Programming and Computer Software, Vol. 29, No. 5, 2003, pp. 245–258. Translated from Programmirovanie, Vol. 29, No. 5, 2003.

Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case

I. B. Bourdonov, A. S. Kossatchev, and V. V. Kuliamin

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

Received May 5, 2003

Abstract—Problems of testing program systems modeled by deterministic finite automata are considered. The necessary (and, sometimes, sufficient) component of such testing is a traversal of the graph of the automaton state transitions. The main attention is given to the so-called irredundant traversal algorithms (algorithms for traversing unknown graphs, or on-line algorithms), which do not require an a priori knowledge of the total graph structure.

1. INTRODUCTION

The graph traversal problem, i.e., the construction of a covering path passing through all graph edges, is well known. The special case of this problem is the Chinese postman problem [1–3], where the covering path must have a minimal length or minimal weight for a graph with given edge weights. In the case of a directed graph, the problem is more complicated, since the path must pass each directed edge (arc) only in its direction.

In the majority of studies, the graph is assumed to be a priori known in an explicit form [4, 5]. 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 much more complicated [6–8]. This is the well-known problem of traversing a maze 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 pass 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 passed. A directed graph corresponds to a maze in which each passage has doors at both ends: the input door can be opened only from the outside (from a junction), and the output door can be opened only from the passage, which permits moving along each passage only in one direction.

An algorithm that works on such an a priori unknown graph is further referred to as an irredundant algorithm. These algorithms are also referred to as on-line algorithms. The particular kind of such an algorithm, corresponding to the case of the internal memory limited to a finite number of states, is a robot (finite automaton) on a graph, which is a kind of Turing machine [7, 10–16]. 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. (To ensure the finiteness of the robot automaton, the outdegree of the graph vertices is to be bounded from above. This restriction can be removed if each arc is made to correspond to a cell, and the cells corresponding to the arcs with a common beginning point are combined in a loop.)

The problem of traversing directed graphs is currently very topical in connection with the problem of testing finite automata (more precisely, objects considered as finite automata). The latter problem is discussed in many publications (see, e.g., extensive surveys in the works [17, 18]). An automaton is determined by a set of its states and transitions. The transition is a quadruple (v, x, y, v'), where v is a pre-state, x is a stimulus, y is a reaction, and v' is a post-state. Usually, an automaton is given by a graph of its state transitions, with the vertices and arcs of the graph being the states and transitions, respectively. An automaton (state transition graph) is fully specified if, in each state v, every stimulus x is admissible, i.e., there exists at least one transition of the form (v, x, y, v'). Otherwise, the automaton is partially specified. An automaton (state transition graph) is said to be deterministic if a pre-state and a stimulus uniquely determine the reaction and the post-state. In this paper, we confine our consideration only to deterministic partially specified automata. The nondeterministic case will be considered in a future paper.

If the state transition graph is known, then whether it satisfies certain requirements is of interest. In this case, the problems are solved analytically, and no testing is needed. The testing is required when the state transition graph is not known. Considering the automaton as a black box and feeding stimuli to its input, we obtain information about the resulting transition, i.e.,

0361-7688/03/2905-0245$25.00

generally, about the automaton reaction and post-state. The goal of the testing is to check whether the automaton satisfies certain a priori given specification requirements. This is a *conformance testing* in a general sense. In the general case, the specification does not mean to check all transitions of the automaton. If, for example, we want to know whether the number of the automaton states is not less than a given number, then the testing is terminated as soon as we make sure of this (the remaining unchecked transitions are of no interest). However, such a case is sort of an exception rather than a rule. Usually, the complete automaton functionality is of interest, and we need to test all automaton transitions. The testing of this kind is based on the following assumptions.

State change. An automaton state changes only in response to a test action (a stimulus on the automaton input). On the one hand, this implies that the automaton is subject to only test actions, and nothing interferes with testing. (To be more precise, external influences do not change the automaton functionality. Information about testing in the presence of disturbances can be found in the works [19, 20] on the "gray-box" and "semicontrollable" testing.) On the other hand, this means that the test cannot change the automaton state other than by inputting a stimulus to the automaton. If the set of states were known (i.e., only transitions are unknown) and the states could arbitrarily be changed by means of a direct record or special (nontested) operation, the problem of searching all automaton transitions would be trivial. An a priori knowledge of the set of the implementation states is a strong requirement. Usually, the information about it is obtained step by step in the course of the testing; i.e., we learn about a state only when the automaton turns into this state.

Admissibility of stimuli. At every moment, we can learn what stimuli are admissible. Clearly, without this assumption, no testing is possible. Note that, in many studies (e.g., [17]), this assumption is replaced by the assumption that the automaton is fully specified, i.e., all stimuli from the automaton alphabet of stimuli are admissible in all states. It is worth noting that, in practice, we are concerned only about the stimuli that are used in the testing, i.e., those determined in the model. The fact that any other stimuli are admissible does not affect the testing. This actually means that, for a given implementation R and model M, a subautomaton $R(M) \subseteq R$ is tested. The latter is determined by the transitions due to the model stimuli and the states reachable from the initial state by means of the above transitions.

Observability of reactions. The reaction of the automaton to a stimulus is to be observable. In fact, when testing an automaton, we check just this reaction. Otherwise, the automaton would perform some transitions and we would not be able to learn whether they are correct. On the other hand, we can judge whether the transitions are correct by the post-states, under the condition that the latter are observable.

A separate question is that of the **observability of the automaton states.** If, at any time, we can learn the automaton state by reading it or by means of the special operation *status message* (assuming that this operation does not change the state) [17], then such a testing is called an open-state testing. Otherwise, we speak of a hidden-state testing.

The special case of the conformance testing (in the narrow sense) is the case where the specification is a model automaton explicitly given by its state transition graph and it is required to check whether the automaton being tested is equivalent to the model one [17]. Two states (of one automaton or two different automata) are equivalent if any sequence of stimuli admissible starting from one state is admissible starting from the other state and results in the same sequence of reactions in both cases. Two automata are equivalent if, for each state of one automaton, there exists an equivalent state of the other. A model automaton describes, thus, a class of implementation automata that are equivalent to it.

If a model automaton is available, the open-state testing reduces to traversing the model graph [17] in the course of which each model transition is followed by feeding the same stimulus to the input of the implementation automaton and checking whether the reaction and post-state of the implementation automaton are the same as in the case of the model transition. Note that, in such a statement of the problem, the irredundancy of the traversal algorithm is not required since the model graph is known. If the information about implementation states is not available (a hidden-state testing), one needs to introduce special restrictions on the implementation and model and take advantage of more complicated checking sequence methods [17]. In this case, the traversal of the model graph is not sufficient (but, clearly, necessary) for solving the problem, and the irredundancy of the traversal algorithm is not required.

Unfortunately, in practice, the specifications do not explicitly describe the model automaton, so that we arrive at the problem of finding its explicit form. Moreover, the implementation is to be equivalent to a certain subautomaton of the model automaton, which is a priori unknown, rather than to the complete model automaton.

The case of implicit specifications given in the form of preconditions and postconditions is most often met in practice. A precondition—a predicate over a pre-state and stimulus—determines the admissibility of stimuli in the states, and a postcondition—a predicate over a pre-state, stimulus, reaction, and post-state—determines possible transitions. The finding of an explicit automaton form from such specifications requires solving a system of general-form equations, which, generally, has no satisfactory solution. However, this is not the only difficulty.

The specification is assumed to describe *possible* rather than obligatory automaton transitions. If a specification admits several transitions from a given pre-

state in response to a given stimulus, this does not necessarily imply indeterminism of the implementation. It is assumed that the implementation has at least one (not necessarily all) such a transition; the deterministic implementation must have exactly one transition. Actually, this implies that a model automaton is associated with a family of classes of equivalent implementation automata rather than with one class. An implementation automaton, or, more precisely (as noted above), its subautomaton $R(M) \subseteq R$ determined by the model stimuli, is equivalent to a certain subautomaton $M(R)$ of the specification automaton M. In each state of the subautomaton $M(R)$, all stimuli admissible in this state of M are admissible; however, not all transitions from this state by a given stimulus available in M are available in $M(R)$. (In this case, R and $M(R)$ are said to be *quasi-equivalent*, and R is said to be a *reduction* of M [18].) Clearly, in this case, the finding of an explicit form of the specification automaton M may be not necessary, since we need only its subautomaton $M(R)$, the numbers of states and transitions in which may be considerably less than those in M. In addition, the subautomaton $M(R)$ itself is a priori unknown, since it is determined not only by M but also by R.

It should be noted also that a nondeterministic specification automaton M can be used for testing deterministic implementation automata R [21]. In this case, the explicit subautomaton $M(R)$ is also deterministic. This fact is important, because the testing of deterministic automata is considerably simpler than that of nondeterministic ones.

Thus, the need in an irredundant algorithm for traversing the state graphs appears to be quite natural. The open-state testing is, in fact, reduced to such an algorithm; in the case of the hidden-state testing, such an algorithm is necessary (but not sufficient).

The paper is organized as follows. In Section 2, formal definitions related to the graphs and traversal algorithms are introduced. In Section 3, the problems of existence of a covering path through the graph and its length are discussed. Irredundant traversal algorithms are suggested in Section 4. The applications of these algorithms to the testing problems are discussed in Section 5.

2. GRAPHS AND ALGORITHMS FOR TRAVERSING THEM

A directed graph (further, simply *graph*) G is an object described by three following sets: a set of vertices VG, a set of stimuli XG, and a set of arcs $EG \subseteq VG \times XG \times VG$.

A stimulus x is *admissible* at a vertex a if there exists an arc (a, x, b) in the graph. For an arc (a, x, b), the vertices a and b are referred to as the beginning and the endpoint of the arc, respectively, and the stimulus x is called an arc *coloring*. If the arc stimulus is not important, we write simply (a, b) instead of (a, x, b).

Remark. When testing, a graph is considered to be a graph of automaton state transitions. However, when studying traversal algorithms, we do not color graph arcs by reactions, because, when passing an arc, it suffice for the algorithm to be able to determine the endpoint (post-state) of the arc that begins at a given vertex (pre-state) and is colored by the given stimulus. In the case of the open-state testing, the post-state is determined directly. In the case of the hidden-state testing, the post-state can sometimes be determined by the reaction. In both cases, a method for determining the post-states is considered to be external with respect to the traversal algorithm itself. Reactions in the course of the testing are discussed in more detail in Section 5.

A graph is said to be finite if the sets of its vertices and arcs are finite. The numbers of vertices and arcs of a finite graph are denoted by n and k, respectively.

A graph is said to be *deterministic* if the endpoint of an arc is uniquely determined by its beginning point and by the stimulus admissible at this point; i.e., for arcs (a, x, b) and (a', x', b'), it follows from $a = a'$ and $x = x'$ that $b = b'$. In this paper, we consider only deterministic finite graphs.

Arcs (a, b) and (a', sb') are said to be adjacent if the endpoint of the first arc coincides with the beginning of the second arc, $b = a'$. A path P of length n in the graph G is a sequence of n adjacent arcs; i.e., for $i = 1, \ldots, n - 1$, the arcs $P[i]$ and $P[i + 1]$ are adjacent. The beginning point a of the first arc of a path is called the *beginning* of the path; the endpoint of the last arc of the path is called its *end*; and the path itself, an $[a, b]$-path. A path consisting of an empty sequence of arcs has zero length; the beginning and the end of such a path coincide. A path is referred to as a *covering* path if it contains all arcs of the graph.

A *graph traversal algorithm* is an algorithm that constructs a path on the graph. Formally, such an algorithm can be defined as a special-purpose abstract-state machine (the Gurevich machine, ASM [22]), in which external operations are partially specified by the graph on which the algorithm operates and by the current vertex. For our purposes, it is sufficient to know that the algorithm has two special-purpose external operations: *status*(), which returns the identifier of the current vertex, and *call*(x), which implements the transition from the current vertex a along the arc with the stimulus x. For a deterministic graph, such an arc (a, x, b) is unique (a unique vertex b). The precondition of the operation *call*(x) is the admissibility of the stimulus x at the current vertex a. The path constructed by the algorithm is a sequence of arcs obtained by means of successive calls of the operation *call*. It should be noted that any external operation (to say nothing of an internal one) does not change the graph. The only operation that can change the current vertex is the operation *call*.

An *irredundant algorithm* is a graph traversal algorithm that takes into account only the traversed part of the graph and the admissibility of stimuli at the current

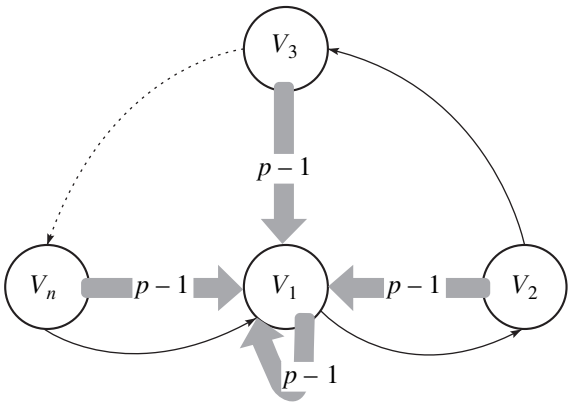

Fig. 1.

vertex. The algorithm determines whether a stimulus is admissible by means of a special-purpose external operation *next*(), which returns a stimulus selected in unspecified way among the stimuli that are admissible at the current vertex and have not been selected yet (iterating, thus, stimuli at the vertex). If all stimuli admissible at the current vertex have already been used, the operation *next*() returns the empty symbol ε.

A *free* algorithm is an irredundant algorithm that learns whether a stimulus that has not been tested yet at the current vertex *a* is admissible when passing an arc colored by this stimulus rather than in advance. In other words, the free algorithm uses the combined external operation *nextcall*(): $x = next()$; *if* $x \neq \varepsilon$ *then call*(x); *return x else return* ε *end* when traversing first time any arc with the beginning at the current vertex that has not been traversed yet. This operation chooses in an unspecified way a stimulus x that has not been tested yet at the current vertex a and makes the algorithm pass along the arc (a, x, b). If all stimuli have already been tested at the current vertex, the empty symbol ε is returned. For the secondary passage along the arc (a, x, b), the operation *call*(x) continues to be used at the moment when a becomes the current vertex.

Any algorithm is designed for solving one or another problem; the problem considered in this work is that of the covering path construction. Of interest are algorithms that stop after a finite number of steps. When the algorithm stops, it can provide us with the information about whether the traversal has been completed, i.e., whether the constructed path is the covering path. It is possible that the path constructed is the traversal, but the algorithm "does not know" about this. The opposite situation is the case where the traversal has not been completed and the algorithm "knows" that there does not exist a covering path at all in this graph. The information of this kind reported by the algorithm when it stops is referred to as a *v*erdict of the algorithm. The verdict is said to be *authentic* if the information contained in it is true.

The algorithm operation depends, generally, on the external operations; for irredundant algorithms, this is operation *next*, which is not uniquely determined. An algorithm is said to perform a *guaranteed* traversal of a given graph with a given initial vertex if it traverses the graph (constructs a covering path) for any admissible results of all external operations (for irredundant algorithms, independent of the stimulus iteration at the vertices (*next*)).

3. GRAPH TRAVERSAL

3.1. Strongly Connected Graphs

A vertex b is *reachable* from a vertex a if there exists an [a, b]-path. A graph is *strongly connected* if any its vertex is reachable from any vertex. A *simple path* is a path that does not contain more than one occurrence of any vertex.

Theorem 3.1. (1) For any strongly connected graph and any pair of its vertices a and b, there always exists a covering [a, b]-path of length $O(nk)$.

(2) For any n and k, there exists a strongly connected graph with n vertices and $k' \geq k$ arcs such that any covering path of the graph has length $\Omega(nk')$.

Proof. (1) Let us introduce an arbitrary linear order on the graph arcs, such that the first arc begins at the vertex a and the last arc ends at the vertex b. For two successive arcs (v_i, v_i') and (v_{i+1}, v_{i+1}') in a strongly connected graph, there always exists a path from the end v_i' of the ith arc to the beginning v_{i+1} of the $(i+1)$th arc. Removing all loops from this path, we obtain a simple path P_i from v_i' to v_{i+1}. The desired covering path is the concatenation of arcs and simple paths: $(v_1, v_1') \wedge P_1 \wedge \ldots \wedge (v_i, v_i') \wedge P_i \wedge \ldots \wedge P_{k-1} \wedge (v_k, v_k')$, where $v_1 = a$ and $v_k = b$. Since the simple path length in the graph does not exceed $n-1$, the length of the covering path is not greater than $k + (n-1)(k-1) = n(k-1) + 1$, i.e., $O(n$k).

(2) An example of the graph in question is depicted in Fig. 1, where $p = k'/n$ is the outdegree of each vertex. A covering path of this graph can be represented as a concatenation of paths $P_1, \ldots, P_t$, where all paths $P_1, \ldots, P_{t-1}$ have the same last arc (v_i, v_1) $(i > 1)$ and all paths $P_2, \ldots, P_{t-1}$ begin at v_1. Each path in the sequence $P_2, \ldots, P_{t-1}$ is terminated by the arc (v_i, v_1) and has length i; the number of these paths is equal to $p-1$. Therefore, assuming that the path P_1 is terminated by the arc (v_j, v_1) and its length is not less than 1 and not taking into account the last path P_t, we obtain the lower bound of the length of the covering path,

$$(p-1) + 2(p-1) + \ldots + n(p-1) - j + 1$$
$$= (p-1)n(n-1)/2 - j + 1$$
$$\geq (p-1)n(n-1)/2 - n + 1 = L.$$

It suffice to prove that $L \geq Cpn^2 = Ck'n$ for some constant $C > 0$ and any $n > 0$. It is easy to show that the above relation holds, for example, for $C = 1/3$ and $p \geq 3$, and, thus, we obtain $k' = \max\{k, 3n\}$.

Note that, in the graph depicted in Fig 1, the outdegrees of all vertices are identical. This has been done in order that not to impose lower bounds on the number of stimuli in addition to the obligatory bound k'/n. Without this requirement, the example can be simplified by replacing all arcs leading from v_i to v_1 by the arcs leading from v_n to v_1, which requires $k' - n + 1$ stimuli (Fig. 2).

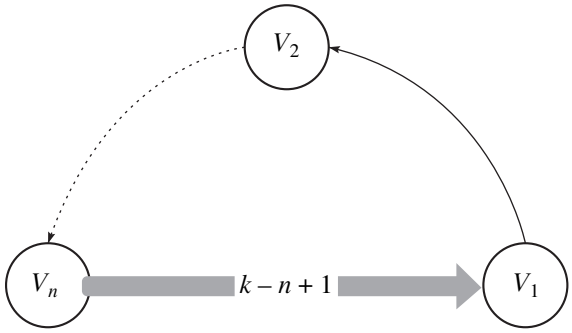

Fig. 2.

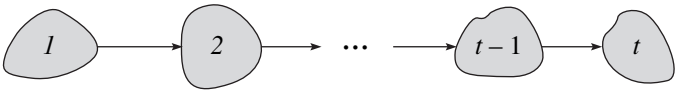

Fig. 3.

3.2. Reachable Graphs

Since mutual reachability of vertices is an equivalence relation, the graph G is partitioned, generally, into strongly connected components, on the set of which the reachability is a partial order relation. A component is a subgraph of the graph G the set of vertices of which is an equivalence class and the arcs are all arcs of the graph G the beginning and end points of which belong to this class. The component containing the vertex a is denoted by $K(a)$. A *connecting* arc is an arc the beginning and endpoint of which belong to different components of the graph G.

A *factor graph* of a graph G with respect to the mutual reachability relation is a graph $F(G)$ the vertices of which are the strongly connected components of G and the arc (A, x, B), where $A \neq B$ are components of the graph G, exists if and only if there exists a connecting arc (a, x, b) in G, where $a \in VA$ and $b \in VB$. The vertices and arcs of the factor graph are referred to as factor vertices and factor arcs, respectively.

A *reachable graph* is a graph all vertices of which are reachable from a given initial vertex. In what follows, we consider only reachable graphs, since, clearly, only these graphs can be traversed. A graph is said to be acyclic if it does not contain cycles. A source is a vertex the has no incoming arcs, and a sink is a vertex without outgoing arcs.

Theorem 3.2. A graph G with an initial vertex v_0 is reachable if and only if its factor graph $F(G)$ is an acyclic graph with one source $K(v_0)$.

Proof. *Necessity.* It is evident that the factor graph $F(G)$ is acyclic. In the reachable graph G, there exists a $[v_0, v]$-path for any vertex v. Keeping only the connecting arcs in the graph and replacing them by the corresponding factor arcs, we obtain a $[K(v_0), K(v)]$-path on the factor graph; i.e., $F(G)$ is also an reachable graph and, hence, has only one source $K(v_0)$.

Sufficiency. In the acyclic factor graph $F(G)$ with one source $K(v_0)$, for each vertex, there exists a $[K(v_0), K(v)]$-path, which is a sequence of the factor arcs corresponding to the connecting arcs (a_i, b_i), $i = 1, \ldots, t$. Introducing the notation $b_0 = v_0$ and $a_{t+1} = v$, we find that, for $i = 0, \ldots, t$, the vertices b_i and a_{i+1} belong to one component, and, hence, there exists a $[b_i, a_{i+1}]$-path P_i in the graph G. The $[v_0, v]$-path in the graph G is constructed as the concatenation $P_0 \wedge (a_1, b_1) \wedge \ldots \wedge P_{t-1} \wedge (a_t, b_t) \wedge P_t$.

3.3. Graphs of the First Kind

A graph of the *first kind* is a graph with a linear reachability order of the components in which each (but the last) component has only one outgoing connecting arc that leads to the next component. By default, the initial vertex v_0 belongs to the first component. In other words, the factor graph of such a graph consists of one acyclic path with the beginning at the component of the initial vertex $K(v_0)$ (Fig. 3).

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

Theorem 3.3. (1) A traversed graph of a path is a graph of the first kind the first and last components of which contain the beginning and the end of the path, respectively.

(2) A covering $[a, b]$-path exists only for graphs of the first kind in which the vertices a and b belong to the first and last components, respectively; the minimum path length is $O(nk)$.

(3) For any n and k, there exists a graph of the first kind with n vertices and $k' \geq k$ arcs for which any covering path has length $\Omega(nk')$.

(4) A covering path from any initial vertex v_0 exists only if the graph is strongly connected.

Proof. Assertion (1) immediately follows from the fact that all vertices of a traversed graph are linearly ordered by the path in the order they have been reached. (2) It follows from assertion (1) that a covering path exists only for graphs of the first kind. Conversely, introducing the notation $b_0 = a$ and $a_t = b$ for a graph of the first kind with the connecting arcs (a_i, b_i), $i = 1, \ldots, t - 1$, we construct a covering $[b_{i-1}, a_i]$-path P_i of the ith

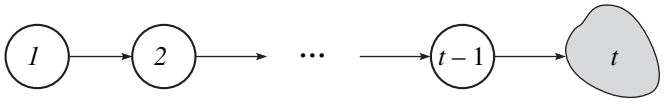

Fig. 4.

component for $i = 1, \ldots, t$ and a covering $[a, b]$-path of the graph as the concatenation $P_1 \wedge (a_1, b_1) \wedge \ldots \wedge P_{t-1} \wedge (a_t, b_t) \wedge P_t$. The length of this covering path does not exceed $t - 1 + \Sigma\{O(n_i k_i) | i = 1, \ldots, t\}$, where n_i and k_i are the numbers of vertices and arcs of the ith component, respectively, i.e., has the order $O(nk)$. Assertion (3) has been proved in Theorem 3.1 for strongly connected graphs ($t = 1$). Assertion (4) immediately follows from assertion (2).

3.4. Coverage of Reachable Graphs

A set of paths beginning at the initial vertex v_0 is called a *coverage* of the graph if each vertex and each arc of the graph belong to at least one of the paths. The coverage length is the sum of lengths of its paths.

Theorem 3.4. A coverage exists if and only if the graph is reachable; its minimum length is equal to $O(nk)$; for any n and k, there exists a reachability graph with n vertices and $k' \geq k$ arcs any coverage of which has length $\Omega(nk')$.

Proof. The assertion of the theorem on the existence of a coverage immediately follows from the definition of a reachable graph G with an initial vertex v_0. Indeed, for each vertex a, there exists a $[v_0, a]$-path $P(a)$, and, for any arc (a, b), there is a path $P(a) \wedge (a, b)$ passing through this arc.

To estimate the coverage length, we consider the factor graph $F(G)$, which, by Theorem 3.2, is an acyclic graph with one source $K(v_0)$. Let t be the number of the components and k_0 be the number of the connecting arcs in the graph G. Let us separate an output directed spanning tree (maximal tree) in $F(G)$ the root of which is the source component $K(v_0)$. It has $t - 1$ arcs, and the remaining $k_0 - (t - 1)$ connecting arcs are chords. To construct a coverage of $F(G)$, it is sufficient to take (a) all factor paths that lead from the root $K(v_0)$ to the leaf components that do not have outgoing chords and, additionally, (b) all factor paths that lead from the root to the initial components of all chords and pass then along these chords. The number of the factor paths of the form (a) does not exceed t, and the number of the factor paths of the form (b) is not greater than $k_0 - (t - 1)$. Thus, the total number of all factor paths is not greater than $k_0 + 1$.

Any above factor path F may be viewed as an alternating sequence of the components and connecting arcs. Let us replace each occurrence of a component K in the factor path F by the path $P(K, F)$ in the graph G that begins at the endpoint of the connecting arc of the factor path F that enters K (for the first component $K(v_0)$, at the vertex v_0) and ends at the beginning point of the connecting arc of the factor path F that goes out of K (for the last component, at any of its vertices). As a result, we obtain a path $P(F)$ on the graph G in the form of an alternating concatenation of the paths $P(K, F)$ and the connecting arcs. For each component K, the path $P(K, F)$ is taken as follows. In one of the paths of $P(F)$ that passes through K, it is a covering path of the component K; in all other paths, it is either a simple path (if K is not the last component in F) or a path of zero length (if K is the last component in F). Clearly, this set of paths $P(F)$ is a coverage of the graph G. If each path in $P(F)$ contained only simple paths in the components rather than traversals, this path itself would be a simple path, or a simple path extended by one arc (a chord of the factor graph); hence, its length would not exceed n, and the sum of lengths would not exceed $(k_0 + 1)n$. Since, for each ith component, only one path $P(F)$ contains the covering path of this component, the coverage length does not exceed $(k_0 + 1)n + \Sigma\{O(n_i k_i) | i = 1, \ldots, t\}$, where n_i and k_i are the numbers of vertices and arcs in the ith component, respectively, i.e., has the order $O(nk)$.

The estimate $\Omega(nk')$ is obtained for the graphs depicted in Figs. 1 and 2 with the initial vertex v_1.

4. TRAVERSAL ALGORITHMS

4.1. Graphs of the Second Kind and Free Algorithms

For a path in a graph, a vertex is said to be *completely traversed* if all outgoing arcs of this vertex are traversed in this path.

A graph of the *second kind* is a graph of the first kind in which all components (but, perhaps, the last one) consist of one vertex and do not contain arcs other than one connecting arc leading to the next component (Fig. 4).

Theorem 4.1. (1) A traversal of a graph by a free algorithm starting from an initial vertex v_0 belonging to the first component and ending at a vertex belonging to the last component is guaranteed only if the graph is a graph of the second kind.

(2) There exists a free algorithm A_1 that stops on any graph after passing a path of length $O(nk)$ and traverses with guarantee any graph of the second kind with the initial vertex v_0 belonging to the first component.

Proof. (1) By Theorem 3.3, only a graph of the first kind can be traversed. If the graph is not a graph of the second kind, then there exists a component (not the last one) of the graph that either consists of more than one vertex or its only vertex has some outgoing arcs in addition to the connecting arcs. In the former case, by virtue of the strong connectivity of the component, there is a path from the beginning of the connecting arc to some other vertex of the component; hence, in addition to the connecting arc (a, x, b), there is another arc (a, x', b') going from the beginning of the connecting arc. In the latter case, the existence of two such arcs is explicitly postulated. When the algorithm deals with the vertex a

for the first time, none of the stimuli at this vertex are tested; therefore, the free algorithm must invoke the operation *nextcall*. Since the algorithm must guarantee the traversal of the graph independent of the result of this operation, we may assume that the operation chooses the stimulus x. Then, we pass the connecting arc (a, x, b) and occur in the next component. Since, we cannot return to the vertex a any more, the arc (a, x', b') remains untraversed.

(2) In the course of the algorithm A_1 operation, we will store the description of the traversed graph by registering all traversed arcs. In addition, we will mark the vertices at which the operation *nextcall* returns the empty symbol ε; clearly, such vertices have been completely traversed. Note that there may be a situation when a vertex has been completely traversed, but we do not know about this yet, and the vertex is not marked.

The algorithm step consists of the following actions:

(1) Execute the operation *nextcall* registering the arcs traversed (the arc stimulus is returned by the operation *nextcall*, and its endpoint is determined by means of the operation *status*) until *nextcall* returns the empty symbol ε. In the latter case, the current vertex is marked.

(2) If all traversed vertices are marked (i.e., are completely traversed), the algorithm stops.

(3) Otherwise, a simple path in the traversed graph from the current vertex to some unmarked vertex is sought. If such a path exists, it is traversed by means of the operation *call*, and the algorithm step terminates.

(4) If such a path does not exist, the algorithm stops.

Algorithm termination. Each above item requires a finite time since the graph has a finite number of arcs (item 1 requires a finite time) and a finite number of simple paths (item 3 requires a finite time). For one step consisting of items 1–3, the algorithm passes at least one untraversed arc and/or marks at least one unmarked vertex. Then, it follows that the algorithm A_1 stops in a finite time on any graph.

Path length. The traversed path can be represented as a concatenation of the first occurrences of the arcs (item 1) and the simple paths connecting them (item 3). Hence, the path length has the order of $O(nk)$.

Guaranteed traversal. In a graph of the second kind, each (not the last) component consists of one vertex that has one outgoing arc leading to the next component. Therefore, an algorithm that starts its operation at the vertex belonging to the first component occurs after the execution of item 1 in the last component and, hence, stops at a certain vertex b of this component. Let there be an arc (c, d) that has not been traversed by the moment when the algorithm stops. Then, its beginning c is not marked. Clearly, it belongs to the last component, and there exists a path from b to c. Removing all loops from the path, we obtain a simple path from the current vertex b to the unmarked vertex c. If this simple path contains only traversed arcs, then the algorithm would not stop (item 3). If this simple path contains untraversed arcs, then we consider the initial fragment of this simple path until the first untraversed arc (c', d'). The vertex c' is not marked, and there is a simple path in the traversed graph from the current vertex to this vertex. Hence, the algorithm would not stop (item 3). Thus, we arrive at the contradiction, which implies that, by the moment when the algorithm stops, all arcs are traversed and the traversal of the graph has been completed.

Different algorithms based on the strategy employed in the algorithm $\mathbb{A}_1$ differ by the ways they select an unmarked vertex v' at the current marked vertex v in item 3 [8]. The algorithms based on the traversal of the graph spanning tree select either the furthest vertex from the root (depth-first search) or the closest to the root vertex (breadth-first search) v' reachable from v. A "greedy" algorithm selects the vertex v' that is nearest to v in terms of the length of the $[v, v']$-path.

Now, we study the question of what authentic verdicts can be returned by a free traversal algorithm.

The free algorithm can learn that all arcs originating from the current vertex have already been traversed only having received the empty stimulus in response to the operation *nextcall*. Like in the algorithm $\mathbb{A}_1$, such a vertex is called marked. If there are no unmarked vertices when the algorithm stops, the algorithm can return the authentic verdict "the traversal has been completed" (recall that we consider only reachable graphs). This may happen only if the graph is strongly connected, since the passage of a connecting arc makes returning to its initial vertex v_0 impossible, and, hence, this vertex remains unmarked. Therefore, if all traversed arcs are marked, even a stronger verdict—"the guaranteed traversal has been completed"—is authentic. The algorithm $\mathbb{A}_1$ can return this verdict when it occurs in item 2. If there are unmarked vertices at the stopping moment (item 4 of the algorithm $\mathbb{A}_1$), one of the two following authentic verdicts can be returned: (i) "it is unknown whether a traversal has been completed; however, if it has, the traversal is guaranteed" if the traversed graph is of the second kind or (2) "it is unknown whether a traversal has been completed; however, if it has, the traversal is not guaranteed" otherwise.

4.2. Irredundant Algorithms

Irredundant algorithms differ from the free ones in that they can obtain advanced information about the stimuli of the untraversed arcs before they are traversed by means of the operation *next*. This allows them to identify the marked vertices and completely traversed vertices and, hence, to return stronger verdicts.

Theorem 4.2. There exists an irredundant algorithm $\mathbb{A}_2$ that stops on any graph after passing a path of length $O(nk)$ and returns one of the three following authentic verdicts: (1) "a guaranteed traversal has been completed" for all graphs of the second kind, or (2) "a tra-

versal has been completed but is not guaranteed" for certain results of external operations for graphs of the first (but not second) kind, or (3) "a traversal has not been completed" for certain results of external operations for graphs of the first (but not second) kind and for all graphs of the first kind.

Proof. The algorithm $\mathbb{A}_2$ is a modification of the algorithm $\mathbb{A}_1$; it constructs the same path. Instead of the operation *nextcall*, the "advanced" operation *next* is used. The result of this operation is stored in the current vertex (when a vertex is visited for the first time, *next* is performed twice at a run if, of course, the vertex has at least one outgoing vertex). Then, when traversing an unmarked arc, we apply the operation *call*(*x*), where *x* is the stimulus produced at the current vertex by the next-to-last operation *next*. Owing to this, the algorithm marks the vertex immediately before it leaves it by the last untraversed arc. Thus, the notions of the marked vertex and completely traversed vertex become identical. When stopping in item 2 (all vertices are marked), the algorithm analyzes the traversed graph. If it is a graph of the second kind, the algorithm returns the verdict "a guaranteed traversal has been completed"; otherwise, "a traversal has been completed, but it is not guaranteed." In item 4, the algorithm returns the verdict "a traversal has not been completed."

4.3. Predicate of Connecting Arcs

The algorithm can traverse with guarantee graphs of the first kind if it receives somehow information about the connecting arcs. Let a predicate $\pi(x)$ of stimulus, which is further referred to as the *predicate of the connecting arcs*, be given at each vertex of the graph. The predicate is said to be *authentic* if it is true on the stimuli of the connecting arcs and is not true on others. The algorithm with the external operations *next*, *call*, and π is, of course, not irredundant; however, in a sense, it is a "minimally redundant" algorithm.

Theorem 4.3. There exists an algorithm $\mathbb{A}_3$ with a predicate of the connecting arcs π that stops on any graph after passing a path of length $O(nk)$ and traverses with guarantee graphs of the second kind and those of the first kind with authentic predicates.

Proof. The algorithm $\mathbb{A}_3$ differs from $\mathbb{A}_2$ in that it passes first only nonconnecting arcs that go out of the traversed vertices and stores stimuli of the connecting arcs in their beginning vertices. When there are no untraversed nonconnecting arcs any more, the algorithm looks for a simple path in the traversed graph to the beginning of one of the stored untraversed connecting arcs. If such a path is found, the algorithm passes it and the connecting arc and starts to look for untraversed nonconnecting arcs again. Whether an arc is connecting is determined by the predicate π. Proofs of the assertions about the algorithm termination and traversal length are trivial.

It is easy to show that, when the algorithm $\mathbb{A}_3$ stops, it can analyze the traversed graph and returns one of the following authentic verdicts:

1. "A traversal has been completed" if the traversed vertices do not have untraversed outgoing arcs.

(a) + "The traversal is guaranteed, and the predicate is authentic" if the predicate in the traversed graph of the first kind is true on, and only on, the connecting arcs.

(b) + "The traversal is guaranteed, but the predicate is not authentic" if the predicate in the traversed graph of the first kind is true on some arcs of the last component and/or false on some connecting arcs the beginnings of which have no other outgoing arcs.

(c) + "The traversal is not guaranteed, and the predicate is not authentic" in all other cases.

2. "A traversal has not been completed" if at least one traversed vertex has untraversed outgoing arcs.

(a) + "Either the predicate is authentic and, then, the original graph is a graph of the first kind or the predicate is not authentic and, then, it is not known whether the original graph is a graph of the first kind" if the predicate is true on, and only on, the connecting and untraversed arcs.

(b) + "The predicate is not authentic, and it is not known whether the original graph is a graph of the first kind" in all other cases.

The predicate π is formally defined on the triples (graph, vertex, and stimulus). A predicate is said to be *irredundant* if it does not depend on the graph. More precisely, the dependence of a predicate on the graph is reduced to the dependence on the set of stimuli admissible at the vertex; i.e., formally, the predicate is defined on the triples (vertex, set of admissible stimuli at the vertex, and stimulus). Considering the irredundant predicate as an internal (rather than external) operation of the algorithm and modifying accordingly the algorithm $\mathbb{A}_3$, we obtain the irredundant algorithm (denote it as $\mathbb{A}_4$) that traverses with guarantee all graphs of the second kind and those of the first kind for which the predicate is authentic.

On the other hand, an irredundant predicate, clearly, cannot be authentic on all graphs with a given initial vertex v_0 isomorphic up to the coloring of arcs by stimuli if they are not graphs of the second kind.

Theorem 4.4. There does not exist an irredundant algorithm that traverses a graph *G* of the first (but not the second) kind and all graphs that differ from *G* only by the coloring of arcs by stimuli.

The *proof* is similar to that of assertion (1) of Theorem 4.1. If *G* is not a graph of the second kind, it contains a connecting arc (*a*, *x*, *b*) the beginning of which has another outgoing arc (*a*, *x*', *b*'). Consider the moment when the algorithm implements the first passage of the second arc (invokes the operation *call*(*x*') at the vertex *a*). If the algorithm traverses the graph, then, at this moment, the first arc has not been traversed yet.

If the algorithm is irredundant, then the information about the graph available at the moment is the same both for the graph G and the graph G' that differs from G by the permutation of the coloring stimuli for these two arcs, (a, x', b) and (a, x, b'). Therefore, in the graph G', the algorithm will go along the arc (a, x', b) when the arc (a, x, b') has not been traversed yet. However, in this graph, the connecting arc is the arc (a, x', b); i.e., the traversal will not be completed.

4.4. A Free Algorithm Optimal in Terms of Time and Memory

Until now, we were interested only in the traversal length (the number of the operations *call*) rather than in the time required for the algorithm operation (the total number of elementary, both internal and external, operations) and in the required memory.

Theorem 4.5. There exists a free algorithm $\mathbb{A}_5$ that stops on any graph and traverses with guarantee all graphs of the second kind with an initial vertex v_0 belonging to the first component. The estimates of the algorithm operation time and the required memory are, respectively, $O(nk)$ and $O(n(\log_2 n + I + X))$ bits, where I and X are the sizes of the vertex identifier and stimulus, respectively, in bits. The verdicts returned by the algorithm are authentic.

Proof. The idea of the algorithm is to construct in the traversed graph a *forward path* (consists of *forward arcs*) and a forest of *back trees* (consists of *back arcs*). According to Theorem 3.3, the traversed graph is a graph of the first kind, the initial vertex of which belongs to the first component and the terminal vertex belongs to the last component. The traversed vertices at which the operation *nextcall* invoked by the algorithm returned the empty symbol ε are marked and, thus, completely traversed. The algorithm operation represents a sequence of steps at the beginning of which the following conditions are fulfilled:

• The forward path is a simple $[v_0, v]$-path all vertices of which are unmarked.

• All other traversed vertices are marked.

• The current vertex is the endpoint of the forward path.

• Each back tree is an input directed spanning tree of a strongly connected component of the traversed graph with the root belonging to the forward path. This root will be referred to as the component root (Fig. 5).

Consider one step of the algorithm operation. If the current vertex (the endpoint of the forward path) has one untraversed outgoing arc e, we go along it. If we occur in a new (not traversed earlier) vertex, the *forward* path is extended by the arc e, and the step is completed. Otherwise, we move along the back arcs until the first vertex belonging to the *forward* path (in the extreme case, we reach the root of the back tree) and, then, along the *forward* arcs until the end of the *forward* path, where the step is completed. If the arc e leads to a vertex below the root of the back tree of the last component, the back trees are corrected. The same motion along the back arcs and, further, along the *forward* arcs is performed in the case where the endpoint of the *forward* path has no untraversed outgoing arcs. In this case, however, the endpoint of the *forward* path is marked, and we stop at the previous vertex on the *forward* path and shorten the *forward* path by removing the last arc. If there is no such a vertex (zero-length *forward* path), the algorithm stops.

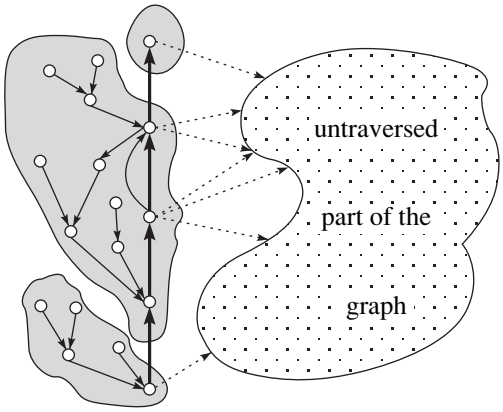

Fig. 5.

Note that the arcs that, in the course of the algorithm $\mathbb{A}_5$ operation, were considered to be *forward* arcs at the beginning of each step, form a *forward tree*, an output directed spanning tree of the traversed graph, when the algorithm stops. In fact, the algorithm traverses this *forward* tree by using the *depth-first search.*

To obtain estimates for the algorithm operation time, let us describe the data structures used by the algorithm.

• To represent the forest of the back trees, the list *Traversed* of all arbitrarily ordered traversed vertices is used. The description of a vertex in the list *Traversed* contains the following data:

• vertex identifier,

• forward reference,

• description of the back arc (the arc of the back tree) originating from this vertex, which contains

• the arc stimulus and

• the reference to the description of the arc endpoint in the list *Traversed* (null reference for the root).

• To represent the *forward* path, the bi-directional list *Forward* of unmarked traversed vertices is used. The description of a vertex in the list *Forward* contains the following data:

• reference to the *Traversed*-description,

• description of the *forward* arc (the arc of the *forward* path) originating from this vertex, which contains

• a *forward* reference to the description of the arc endpoint in the list *Forward* and

• the arc stimulus,

• a backward reference in the list *Forward*,

• the index of the strongly connected component to which the vertex belongs (for this index, the index of its root in the list *Forward* is used).

• Length of the list *Forward*.

A reference to a vertex in the list and the component index do not exceed n and, thus, require $O(\log_2 n)$ bits of memory. Since the number of the descriptions is not greater than the number of vertices n, the total amount of memory required is $O(n(\log_2 n + I + X))$ bits. This is an exact estimate: when the algorithm stops on a graph of the second kind, the list *Traversed* contains n descriptions, which requires $O(n(\log_2 n + I + X))$ bits.

Remark. The external operation *next*, generally, requires memory for each vertex (the last stimulus (or its number) produced or the stimulus to be produced). The additional amount of the required memory is $O(nI)$ bits.

Now, we describe the algorithm step in more detail.

(1) Execute the operation *nextcall*.

(2) If *nextcall* returns a stimulus $x \neq \varepsilon$, determine the endpoint b of the arc (a, x, b) by means of the operation *status* and find it in the list *Traversed*.

(3) If it is not found (the vertex b is new), create descriptions of the vertex b in the lists *Traversed* and *Forward* and extend the *forward* path by the *forward* arc (a, x, b). Since b is the root of the last component $K(b)$, the index of $K(b)$ is set equal to the length of the list *Forward*, i.e., to the index of the *Forward*-description of the vertex b in this list. The step is completed.

(4) If b is an old vertex, it is required to return to the vertex a and to correct, if needed, the back trees and the indices of the components. If b does not belong to the *forward* path, move along the back arcs by means of the operation *call* until the first vertex c on the *forward* path. Compare indices of the components at the vertices a and c. If they are equal each other, move along the *forward* arcs until the end of the forward path by means of the operation *call*. If they are not equal, then a correction is needed. While moving, the index of the component for each vertex *after* c, starting from the first root (inclusively), is set equal to that in c; for the back arc, the forward arc is taken, except for the endpoint of the forward path where the arc (a, x, b) becomes the back arc. Note that, in item 4, the traversed cycle in the graph consists of the back and forward simple paths and its length is always not greater than n. The step is completed.

(5) If *nextcall* returns the empty symbol ε, it is required to go to the next-to-last vertex a^{-1} of the *forward* path and shorten the path by removing the last arc. First, it is checked whether the vertex a is the root of the back tree. If it is, the algorithm stops. In this case, if $a = v_0$ is the initial vertex of the graph, the algorithm returns the verdict "*the guaranteed traversal has been completed*." Otherwise, the lists *Traversed* and *Forward* are analyzed in order to check whether the traversed graph is a graph of the second kind. If it is, the verdict is "it is unknown whether the traversal has been completed; however, if it has, the traversal is guaranteed"; otherwise, the verdict is "*it is unknown whether the traversal has been completed; however, if it has, the traversal is not guaranteed*." If a is not the root of the back tree, then the vertex a^{-1} is found (by taking advantage of the bi-directionality of the list *Forward*), the back arcs are traversed by means of the operation *call* until the first vertex c lying on the forward path, and, further, the forward arcs, until the vertex a^{-1}. The description of the vertex a is deleted from the list *Forward* (and, thus, a is marked), and the forward path is shortened. The step is completed. Note that, in item 5, the traversed path in the graph also consists of the back and forward simple paths.

It is not difficult to see that, if the algorithm does not stop at the given step, all necessary conditions required to begin the next step are fulfilled by the end of the current step.

Let t_i and $call_i$ be the numbers of operations and arc passages (*call* and *nextcall*), respectively, in item i, and let C be a constant depending on a particular program implementation. We have $t_1 \le C$, $call_1 \le 1$, $t_2 \le Cn$, $call_2 = 0$, $t_3 \le C$, $call_3 = 0$, $t_4 \le Cn$, $call_4 \le n$, $t_5 \le Cn$, and $call_5 \le n$. At each step, but the last, either (1) items 1, 2, and 3 or items 1, 2, and 4 are executed and exactly one earlier untraversed arc is traversed or (2) items 1 and 5 are executed and exactly one earlier unmarked vertex is marked. Clearly, the number of steps of form (1) does not exceed the number of arcs k, and that of form (2) does not exceed the number of vertices n. Hence, the algorithm stops after a finite number of elementary operations, and the total number of operations does not exceed $\max[k(t_1 + t_2 + \max(t_3, t_4)), n(t_1 + t_5)] \le k(t_1 + t_2 + t_4) + n(t_1 + t_5) \le k(C + 2Cn) + n(C + Cn)$, i.e., taking into account that $n \le k - 1$, is equal to $O(nk)$. The length of the traversed path does not exceed $\max[k(call_1 + call_2 + \max(call_2, call_4)), n(call_1 + call_5)] \le k(call_1 + call_2 + call_4) + n(call_1 + call_5) \le k(1 + n) + n(1 + n)$, i.e., taking into account that $n \le k - 1$, is equal to $O(nk)$.

It is evident also that the algorithm traverses with guarantee any graph of the second kind and returns an authentic verdict on any graph.

4.5. Modification of the Optimal Algorithm

The algorithm $\mathbb{A}_5$ is a modification of the algorithm suggested in 1971 [11] by one of the authors of this paper for solving a more complicated problem of traversing strongly connected directed graphs by using a robot on the graph. Whereas an irredundant algorithm can be viewed as an ASM on the graph, the robot is a finite automaton on the graph. The estimate of the traversal length for the robot is $O(nk + n^2\log_2 n)$. The second addend is explained by the fact that, in contrast to the algorithm $\mathbb{A}_5$, which, using the bi-directionality of

the list *Forward*, comes to the next-to-last vertex of the forward path (step 5) for one passage following a cyclic path, the robot has to do many passages. Without applying special optimizing expedients, the estimate for the robot is given by $O(nk + n^3)$. In [11], to traverse the forward tree, the robot uses the *breadth-first search*, which reduces the estimate to $O(nk + n^2\log_2 n)$. In 1994, Afek and Gafni [14] suggested a robot that uses the *depth-first search* together with a special optimizing technique, which also results in the estimate $O(nk + n^2\log_2 n)$ for the traversal length. We believe that the combination of the breadth-first search with this optimizing technique will ensure the estimate $O(nk + n^2\log_2\log_2 n)$ for the robot. For the sake of comparison, we note that the well-known Tarry algorithm [9] for traversing undirected graphs is also a free algorithm and can be implemented as a robot with the traversal length $2k$ and the operation time of the order of $O(k)$.

The free algorithm $\mathbb{A}_5$ is associated with the irredundant algorithm $\mathbb{A}_6$ that passes the same path but returns more accurate authentic verdict. Moreover, this algorithm can be optimized, so that, after passing the last chord that begins at the end of the forward path, it returns to the last vertex of the forward path that contains the outgoing arcs that have not been traversed yet (rather than to the next-to-last vertex of the forward path). The algorithm $\mathbb{A}_6$ can be modified to work with the predicate of the connecting arcs (a "minimally irredundant" algorithm) or with the irredundant predicate (irredundant algorithm). In all these modifications, the upper estimates of the operation time and the traversal length are the same. However, for certain classes of graphs, the upper estimate $O(nk)$ of the minimal traversal length can be improved, such that the irredundant algorithm works better than the free one on these graphs. For example, for a graph depicted in Fig. 6, the free algorithm $\mathbb{A}_5$ has the estimate $\Omega(n^2)$, whereas the estimate for the irredundant algorithm $\mathbb{A}_6$ is $\Omega(n)$.

The algorithms discussed can also be used for covering any reachable graphs by using repeated runs of the algorithms and saving the information obtained during the previous runs. In other words, these algorithms can work with the graphs for which the authentic operation *reset* [17] is defined. This operation is used only when needed (in particular, it is not used on strongly connected graphs).

For each arc that can be traversed repeatedly (arcs of the back tree and forward paths), the algorithm $\mathbb{A}_5$ keeps a reference to the description of the arc end in the description of the arc beginning. Therefore, it can work on arbitrary graphs controlling the determinism in the course of its operation. For this purpose, after each operation *call*, the operation *status* is invoked, and the vertex returned by this operation is compared with the vertex stored. If some indeterminism is detected, the algorithm stops and returns the corresponding verdict. Of course, if no signs of indeterminism are found during the algorithm operation, this does not necessarily mean that the graph is deterministic and may simply mean that we were "lucky" this time.

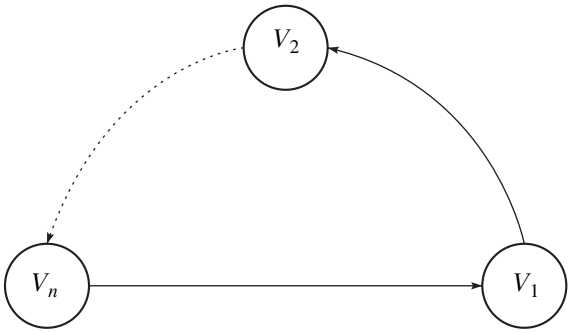

Fig. 6.

Another field of application of the algorithm with the predicate (both redundant and irredundant) of the connecting arcs is multilevel graphs. A two-level graph can be defined as a second-level graph the vertices of which are first-level graphs, with all these graphs being strongly connected. In stricter terms, a two-level graph is a graph some arcs of which are marked. Removing marked arcs, we obtain a set of isolated strongly connected graphs (first-level graphs). By factoring a two-level graph with respect to the mutual reachability of vertices through unmarked arcs, we obtain a strongly connected graph of the second level.

If the predicate is meant to be the predicate of the marked arcs, the algorithm will traverse the two-level graph by levels: when the algorithm enters a first-level graph first time, it, first, completely traverses this graph by the unmarked arcs; then, goes to the next first-level graph by the marked arc. When the algorithm enters the first-level graph next time, it passes only the simple path until the required marked arc that goes out of this graph. To obtain the estimate of the traversal length, we consider the class of two-level graphs in which all first-level graphs have the same number of vertices n_1 and arcs k_1, and the graphs of the second level have n_2 vertices (the number of the first-level graphs) and k_2 arcs. The numbers of vertices and arcs in the two-level graph are given by $n = n_1n_2$ and $k = k_2 + n_2k_1$, respectively. The traversal length is equal to $O(n_2k_2O(n_1) + n_2O(n_1k_1) = O(n(k_1 + k_2)))$, which, as k_1 grows, gives a limit gain of n_2 times. An example of a two-level graph is shown in Fig. 7. The first-level graphs (highlighted by the gray color) are similar to the graph with the initial vertex v_{n1} depicted in Fig. 2 and the second-level graph is similar to the graph shown in Fig. 6. The ith arc of the second level leads from the vertex v_{n1-1} of the ith first-level graph to the beginning vertex v_{n1} of the $(i + 1)$th first-level graph (i varies through 0, …, n_2 and is shown as the superscript of the vertices).

Let two arcs originating from the vertices v_{n1-1} of the first-level graphs be ordered such that the arc $(v^i_{n1-1}, v^{i+1}_{n1})$ precedes the arc (v^i_{n1-1}, v^i_{n1}) and the

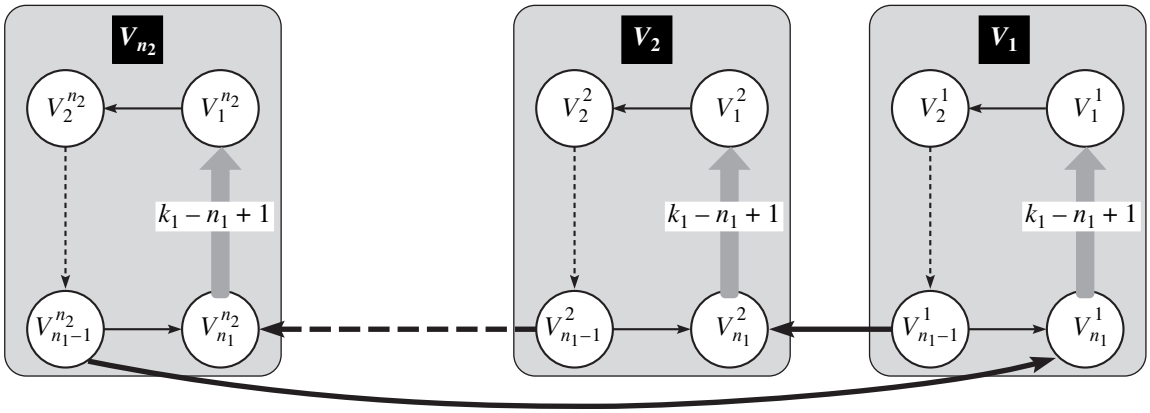

Fig. 7.

arc (v_{n1-1}^{n2}, v_{n1}^1) precedes the arc (v_{n1-1}^{n2}, v_{n1}^{n2}). Then, the algorithms $\mathbb{A}_5$ and $\mathbb{A}_6$, which "do not know" about the two-level structure of the graph, traverse first a Hamilton circuit (a path that passes through all vertices of the graph, with each vertex being traversed once), all arcs of which, except for the first arc (one of the arcs (v_{n1}^1, v_1^1)), become the back arcs. Afterwards, after passing all chords from the set of $k_1 - n_1 + 1$ arcs (highlighted by the gray color in the figure), except for the first and last chords in each set, it will require to traverse this Hamilton circuit again. The number of such chords is $\Omega(n_2k_1)$, and the path length is $\Omega(n)$; thus, the covering path length is equal to $\Omega(nn_2k_1)$. On the other hand, the algorithm that uses marking of the second-level arcs traverses the first-level graphs in turns: after traversing the ith graph, it goes along the marked arc to the next, $(i + 1)$th graph. Hence, the covering path length is $\Omega(n_2n_1k_1) = \Omega(nk_1)$, which gives a gain of n^2 times. Thus, if the algorithm "knows" about the two-level structure of the graph and this "knowledge" is formalized in the form of the authentic predicate of marked arcs, the traversal of the graph is closer to the optimal one for graphs from this class.

A p-level graph is defined by induction as a two-level graph the components of which are $(p - 1)$-level graphs. The algorithm discussed can be modified for work with any predetermined number p of graph levels.

5. TESTING BASED ON IRREDUNDANT TRAVERSAL ALGORITHMS

The suggested irredundant algorithms for traversing deterministic directed graphs can be used for the open-state testing. In addition to the algorithm, the test contains the following components:

• An *iterator* of stimuli (operation *next*) determined by the specification precondition, which specifies admissibility of stimuli x at each state v of the automaton, $PRE(v, x) = true$.

• A *mediator* (operations *call* and *status*) designed for supplying stimuli to the tested automaton and observing reactions and post-states.

• An *oracle* that checks the transition correctness and is determined by the specification postcondition, which specifies admissibility of the reactions y and post-states v' upon transition from a pre-state v by a stimulus x, $POST(v, x, y, v') = true$.

We assume that the following *admissibility hypothesis* for the open-state testing for partially defined automata holds: for any state reachable from the initial state in the model, all stimuli admissible in the model are admissible in the implementation (the converse is not required).

The hidden-state testing is a much more complicated task. First of all, we face the problem of stimulus admissibility for partially defined automata. If we do not know the current state of the automaton being tested, we do not know what stimuli are admissible. If no assumptions on the implementation are made, we may input only those that are admissible in all states. It is because of this reason that only completely defined automata are usually considered [17].

The automaton specification may, generally, define several specified transitions from a given pre-state by a given stimulus with the same reaction. Such transitions differ only by their post-states. In other words, the postcondition equation $POST(v, x, y, v') = true$ may have more than one solution in v'. If, for any admissible v, x, and y, the number of such solutions is not greater than one, the specification and the corresponding model automaton are said to be *weakly deterministic*. This is the case of the so-called *observable* indeterminism [23].

It should be noted that the weak determinism does not reduce the descriptive ability of the specifications up to the equivalence of the automata being specified. The point is that the class of equivalent finite automata corresponds to a regular set of sequences in the alphabet of stimuli and reactions (on the Cartesian product of their alphabets), which, by the known theorem on

regular sets [24–26], can be generated by a deterministic finite graph. Here, the determinism is understood as the lack of two identically colored (by a stimulus and reaction) outgoing arcs of one vertex, which is equivalent to the above-defined weak determinism. Of course, without the condition of the weak determinacy, writing of the specifications is simplified, whereas finding of equivalent weakly deterministic specifications may be a rather difficult task.

For the hidden-state testing of partially defined automata with weakly deterministic specifications, we assume that the following *admissibility hypothesis* holds: for a given pre-state, a stimulus admissible in it, and a given reaction, any stimulus admissible in the post-state of the specified transition is admissible in the post-state of the implemented transition. Let $x(v)$ denote the set of stimuli admissible in the state v. Then, the above hypothesis means that the inclusion $x(v') \subseteq x(v_R)$ holds, where v_R is the post-state of the implemented transition (v, x, y, v_R) and v' is a solution of the postcondition equation $POST(v, x, y, v') = true$. If this equation has no solutions, then this implies that the reaction y is not correct. In this case, the testing is terminated.

At first glance, the admissibility hypothesis does not seem motivated; however, it is quite natural from the standpoint of practice. It implies that the admissibility of stimuli at any time is uniquely determined by the history (by the sequence of stimuli and the corresponding reactions), which is quite natural from the standpoint of the user working with a software system modeled by an automaton. It is assumed, of course, that the errors that may appear in the implementation can be detected (by the reactions observed) before they result in the violation of the admissibility hypothesis.

Let us assume that we are able to determine whether a reaction is correct and, for a correct reaction, to calculate the corresponding post-state. For example, let the precondition have the form "ReactionChecking(v, x, y)&$v' = Poststate(v, x, y)$", where *ReactionChecking* is the predicate determining correctness of the reaction and *Poststate* is an explicit function calculating the post-state for a correct reaction. Then, the corresponding test can be organized as follows.

Let the initial state $v_0$be known. Feeding a stimulus x_0 to it and observing a reaction y_0, we check whether the reaction is correct and, if it is correct, calculate the post-state v_1. Then, according to the admissibility hypothesis, the implementation occurs in the post-state in which all stimuli admissible in v_1 are admissible. At the next step, we select a stimulus that is admissible in v_1, and so on. In the course of this testing, we construct a hypothetical graph and a path in it drawing the arcs corresponding to the hypothetical automaton transitions. If a reaction is rejected, the test notes an error and terminates.

It is important to emphasize that, if the implementation automaton is deterministic, then the hypothetical graph constructed is also deterministic. This makes it possible to control the assumption on the determinism of the implementation graph. The more important consequence of this fact is that, if the traversed path is a covering path of the constructed graph, this graph may be considered the state transition graph of the explicated subautomaton $M(R)$ of the model automaton M. The implementation automaton R (the automaton in which the "redundant" stimuli, which are admissible in the implementation but not admissible in the specification, are not taken into account) satisfies the specification if and only if it is equivalent to the subautomaton $M(R)$. Then, we can check this equivalence using ordinary methods of the conformance testing by means of the checking sequences and considering the graph $M(R)$ as the model graph.

Thus, we see that the irredundant traversal algorithms can be used as based ones at the first stage of the hidden-state testing, which can be referred to as *model explication phase*. Necessary conditions for this are the determinism of the implementation, the weak determinism of the specifications, and the possibility to determine whether the reactions are correct and to calculate the post-states for the correct reactions; in addition, the admissibility hypothesis is required. The second testing phase is the ordinary automaton conformance testing with the use of the explicit graph for the model.

A nondeterministic (in particular, weakly deterministic) specification may, of course, correspond to a nondeterministic implementation. In this case, the specification is often factored by introducing transition equivalence [27, 28]. The objective of such a factorization is to reduce the required number of test actions through the reduction of the number of states and transitions in the factored model automaton. Such a testing requires checking of only factor transitions (for this purpose, any transition from the corresponding equivalence class may be chosen). Clearly, if the given equivalence "is broken into fragments," then the testing of the factor model with the "fragmented" equivalence will result in a better test coverage. There exist methods for the "equivalence splitting"; in certain cases, they result in deterministic factor models, still, keeping them "small" compared to the original model [7]. Thus, the suggested irredundant algorithms for traversing deterministic graphs can be used in factored testing of nondeterministic implementations.

6. CONCLUSIONS

The irredundant graph traversal algorithms discussed in this paper and tests based on them have been developing and testing since 1995 by the group RedVerst [29] in the course of the execution of several large-scale projects on testing various software [27, 28]. The latter testing was based on functional specifications obtained on the design or reengineering stages.

As a rule, in testing, not only the complexity of the traversal algorithm in terms of the time and memory required is critical. A more important characteristic is the number of test actions, i.e., the constructed covering path length. The free algorithm $\mathbb{A}_5$ and its irredundant modification $\mathbb{A}_6$ guarantee only that the order of the covering path length is not greater than *nk* in the worst case. At the same time, for many graphs with the minimal covering path length less than *nk*, they construct as lengthy covering paths as in the worst case. The studies of irredundant algorithms that tend to construct covering paths of minimal length are extensively developing (e.g., [8]).

In the next paper, we will consider algorithms for traversing nondeterministic graphs, which can be used for testing nondeterministic automata. We plan to consider two following cases: (1) the open-state testing and (2) the first stage (*the explication of the model automaton*) of the hidden-state testing for weakly deterministic specifications.

REFERENCES

1. Edmonds, J. and Johnson, E.L., Matching, Euler Tours and the Chinese Postman *Math. Programming*, 1973, vol. 5, pp. 88–124.
2. Lenstra, J.K. and Rinnooy Kan, A.H.G., On General Routing Problems, *Networks*, 1976, vol. 6, pp. 273–280.
3. Thimbleby, H., The Directed Chinese Postman Problem, *Techn. Report*, School of Computing Sci., Middlesex Univ., London, 2000.
4. Hoffman, D. and Strooper, P., ClassBench: A Framework for Automated Class Testing, *Software Maintenance: Practice Experience*, 1997, vol. 27, no. 5, pp. 573–579.
5. 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.
6. Deng, X. and Papadimitriou, C.H., Exploring an Unknown Graph, *J. Graph Theory*, 1999, vol. 32, no. 3, pp. 265–297.
7. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Use of Finite Automata for Program Testing, *Programmirovanie*, 2000, no. 2, pp. 12–28.
8. Albers, S. and Henzinger, M.R., Exploring Unknown Environments, *SIAM J. Comput.*, 2000, vol. 29, no. 4, pp. 1164–1188.
9. Ore, O., *Theory of Graphs*, Providence: AMS, 1962. Translated under the title *Teoriya grafov*, Moscow: Nauka, 1980.
10. Rabin, M.O., Maze Threading Automata. Lecture presented at MIT and UC Berkley, 1967.
11. Bourdonov, I.B., Study of the Automaton Behavior on Graphs, *MS Dissertation*, Moscow: Moscow State University, 1971.
12. 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.
13. Even, S., *Graph Algorithms*, Comput. Sci., 1979.
14. Afek, Y. and Gafni, E., Distributed Algorithms for Undirectional Networks, *SIAM J. Comput.*, 1994, vol. 23, no. 6, pp. 1152–1178.
15. Even, S., Litman, A., and Winkler, P., Computing with Snakes in Directed Networks of Automata, *J. Algorithms*, 1997, vol. 24, pp. 158–170.
16. 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.
17. Lee, D. and Yannakakis, M., Principles and Methods of Testing Finite State Machines: A Survey, *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, Berlin: IEEE Computer Society, 1996.
18. von Bochmann, G. and Petrenko, A., Protocol Testing: Review of Methods and Relevance for Software Testing, *Proc. of ISSTA*, 1994, pp. 109–124.
19. Petrenko, A., Yevtushenko, N., and Dssouli, R., Grey-Box FSM-Based Testing Strategies, *Department Publication 911*, Univ. de Montreal, 1994.
20. Fecko, M.A., Uyar, M.U., Sethi, A.S., and Amer, P.D., Conformance Testing in Systems with Semicontrollable Interfaces, *Ann. Telecommunications*, 2000, vol. 55, no. 1, pp. 70–83.
21. Petrenko, A., Yevtushenko, N., and von Bochmann, G., Testing Deterministic Implementations from Nondeterministic FSM Specifications, *Selected Proc. of the IFIP TC6 9th Int. Workshop on Testing of Communicating Systems*, 1996.
22. Gurevich, Yu., Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Trans. Computational Logic*, 2000, vol. 1, no. 1, pp. 77–111.
23. Tabourier, M., Cavalli, A., and Ionescu, M., A GSM-MAP Protocol Experiment Using Passive Testing, *Proc. of the World Congr. on Formal Methods in Development of Computing Systems (FM'99)*, Toulouse, 1999.
24. Rabin, M. and Scott, D., Finite Automata and Their Decision Problem, *IBM J. Research Development*, 1959, vol. 3, pp. 114–125.
25. Ginsburg, S., *The Mathematical Theory of Context-Free Languages*, New York: McGraw-Hill, 1966. Translated under the title *Matematicheskaya teoriya kontekstnosvobodnykh yazykov*, Moscow: Mir, 1970, pp. 71–78.
26. Varsanof'ev, D.V. and Dymchenko, A.G., *Osnovy kompilyatsii* (Fundamentals of Compilation), 1991, http://www.code-net.ru/progr/compil/cmp/intro.php.
27. Bourdonov, I, Kossatchev, A., Kuliamin, V., and Petrenko, A., UniTesK Test Suite Architecture, *Proc. of FME 2002*, Lecture Notes in Computer Science, vol. 2391, pp. 77–88, Berlin: Springer, 2002.
28. 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.
29. http://www.ispras.ru/RedVerst.