Programming and Computer Software, Vol. 30, No. 1, 2004, pp. 2–17. Translated from Programmirovanie, Vol. 30, No. 1, 2004.
Original Russian Text Copyright © 2004 by Bourdonov, Kossatchev, Kuliamin.

Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic 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

1. INTRODUCTION

This work is a sequel to the paper [1], which was devoted to the traversal of deterministic graphs as the first step of the specification-based testing of finite automata (more precisely, objects considered as finite automata).

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 directed graph, called the state transition graph, 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 the paper [1], we considered deterministic partially specified automata. This study is devoted to the nondeterministic case.

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., 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 the so-called *conformance testing* in the broad 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 if possible. 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). On the other hand, this means that the test cannot change the automaton state other than by inputting a stimulus to the automaton.

Admissibility of stimuli. At every moment, we can learn what stimuli are admissible. Not all admissible stimuli are necessarily used in 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 by 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) [2], 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 [2]. 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.

Three basic problems here are as follows:

1. **Nondeterminism problem.** In the general case, we can never be sure whether all transitions of the implementation automaton have been checked. To solve this problem, some special assumptions are sometimes introduced [2]. For example, it is assumed that, if the number of passages of a given vertex v with a given stimulus x is sufficiently large, all transitions of the form (v, x, y, v') will be used. It is also possible to introduce probabilities of the transitions and carry out a sufficient number of tests to ensure the probabilistic completeness of the test. Another approach is based on the assumption of *test equivalence* of the transitions. In this approach, it is sufficient to check only one transition belonging to an equivalence class. If the latter includes all transitions from a given state by a given stimulus, then it suffices to check each stimulus in each state. This is an analogue of the graph traversal, which is referred to as a *stimulus-based traversal.*

2. **Testing completeness problem.** How to check the equivalence of model and implementation automata by means of testing? In the deterministic case, there exist well-developed methods for solving this problem. For an open-state testing, the problem reduces to the traversal of the model graph [2], i.e., to the construction of a path passing through all model transitions. Moving along the path, for each model transition (v, x, y, v'), we feed one and the same stimulus x to the implementation automaton and check whether the reaction and the post-state of the implementation automaton coincide with the model ones (y, v'). If the information about implementation states is not available (hidden-state testing), one needs to introduce special restrictions on the implementation and model and take advantage of more complicated checking sequence methods [2]. In the nondeterministic case, the problem is much more complicated in view of the nondeterminism. However, at least, the stimulus-based traversal is to be always executed, although it is not sufficient even in the case of an open-state testing (if we do not confine ourselves to checking of only one transition at each state by every stimulus).

3. **Specification implicitness problem.** Unfortunately, in practice, the specifications do not explicitly describe the model automaton, so that we face with 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 reduces to 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. In stricter terms, if a specification admits several transitions from a given pre-state in response to a given stimulus, then it is assumed that the implementation has at least one (not necessarily all) such a 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 to be 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* [3].) 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.

The problems of construction of paths in the state transition graph of a finite automaton are known for a long time. They were formulated for paths of certain specific classes at the very beginning of the theory of finite automata. Problems of this kind occasionally attracted attention of the researchers in connection with the testing based on a finite automaton model [4–6]. Various testing methods based on nondeterministic models are being intensively studied (see Petrenko, *et al.* [7] or the ASM group of Microsoft Research [8]). Certain restrictions on the behavior of the implementation are usually imposed. For example, the implementation is assumed to be deterministic, as in [9], or certain complicated test hypotheses that all possible transitions could be obtained after several attempts are assumed [10, 11].

The paper is organized as follows. In Section 2, an approach to solving these problems is suggested. The first stage of testing by implicit specifications is separated. The goal of this stage is to find an explicit model subautomaton (to be more specific, a subgraph of the state transition graph that contains all reachable states and, at least, one transition for each stimulus at each state), which could be used at the following stages. This problem is reduced to the stimulus-based traversal of an unknown nondeterministic graph. The corresponding algorithms that obtain information about the graph in the course of its stimulus-based traversal are called *irredundant*.

In Section 3, the graph terminology is introduced, and the notions of the stimulus-based traversal and traversal algorithms are defined. Problems of existence of a covering path through the graph and finding its length are discussed in Section 4. Particular irredundant traversal algorithms are suggested in Section 5.

2. TESTING OF NONDETERMINISTIC AUTOMATA

We assume that the model M is described by its implicit specification in terms of preconditions and postconditions. The precondition determines admissibility of each stimulus x at each state v of the model: $PRE(v, x) = true$. The postcondition determines admissibility of the obtained reaction y and post-state v' upon transition from the pre-state v by the stimulus x: $POST(v, x, y, v') = true$.

We suggest to consider testing based on implicit specifications as a two-stage testing. The goal of the first stage is to find an explicit subautomaton $M(R) \subseteq M$ for the implementation R, and the goal of the second stage is to test the implementation subautomaton $R(M) \subseteq R$ by the model automaton $M(R)$.

Let us consider first how to determine the automata $R(M)$ and $M(R)$ in the course of the *open-state* testing. The initial state v_0 of the model M is assumed to be the same as the initial state of the implementation R (which can be checked by means of the *status message*), and we place it into $R(M)$ and $M(R)$. Then, we feed the stimulus x_0 that is admissible in M at the state v_0. For such a stimulus, we can take any solution of the specification precondition equation $PRE(v_0, x_0) = true$. The implementation automaton performs a transition (v_0, x_0, y_0, v_1), which is added to $R(M)$. Having obtained the reaction y_0 and post-state v_1, we check whether there exists the transition (v_0, x_0, y_0, v_1) in M, i.e., verify the specification post-condition $POST(v_0, x_0, y_0, v_1) = true$. If the postcondition does not hold, we report about an error in the implementation. Otherwise, we add the transition (v_0, x_0, y_0, v_1) to $M(R)$. Next, we feed the stimulus x_1 admissible in M at the post-state v_1, and so on. In practice, the test will construct only a subautomaton $M(R)$.

Such testing suggests that, for any state v reachable in M from an initial state v_0, all stimuli admissible in M are admissible in R (the opposite is not required) if, of course, no implementation errors are found "on the way" from v_0 to v. This is the so-called *admissibility hypothesis* for the open-state testing.

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 the case of the open-state testing, the only current post-state is known. If the states are hidden, we require that there exist only one such a post-state in the model; i.e., the postcondition equation $POST(v, x, y, v') = true$ must not have more than one solution in v'. In this case, the specification and the corresponding model automaton are said to be *weakly deterministic*. This is the case of the so-called *observable* nondeterminism [12].

Let us assume that the postcondition equation has only one solution and that this solution can be found. For example, let the postcondition 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, when determining the transition in the automaton $M(R)$ that corresponds to the transition (v_i, x_i, y_i, v_{i+1}) in $R(M)$, we calculate the model post-state v^*_{i+1} instead of the unknown implemented post-state v_{i+1} and add the transition $(v^*_i, x_i, y_i, v^*_{i+1})$ to $M(R)$; where v^*_i is a model state calculated at the previous step and $v^*_0 = v_0$.

Note that, in constructing $M(R)$, the uniqueness and computability of the post-states are required not for the entire model M but only for that part of the model that is used in such a construction. The model is not required to be weakly deterministic in those parts that can be reached only by the reactions that are lacking in the implementation and, hence, in $M(R)$. In what follows, speaking of weak determinacy, we mean the weak determinacy of $M(R)$.

In the case of the hidden-state testing, we also use the *admissibility hypothesis*. However, in this case, we speak not about identical states but about *corresponding* states of the implementation and the model, v and v^*, respectively, and require that any stimulus admissible in the model M at v^* be admissible in the implementation R at v. The correspondence here is meant in the following sense: the states v and v^* are reachable in R and M, respectively, from the initial state v_0 by means of one and the same sequence of stimuli and reactions.

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.

We have described the construction of the subautomaton $M(R)$ in the course of testing. Here, we arrive at the question of how to determine whether such a construction is complete? One condition is quite evident: at each model state v^* in $M(R)$, all stimuli admissible in this state are to be tested; i.e., for each stimulus x admissible at v^* in M, the automaton $M(R)$ contains, at least, one transition of the form $(v^*, x, y, v^{*\prime})$. In this case, the graph $M(R)$ is said to be *traversed by stimuli*. In the deterministic case, this implies that $M(R)$ has been constructed and traversed. In the nondeterministic case, this condition is necessary but not sufficient, because there may be several such transitions and, without additional assumptions, there is no guarantee that all these transitions are contained in $M(R)$. Therefore, we assume in this case that a certain approximation $M'(R) \subseteq M(R)$ has been constructed.

Under some natural assumptions on guaranteed reachability of one state from another, $M'(R)$ contains all states of $M(R)$ and may be viewed as its covering subgraph. The graph $M'(R)$ does not contain only some transitions of the form $(v^*, x, y, v^{*\prime})$, but it contains the states v^* and $v^{*\prime}$, as well as, at least, one transition from v^* by the stimulus x. It is possible that some of these transitions will be passed during the second testing stage and, thus, will be added to $M'(R)$.

The test system at the first stage contains the following components:

• *An algorithm of stimulus-based traversal.*

• An *iterator* of stimuli determined by the specification precondition, which specifies admissibility of stimuli x at each state v of the automaton,

$$PRE(v, x) = true.$$

• A *mediator* designed for supplying stimuli to the tested automaton and observing the reactions.

• An *oracle* that checks the transition correctness and is determined by the specification postcondition, which specifies admissibility of the reaction y and post-state v' upon transition from a pre-state v by a stimulus x,

$$POST(v, x, y, v') = true.$$

• *Post-state computation procedure*:

(a) for an open state, operation *status message*,

(b) for a hidden state, explicit function *Poststate*.

In conclusion, we briefly mention two problems that are usually encountered when testing automata in practice: nonidentity of the mediator correspondence and testing based on automaton factorization.

So far, we assumed that the alphabets of stimuli, reactions, and states are the same in the model and in the implementation, and the mediator and the function *Poststate* implement, in fact, identical transformations. However, such a situation is seldom met in practice. On the one hand, this is explained by technical reasons: for example, by the fact that the specifications are written in a specification language that is different from the language used for programming the implementation and has another type structure. The fact that the specification is always an abstraction, which serves as a model for several (sometimes, very different) implementations, is even more important. In the general case, the mediator implements down-transformations of model stimuli into implementation ones and up-transformations of implementation reactions into model ones, which are not identical transformations. In the case of the open-state testing, the function *Poststate* transforms the implementation states into model ones (up-transformations), and, in the hidden-state testing, such a correspondence of states exists implicitly. In any case, the up-transformations may be not injective and result in certain "roughening" of tests, which correspond to the model abstraction level. Nevertheless, in the case of the open-state testing, transformations of stimuli and reactions may depend on the implementation state and, generally, even on prehistory of interactions with the implementation automaton.

The test roughening is often made purposefully as a factorization of the automaton, when the number of states and stimuli (even in the subautomata $R(M)$ and $M(R)$) is too great [13]. The factorization is performed by a given equivalence of states and/or stimuli and, in the general case, transitions. Thus, it is required to check only factor transitions, which means that we can take any transition from the corresponding equivalence class. Note that, if we consider transitions from a given state by a given stimulus as equivalent, then the subautomaton $M'(R)$ obtained at the end of the first stage of testing may be viewed as a factor automaton (if we do not introduce additional equivalence of states). Generally, in addition to two levels—levels of the implementation R and the specification model M—we get the third level, the level of the factor model $F(M)$. Thus, the test works not on the level M, but rather on the level $F(M)$, which may be referred to as the test model level. Accordingly, after the first stage, we obtain a subautomaton $F(M)(R)$ of the test model (rather than a subautomaton $M(R)$ of the specification model), which coincides with the factorization of $M(R)$, i.e., with $F(M(R))$.

The factorization may be viewed as a particular case of the general mediator correspondence between the levels of the model and factor model. Thus, one can consider multilevel systems of automaton descriptions, where each level is connected with the neighboring levels by mediator correspondences, the lower level is the implementation, and the upper level is the test model, i.e., the model that is directly used for testing the implementation.

Problems of the second stage of testing, nonidentity mediator correspondences, and testing for multilevel systems are not considered in this work. In the remaining part of this paper, we will discuss only the central component of the first testing stage, namely, irredundant algorithms of stimulus-based traversal of nondeterministic graphs. Recall that irredundant algorithms are those that obtain information about the graph only in the course of its traversal.

3. GRAPHS AND TRAVERSAL ALGORITHMS

A directed graph (further, simply *graph*) G is a collection of three following objects: 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. 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). A *Δ-arc* (a, x) is a set of the graph arcs that begin at the vertex a (the beginning of the Δ-arc) and are marked by the stimulus x admissible in a (coloring of the Δ-arc), $(a, x) = \{(a, x, b) \in EG\}$.

Remark. When testing, the graph is considered to be the state transition graph of an automaton. However, in the traversal algorithms, the graph arcs are not colored by reactions, because, when passing an arc, it suffice for the algorithm to be able to determine the endpoint (post-state) of the arc. This problem is considered to be external from the standpoint of the traversal algorithm, and it is solved by the post-state computation procedure. In connection with this, we do not distinguish between the arcs that differ only by the reactions.

A graph is said to be finite if the sets of its vertices and arcs are finite. The numbers of vertices, arcs, and Δ-arcs of a finite graph are denoted by n, k, and m, 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 $b = b'$. All Δ-arcs of such a graph are singletons; i.e., they consist of one arc. In a nondeterministic graph, a Δ-arc may contain several arcs, which differ by their endpoints.

Arcs (a, x, b) and (a', x', b') 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 and a *stimulus-based covering path* if it contains, at least, one arc from every Δ-arc of the graph.

A *Δ-path* is a set D of paths that begin at one vertex and "fork" at each Δ-arc being passed. In stricter terms, for each path $P \in D$ and any i less than the length of P, the set of the $(i + 1)$th arcs of the paths from D that have the same first i arc as P forms a Δ-arc to which the $(i + 1)$th arc of P belongs: for $P[i + 1] = (a, x, b)\{Q[i + 1] | Q \in D \& Q[1\ldots i] = P[1\ldots i]\} = (a, x)$. The beginning a of the paths belonging to the Δ-path D is called the *beginning* of the Δ-path. A vertex b is called the *end* of a Δ-path D if all paths of this Δ-path are terminated at this vertex; and the Δ-path itself is called an $[a, b]$-Δ-path. The *length* of a Δ-path D is the maximum length of its paths. A *covering Δ-path* is a Δ-path all paths of which are stimulus-based covering paths.

The above-introduced notion of a Δ-path is close to the notions of an adaptive test sequence or a test tree, which are used for describing the selection of stimuli depending on the previous transitions [2]. For a deterministic graph, the notions of a path and a Δ-path coincide; to be more specific, any Δ-path is a singleton. Accordingly, the notions of a covering path, stimulus-based covering path, and covering Δ-path are identical.

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 [14]), 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 an arc (a, x, b) selected in unspecified way among the arcs belonging to the Δ-arc (a, 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 (not to mention internal ones) 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 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 untraversed Δ-arc with the beginning at the current vertex. 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 an arc (a, x, b) selected in unspecified way from the Δ-arc (a, x). If all stimuli have already been tested at the current vertex, the empty symbol ε is returned. For the second passage along the Δ-arc (a, x), 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 construction of a stimulus-based covering path. The algorithm functioning depends, generally, on the external operations *call* and *next*. In a nondeterministic graph, a Δ-arc contains, generally, several arcs that differ by their endpoints, such that the result of the operation *call*(x) is not uniquely determined by the current vertex and stimulus x. Therefore, the stimulus-based traversal performed by the algorithm depends on the results of the operation *call* and, strictly speaking, is not a graph traversal. We may say only about a *call*-independent set of such paths, i.e., about a set of all paths that could be passed by the algorithm upon fixing all, but *call*, external operations and for all possible results of the operation *call*. This set is obviously a Δ-path in the graph, and each path from this Δ-path corresponds to a certain set of results of the operation *call*. If this Δ-path is a covering Δ-path, the algorithm is said to perform a Δ-traversal of the graph with a given initial vertex. An algorithm is said to perform a *guaranteed* Δ-traversal of a given graph with a given initial vertex if it traverses the graph by stimuli for any admissible results of all (not only *call*) external operations (for irredundant algorithms, independently of the stimulus iteration at the vertices performed by *next*).

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 stimulus-based traversal has been completed, i.e., whether the path constructed is a covering path. It is possible that the path constructed is a stimulus-based covering path, 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 *verdict* of the algorithm. The verdict is said to be *correct* if the information contained in it is true.

4. Δ-TRAVERSALS OF GRAPHS

4.1. Strongly Δ-Connected Graphs

Lemma 4.1. If any path of a Δ-path D beginning at a vertex a passes through a vertex b, then there exists an [a, b]-Δ-path.

Proof. The desired [a, b]-Δ-path is constructed as a set of initial segments of paths from D that are terminated by the first occurrence of the vertex b.

A nonempty proper subset U of vertices of a graph is called a Δ-*isolated set* if each Δ-arc beginning in U contains an arc with the end in U. If a vertex a belongs to a Δ-isolated set U, and a vertex b does not belong to U, then such a set U is called [a, b]-Δ-isolated.

Lemma 4.2. If there exists an [a, b]-Δ-path D in a graph, then there does not exist an [a, b]-Δ-isolated set of vertices in this graph.

Proof. Let us assume the contrary; i.e., let such a set U exist. Any path $P \in D$ leads from $a \in U$ to $b \notin U$. Hence, there exists an arc in P that leads from U to the outside. Let i be the number of the first arc of this kind, $P[i] = (c, x, d)$, where $c \in U$ and $d \notin U$. Consider a path P from D that has a maximal index i among the paths belonging to D. Since U is a Δ-isolated set, there must exists an arc (c, x, d') in the Δ-arc (c, x) such that $d' \in U$. Then, since D is a Δ-path, there must exist a path $P' \in D$ such that $P'[1, i-1] = P[1, i-1]$ and $P'[i] = (c, x, d')$. It is evident that the number of the first arc in the path P' that leaves the set U is greater than i. Thus, we arrived at the contradiction, which proves the lemma.

A Δ-*subgraph* is a subgraph that, for any Δ-arc of the graph, either contains all its arcs or does not contain it at all. A *simple* Δ-*path* is a Δ-path in which all paths are simple paths. Clearly, the length of a simple Δ-path does not exceed $n - 1$. A graph is called acyclic if it does not contain cyclic paths; a source is a vertex that has no incoming arcs; and a sink is a vertex without outgoing arcs.

Lemma 4.3. If there does not exist an [a, b]-Δ-isolated set of vertices in a graph G, then there exists a simple [a, b]-Δ-path in it.

Proof. First, we describe an algorithm for constructing an acyclic Δ-subgraph in which the vertex a is a source (which is, perhaps, not the only one) and b is the only sink. At the beginning of each algorithm step, we have an acyclic Δ-subgraph H with the only sink b. At the very beginning, H consists of one vertex b and has no arcs. The algorithm step consists in the following: if a does not belong to VH and there exists a Δ-arc (c, x) such that its beginning $c \notin VH$ and the endpoints of all arcs belong to VH, then this Δ-arc is added to the subgraph H (its beginning is added to VH, and the arcs, to EH). Otherwise, the algorithm stops.

Note that, if $a \notin VH$, then such a Δ-arc (c, x) exists, since, otherwise, the set of vertices $VG\backslash VH$ would be [a, b]-Δ-isolated. Hence, taking into account that the number of Δ-arcs in the graph is finite, the algorithm will stop after a finite number of steps immediately after the vertex a occurs in VH and, evidently, becomes its source. Since, at each step, the whole Δ-arc is added to H, H remains a Δ-subgraph. Since the beginning of the added Δ-arc did not belong to VH, H remains a cyclic graph.

The desired simple [a, b]-Δ-path is constructed as a set of paths in the Δ-subgraph H that start at a and terminate at b.

The next theorem follows immediately from Lemmas 4.1–4.3 and the definition of a simple [a, b]-Δ-path.

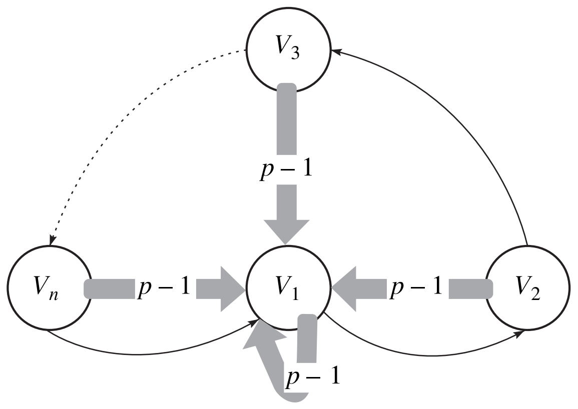

Fig. 1.

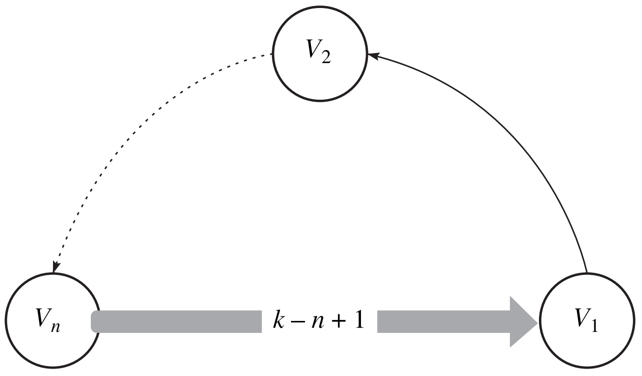

Fig. 2.

Theorem 4.1. For vertices a and b of a graph, the following assertions are equivalent:

(1) there exists a Δ-path beginning at a any path of which passes through b,

(2) there exists an $[a, b]$-Δ-path,

(3) there does not exist an $[a, b]$-Δ-isolated set of vertices,

(4) there exists a simple $[a, b]$-Δ-path.

The vertex b is said to be Δ-*reachable* from the vertex a if any assertion of Theorem 4.1 is valid. A graph is said to be *strongly* Δ-*connected* if any vertex of the graph is Δ-reachable from any other vertex.

Theorem 4.2. (1) For any strongly Δ-connected graph and any pair of its vertices a and b, there always exists a covering $[a, b]$-Δ-path of length $O(nm)$.

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

Proof. (1) Let us introduce an arbitrary linear order on the Δ-arcs of the graph, such that the first Δ-arc begins at the vertex $v_1 = a$: $(v_1, x_1), (v_2, x_2), \ldots, (v_m, x_m)$. Denote $v_{m+1} = b$. In a strongly Δ-connected graph, there always exists a simple $[v'_i, v_{i+1}]$-Δ-path $D_i(v'_i)$ from the endpoint v'_i of any arc belonging to the ith Δ-arc $(v_i, x_i, v'_i) \in (v_i, x_i)$ to the beginning v_{i+1} of the next $(i + 1)$th Δ-arc (for $i = m$, to the vertex $v_{m+1} = b$). Denote by $D_i = \{D_i(v'_i) | v'_i \in (v_i, x_i)\}$ the union of all these simple Δ-paths. The desired covering Δ-path is the set of all possible concatenations of arcs and simple paths: $D = (v_1, v'_i) \wedge D_1 \wedge \ldots \wedge (v_i, v'_i) \wedge D_i \wedge \ldots \wedge (v_m, v'_m) \wedge D_m$. Each path in D passes along, at least, one arc from each Δ-arc; it begins at a, terminates at b, and, therefore, is a stimulus-based covering $[a, b]$-path. Since each stimulus-based covering path consists of m arcs (one arc from every Δ-arc) and m connecting simple paths (belonging to the connecting simple Δ-paths), its length does not exceed $m + m(n - 1) = O(nm)$.

(2) Since, in a deterministic graph, a Δ-arc coincides with an ordinary arc, and a covering Δ-path is the same as just covering path, we may consider a deterministic graph 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 length of the covering path P_1 is not less than one and that its last arc is (v_j, v_1), we obtain the lower bound of the length of the covering path (without regard to the last covering path P_t),

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

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).

In examples depicted in Figs. 1 and 2, there is a strong relationship between the numbers of arcs and Δ-arcs, $k' = m'$. If we require, for example, that the number of arcs be t times greater than the number of Δ-arcs, we can add, to each arc of this graph, one Δ-arc that has the same beginning and consists of $2t - 1$ arcs leading to any vertices. The number of vertices will not be changed; the number of arcs will be equal to $k'' = k' +$

$k'(2t-1) = 2tk'$; and the number of Δ-arcs will be $m'' = 2k'$. It is evident that the graph obtained is strongly Δ-connected, and the length of any covering Δ-path of this graph is equal to $\Omega(nk') = \Omega(nm'')$.

4.2. Δ-Reachable Graphs

Since mutual Δ-reachability of vertices is an equivalence relation, the graph G is partitioned, generally, into strongly Δ-connected components, on the set of which the Δ-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 the 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 $K(a)$. An arc (a, x, b) is a *forward* arc if $K(a)$ is Δ-unreachable from $K(b)$. A Δ-arc is called *connecting* if the endpoints of all of its arcs belong to one component B that is different from the component A of the beginning of the Δ-arc; the connecting Δ-arc is said to lead from A to B.

A *factor graph* of a graph G with respect to the mutual Δ-reachability relation is a graph $F(G)$ the vertices of which are the strongly Δ-connected components of G and the arc (A, x, B), where $A \neq B$ are components of the graph G, is a connecting Δ-arc in G leading from A to B and colored by the stimulus x. It is evident that the factor graph is a deterministic acyclic graph.

A Δ-*reachable graph* is a graph all vertices of which are Δ-reachable from a given initial vertex. By virtue of Lemma 4.1, a Δ-traversal is possible only in a Δ-reachable graph; therefore, in what follows, we consider only Δ-reachable graphs.

Theorem 4.3. A graph G with an initial vertex v_0 is Δ-reachable if and only if its factor graph $F(G)$ with respect to Δ-reachability is an acyclic graph with one source $K(v_0)$.

Proof. *Necessity.* Any component $K \neq K(v_0)$ contains, at least, one connecting Δ-arc (v, x), since, otherwise, the $VG\backslash VK$ would be Δ-isolated in G, and none of vertices from VK would be Δ-reachable from v_0. Hence, any factor vertex of the factor graph, except for $K(v_0)$, has, at least, one incoming factor arc, and, being an acyclic graph, $F(G)$ has only one source $K(v_0)$.

Sufficiency. For any vertex v, consider a simple path F of length t in $F(G)$ from $K(v_0)$ to $K(v)$. Its ith factor arc is a connecting Δ-arc (v_i, x_i) of the graph G, where $i = 1, \ldots, t$. Considering all its arcs $(v_i, x_i, v'_i) \in (v_i, x_i)$ and introducing the notation $v_{t+1} = v$, we find that the vertices v'_i and v_{i+1}, where $i = 0, \ldots, t$, belong to one component, and, hence, there exists a $[v'_i, v_{i+1}]$-Δ-path $D(v'_i)$. Denote by $D_i = \{D_i(v'_i) | v'_i \in (v_i, x_i)\}$ the union of all such Δ-paths. Let D_0 denote the $[v_0, v_1]$-Δ-path in the component $K(v_0)$. The desired $[v_0, v]$-Δ-path is constructed as a set of all possible concatenations $D_0 \wedge (v_1, x_1) \wedge D_1 \wedge \ldots \wedge (v_t, x_t) \wedge D_t$.

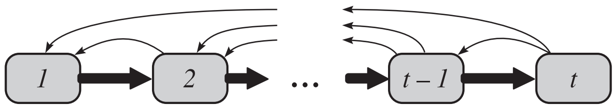

Fig. 3.

4.3. Graphs of the First Kind

A graph of the *first kind* is a graph with a linear Δ-reachability order of the components in which all outgoing *forward* arcs (arcs leading to components with greater numbers) of any (but the last) ith component form one connecting Δ-arc that leads to the next, $(i+1)$th, component. Note that some arcs (but not Δ-arcs!) may lead to components with lesser numbers (in the *backward* direction). By default, the initial vertex v_0 belongs to the first component (Fig. 3). In the deterministic case, the above definition of a graph of the first kind coincides with that in [1].

A *traversed graph* of a Δ-path D is a subgraph GD consisting of all arcs of all paths of this Δ-path and of the incidental vertices. It is evident that a traversed graph is a Δ-subgraph. Note that D is not necessarily a covering Δ-path of GD, in contrast to the deterministic case, where a path is always a covering path on the traversed graph [1].

Theorem 4.4. (1) If a Δ-path D is a covering Δ-path of its traversed graph GD, then GD is a graph of the first kind, and the initial vertex v_0 of the Δ-path belongs to its first component.

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

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

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

Proof.

(1) For any two vertices b and c, any path from the covering Δ-path passes through both these vertices. Let the first occurrence of b in the path $P \in D$ precede the first occurrence of c, and let $P(b)$ be the initial segment of P that ends by the first occurrence of b. The extensions $Q\backslash P(b)$ of all paths $Q \in D$ extending $P(b)$ must contain c. The set of all these extended paths, clearly, forms a Δ-path in the traversed graph, and, by Lemma 4.1, there exists a $[b, c]$-Δ-path in the traversed graph. Thus, for any pair of vertices of the traversed graph, one of them is Δ-reachable from another in the traversed graph. Hence, the traversed graph has a linear order of Δ-reachability of the components. Since, for

any path from D, the initial vertex v_0 is passed before any other vertex, all traversed vertices are Δ-reachable from v_0, and, hence, v_0 belongs to the first component.

Now, it remains to show that any arc (b, x, c) in the traversed graph that leads from the ith component to the component with a greater number belongs to the connecting Δ-arc (a_i, x_i). Let us assume the contrary, namely, that $(b, x) \neq (a_i, x_i)$. Any path $P \in D$ must pass through both these Δ-arcs. If P passes first through the Δ-arc (a_i, x_i), $P[j] \in (a_i, x_i)$, then it occurs in the $(i + 1)$th component, from which the ith component is Δ-unreachable. Hence, among the paths D that are extensions of $P[1, \ldots, j]$, there is, at least, one that does not lead to b and, thus, does not pass through the Δ-arc (b, x), which is impossible on the strength of the fact that D is a covering Δ-path of GD. If P passes first through the Δ-arc (b, x), $P[j] \in (b, x)$, then there exists a path $P' \in D$ that has first $j - 1$ arcs identical to those of P, passes through the arc $P'[j] = (b, x, c) \in (b, x)$, and leads to the component from which the ith component is Δ-unreachable. Hence, among the paths D that are extensions of $P'[1, \ldots, j]$, there is, at least, one that does not lead to a_i and, thus, does not pass through the Δ-arc (a_i, x_i), which is impossible in view of the fact that D is a covering Δ-path of GD.

(2) *Necessity follows from assertion (1).* It is required only to show that the vertex b belongs to the last component. Since all paths in D pass through each vertex c, their extensions after the first occurrence of c forming a Δ-path end at b; hence, b is Δ-reachable from any vertex c, i.e., belongs to the last component.

Sufficiency. We describe an algorithm for constructing a covering $[a, b]$-Δ-path. At the beginning of an ith step, we have a Δ-path D_i in which all paths $P \in D_i$ are initial segments of the paths from the covering Δ-path to be constructed. Each path P begins at the vertex a, ends at a vertex c (different paths may have different ends), and passes through all Δ-arcs of all components preceding the component of the path end $K(c)$. The step is applied to each path from D_i. If the current path is not a stimulus-based covering $[a, b]$-path, it is replaced by a set of paths extending it, which results in a new Δ-path D_{i+1}. The algorithm stops when all paths are stimulus-based covering $[a, b]$-paths. At the very beginning, we have one path of zero length with the initial vertex a. The algorithm step consists in the following:

1. Let there exist a path P such that all Δ-arcs originating from the vertices of the end component $K(c)$ are traversed in P. Then, in view of the step assumptions, P is a stimulus-based covering path; however, it is not necessarily terminated at b. Construct a $[c, b]$-Δ-path Q_b and, instead of the path P, take a set of all concatenations of this path with the paths from Q_b. The step assumptions clearly hold.

2. Let there exist a path P such that the vertex $d \in VK(c)$ has an outgoing Δ-arc (d, x) that has not been traversed in P yet. The priority is given to nonconnecting Δ-arcs; i.e., a connecting Δ-arc is selected only when all other Δ-arcs originating from the vertex $K(c)$ have already been traversed in P. Construct a $[c, d]$-Δ-path Q_d and, instead of the path P, take a set of all concatenations of this path with the paths from Q_d and, further, with arcs of the Δ-arc (d, x). If all arcs of the Δ-arc (d, x) end at the component $K(c)$ or the Δ-arc (d, x) is connecting, then the step assumptions obviously hold.

3. Otherwise, some arcs $(d, x, e) \in (d, x)$ end at components $K(e) \neq K(c)$. In a graph of the first kind, the component $K(e)$ precedes $K(c)$; i.e., all vertices $K(c)$ are Δ-reachable from e. Then, for each path P obtained in item 2 that ends at such a vertex e, construct an $[e, c]$-Δ-path Q_c, and, instead of the path P, take all concatenations of this path with the paths from Q_c. The step assumptions obviously hold.

Since, at each step, every path that is not a stimulus-based covering $[a, b]$-path is replaced by the paths extending it, which is followed by an increase of the number of the traversed Δ-arcs, the algorithm terminates in not more than m steps by constructing a covering $[a, b]$-Δ-path. At each step, any path is extended by not more than the length of the path Q_b (item 1) or the length of the path Q_d plus one arc (item 2) plus the length of the path Q_c (item 3). Since the path length does not exceed $n - 1$, the length of each path and, thus, the length of the constructed covering Δ-path do not exceed $m(2n - 1) = O(nm)$.

Assertion (3) immediately follows from Theorem 4.2 for strongly Δ-connected graphs.

Assertion (4) immediately follows from assertion (2) and the definition of a strongly Δ-connected graph as a graph with one strongly Δ-connected component.

4.4. Coverage of Δ-Reachable Graphs

A Δ-path is said to *cover* a vertex (Δ-arc) of a graph if each its path passes through this vertex (Δ-arc). In this sense, a covering Δ-path is simply a Δ-path that covers all vertices and Δ-arcs of the graph. The set of all Δ-paths that begin at a given initial vertex is called a Δ-coverage of the graph if any vertex and any Δ-arc of the graph are covered by, at least, one Δ-path from the set. The length of the Δ-coverage is the sum of lengths of its Δ-paths.

Theorem 4.5. A Δ-coverage exists if and only if the graph is Δ-reachable; its minimum length is equal to $O(nm)$; for any n and m, there exists a Δ-reachability graph with n vertices and $m' \geq m$ Δ-arcs any Δ-coverage of which has length $\Omega(nm')$.

Proof. The assertion of the theorem on the existence of a Δ-coverage immediately follows from the definition of a Δ-reachable graph G with an initial vertex v_0. Indeed, for any vertex v, there exists a $[v_0, v]$-Δ-path $D(v)$, and, for any Δ-arc (v, x), there is a Δ-path $D(v) \wedge (v, x)$ (the set of all concatenations of paths from $D(v)$ and arcs from (v, x)) passing through this arc.

To estimate the length of the Δ-coverage for a Δ-reachable graph G, we consider the factor graph $F(G)$, which, by Theorem 4.3, is a deterministic acyclic graph with one source $K(v_0)$. Let t be the number of the components and m_0 be the number of the connecting Δ-arcs in the graph G. Let us separate an output directed spanning tree (maximal tree) in the factor graph, the root of which is a source component. It has $t - 1$ factor arcs, and the remaining $m_0 - (t - 1)$ factor arcs are factor chords. To construct a coverage of the factor graph, it is sufficient to take the set $\mathbb{F}$ of the following factor paths: (a) all factor paths that lead from the root to the leaf factor vertices that do not have outgoing factor chords and, additionally, (b) all factor paths that lead from the root to the beginnings 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 $m_0 - (t - 1)$. Thus, the total number of all factor paths is not greater than $m_0 + 1$.

Any factor path $F \in \mathbb{F}$ may be associated with an alternating sequence of the components of the graph G (beginnings and endpoints of the factor arcs of F) and the connecting Δ-arcs of G (factor arcs of F): $K(v_1)$, (v_1, x_1), $K(v_2)$, (v_2, x_2), …, (v_f, x_f), $K(v_{f+1})$ with $K(v_1) = K(v_0)$. Each, but the last, component $K(v_i)$ contains a connecting Δ-arc (v_{i-1}, x_{i-1}) from the previous component, and each, but the last, component $K(v_i)$ has one outgoing connecting Δ-arc (v_i, x_i) that leads to the next component. In each component $K(v_i)$ (except for the first and last), for any incoming arc $(v_{i-1}, x_{i-1}, v'_{i-1}) \in (v_{i-1}, x_{i-1})$, we select a $[v'_{i-1}, v_i]$-Δ-path $D_i(v'_{i-1})$ from the endpoint of the incoming arc to the beginning of the outgoing Δ-arc. In the last component, such a Δ-path $D_{f+1}(v'_f)$ can be terminated at any vertex; in the first component, the Δ-path D_0 begins at the initial vertex v_0. For $i > 0$, we denote the union of these Δ-paths as $D_i = \cup\{D_i(v'_i) | (v_i, x_i, v'_i) \in (v_i, x_i)\}$. Replacing each occurrence of a component by the corresponding set of paths, we associate the factor path F with a set of all possible alternating concatenations of sets of paths and the connecting Δ-arcs, $D(F) = D_0 \wedge (v_1, x_1) \wedge D_1 \wedge (v_2, x_2) \wedge \ldots \wedge (v_f, x_f) \wedge D_{f+1}$, which is obviously a Δ-path beginning at the vertex v_0.

Now, for each component K of the graph G, we select one factor path F that passes through this component $K = K(v_i)$. In the Δ-path $D(F)$, we replace each Δ-path $D = D_i(v'_{i-1})$ in the component K by a covering Δ-path of the component with the same beginning and endpoint. The desired Δ-coverage is the set of all such Δ-paths, $D = \{D(F) | F \in \mathbb{F}\}$. If all Δ-paths $D_i(v'_{i-1})$ remained simple Δ-paths, each Δ-path $D(F)$ would also be a simple Δ-path or a simple Δ-path extended by one Δ-arc (factor chord) and, hence, its length would not exceed n, and the sum of lengths would not exceed $(m_0 + 1)n$. Since, for each ith component of G, only one

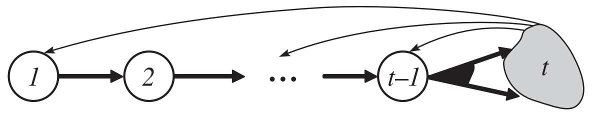

Fig. 4.

path F contains the covering Δ-path of this component, the Δ-coverage length does not exceed $(m_0 + 1)n + \Sigma\{O(n_i m_i) | i = 1, \ldots, f\} = O(nm)$, where n_i and m_i are the numbers of vertices and Δ-arcs in the ith component, respectively.

The estimate $\Omega(nm')$ is reached on the graph depicted in Figs. 1 and 2 with the initial vertex v_1.

5. Δ-TRAVERSAL ALGORITHMS

5.1. Graphs of the Second Kind and Free Algorithms

For a path in a graph, a Δ-arc is said to be traversed if, at least, one arc from this Δ-arc is traversed; a vertex is said to be *completely traversed* if all outgoing Δ-arcs of this vertex are traversed.

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 Δ-arc leading to the next component (Fig. 4). Since all, but the last, components consist of one vertex, all connecting Δ-arcs, but the last one, consist of one arc. In the deterministic case, the above definition of a graph of the second kind coincides with that in [1].

Theorem 5.1. A Δ-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.

Proof. We will prove the assertion by *reduction ad absurdum.* By Theorem 4.4, a covering Δ-path exists only for a graph of the first kind. 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 Δ-connecting arc. In the former case, by virtue of the strong Δ-connectivity of the component, there is a Δ-path from the beginning of the connecting Δ-arc to some other vertex of the component; hence, in addition to the connecting Δ-arc (a, x), there is another Δ-arc (a, x') going from the beginning of the connecting Δ-arc. In the latter case, the existence of two such Δ-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 Δ-traversal must be guaranteed, the path being passed is to be a stimulus-based covering path, independent of the result of the operation *nextcall*. Suppose that the operation *nextcall* chooses the stimulus x.

Then, we pass along an arc from the connecting Δ-arc (a, x) to a vertex in the next component, from which there is no Δ-path leading to the vertex a. Thus, the Δ-arc (a, x') remains untraversed, and we arrived at the contradiction.

5.2. A Free Optimal-Complexity Algorithm

Theorem 5.2. There exists a free algorithm $\mathbb{B}_1$ that stops on any graph and completes guaranteed Δ-traversal of any graph of the second kind with an initial vertex belonging to the first component. The length of the traversed path is $O(nm)$. The algorithm operation time depends of the comparison operations used for the vertex identifiers. If only equality comparisons (matchings) are involved, the algorithm operation time is $O(n^2m)$; if greater/less comparisons are also used, then the time is $O(nm\log_2 n)$. The required memory is $O(nI + mX + m\log_2 m)$, where I and X are the sizes of the vertex identifier and stimulus, respectively, in bits.

First, we define some terms and discuss a general idea of the algorithm.

In a deterministic graph, the distance from a vertex a to a vertex b is the length of a minimal $[a, b]$-path, and the distance from a to a *set* of vertices B is defined as a minimal distance from a to vertices $b \in B$. Similarly, in the nondeterministic case, we can introduce a *Δ-distance* $\rho(a)$ *from a vertex a* to a set B as the length of a minimal $[a, B]$-Δ-path beginning at a and ending in B (i.e., ends of all of its paths belong to B). In a strongly Δ-connected graph, $[a, B]$-Δ-paths always exist, and, obviously, the minimal path among them is a Δ-path; hence, $\rho(a) \le n - 1$. A *Δ-distance of a Δ-arc without loops* $\rho(a, x)$ is defined as the maximum of the Δ-distances of the endpoints of its arcs increased by one, $\rho(a, x) = \max\{\rho(c) + 1 | (a, x, c) \in (a, x)\}$. It is evident that, for $a \in B$, $\rho(a) = 0$, and, for $a \notin B$, $\rho(a) = \min\{\rho(a, x)\}$, where (a, x) runs through all Δ-arcs without loops that originate from the vertex a.

For a free algorithm that, at a certain moment, traversed a path P, an *unmarked* vertex is the vertex at which the operation *nextcall* returned the empty symbol ε, i.e., the vertex that is "known" to be completely traversed. In what follows, Δ-distances of vertices and Δ-arcs are always assumed to be measured from the set of *unmarked* vertices.

If, at any time, the algorithm knew Δ-distances of all vertices and Δ-arcs, it could work by the following simple scheme:

1. If all traversed vertices are marked, the algorithm stops.

2. Otherwise, move from unmarked vertices along untraversed Δ-arcs by means of the operation *nextcall* until occur in a marked vertex or *nextcall* returns the empty symbol ε and the current vertex becomes marked.

3. If the marked vertex does not have outgoing Δ-arcs for which the Δ-distances are determined, the algorithm stops.

4. Otherwise, move along the outgoing Δ-arc with the least Δ-distance until occur in an unmarked vertex or in a marked vertex that does not have such Δ-arcs.

In item 4, the algorithm passes along a path no longer than $n - 1$ and, if occurs in an unmarked vertex, passes a new Δ-arc in item 2. Thus, until it stops (in item 1 or 3), the algorithm will pass a path of length $O(nm)$. If the graph is strongly Δ-connected, the situation described in item 3 is impossible, and the algorithm will stop in accordance with item 1, after the whole graph has been traversed by stimuli. In the case of a graph of the second kind with an initial vertex belonging to the first component, the algorithm, first, moves along the connecting Δ-arcs until it occurs in the last component, which is traversed then by stimuli. However, in this case, vertices of (not last) components will remain unmarked, and, although all Δ-arcs have been traversed with guarantee, the algorithm will stop at item 3.

Obviously, such an algorithm is redundant. Its irredundant version can be created if we replace the Δ-distances calculated over the entire graph by the Δ-distances calculated over the *traversed* graph. These Δ-distances are to be recalculated every time when a new arc is traversed. The length of the stimulus-based traversal is, as before, $O(nm)$; however, the algorithm operation time will increase considerably.

The basic idea of the algorithm described below consists in the use of a local approximation of the Δ-distance, which will be called a *rank* and denoted by $r(v)$ and $r(v, x)$, for a vertex v and a Δ-arc (v, x), respectively. In the course of the algorithm operation, ranks may only be increased. If the rank of a vertex becomes equal to or greater than the number of the traversed vertices, the algorithm stops. This may happen only if the graph is not strongly Δ-connected.

The algorithm manages a list of the traversed vertices. For each traversed vertex v, there is a unidirectional list of ranks of the outgoing Δ-arcs without loops in the increasing order of their ranks. For each rank r, there is a bidirectional list of the traversed Δ-arcs without loops that begin at the vertex v and have rank r.

The algorithm uses the following data structures:

• List of the traversed vertices. The descriptor of vertex v contains

◦ a vertex identifier (returned by the operation *status*),

◦ a marked vertex attribute (the operation *nextcall* returned the empty symbol ε), and

◦ a reference to the first element of the rank list.

• List of ranks of the traversed Δ-arcs without loops that originate from vertex v. The descriptor of rank r contains

◦ the rank value,

◦ a reference to the rank list, and

◦ a reference to the first element of the list of Δ-arcs.

• List of the traversed Δ-arcs without loops that originate from vertex *v* and have rank *r*. The descriptor of the Δ-arc (*v*, *x*) contains

◦ a stimulus of the Δ-arc,

◦ a "forward" reference in the list of Δ-arcs, and

◦ a "backward" reference in the list of Δ-arcs.

• *N* is a counter of the traversed vertices.

• *L* is a counter of the marked vertices.

• *v* is a reference to the descriptor of the current vertex.

The rank of an unmarked vertex is considered to be equal to zero, and the rank of a marked vertex is equal to the maximal rank of the outgoing Δ-arcs without loops, i.e., to the first rank in the list of ranks of the outgoing traversed Δ-arcs without loops. If the list of ranks for a marked vertex is empty, then the vertex rank is not defined.

At the beginning of the algorithm operation, lists of vertices and the corresponding lists of ranks and Δ-arcs are empty; both counters are set equal to zero; and the reference to the current vertex *v* is also empty. Using the operation *status*, the algorithm determines the identifier of the initial vertex *v* and creates a descriptor for it with an empty list of ranks and the vertex being unmarked. The counter of the traversed vertices is set equal to one.

The further operation of the algorithm consists in the execution of a sequence of steps. Each step contains one call of the operation *nextcall* or *call*. As a result of this operation, either one arc (*v*, *x*, *v*') is traversed, or the current unmarked vertex *v* is marked (*nextcall* returns the empty symbol ε). We will use the prime symbol to denote the ranks of vertices and Δ-arcs, the current vertex, and the values of the counters at the end of a step, i.e., *r*', *v*', *N*', and *L*'. The algorithm step is as follows:

1. The current vertex *v* is unmarked. Call the operation *nextcall*.

A. *nextcall* returns the empty symbol ε. Mark the current vertex and increase the counter of the marked vertices: $L' = L + 1$.

a. If the values of the counters of the traversed and marked vertices are equal each other, $N' = L'$, the algorithm stops (*stop 1*).

b. Otherwise, if $N' > L'$, analyze the rank of the vertex *v*.

I. If the list of ranks of the Δ-arcs is empty (i.e., $r'(v)$ is not defined) or $r'(v) \geq N$, the algorithm stops (*stop 2*).

II. Otherwise, *the end of Step 1.*

B. *nextcall* performs a transition along an untraversed Δ-arc (*v*, *x*) and returns its stimulus *x*. By means of the operation *status*, determine the identifier of the new current vertex *v*', i.e., the end of the traversed arc (*v*, *x*, *v*'). By means of this identifier, seek for *v*' in the list of descriptors of the traversed vertices.

a. If the vertex *v*' is not found (a new one), create the descriptor for it: the list of ranks is empty, and the vertex is unmarked. Increase the counter of the traversed vertices, $N' = N + 1$. Create the descriptor of the Δ-arc (*v*, *x*) and place the arc to the list of Δ-arcs of rank 1 originating from the vertex *v*. If required, create the descriptor of rank 1 for the vertex *v*. *The end of Step 2.*

b. If the vertex *v*' is not found (an old one), check whether the traversed arc (*v*, *x*, *x*') is a loop.

I. It is a loop, $v' = v$. *The end of Step 3.*

II. It is not a loop, $v' \neq v$. The rank of the Δ-arc (*v*, *x*) is set equal to $r'(v, x) = r(v') + 1$. Create the descriptor of the Δ-arc and place it to the list of Δ-arcs of rank $r'(v, x)$. If required, create the descriptor of rank $r'(v, x)$ for the vertex *v*. *The end of Step 4.*

2. The current vertex *v* is marked. Select the Δ-arc (*v*, *x*) originating from *v* with the least rank. For the stimulus *x* of the selected Δ-arc, execute *call*(*x*). By means of the operation *status*, determine the identifier of the new current vertex *v*', i.e., the end of the traversed arc (*v*, *x*, *v*'). By this identifier, seek for *v*' in the list of the descriptors of the traversed vertices.

A. If the vertex *v*' is not found (a new one), create the descriptor for it: the list of ranks is empty, and the vertex is unmarked. Increase the counter of the traversed vertices, $N' = N + 1$. *The end of Step 5.*

B. If the vertex *v*' is not found (an old one), check whether the traversed arc (*v*, *x*, *x*') is a loop.

a. It is a loop, $v' = v$. The descriptor of the Δ-arc (*v*, *x*) is extracted from the list of Δ-arcs of rank $r(v, x)$. If the list of Δ-arcs of rank $r(v, x)$ became empty, remove the descriptor of this rank from the list of ranks and delete it. Correct the reference from the descriptor of the vertex *v* to the descriptor of the next rank. If the list of ranks is empty, this reference will also be empty. Analyze the rank of the vertex *v*.

I. If the list of ranks of Δ-arcs is empty, i.e., $r'(v)$ is not defined, or $r'(v) \geq N$, then the algorithm stops (*stop 2*).

II. Otherwise, *the end of Step 6.*

b. It is not a loop, $v' \neq v$. Compare the rank $r(v')$ with the rank of the beginning of the traversed arc $r(v) = r(v, x)$.

I. $r(v) > r(v')$. The ranks of the Δ-arc (*v*, *x*) and the vertex *v* have not been changed. *The end of Step 7.*

II. $r(v) \leq r(v')$. The rank of the Δ-arc (*v*, *x*) is increased: $r'(v, x) = r(v') + 1$. The descriptor of the Δ-arc is removed from the list of Δ-arcs of rank $r(v, x)$ and placed to the list of Δ-arcs of rank $r'(v, x)$. If required, create the descriptor of this rank. If the list of Δ-arcs of rank $r(v, x)$ is empty, delete the descriptor of this rank and modify the reference from the descriptor of the vertex *v* to the next rank in the list of ranks. Analyze the rank of the vertex *v*.

(i) If the list of ranks of Δ-arcs is empty, i.e., $r'(v)$ is not defined, or $r'(v) > N$, the algorithm stops (*stop 2*).

(ii) Otherwise, *the end of Step 8.*

Lemma 5.1. The rank of a traversed Δ-arc without loops is less than or equal to the maximum rank of the endpoints of its traversed arcs plus one.

Proof. When the rank of a Δ-arc is set for the first time (items 1.B.a and 1.B.b.II) or corrected afterwards (item 2.B.b.II), it becomes equal to the maximum of the determined ranks of the endpoints of its traversed arcs increased by one. Since ranks of the Δ-arcs and, thus, ranks of vertices do not decrease, the ranks of the endpoints of the arcs belonging to the Δ-arc do not decrease between the corrections of its rank. The lemma is proved.

Lemma 5.2. If there is an unmarked vertex in a strongly Δ-connected component K, then, for any vertex $a \in VK$ of rank $r(a) > 0$, there exists a vertex $b \in VK$ of rank $r(b) = r(a) - 1$.

Proof. Let us assume the contrary. Let, for some vertex $a \in VK$ of rank $r(a) > 0$, the rank of any vertex $b \in VK$ satisfy the condition $r(b) \neq r(a) - 1$. Consider the set H of vertices that have rank equal to or greater than $r(a)$. Since K contains unmarked vertices of rank 0, H is a nonempty proper subset of VK. Since K is a strongly Δ-connected component, H cannot be Δ-isolated; i.e., there exists a Δ-arc (v, x) with the beginning $v \in H$ the endpoints of all arcs of which belong to $VK \backslash H$. It is evident that this Δ-arc is traversed (the rank of its beginning v is greater than zero), and the endpoints of all of its traversed arcs have ranks lesser than $r(a) - 1$. Then, by Lemma 5.1, $r(v, x) \leq r(a) - 1$ and $r(v) \leq r(v, x) \leq r(a) - 1$, which contradicts the condition $v \in H$. The lemma is proved.

Lemma 5.3. When the algorithm stops at a vertex v, the strongly Δ-connected component $K(v)$ is traversed by stimuli (all Δ-arcs originating from its vertices are traversed).

Proof. If the algorithm termination corresponds to *stop 1*, then all traversed vertices are marked.

If the termination corresponds to *stop 2*, then all traversed vertices of the component $K(v)$ are marked. Indeed, this may happen in two following cases:

(1) all Δ-arcs originating from v have loops, or

(2) the vertex v has too large rank, $r(v) \geq N$.

In the first case, v is the only vertex in $K(v)$, and it is completely traversed. In the second case, clearly, $r(v) \geq N(v)$, where $N(v)$ is the number of the traversed vertices of the component $K(v)$. If there were unmarked vertices in $K(v)$, then, by Lemma 5.2, $K(v)$ would contain vertices with the ranks $r(v), r(v) - 1, \ldots, 0$. In this case, we would have $N(v) \geq r(v) + 1 \geq N(v) + 1$, which is impossible.

Thus, in both cases of the algorithm termination, all traversed vertices of the component $K(v)$ are marked; i.e., all Δ-arcs originating from them are traversed. If there were untraversed Δ-arcs in $K(v)$, then the set of the traversed vertices $K(v)$ would be a Δ-isolated nonempty proper subset of the set $VK(v)$, which is not the case. Hence, $K(v)$ has been traversed by stimuli.

Guaranteed Δ-traversal of a graph of the second kind with an initial vertex belonging to the first component. In such a graph, the algorithm passes a simple path leading from the initial vertex to the last component. If it stops, then, by Lemma 5.3, it has traversed the graph by stimuli. The fact that the algorithm stops on any graph is proved below.

Lemma 5.4. The rank of a vertex is less than n, and the rank of a Δ-arc does not exceed n.

Proof. The rank of a vertex before the algorithm stops is less than N (*stop 2*). Since $N \leq n$, the vertex rank is less than n. On the strength of Lemma 5.1, the rank of a Δ-arc does not exceed n.

The algorithm stops on any graph, with the length of the traversed path being estimated as $O(nm)$. Let the rank of a Δ-arc with a loop be defined equal to n. Consider the sum $S = R_\Delta - r(v) + nL$, where R_Δ is the sum of ranks of the traversed Δ-arcs (including Δ-arcs with loops) and $r(v)$ is the rank of the current vertex. Taking into account that $L \leq N \leq n \leq m$ and using Lemma 5.4, we find that $S = O(nm)$. At each step, the algorithm traverses not more than one arc. Therefore, to prove the assertion, it is sufficient to show that, at each step (but, perhaps, the last one), the sum S is increased by, at least, one; i.e., $S' \geq S + 1$. Let us show that this is true for all eight kinds of the step termination.

Step termination 1. $R'_\Delta = R_\Delta$, $v' = v$, $r'(v) < n$, $r(v) = 0$, $L' = L + 1$. Therefore, $S' - S = (-r'(v) + r(v)) + n \geq 1$.

Step termination 2. $R'_\Delta = R_\Delta + 1$, $r'(v') = r(v) = 0$, $L' = L$. Therefore, $S' - S = 1$.

Step termination 3. $R'_\Delta = R_\Delta + r'(v, x)$, $r'(v, x) = n$, $v' = v$, $r'(v) = r(v)$, $L' = L$. Therefore, $S' - S = n \geq 1$.

Step termination 4. $R'_\Delta = R_\Delta + r'(v, x)$, $r'(v, x) = r(v') + 1$, $r'(v') = r(v')$, $r(v) = 0$, $L' = L$. Therefore, $S' - S = r'(v, x) - r'(v') = 1$.

Step termination 5. $R'_\Delta = R_\Delta$, $r'(v') = 0$, $r(v) > 0$, $L' = L$. Therefore, $S' - S = r(v) - r'(v') \geq 1$.

Step termination 6. $R'_\Delta = R_\Delta + r'(v, x) - r(v, x)$, $r'(v, x) = n$, $r(v, x) = r(v) < n$, $v' = v$, $r'(v') < n$, $r(v) > 0$, $L' = L$. Therefore, $S' - S = r'(v, x) - r(v, x) + r(v) - r'(v') = n - r(v) + r(v) - r'(v') \geq 1$.

Step termination 7. $R'_\Delta = R_\Delta$, $r'(v') = r(v')$, $r(v) > r'(v')$, $L' = L$. Therefore, $S' - S = r(v) - r'(v') \geq 1$.

Step termination 8. $R'_\Delta = R_\Delta + r'(v, x) - r(v, x)$, $r'(v, x) = r(v') + 1$, $r(v, x) = r(v) < n$, $r'(v') = r(v')$, $r(v') \leq r(v')$, $L' = L$. Therefore, $S' - S = r'(v, x) - r(v, x) + r(v) - r'(v') = r(v') + 1 - r(v) + r(v) - r(v') = 1$.

Algorithm operation time. At each step, the numbers of elementary operations required are not bounded by a constant (depend on the graph characteristics) for only two following tasks: (1) searching for the identifier of a vertex (the end of the traversed arc) among the identifiers of the traversed vertices and (2) moving Δ-arcs from one list to another when their ranks are increased. As to (2), it suffice to note that, since the rank of a Δ-arc is not decreased and does not exceed n, it is totally (over all steps) moved by not more than n positions. Therefore, not more than $O(nm)$ elementary operations (over all steps) are required to move all Δ-arcs.

Time required for item (1) depends on the comparison operations defined for the vertex identifiers. If only the matching operation is defined, the number of the comparison operations required for each search is $O(n)$. Multiplying it by the number of searches $O(nm)$, we find that the total number of matching operations is $O(n^2m)$. If the comparison operations ("greater/less") are also defined, then, instead of a simple list of vertex descriptors, we can organize a balanced tree of such descriptors. Each search in such a tree requires $O(\log_2 n)$ comparisons; multiplying it by the number of searches $O(nm)$, we obtain the total number of operations $O(nm\log_2 n)$. The correction of a tree when a new vertex is added requires $O(\log_2 n)$ operations; the number of such corrections is $O(n)$; and the total number of operations is $O(n\log_2 n)$.

Required memory. Since the rank does not exceed n, the rank value and a reference in the rank list occupy $O(\log_2 n)$ bits. A reference in the list of Δ-arcs occupies $O(\log_2 m)$ bits. Let I and X be sizes of the vertex identifier and stimulus, respectively, in bits. Then, the vertex descriptor occupies $O(I + \log_2 n)$ bits; the rank descriptor, $O(\log_2 n + \log_2 m)$ bits; and the Δ-arc descriptor, $O(X + \log_2 m)$ bits. Since $n \le m - 1$ and each rank descriptor is associated with a nonempty list of Δ-arcs, the total estimate is $O(nI + m_X + m\log_2 m)$ bits. If a balanced tree of vertex descriptors is supported, then each reference in a tree requires $O(\log_2 n)$ bits of memory, and the total amount of memory is $O(n\log_2 n)$, which does not change the order of the estimate.

This completes the proof of Theorem 5.2.

5.3. Modifications of the Algorithm

An irredundant version $\mathbb{B}_2$ of the free algorithm $\mathbb{B}_1$ can easily be constructed. To this end, a vertex is marked when the algorithm leaves the vertex by the last untraversed Δ-arc originating from this vertex. Now, let us see what correct verdicts can the free algorithm $\mathbb{B}_1$ and the irredundant algorithm $\mathbb{B}_2$ return? If there are no unmarked vertices when any of these algorithms stops, the set of the traversed vertices is Δ-isolated, and the algorithm has completed a stimulus-based traversal of the graph. For $\mathbb{B}_1$, the following verdict is correct: "*if the graph is strongly Δ-connected, then the guaranteed stimulus-based traversal has been completed*" (recall that only Δ-reachable graphs are considered). $\mathbb{B}_2$ returns the following, stronger, verdict: "*if the graph is a graph of the second kind, then the guaranteed stimulus-based traversal has been completed.*" If there remained unmarked vertices at the moment when the algorithm stops, the free algorithm $\mathbb{B}_1$ does not know whether they have been completely traversed. It may state only that the strongly Δ-connected component of the traversed graph has been traversed by stimuli, that the algorithm stopped at this component, and that the set of vertices of this component is Δ-isolated. On the other hand, the irredundant algorithm $\mathbb{B}_2$ knows for sure that the stimulus-based traversal has not been completed.

To return more accurate verdicts, the traversed graph is to be analyzed. In the above-discussed version of the algorithm, the traversed graph was not saved: we stored only the traversed vertices and stimuli of the traversed Δ-arcs. The algorithm can easily be modified in order to keep information about all traversed arcs. This will not change the order of the path length and the algorithm operation time, but will certainly increase the required amount of memory.

The algorithms discussed can be used also for constructing Δ-coverage of any Δ-reachable graphs, which can be achieved by repeatedly running the algorithms and saving information on the results of the previous runs. In other words, the algorithms can work with graphs for which the reliable operation *reset* [2] is defined. This operation is used only when needed instead of *stop 2* (in particular, it is not used on strongly Δ-connected graphs). The algorithm $\mathbb{B}_2$ can be modified such that the modified version (denote it as $\mathbb{B}_3$) can perform guaranteed stimulus-based traversals of graphs of the first kind if it receives somehow information about the connecting Δ-arcs. Let a predicate $\pi(x)$ of stimulus, which is further referred to as the *predicate of the connecting* Δ-*arcs*, be given at each vertex of the graph. The predicate is said to be *correct* if it is true on the stimuli of the connecting Δ-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. The algorithm $\mathbb{B}_3$ differs from $\mathbb{B}_2$ in that it passes first only nonconnecting Δ-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 Δ-arcs any more and all vertices are marked, the algorithm considers the beginnings of the connecting Δ-arcs as unmarked and, accordingly, corrects ranks of the vertices and Δ-arcs (for this purpose, it looks through the traversed graph again).

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 $\mathbb{B}_3$, we obtain the irredundant algorithm (denote it as $\mathbb{B}_4$) that can perform guaranteed stimulus-based traversal of all graphs of the second kind and those of the first kind for which the predicate is correct. It should be noted, however, that an irredundant predicate cannot be correct, of course, on all graphs with given initial vertices v_0 that are isomorphic up to coloring of Δ-arcs by stimuli if these are not graphs of the second kind.

Another field of the application of the algorithm with the predicate (both redundant and irredundant) of the connecting Δ-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 Δ-connected. In stricter terms, a two-level graph is a graph some Δ-arcs of which are marked. Removing marked Δ-arcs, we obtain a set of isolated strongly Δ-connected graphs (first-level graphs). By factoring a two-level graph with respect to the mutual Δ-reachability of vertices through unmarked Δ-arcs, we obtain a strongly Δ-connected graph of the second level. If the predicate is meant to be the predicate of the marked Δ-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 stimuli of the unmarked Δ-arcs; then, goes to the next first-level graph by the marked Δ-arc. When the algorithm enters the first-level graph next time, it passes only the simple path till the required marked Δ-arc that originates from this graph. 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. The length of the algorithmic traversal of a *p*-level graph is, in many cases, almost optimal, which is less than $O(nm)$

6. CONCLUSIONS

The free and irredundant algorithms of graph stimulus-based traversal discussed in this paper and tests based on them have been developing and testing since 1995 by the group RedVerst [15] in the course of the execution of several large-scale projects on testing various software [16, 17]. 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 algorithm of Δ-traversal in terms of time and memory required is critical. A more important characteristic is the number of test actions, i.e., the length of the Δ-path constructed. The free algorithm $\mathbb{B}_1$ and its irredundant modification $\mathbb{B}_2$, which have nice complexity estimates, ensure only that the length of the Δ-path in the worst case has the minimal order *nm*. At the same time, for many graphs with the minimal covering Δ-path length much less than *nm*, they construct as lengthy covering Δ-paths as in the worst case. The study of irredundant algorithms that tend to construct covering Δ-paths of minimal length is a promising problem for future researches. Similar results for deterministic graphs can be found, e.g., in [18].

REFERENCES

1. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case, *Programmirovanie*, 2003, no. 5, pp. 11–30.
2. 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.
3. Von Bochmann, G. and Petrenko, A., Protocol Testing: Review of Methods and Relevance for Software Testing, *Proc. of ISSTA*, 1994, pp. 109–124.
4. Burch, J.R., Clark, E.M., McMillan, K.L., and Dill, D.L., Sequential Circuit verification Using Symbolic Model Checking, *Proc. of the Design Automation Conf.*, 1990, pp. 46–51.
5. Cabodi, G., Lavagno, L., Macci, E., Poncino, M., Quer, S., Camurati, P., and Sentovicha, E., Enhancing FSM Traversal by Temporary Re-Encoding, *Proc. of IEEE Int. Conf. on Comput. Design*, 1996, pp. 6–11.
6. Swamy, G.M., Singhal, V., Brayton, R.K., Incremental methods for FSM Traversal, *Techn. Report*, Electronics Research Lab., Univ. of California, Berkeley, 1995.
7. Luo, G., Petrenko, A., and von Bochmann, G., Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-Method, *IEEE Trans.*, 1994,vol. SE-20, no. 2.
8. Barnett, M., Nachmanson, L., and Schulte, W., Conformance Checking of Components against Their Non-Deterministic Specifications, *Microsoft Research Techn. Report MSR-TR-2001-56.*
9. 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.
10. Fujiwara, S. and von Bochmann, G., Testing Nondeterministic Finite State Machine with Fault Coverage, *Proc. of IFIP TC6 Fourth Int. Workshop on Protocol Test Systems*, (1991), Kroon, J., Hei-jink, R.J., and Brinksma, E., Eds., North-Holland, 1992, pp. 267–280.
11. Milner, R., A Calculus of Communicating Systems, in *Lecture Notes in Computer Science*, Berlin: Springer, 1980, vol. 92.
12. Tabourier, M., Cavalli, A., and Ionescu, M., A GSM-MAP Protocol Experiment Using Passive Testing, *Proc. of the World Congr. on Formal Methods in Devel-*

opment of Computing Systems (FM'99), Toulouse, 1999.

13. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Use of Finite Automata for Program Testing, *Programmirovanie*, 2000, no. 2, pp. 12–28.
14. Gurevich, Yu., Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Trans. Computational Logic*, 2000, vol. 1, no. 1, pp. 77–111.
15. http://www.ispras.ru/RedVerst/.
16. 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.
17. 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.
18. Albers, S. and Henzinger, M.R., Exploring Unknown Environments, *SIAM J. Comput.*, 2000, vol. 29, no. 4, pp. 1164–1188.