active path is changed. Next, unity is added again, and so on. Note that the addition of unity does not affect the terminal interval of numbered vertices (corresponding to the bits of number 0 of the first level) but simply shifts it along the active path. When the active path is changed, the work in the nested ranges is interrupted, but it continues after completing the unity addition stage in the first range. Clearly, these modifications do not affect the estimate of the number of passages in each level, which remains equal to $O(n)$. Since the ranges of all current levels form a strictly nested structure, the number of levels, as before, is limited by the number $O(\log^*(n))$. This yields the same estimate $O(n\log^*(n))$ both for the number of passages on all levels and for the number of passages required for changing the levels, firs of all, for searching for the enclosing range when closing the unit range (function $R1\langle Traverse\_filter\rangle$). Thus, we have the same estimate $O(n\log^*(n))$ for the total number of passages. □

5. TRAVERSAL OF STRONGLY CONNECTED GRAPHS

A *strongly connected component* (further referred to as simply *component*) is the largest strongly connected subgraph that is not a subgraph of any other strongly connected subgraph. The notation $K(v)$ denotes the component to which the vertex v belongs. 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. 16). The latter arcs are said to be *connecting*.

In a graph of the first kind with an arbitrary initial vertex belonging to the first component, there always exist an output spanning tree T_{out} with the root coinciding with the initial vertex, which contains all connecting arcs, and a spanning forest of input trees F_{in} (a forest of input trees that is a spanning subgraph) consisting of the input spanning trees of the components the roots of which are endpoints of the connecting arcs (the root of the first component is the initial vertex). The arcs of the output tree are called *out*-arcs, and the arcs of the input trees, *in*-arcs. The root of the input tree of the forest F_{in} is also referred to as the root of the component K for which this tree is a spanning tree and is denoted as $r(K)$.

A *traversed graph* of a path is a subgraph consisting of the arcs belonging to the path and the incidental vertices. Clearly, a traversed graph is a graph of the first kind.

The known DFS and BFS robots use the following two algorithms for traversing a strongly connected graph: (1) the *algorithm of constructing* the output spanning tree T_{out} and the spanning forest of input trees F_{in} and (2) the *algorithm of backtracking* along the output spanning tree T_{out}. The tree T_{out} is traversed by means of DFS or BFS methods. When using the DFS algorithm [4], in the tree T_{out}, one active $[v_1, v_A]$-*out*-path from the root (initial vertex) v_1 to a nonterminated vertex v_A is selected. The robot *searches for an untraversed arc* as follows: starting from the root of the last component $r(K(v_A))$, it moves along an active path to its terminal vertex v_A and traverses the untraversed arc e originating from the vertex v_A ($\alpha e = v_A$). If the arc e is a chord, the robot *returns along the chord*, namely, using the $[\beta e, r(K(v_A))]$-*in*-path in the forest F_{in} that leads to the root of the last component and the $[r(K(v_A)), v_A]$-segment of the active path, it returns to the beginning of the chord, to the vertex v_A. Otherwise, the arc e becomes a new *out*-arc; i.e., it is added to the tree T_{out}, and its end βe becomes the only root of the last component of the traversed graph. When all arcs originating from the terminal vertex v_A are traversed, the backtracking algorithm starts working; the goal of this algorithm is to terminate the vertex v_A, reducing thus the active path by one arc, and to return to the root $r(K(v_A))$ of the last component.

The BFS algorithm [1, 3] differs from the DFS algorithm in that, in *searching for an untraversed arc*, the robot modifies the active path itself. To this end, for each vertex v of the tree T_{out}, one active arc originating from it is marked: the robot makes active the next arc in the v-loop and moves along it. The active $[v_1, v_A]$-*out*-path is the path consisting of the active arcs that starts at the initial vertex.

The backtracking problem differs from that considered in the previous section in the following three respects:

(1) The tree T_{out} is not a priori given but is constructed in the course of the operation of the first algorithm.

(2) To make the tree cyclic, instead of the chords that are used for the direct return from the leaves to the root, the *in*-paths of the tree F_{in} are used.

(3) These *in*-paths are used for returning to the root of the last component rather than to the initial vertex.

If the last (third) distinction is lacking, i.e., the traversed graph at the moment when the backtracking begins consists of only one component (a strongly connected graph), then the backtracking in the DFS algorithm can be implemented by means of a modification of robot $R8$ that takes into account only the first two distinctions.

(1) The backtracking is performed only for one terminated vertex v_A; then, the algorithm of the tree con-

struction works again until all arcs originating from the new terminal vertex of the active path become traversed.

(2) To return to the root of the last component from the terminal vertex v_A of the active path, instead of the chord leading to the initial vertex v_1, the $[v_A, v_1]$-*in*-path is used.

The presence of several components $K_1, K_2, \ldots, K_t$ in the traversed graph (the third distinction) makes the backtracking problem more complicated. If the modification of robot $R8$ is used, then, in each component K_i, its own system of nested ranges may be formed. Later on, when traversing the chord leading from the end of the active path (from the component K_t) to a component K_i, several components $K_i, K_{i+1}, \ldots, K_t$ will merge. The robot will occur in the root of the component K_i and will have either to merge the ranges of the components $K_i, K_{i+1}, \ldots, K_t$ into one range or to seek the beginning of the last range, the root of the last component K_t. Both these variants may considerably increase the estimate $O(n^2\log^*(n))$ for the backtracking.

The number of the components in the traversed graph and the configuration itself of the output tree and the forest of the input trees depend not only on the given (unordered) graph but also on its ordering, i.e., on the order of arcs in the v-loops for all vertices v. This order is defined *ab extra.* It determines the selection of a current untraversed arc by the robot from the set of the untraversed arcs originating from the current vertex: the robot selects the first untraversed arc in the v-loop. We will show that, for any strongly connected graph, there exists an arc ordering for which the traversed graph always contains only one component. In stricter terms, each component of the traversed graph (except, perhaps, the first one) consists of one vertex incidental to only the connecting arcs. This implies that, after the chord is traversed, all components are glued into one. Moreover, we have only one component at the beginning of the backtracking. We will show also that such an arc ordering can be constructed by a finite robot.

To prove this assertion, we consider an arbitrary input spanning tree T_{in} of the graph. If the arcs in the v-loops of all, but root, vertices are arranged such that the arcs of the spanning tree T_{in} are the first arcs and the other outgoing arcs are numbered arbitrarily, then we get the desired order of the arcs. Indeed, let us assume the contrary. Suppose that, at some moment, there is a (not first) component K consisting of more than one vertex. Then, K contains an arc e_1, and, since this is not the first component and the vertex αe is not initial, this vertex has an outgoing arc belonging to the spanning tree T_{in}, which is the first arc e_1^1 in the αe-loop. The arc e_1^1 either coincides with the arc e_1 or has been traversed earlier; hence, it cannot be a connecting arc and, thus, also belongs to the component K. Since the component is strongly connected, it contains an arc e_2 originating from βe_1^1. Continuing this reasoning, we obtain an infinite sequence of the first arcs $e_1^1, e_2^1, \ldots$ belonging to the component K. Since the number of different arcs in the graph is finite, we have a closed path consisting of the first arcs, which is impossible since they are arcs of a tree.

Such an order of arcs can easily be specified by marking arcs of the spanning tree T_{in}. This can be done by the robots that traverse the graph in the framework of the DFS [4] or BFS [1, 3] algorithms. These robots construct a forest of input trees F_{in} marking its arcs as *in*-arcs, and, in the end of the traversal, this forest consists just of one spanning tree T_{in}.

Thus, there exists a robot that traverses the graph twice, such that the first traversal has the estimate $O(nm + n^2\log\log n)$, and the second traversal, the estimate $O(nm + n^2\log^*(n))$.

6. CONCLUSIONS

The general problem of the traversal of an unknown strongly connected directed graph by a finite robot is still not solved. The best known results are the algorithm of the single traversal with the estimate $O(nm + n^2\log\log n)$ suggested by the author of this paper in [1] and the algorithm of a repeated traversal with the estimate $O(nm + n^2\log^*(n))$ suggested in this paper. An exact estimate for this problem (minimum of the upper bounds of the algorithms over all possible traversal algorithms) remains unknown. Moreover, although it seems unlikely that a finite robot could traverse a graph with the estimate $\Omega(nm)$, this fact has not been proved yet.

REFERENCES

1. Bourdonov, I.B., Traversal of an Unknown Directed Graph by a Finite Robot, Programining and Computer Software, 2004, vol. 30, no. 4, pp. 188–203.
2. Rabin, M.O., Maze Threading Automata. Lecture presented at MIT and UC Berkley, 1967.
3. Bourdonov, I.B., Study of the Automaton Behavior on Graphs, *MS Dissertation*, Moscow: Moscow State University, 1971.
4. Afek, Y. and Gafni, E., Distributed Algorithms for Unidirectional Networks, *SIAM J. Comput.*, 1994, vol. 23, no. 6, pp. 1152–1178.