ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 1, pp. 1–13. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © I.B. Bourdonov, A.S. Kossatchev, V.V. Kulyamin, 2015, published in Programmirovanie, 2015, Vol. 41, No. 1.

Parallel Computations on a Graph

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

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

Received September 7, 2014

Abstract—The problem of parallel computation of the value of a function of multiset of values recorded at the vertices of a directed graph is considered. Computation is performed by automata that are located at the vertices of the directed graph and exchange messages with each other that are transmitted along the arcs of the graph (in the direction of their orientation). It is assumed that the arc capacity, that is, the number of messages that can be simultaneously transmitted through an arc, is limited. Computation is initiated by a message coming from outside to the automaton located at the initial vertex of the graph. At the end of work, the same automaton sends outside the calculated value of the function. Two algorithms are proposed to solve this problem. The first algorithm carries out an analysis of the graph based on its traversal. Its purpose is to mark the graph by a change of the states of the automata at the vertices. Direct and back spanning trees of a graph are constructed. The direct spanning tree is oriented from the root, which is the initial vertex of the graph. The back spanning tree is oriented to the root. In addition, at each vertex, a value of the counter of incoming arcs of the back spanning tree is set. Such marking is used by the second algorithm, which calculates the value of one or other function. This calculation is based on a pulsation algorithm: first, *request messages* are distributed from the automaton of the initial vertex over the graph, which should reach each vertex, and then *response messages* are sent from each vertex back to the initial vertex. In fact, the pulsation algorithm calculates aggregate functions for which the value of a function of a union of multisets is calculated by the values of the function of these multisets. However, it is shown that any function $f(x)$ has an aggregate extension; that is, an aggregate function can be calculated as $h(f'(x))$, where f' is an aggregate function. Note that the marking of a graph does not depend on a function that will be calculated. This means that the marking of a graph is carried out once; after that, it can be reused for calculating various functions.

DOI: 10.1134/S0361768815010028

1. INTRODUCTION

The study of directed graphs is a root problem in many applications. It suffices to mention the study of communication networks, including Internet and GRID networks, and testing software and hardware systems modeled by transition graphs. The study of a graph, as a rule, is based on its traversal, which is an old classical problem of traversal of a labyrinth. This problem is nontrivial if the graph is directed, that is, in the case of a one-way streetå labyrinth.

The traversal of a directed graph takes time on the order of nm, where n is the number of vertices of the graph and m is the number of its arcs. Such traversal time is reached by many well-known algorithms: depth-first traversal and width-first traversal algorithms, a greedy algorithm, etc. [1, 2, 3].

In 1976, Rabin set up the problem of traversal of a directed graph by a finite automaton [4]. The automaton on the graph is similar to Turing's machine: to a cell of a tape, there corresponds a vertex of the graph, and motion to the left or to the right along the tape corresponds to a transition along one of the arcs emanating from a current vertex of the graph. To date, the fastest algorithm has been proposed in [5]; it has an estimate $nm + n^2\log\log n$. Under a repeated traversal when the automaton can use the marks left by it after the first traversal, the estimate decreases to $nm + n^2 l(n)$, where $l(n)$ is the number of operations of taking the logarithm for which the relation $1 \le \log(\log...(n)...) < 2$ is satisfied [6]. The difference from the lower bound nm is associated with the fact that automaton has to "return" to the beginning of the just traversed arc.

In recent years, the size of actually used systems and networks and, hence, the size of the graphs studied has been continuously growing. Problems arise when the study of a graph by one automaton (computer) either requires inadmissibly large time, or the graph is not located in the memory of one computer, or both. Therefore, the problem arises of a parallel and distributed study of graphs. This problem is formalized as the study of a graph by a collective of automata.

In [7] and [8], the authors proposed operation algorithms for such a collective of automata. It was assumed that automata cannot record anything at vertices of the graph or read from them, but they can exchange messages with each other by means of a communication network orthogonal to the graph, as

well as generate new automata. The best estimate obtained is $m + nD$, where D is the diameter of the graph, i.e., the length of the maximum simple path (a path without self-intersections) in the graph.

In this work, we consider a classical problem of analyzing a graph by automata the exchange of information between which occurs only through the memory of the graph vertices. This is equivalent to studying a graph by means of messages which the automata fixed at the vertices of the graph exchange with each other, and the arcs of the graph play the role of message transmission channels. An automaton at a vertex sends a message along one of arcs emanating from this vertex, and, in some time, this message is received by the automaton at the end of the arc. An estimate for the operating time of the algorithm depends on the number of messages that can be simultaneously transmitted along an arc. We will call this number the *arc capacity* and denote it by k.

Both the algorithms for the analysis of a graph and an estimate for their operating time significantly depend on whether the automata at the vertices of the graph have any information on the graph or each automaton is in its initial state and "knows nothing" about the graph. In this work, we propose two algorithms. The first algorithm carries out a primary traversal of the graph by means of transmitted messages when at the initial instant of time all automata are in their initial states. After completing the traversal, the automata are in such final states that allow one to perform a parallel calculation of the required function by the second algorithm. The traversal of the graph is performed in time of about $n/k + D$, and the calculation of the function, in time of about D.

The second algorithm is a *pulsation algorithm*, which is based on the fact that, first, *request messages*, which should reach each vertex, propagate from the automaton of a distinguished initial vertex along the whole graph. Then *response messages* propagate from each vertex back to the initial vertex. Using the pulsation algorithm, one can calculate in parallel any function of a multiset of values recorded in the memory of the automata over all vertices of the graph (we will say "recorded at vertices"). Here are some examples of such functions: 1. The maximum of numbers recorded at the vertices of a graph. 2. More generally, instead of the maximum, one can use any commutative and associative operation over numbers: a minimum, addition, product, etc. 3. Special cases: the number of vertices in the graph if "1" is recorded at each vertex, and the number of arcs in the graph if the number of outgoing arcs is recorded at each vertex. 4. Disjunction of logical values recorded at the vertices of a graph. 5. More generally, instead of a disjunction, one can use any commutative and associative operation on logical values: conjunction, equivalence, etc. 6. Even more generally, instead of numbers or logical values, one can use any values and any commutative and associative operations on them. 7. Arithmetic mean, geometric mean, or the root mean square of the numbers recorded at the vertices of the graph.

2. DESCRIPTION OF THE TRAVERSAL ALGORITHM

Whenever it does not lead to confusion, we will say a "vertex" for short, meaning "the automaton of a vertex" or "the memory of the automaton of a vertex." While sending a message, the automaton at a vertex has to specify an arc outgoing from the vertex along which the message is sent. We will assume that the arcs outgoing from a vertex are numbered starting with 1, and the automaton specifies the arc number.

Suppose that the number of arcs in a graph is m, the number of vertices is n, the length of the maximum simple path (a path without self-intersections) is D, the number of arcs outgoing from a vertex does not exceed s_0, and the number of arcs incoming to a vertex does not exceed s_1. Obviously, $m \le ns_0$. We propose a graph traversal algorithm with the following characteristics:

- the memory of the automaton at a vertex $O(nD\log s_0)$,
- the message size $O(D\log s_0)$,
- the arc capacity k,
- the operating time of the algorithm $O(n/k + D)$.

Remark 2.1. It is obvious that simultaneous sending of $k' \le k$ messages with a size of z bits is equivalent to sending a single message of size $k'z$ bits, or, generally, to sending k_1 messages of size k_2z, where $k_1k_2 = k$. Thus, an estimate for the operating time of the algorithm for a message size of z bits and the arc capacity of k messages coincides with the estimate for the operating time of the algorithm for a message size of k_1z bits and the arc capacity of k_2 messages, where $k_1k_2 = k$. Therefore, after a trivial modification, our algorithm has the following characteristics:

- the memory of the automaton at a vertex is $O(nD\log s_0)$,
- the message size is $O(k_2D\log s_0)$,
- the arc capacity is k_1,
- the operating time of the algorithm is $O(n/k_1k_2 + D)$.

To estimate the operating time of the algorithm, we assume that the actuation time of the automaton at a vertex can be neglected, while the message transfer time along an arc is limited from above by 1 time step.

Some messages are sent from a vertex simultaneously along all outgoing arcs, some other, only along a part of outgoing arcs, and the third ones, only along a single outgoing arc. Nevertheless, we will assume for simplicity that a message expects at a vertex that all the outgoing messages will be released simultaneously. Thus, the automaton of a vertex does not need a signal

on the release of a given outgoing arc; a general signal on the release of all outgoing arcs suffices. We will also assume that an automaton generally sends several messages simultaneously along an arc, but not more than k, the capacity of the arc.

During its operation, the algorithm constructs spanning trees of a graph: a *direct spanning tree* directed from the root, and a *back spanning tree* directed to the root. The arcs going out from a vertex: *direct arcs*—the arcs of the direct spanning tree, *chords*—other outgoing arcs, and a *back arc*—an arc of the back spanning tree (it may be both a direct arc and a chord). A *direct simple path* is a simple path consisting of arcs outgoing from the root. A *back simple path* is a simple path consisting of direct arcs from the root. By a *vector of a path* is meant the list of the numbers of arcs of a path in a graph. The algorithm uses only vectors of simple paths of length of at most D and contours (cycles in which the same vertex is not met twice) of length of at most $D + 1$. The *vector of a vertex* is a vector of a direct simple path to this vertex. The root has an empty vector ε. The size of a vector of simple path or of a contour is $O(D\log s_0)$. The message size in the algorithm described below is equal to $O(1)$ vectors, i.e., $O(D\log s_0)$ bits.

We will divide the description of the algorithm into four parts. The first part constructs a back spanning tree, the second part is a "stopper" that determines the end of construction of the back spanning tree, the third part is intended for marking the arcs outgoing from a vertex into straight lines and chords, and the fourth part installs counters of incoming back arcs at the vertices. Such marking of the graph (through the memory of the vertex automata) is used further by the pulsation algorithm.

2.1. The First Part of the Algorithm

Four types of messages are used: *Start*, *Root search*, *Direct*, and *Reverse*. A message can either be created at a vertex or arrive at a vertex along an incoming arc. In both cases, we will say for brevity "a vertex receives a message."

A *Start* message is created by the root at the beginning of operation (when a *Start* message, which initiates the operation of the algorithm, is received from outside) and is sent to all vertices. Its purpose is to report to each vertex its vector and to initiate sending a *Root search* message. Upon receiving a *Start* message for the first time, each vertex (starting from the root) distributes it over all outgoing arcs and ignores repeated *Start* messages. The message contains a *path vector* along which it passes. A vertex (including the root), while sending a message along an outgoing arc with number i, adds i to the end of the *path vector*. Receiving the message for the first time, the vertex stores the *path vector* (obviously, this is a simple path) from the message as its own *vertex vector*; the root vector is empty.

A *Root search* message is created by a vertex when it receives a *Start* message for the fist time. We will call this vertex an *initiator* of a *Root search* message. The goal of the *Root search* message is to pass a certain simple path from the initiator to the root and to report the vector of this simple path to the root. In response, the root sends a *Direct* message, which is addressed to the initiator. Both these messages contain a *vector of initiator* as a parameter. Each vertex stores a *list of vectors of initiators* from those *Root search* or *Direct* message that appeared at this vertex. In the beginning, the list is empty. As soon as a vertex receives a *Root search* or *Direct* message with a given *vector of initiator* for the first time, it places this message in the list. The proper *vector of vertex* can be stored as the first element of the list; this element will be empty if the vertex does not yet have a proper vector, i.e., if it has not yet received a *Start* message.

A *Root search* message is distributed "fanwise" so that it passes along each arc exactly once. For this purpose, the *vector of initiator* in the message is compared with the vectors from the *list of vectors of initiators*. If the message is the first one, it is distributed over all outgoing arcs, and the *vector of initiator* from the message is added to the *list of vectors of initiators*. Repeated messages are ignored. In addition to the *vector of initiator*, the *Root search* message contains a *path vector* along which it passes. When the root receives the *Root search* message with the given *vector of initiator* for the first time, it creates a *Direct* message with the same *vector of initiator* and *path vector*. Note that here the *vector of initiator* is a vector of direct simple path from the root to the initiator, and the *path vector* is a vector of some simple path from the initiator to the root.

A *Direct* message is created by the root when it receives a *Root search* message with a given *vector of initiator* for the first time. It is sent along a direct simple path to the initiator. The goal of this message is to communicate a *simple path vector* from the initiator to the root to the initiator. When a vertex receives a *Direct* message with a given *vector of initiator* x, it compares it with a proper *vertex vector* y. Since a *Direct* message moves along a direct simple path to the initiator, $y \le x$. If $y < x$, then $y \cdot i \le x$ for some arc number i, and the message is sent unchanged along the outgoing arc i. If $y = x$, i.e., if the vertex is the initiator for this message, then the vertex creates a *Reverse* message, which contains the same *vector of simple path* from the initiator to the root. Let us recall that, as soon as a vertex receives a *Direct* message with a given *vector of initiator* for the first time, it places it into the list in order that further *Root search* messages with the same initiator be ignored.

A *Reverse* message is created by a vertex when it receives a *Direct* message addressed to it. The goal of this message is to mark a simple path back from this vertex to the root, defined by the *simple path vector*

from the *Direct* message. When a vertex (different from the root) receives (creates or receives along the incoming arc) a *Reverse* message with a *simple path vector* of the form $i \cdot x$, the arc with number i is marked as a back arc, and a *Reverse* message with the *simple path vector* x is sent along this arc. The root ignores the *Reverse* message (which, obviously, arrives at the root with empty *simple path vector*).

Reverse messages are "summed" at a vertex in anticipation of the release of outgoing arcs. More precisely, if there is a *Reverse* message at the vertex, that waits for the release of arcs, then newly arriving messages of this type are ignored.

Proposition 2.1. *The first part of the algorithm constructs a back spanning tree of a strongly connected graph.*

Proof. According to the description of the algorithm, at most one back arc appears at each vertex except for the root. Each time when a certain arc outgoing from a vertex is declared back, a message is sent along it that guaranteedly reaches the root, possibly, "being summed" with waiting *Reverse* messages. Therefore, if a back arc appears at a vertex, then a simple path along back arcs appears from this vertex to the root.

The number of all transfers of messages along arcs is finite. Indeed, a *Start* message passes exactly once along each vertex; therefore, the number of such transfers of messages along arcs is equal to the number of arcs m. The *Root search* message with this *vector of initiator* passes along each arc also exactly once, and the number of different initiators is equal to the number of vertices other than the root, i.e., to $n - 1$; therefore, the number of transfers of *Root search* messages along arcs is not greater than $(n - 1)m$. The *Direct* and *Reverse* messages are created once for each initiator, and each such message passes a simple path of length at most D; therefore, the total number of transfers of such messages along arcs is at most $2(n - 1)D$. Thus, the total number of all transfers of messages along arcs is finite. Then, since a message is sent along each arc in finite time (not greater than the time step), after a finite time, there will be no messages of these four types on the graph. We will show that, at this instant of time, a back spanning tree is constructed.

To this end, it suffices to show that a back arc appears at each vertex except for the root. Suppose that this is not so: there exists a vertex a, other than the root, at which no back arc has appeared. Since the graph is strongly connected, the vertex a is reachable from the root. Hence, the vertex a receives a *Start* message; after that, a *Root search* message is distributed "fanwise" from the vertex a. Since the graph is strongly connected, this message reaches the root; after that, the root sends a *Direct* message to the vertex a along a direct simple path. When this message arrives at the vertex a, a back arc appears at this vertex, unless it has already appeared there. We arrive at a contradiction; hence, our assumption is not correct.

The proposition is proved.

2.2. The Second Part of the Algorithm

The task of this part is to determine the end of construction of the back spanning tree. The idea is based on the calculation of the number of arcs of the graph at the root in such a way that, for every arc $a \longrightarrow b$, the root first receives a message from the beginning a of the arc at which there is "+1" for the arc $a \longrightarrow b$, and, a fortiori later, it receives a message from the end b of the arc at which there is "−1" for the arc $a \longrightarrow b$. In addition, this last message from the end b of the arc arrives at the root a fortiori later than the message that is sent from b and contains "+1" for every arc outgoing from b. To this end, the *Root search* message is modified, and two additional messages *Finish* and *Minus* are used.

When creating a *Root search* message, the initiator adds one more parameter to the message: the *number of arcs* outgoing from the initiator. We will say that, for every arc outgoing from the initiator, there is "+1" in the *Root search* message. The root conducts the *counter of arcs*, which is first set equal to the number of arcs outgoing from the root. When obtaining an (unignored) *Root search* message, the root adds the *number of arcs* from the message to this *counter of arcs*.

A *Finish* message is created by the root at the beginning of operation (when receiving a *Start* message from outside) and is sent along all arcs outgoing from the root. Any other vertex (other than the root) creates a *Finish* message when obtaining a *Direct* message and sends the *Finish* message along all outgoing arcs. The goal of the *Finish* message is to initiate the creation of a *Minus* message at the end of the arc along which it is sent.

As long as there is no back arc at a vertex different from the root, the vertex conducts the *counter of the number of incoming arcs* as the number of *Finish* messages received by it. When a back arc appears, the vertex sends, along the back simple path, a *Minus* message with the *number of arcs* parameter equal to the current value of the *counter of the number of incoming arcs*, provided that it is greater than zero. We will say that there is "−1" in the *Minus* message for each of these arcs. When a back arc appears at a vertex, every reception of a *Finish* message initiates sending a *Minus* message with the *number of arcs* parameter equal to 1 from the vertex along the back simple path. We will say that there is "−1" in the *Minus* message for this arc. The *Minus* message is transferred without change by vertices along the back simple path to the root. The root, having received the *Minus* message, subtracts the *number of arcs* parameter in the *Minus* message from its *counter of arcs*.

The *Minus* messages are summed at a vertex in anticipation of the release of arcs. If the vertex already

contains a *Minus* message that waits for the release of arcs and a new *Minus* message arrives, then the *Minus* message is preserved in which the *number of arcs* parameter is equal to the sum of the corresponding parameters of both these messages.

Proposition 2.2. *After finite time, the counter of arcs at the root is zeroed. By this instant of time, the back spanning tree has already been constructed.*

Proof. Consider an arbitrary arc *ab*. If *a* is the root, then, right at the beginning, the *counter of arcs* at the root increases by 1 for the arc *ab*. Then the root sends a *Finish* message, which reaches the vertex *b* in finite time. If *a* is a nonroot vertex, it receives the first *Start* message in finite time; after that the vertex creates a *Root search* message in which there is "+1" for the arc *ab*. After a finite time, this message reaches the root, and the *counter of arcs* at the root increases by 1 for the arc *ab*. Then a *Direct* message from the root reaches the vertex *a*. After that, a *Finish* message is sent along the *ab* arc, which reaches the vertex *b* in finite time. Also in finite time, a back arc appears at the vertex *b*. Hence, both events occur in finite time: reception of a *Finish* message by the vertex *b* and the emergence of a back arc at this vertex. At this instant of time, a *Minus* message with "−1" for the arc *ab* is sent from the vertex *b* along the back arc. After a finite time, this message reaches the root, and the root subtracts 1 for the arc *ab* from its *counter of arcs*. Thus, for every arc, the root first once adds 1 to its counter of arcs and then once subtracts 1. Thus, after a finite time when this occurs with all arcs, the *counter of arcs* at the root is zeroed.

Let us show that, at the instant of time when the counter of arcs at the root is zeroed, the back spanning tree is already constructed. With respect to every arc, the root first once receives "+1" in the *Root search* message sent from the beginning of the arc and then once receives "−1" in the *Minus* message sent from the end of the arc. Therefore, at the instant when the *counter of arcs* is zeroed, with respect to each arc, the root either receives "+1" and "−1" or receives neither "+1" nor "−1." Suppose that the proposition does not hold: there exists a vertex at which a back arc has not yet appeared. Since the graph is strongly connected, there is a simple path from the root to this vertex. With respect to the last arc of this simple path, the root has not received "−1"; hence, it has not received "+1." Denote by *bc* the first arc on this simple path with respect to which the root has received neither "+1" nor "−1." Since the root immediately adds "+1" to the counter of arcs for each arc outgoing from the root, the vertex *b* is not the root. Then we denote by *ab* the preceding arc on the simple path. The root received a *Minus* message with "−1" with respect to the arc *ab* from the vertex *b*. But then the root had received a *Root search* message with "+1" with respect to the arc *bc* from the vertex *b*, which is not the case. We arrive at a contradiction; hence, our assumption is not valid, and the back spanning tree is already constructed.

The proposition is proved.

2.3. The Third Part of the Algorithm

This part marks outgoing arcs from each vertex as direct arcs and chords. First, we assume that all outgoing arcs are chords. Then, each arc *ab* along which a *Direct* message is sent is marked as a direct arc at the vertex *a*.

Proposition 2.3. *When the counter of arc is zeroed at the root, all arcs are correctly marked as chords or straight lines.*

Proof. Since a *Direct* message is sent from the root to each vertex along a direct simple path, sooner or later, all direct arcs will be marked. Let us show that this will occur before the *counter of arcs* at the root is zeroed. Consider an arbitrary direct arc *ab*. Since the graph is strongly connected, there exists an arc *bc*. The zeroing of the *counter of arcs* at the root occurs only after a *Minus* message with "−1" for the arc *bc* is sent from the vertex *c* and this message reaches the root. But such a message is sent only after a *Finish* message is sent along the arc *bc*, and this message is sent only after the vertex *b* receives a *Direct* message, which has to pass along the direct arc *ab* and mark the arc *ab* at the vertex *a* as a direct arc.

The proposition is proved.

2.4. The Fourth Part of the Algorithm

This part sets the values of the *counters of incoming back arcs* at vertices. At the beginning, the *counters of incoming back arcs* is set to zero. Let us add two new messages *Start of calculation* and *End of calculation*. The *Start of calculation* message is sent from the root along direct arcs after zeroing the *counter of arcs* at the root. Each vertex except for the root, having received the *Start of calculation* message, first, sends the same *Start of calculation* message along all outgoing direct arcs and, second, creates the *End of calculation* message and sends it along the back arc.

The *End of calculation* message contains a sign of the *first back arc*. When a vertex creates the *End of calculation* message, it supplies it with this sign. When a vertex receives the *End of calculation* message with a sign of the *first back arc* along an incoming arc, it adds 1 to its *counter of incoming back arcs*, and if this vertex is not the root, it sends the *End of calculation* message further along the back arc, now without this sign. When a vertex receives the *End of calculation* message without the sign of the *first back arc* along an incoming arc, it simply forwards the *End of calculation* message without change further along the back arc.

The *End of calculation* messages are summed at vertices in anticipation of sending along the back arc.

For this purpose, a message is supplied with the *number of summed messages* parameters. When an *End of calculation* message at a vertex waits for sending along a back arc and a new *End of calculation* message arrives, their *number of summed messages* parameters are added up, and one *End of calculation* message is preserved with this number; when the back arc is released, the *End of calculation* message with the accumulated *number of summed messages* parameter is sent along it.

When the *counter of arcs* at the root is zeroed, there is a *list of vectors of initiators* at the root that contains the vectors of all vertices except for the root. The root calculates the length of the list and writes it in the *counter of nonroot vertices*. The root, having received the *End of calculation* message, subtracts the *number of summed messages* parameter from the *counter of nonroot vertices*. When this counter becomes equal to zero, the operation of this part and of the whole algorithm is completed.

Proposition 2.4. *When the counter of nonroot vertices is zeroed at the root, all vertices have correct values of the counters of incoming back arcs.*

Proof. A *Start of calculation* message sent along direct arcs reaches each vertex once. Each vertex sends once an *End of calculation* message with the *first back arc* sign. Thus, exactly one *End of calculation* message with the *first back arc* sign passes along each back arc. Therefore, a correct value of the *counter of incoming back arcs* is set at each vertex after the *End of calculation* message with the *first back arc* sign is received along all back arcs.

Moreover, for each back arc, an *End of calculation* message with "1" for this arc in the *number of summed messages* parameter reaches the root. The number of such "1" is equal to the number of back arcs, which is equal to the number of nonroot vertices. Therefore, when the *counter of nonroot vertices* becomes equal to zero, no *Beginning of calculation* and *End of calculation* messages are left on the arcs of the graph, and the values of the *counters of incoming back arcs* at all vertices will be correct.

The proposition is proved.

Thus, after the end of the operation of the algorithm, direct and back spanning trees of the graph will be constructed: at each vertex, direct arcs and (if the vertex is not the root) one back arc will be marked. Moreover, a value of the counter of incoming back arcs will be set at each vertex.

3. ESTIMATES FOR THE MEMORY AND THE OPERATING TIME OF THE ALGORITHM

3.1. Estimate for the Memory Size of a Vertex Automaton

The memory of the automaton of a nonroot vertex contains:

– 1 bit—a sign of that a *Start* message waits for the release of outgoing arcs;

– for each initiator, the *list of vectors of initiators* stores

° 1 vector—a *vector of initiator*;

° 1 bit—a sign of that a *Root search* message waits for the release of outgoing arcs;

° 1 bit—a sign of that a *Direct* message waits for the release of outgoing arcs;

° 1 vector—the *number of arcs* parameter waiting for a *Root search* message or the *vector of back simple path* parameter waiting for a *Direct* message; note that, for a given initiator, only one of the messages *Root search* or *Direct* can be expecting, and the number of arcs is a vector of length 1;

– 1 vector—the *vector of back simple path* parameter waiting for a *Reverse* message;

– 1 bit—a sign of waiting for a *Finish* message;

– $\log s_1$ bit—the *number of arcs* parameter waiting for a *Minus* message;

– 1 bit—a sign of waiting for a *Start of calculation* message;

– $\log n$ bit—the *number of summed messages* parameter waiting for an *End of calculation* message; it is equal to zero if there is no waiting message;

– $\log s_1$ bit—the *counter of incoming back arcs*.

Altogether, the memory size of the automaton of a nonroot vertex is 1 bit + n (1 vector + 1 bit + 1 bit + 1 vector) + 1 vector + 1 bit + $\log s_1$ bits + 1 bit + $\log n$ bits + $\log s_1$ bit = $(2n + 1)$ vectors + $(2n + \log n + 2\log s_1 + 3)$ bits $\le (2n + 1)$ vectors + $(2n + \log n + 2\log m + 3)$ bits $\le (2n + 1)$ vectors + $(2n + \log n + 2\log n s_0 + 3)$ bits = $(2n + 1)O(D\log s_0) + 2n + \log n + 2\log n s_0 + 3$, which is equal to $O(nD\log s_0)$ for $D > 0$.

The memory of the automaton of the root contains,

– for each initiator in the *list of vectors of initiators*,

° 1 vector—a *vector of initiator*;

° 1 bit—a sign of that a *Direct* message waits for the release of outgoing arcs;

° 1 vector—the *vector of back simple path* parameter waiting for a *Direct* message;

– 1 bit—a sign of waiting for a *Finish* message;

– 1 bit—a sign of waiting for a *Start of calculation* message;

– $\log m$ bits—the *counter of arcs*;

– $\log n$ bits—the *counter of nonroot vertices*.

Altogether, the memory size of the automaton of the root is n(1 vector + 1 bit + 1 vector) + 1 bit + 1 bit +

$\log m$ bits + $\log n$ bits = $2n$ vectors + $(n + \log m + \log n + 2)$ bits $\le 2n$ vectors + $(n + \log n s_0 + \log n + 2)$ bits = $2nO(D\log s_0) + n + \log n s_0 + \log n + 2$, which is equal to $O(nD\log s_0)$ for $D > 0$.

Thus, the memory of the automaton of any vertex is equal to $O(nD\log s_0)$ for $D > 0$.

3.2. Estimate for the Operating Time of the Traversal Algorithm

We can always assume that messages are sent from a vertex in the order of decreasing their priorities: (1) *Start*, (2) *Root search*, (3) *Direct*, (4) *Reverse*, (5) *Finish*, (6) *Minus*, (7) *Start of calculation*, and (8) *End of calculation*.

Below we prove a proposition stating that (1) if there are no other messages except for *Root search*, each such message reaches the root in time of at most $T(n, k, D)$ and (2) if there are no other messages except for *Direct*, then each such message reaches its destination in time of at most $T(n, k, D)$. Here we will represent the operation of the algorithm as a sequence of eight stages in eight priorities of messages and estimate the operating time of each stage.

(1) A *Start* message, as the most priority one, can wait for the release of outgoing arcs at most for one time step. The transmission of the message along an arc may take at most another time step. Since a *Start* message is sent exactly once along each arc, the *Start* messages reach all the vertices in time of at most $2D$.

(2) Consider the flow of *Roots search* messages after all *Start* messages reach all the vertices. Since there are no *Start* messages any more and other messages are of lower priority than a *Root search*, each such message waits for the release of outgoing arcs occupied by messages of other types for at most one time step. Thus, an estimate for $T(n, k, D)$ increases at most twofold; i.e., all *Root Search* messages reach the root in time of at most $2T(n, k, D)$.

(3) Consider the flow of *Direct* messages after all *Root search* messages reach all the vertices. Since there are no messages of higher priority any more and other messages are of lower priority than a *Direct* message, each such message waits for the release of outgoing arcs occupied by messages of other types for at most one time step. Thus, an estimate for $T(n, k, D)$ increases at most twofold; i.e., all *Direct* messages reach the root in time of at most $2T(n, k, D)$.

(4) Consider the flow of *Reverse* messages after all *Direct* messages reach all the vertices. These messages move along back arcs and are summed at vertices in anticipation of the release of outgoing arcs. Each such message passes a simple path of length of at most D. Since there are no messages of higher priority any more, such a summing *Reverse* message waits at a vertex for the release of outgoing arcs for at most one time step. Thus, all *Reverse* messages reach the root in time of at most $2D$.

(5) Consider the flow of *Finish* messages after all *Reverse* messages reach the root. From this instant of time, there are back arcs at all vertices except for the root. *Finish* messages move along arcs, passing along each arc exactly once. Each such message passes a simple path of length of at most D. Since there are no messages of higher priority any more and there are back arcs everywhere, each *Finish* message waits at a vertex for at most one time step. Thus, all *Finish* messages pass over all arcs in time of at most $2D$.

(6) Consider the flow of *Minus* messages after all *Finish* messages pass over all arcs. These messages move along back arcs up to the root, being summed at each vertex. Each such message passes a simple path of length of at most D, waiting at a vertex for at most one time step. Thus, all *Minus* messages reach the root in time of at most $2D$.

Thus, the *counter of arcs* at the root is zeroed in time of at most $2D + 2T(n, k, D) + 2T(n, k, D) = 2D + 2D + 2D = 4T(n, k, D) + 8D$. After that, the *counters of incoming back arcs* are set.

(7) *Start of calculation* messages are sent along direct arcs to each vertex. Therefore, each such message passes a simple path of length of at most D. Since there are no messages of higher priority any more, each *Start of calculation* message waits at a vertex for the release of outgoing arcs for at most one time step. Thus, all *Start of calculation* messages stop in time of at most $2D$.

(8) Consider the flow of *End of calculation* messages after all *Start of calculation* messages stop. These messages move along back arcs up to the root, being summed at each vertex. Each such message passes a simple path of length of at most D, waiting at a vertex for at most one time step. Thus, all *End of calculation* messages reach the root in time of at most $2D$.

Thus, the whole algorithm stops in time of at most $4T(n, k, D) + 8D + 2D + 2D = 4T(n, k, D) + 12D$.

3.3. A Formal Model of Flow of Root search and Direct Messages

$T(n, k, D)$ is an upper bound for the flow time of all *Root search* messages under the assumption that there are no other messages, as well as the flow time of all *Direct* messages under the assumption that there are no other messages.

First, consider the flow of the *last Root search* message, i.e., the message that arrives last at the root. This message passes a simple path of length $d \le D$ from the initiator vertex to the root. Other *Root search* messages can arrive at the vertices of this simple path "sideways," i.e., along incoming arcs that do not belong to the simple path. In this case, we will say that a message *is born* at a vertex. Since repeated messages with the same initiator are ignored at a vertex, we will say that they *die* at a vertex.

Now, consider the flow of the *last Direct* message, i.e., the message that arrives last from the root to the initiator. The *Direct* message moves from the root to the initiator vertex along a direct simple path of length $d \le D$. Since *Direct* messages are not distributed fanwise and move only along direct arcs, they, unlike *Root search* messages, do not arrive at vertices "sideways," and there are no repeated messages with the same initiator at vertices. Obviously, this is a particular case of the flow of *Root search* messages.

Let us formalize the description of the flow of *Root search* messages.

There is a simple path of length $d \le D$. We will refer to the vertices and arcs as *positions*. Let us renumber the positions as follows: the initial vertex of the simple path has number 0, the number of an arc is 1 greater than the number of its beginning, and the number of the end of an arc is 1 greater than the number of the arc. Thus, vertices have even numbers 0, 2, ..., $2d$, while arcs have odd numbers 1, ..., $2d - 1$. If $i < i'$, then we will say that position i is *to the left of* position i' and position i' is *to the right of* position i. The positions may contain messages that are called *points* in the formal model.

The following events are possible for the points: (1) the birth of a point at a vertex (a sideways arrival of a message), (2) the death of a point at a vertex (a message is ignored), (3) the flow of a point from the beginning of an arc onto the arc (a message is sent along the arc from its beginning), and (4) the flow of a point from an arc to the end of the arc (a message is received from an arc to its end). We will assume that all these events occur instantaneously.

By the event of *appearance* of a point in a position we mean the event of birth of this point in this position (if the position is a vertex) or the event of flow of this point to this position from the previous position. Accordingly, by the event of *disappearance* of a point from a position we mean the event of death of this point in this position (if the position is a vertex) or the event of displacement of this point from this position to the next position.

The following rules of flow of points follow from the description of the algorithm: 1. A point is on an arc for at most one time step. 2. Points move onto an arc when no points remain on the arc; the minimum from among the points at the beginning of the arc and the "capacities" k of the arc move at once onto the arc. 3. If a point dies at a vertex, this occurs at that very moment when it appears at the vertex (either is born or is displaced from an incoming arc). 4. The number of points that died at a vertex is not greater than the number of points that are born at the vertex (this follows from the fact that a dying point is a repeated message with the same initiator). 5. At the initial vertex 0 at time step 0, there is a point that will not die. We will call it *our* point (this is the last message). 6. The total number of points that appear in all positions is finite (the number of *Root search* messages is finite, which was proved above in Proposition 2.1). 7. The number of points that appear but do not die in the final position $2d$ up to the appearance of our point in this position is not greater than n (actually, together with our point, the number of such points does not exceed the number $n - 1$ of nonroot vertices).

If a point is at some vertex at the initial instant of time 0, we will assume that the point is born at this vertex at the instant of time 0. If a point is on some arc at the initial instant of time 0, we will assume that the point is born at the beginning of this arc at the instant of time 0 and is instantly displaced onto an empty arc together with several points.

We will say that a *behavior of points* is defined if, for every point, the place (position) and the times of its birth and death, as well as the delay of this point on each arc, are specified so that the above-mentioned rules of flow of points are satisfied.

Denote by $U_{i,j}$ the number of points that appeared but not died in position i at time step $i + j$.

Denote by $T = T(n, k, D)$ the number of the a time step at which our point reaches the final position $2d$.

Lemma 3.1. Suppose given the behavior of points satisfying rules 1–4 (which may not satisfy rules 5–7). Suppose that, in this behavior, $U_{0,j} \ge kj$ for every j such that $i + j \le T$. Then, $U_{i,j} \ge kj$ for every position i and every j such that $i + j \le T$.

Proof. We will prove the lemma by induction on i and j.

The base of induction on i for $j \le T$ is $U_{0,j} \ge kj$, which is defined by the hypothesis of the lemma.

The base of induction on j for $i \le T$ is $U_{i,0} \ge k \cdot 0 = 0$, which is obvious.

The induction step: suppose that the assertion of the lemma holds both for i and $j + 1$ and for $i + 1$ and j, where $i < 2d$; i.e., $U_{i,j} + 1 \ge k(j + 1)$ and $U_{i+1,j} \ge kj$. We have to prove that if $i + j + 2 \le T$, then the assertion of the lemma holds for $i + 1$ and $j + 1$; i.e., $U_{i+1,j+1} \ge k(j + 1)$. Consider two possible cases. i is even; i.e., position i is an arc. Introduce the notation:

A is the number of points that moved from arc i to vertex $i + 1$ and did not die there at all time steps from 0 to $i + j + 1$;

A' is the number of points that moved from arc i to vertex $i + 1$ and died there at all time steps from 0 to $i + j + 1$;

B is the number of points that are born at vertex $i + 1$ and did not die there at all time steps from 0 to $i + j + 1$;

B' is the number of points that are born at vertex $i + 1$ and died there at all time steps from 0 to $i + j + 1$;

A_Δ is the number of points that moved from arc i to vertex $i + 1$ and did not die there during the time step $i + j + 2$;

A'_Δ is the number of points that moved from arc i to vertex $i+1$ and died there during the time step $i+j+2$;

B_Δ is the number of points that are born at vertex $i+1$ and did not die there during the time step $i+j+2$;

B'_Δ is the number of points that are born at vertex $i+1$ and died there during the time step $i+j+2$;

a is the number of points on arc i at the end of the time step $i+j+1$ that will not die at vertex $i+1$.

a' is the number of points on arc i at the end of the time step $i+j+1$ that will die at vertex $i+1$.

Taking into account that, according to rule 3, a point either dies at a vertex when it appears there or does not die at all at this vertex, by rule 4, we have $A' + A'_\Delta + B' + B'_\Delta$. Hence, $A' + A'_\Delta \le B + B_\Delta$. By definition, $U_{i+1,j} = A + B$. The point appearing on an arc does not die on the arc and is either on the arc or among the points ($A + A'$ in number) that move to the end of the arc. Therefore, $U_{i,j+1} = a + a' + A + A'$. The points that arrive at vertex $i+1$ in a time step and do not die at this vertex are of two kinds: points (A_Δ in number) that have moved from arc i and did not die at vertex $i+1$ and points (B_Δ in number) that were born at vertex $i+1$ and did not die there. Therefore, $U_{i+1,j+1} = U_{i+1,j} + A_\Delta + B_\Delta$. By rule 1, all points ($a + a'$ in number) that were on arc i at the beginning of a time step move from this arc to vertex $i+1$ during the time step; however, even a greater number of points ($A_\Delta + A'_\Delta$ in number) may move to the vertex, provided that their delays are small enough and the arc is released several times during a time step. Therefore, $A_\Delta + A'_\Delta \ge a + a'$. Then

$$U_{i+1,j+1} = U_{i+1,j} + A_\Delta + B_\Delta = A + B + A_\Delta + B_\Delta =$$

$$A + A_\Delta + B + B_\Delta \ge A + A_\Delta + A' + A'_\Delta = A + A' + A_\Delta +$$

$$A'_\Delta \ge A + A' + a + a' = U_{i,j+1} \ge k(j+1).$$

Thus, $U_{i+1,j+1} \ge k(j+1)$, which was to be proved.

i is odd; i.e., position i is a vertex.

Denote by x the number of points that are at vertex i at the end of time step $i+j+1$. The points that arrived at vertex i and did not die there up to the time step $i+j+1$ inclusive either are at the vertex at the end of this time step (their number is x) or moved to arc $i+1$ and did not die there because points on an arc do not die. Therefore, $U_{i,j+1} = x + U_{i+1,j}$. Consider the case of $x \le k$. By rule 1, arc $i+1$ is released at least once during a time step, and, since $x \le k$, by rule 2, all the points that were at the vertex at the beginning of a time step will move to the arc. Notice that a greater number of points can move onto the arc if they appear at the vertex before the first release of the arc or appear later, but within a time step, while delays on the arc are small enough and the arc is released several times during a time step. Therefore, $U_{i+1,j+1} \ge U_{i+1,j} + x = U_{i,j+1} \ge k(j+1)$. Thus, $U_{i+1,j+1} \ge k(j+1)$, which was to be proved. Consider the case of $x > k$. By rule 1, arc $i+1$ is released at least once during a time step, and, since $x > k$, by rule 2, k points move to the arc. Notice that, just as in the previous case, even a greater number of points can move onto the arc. Therefore, $U_{i+1,j+1} \ge U_{i+1,j+k} \ge kj + k = k(j+1)$.

Thus, $U_{i+1,j+1} \ge k(j+1)$, which was to be proved.

The induction step is proved.

The lemma is proved.

Proposition 3.1. Our point reaches the final position $2d$ at time step $T \le n/k + 2d + 1$.

Proof. First of all, we will notice that our point reaches the final position $2d$ in finite time. Indeed, since our point does not die by rule 5, it suffices that our point stays in each position for a finite time. The delay of our point on an arc is finite by rule 1 (for at most one time step). The finiteness of the delay of our point at a vertex is determined by the finiteness of the delays of points on an outgoing arc (rule 1), the necessity of the flow of points onto an arc as the arc is released (rule 2), and the finiteness of the number of points (rule 6).

The points that are always to the left of our point obviously do not influence the flow of our point; therefore, they can be ignored: we will assume that there are no such points. If a point is born to the left of our point but overtakes our point at some vertex at some instant of time, this is equivalent to that this point is born at this instant of time at this vertex. Considering the aforesaid, below we will assume that there are no points that are born to the left of our point. All the more so, no points are born to the left of the leftmost point in the current disposition of points.

Consider an *extended* behavior of points, which differs in that an additional number of points are born in positions 0 at each time step $t \le T$, so that the hypothesis $U_{0,t} \ge kt$ of Lemma 3.1 is satisfied for $t \le T$. We will assume that each of these additional points (1) is born in the initial position 0 and does not die (at any vertex), (2) has the maximum delay of one time step on arcs, and (3) moves from a vertex onto an arc outgoing from it only after original (not additional) points.

The extended behavior of points satisfies rules 1–6. Only the fulfillment of rule 2 needs explanation, because additional points move from a vertex onto an arc outgoing from it only after the original points. There is no contradiction here, because the original points are not born to the left of the leftmost original point. Therefore, at the instant of time when, according to rule 2, additional points have to move from a vertex onto an outgoing arc, either there are no original points at this vertex or the number of these points is less than k and they move together with a few first additional points. Thus, the leftmost original point subsequently turns out to be to the right of this vertex,

and, therefore, no original points arise at this vertex any more.

By definition, additional points do not overtake the original points; therefore, the additional points do not influence the flow of the original points. Thus, in the extended behavior of points, our point still reaches the final position $2d$ at time step T, and, up to this time step, the number of original points that appeared but did not die in the final position $2d$ does not exceed n according to rule 7. Together with our point, a few additional points may move to the final position from an incoming arc, but the number of these additional points is not greater than $k - 1$. Therefore, the total number of points (both original and additional) that appeared but did not die in the final position $2d$ up to the time step T inclusive is $U_{2d, T-2d} \le n + k - 1$.

Since the extended behavior of points satisfies the hypotheses of Lemma 3.1, for $T \ge 2d$, we have $U_{2d, T-2d} \ge k(T - 2d)$. Hence, $k(T - 2d) \le n + k - 1$. Thus, $T \le n/k + 1 - 1/k + 2d \le n/k + 2d + 1$. If $T < 2d$, then the more so $T \le n/k + 2d + 1$.

The proposition is proved.

Thus, $T(n, k, D) \le n/k + 2d + 1$. Hence, the operating time of the algorithm is not greater than $4T(n, k, D) + 12D = 4n/k + 8D + 4 + 12D = 4n/k + 20D + 4 = O(n/k + D)$. In a more general case, this time is $O(n/k_1k_2 + D)$, where k_1 is the arc capacity and k_2 is the message size in vectors.

4. AGGREGATE FUNCTIONS AND AGGREGATE EXTENSIONS OF FUNCTIONS

In fact, the pulsation algorithm calculates aggregate functions for which the value of a function of a union of multisets is calculated by the values of the function of these multisets. In this section, we give a formal definition of an aggregate function, prove the aggregation criterion, and show that any function $f(x)$ has an aggregate extension, i.e., any function can be calculated as $h(g(x))$, where g is an aggregate function. We also show that there is a unique minimum aggregate extension that calculates the minimum information by which one can reconstruct a function f. The theory of aggregate functions presented in this section is a modification of the theory of inductive functions presented in [9].

Next, we consider functions on finite multisets consisting of elements of some basic set X. Denote by X^- the set of all finite multisets of elements of X. For $a \in X^-$, we denote by a the cardinality of the multiset a. By the operations of union, intersection, complementation, etc., we mean operations on multisets, i.e., operations that take into account the multiplicities of elements. Denote by N the set of natural numbers.

An *aggregate function* g: $X^- \longrightarrow A$ is a function such that $\exists e: A \times A \longrightarrow A\ \forall a, b \in X^-\ g(a \cup b) = e(g(a), g(b))$.

Remark 1. One can easily show that the aggregate function g satisfies the condition: $\forall r \in N\ \exists e_r : A^r \longrightarrow A\ \forall a_1, ..., a_r \in X^-\ g(\cup\{a_i | 1 \le i \le r\}) = e_r(g(a_1), ..., g(a_r))$.

Remark 2. The set of aggregate functions is nonempty: for example, such functions are given by the sum of all terms of a multiset and their minimum; in either case the functions e are given by sums and minima.

Lemma 4.1 (aggregation criterion). *A function $g : X^- \longrightarrow A$ is aggregate if and only if $\forall a, b \in X^-\ \forall x \in Xg(a) = g(b) \Rightarrow g(a \cup \{x\}) = g(b \cup \{x\})$.*

Proof.

Necessity.

$\forall a, b \in X^-\ \forall x \in Xg(a \cup \{x\}) = e(g(a), g(\{x\}))$ and $g(b \cup \{x\}) = e(g(b), g(\{x\}))$.

Therefore, if $g(a) = g(b)$, then $g(a \cup \{x\}) = e(g(a), g(\{x\})) = e(g(b), g(\{x\})) = g(b \cup \{x\})$.

Sufficiency. First, by induction on $\#c$, we prove that $\forall a, b, c \in X^-\ g(a) = g(b) \Rightarrow g(a \cup c) = g(b \cup c)$,

For $\#c = 0$, the proposition is trivial.

For $\#c = 1$, $\exists x \in X = \{x\}$, and we obtain the given condition:

$\forall a, b \in X^-\ g(a) = g(b) \Rightarrow g(a \cup c) = g(a \cup \{x\}) = g(b \cup \{x\}) = g(b \cup c)$.

Suppose that the proposition is proved for $\#c = n - 1$; let us prove it for $\#c = n > 1$.

We have $\forall a, b, c \in X^-\ \exists x \in g(a) = g(b) \Rightarrow g(a \cup c\backslash\{x\}) = g(b \cup c\backslash\{x) \Rightarrow g(a \cup c) = g((a \cup c\backslash\{x\}) \cup \{x\}) = g((b \cup c\backslash\{x\}) \cup \{x\}) = g(b \cup c)$.

Now, we prove that $a_1, a_2, b_1, b_2 \in X^-(g(a_1) = g(b_1) \& g(a_2) = g(b_2) \Rightarrow g(a_1 \cup a_2) = g(b_1 \cup b_2)$.

We have $g(a_2) = g(b_2) \Rightarrow g(a_1 \cup a_2) = g(a_1 \cup b_2)$ and $g(a_1) = g(b_1) \Rightarrow g(a_1 \cup b_2) = g(b_1 \cup b_2)$. Thus, $g(a_1 \cup a_2) = g(a_1 \cup b_2) = g(b_1 \cup b_2)$. Now, we define $e : g(X^-) \times g(X^-) \longrightarrow A$ as $e(g(a), g(b)) = g(a \cup b)$. Since, by what has been proved, the choice of specific a and b is inessential and only the values of $g(a)$ and $g(b)$ are important, this function is well defined. It follows from its definition that g is an aggregate function.

The lemma is proved.

Remark 4.3. It follows from the aggregation criterion that $g : X^- \longrightarrow A$ is aggregate if and only if $\exists g' : A \times X \longrightarrow A\ \forall a \in X^- g(a \cup \{x\}) = g'(g(a), x)$. Indeed, it suffices to define $g'(g(a), x) = e(g(a), g(\{x\}))$.

An *aggregate extension* of a function $f: X^- \longrightarrow A$ is an aggregate function $g : X^- \longrightarrow B$ such that $\exists h : B - A\ \forall a \in X^- f(a) = h(g(a))$.

The aggregate extension $g : X^- \longrightarrow B$ of the function $f: X^- \longrightarrow A$ is said to be *minimal* if $g(X^-) = B$ and, for any $\forall g': X^- \longrightarrow C$, which is an aggregate extension of f, we have $\exists i: C \longrightarrow Bg = ig'$.

Remark 4.4. An aggregate extension g of a function f represents an aggregate function by which one can calculate the function f. Here extensions are possible that are of no help in practice; for example, one can take the identity function on X^- as g and the function f itself as h. To avoid such a situation, one uses a minimum aggregate extension; intuitively, this is an aggregate function that calculates the minimum information by which one can recover f.

Proposition 4.2 (uniqueness of the minimum aggregate extension). *The minimum aggregate extension of a function $f: X^- \longrightarrow A$ is unique up to one-to-one mappings.*

Proof. If $g: X^- \longrightarrow B$ and $g': X^- \longrightarrow C$ are minimum aggregate extensions of the function f, then $\exists i: C \longrightarrow B$, $i': B \longrightarrow C$ are one-to-one and such that ii' and $i'i$ are identity mappings on B and C, $g = ig'$, and $g' = i'g$.

The proposition is proved.

Proposition 4.3 (existence of the minimum aggregate extension). *For every $\forall f: X^- \longrightarrow A$, there exists a minimum aggregate extension.*

Proof. Define a relation on X^-. For $a, b \in X^-$, define $a \sim b \Leftrightarrow \forall c \in X^- f(a \cup c) = f(b \cup c)$. This relation is reflexive, symmetric, and transitive; hence, it is an equivalence relation. This relation is compatible with the function f; i.e., $\forall a, b \in X^- a \sim b \Rightarrow f(a) = f(b)$.

For any equivalence relation compatible with f, one can define a quotient function $h: X^-/\sim \longrightarrow A$ such that $h(\pi^\sim(a)) = f(a)$, where $\pi^\sim$ is a projection to a quotient set with respect to the relation $\sim$. The function $h(\pi^\sim(a))$ is defined as $f(a)$ for any $a \in \pi^\sim(a)$. Since the relation $\sim$ is compatible with f, this definition is well defined: the values of f are equal for any representatives of the same equivalence class.

In addition, $\forall a, b \in X^- \forall x \in Xa \sim b \Rightarrow a \cup \{x\} \sim b \cup \{x\}$.

Indeed, if $\forall c \in X^- f(a \cup c) = f(b \cup c)$, then also $\forall c \in X^- f(a \cup \{x\} \cup c) = f(b \cup \{x\} \cup c)$. Hence, the projection $\pi^\sim$ satisfies the aggregation criterion:

$\forall a, b \in X^- \forall x \in X(\pi^\sim(a) = \pi^\sim(b) \Leftrightarrow a \sim b) \Rightarrow (a \cup \{x\} \sim b \cup \{x\} \Leftrightarrow \pi^\sim(a \cup \{x\}) = \pi^\sim(b \cup \{x\}))$.

That is, $\pi^\sim: X^- \longrightarrow X^-/\sim$ satisfies the definition of the aggregate extension of the function f. Now we will take an arbitrary function $g': X^- \longrightarrow C$, which is an aggregate extension of the function f. By the definition of extension, $\exists h': C \longrightarrow A \, \forall a \in X^- f(a) = h'(g'(a))$. Let us show that $\forall a, b \in X^- g'(a) = g'(b) \Rightarrow a \sim b$. Since g' is an aggregate function, $g'(a) = g'(a) = g'(b) \Rightarrow \forall c \in X^- g'(a \cup c) = g'(b \cup c)$. It follows from $f = h'g'$ that $g'(a) = g'(b) \Rightarrow f(a) = f(b)$. Therefore, $g'(a) = g'(b) \Rightarrow \forall c \in X^- g'(a \cup c) = g'(b \cup c) \Rightarrow \forall c \in X^- f(a \cup c) = f(b \cup c) \Rightarrow a \sim b$. Hence, $\forall a, b \in X^- g'(a) = g'(b) \Rightarrow a \sim b \Rightarrow \pi \sim (a) = \pi \sim (b)$. Thus, one can define a function $i: g'(X^-) \longrightarrow X^-/\sim$ such that $i(g'(a)) = \pi \sim (a)$. This definition is well defined because, as we have just shown that the value of $\pi \sim (a)$ does not depend on the choice of a specific a provided that the value of $g'(a)$ is preserved. Thus, π is an aggregate extension of f; it is calculated by any aggregate extension of f; moreover, the image of $\pi \sim$ is given by the whole set $X^-/\sim$. Hence, $\pi \sim$ satisfies the definition of the minimum aggregate extension of f.

The proposition is proved.

Remark 4.5. Examples of the minimum aggregate extension (π_i is a projection of a tuple to the ith component). One can easily show that

• for the function of calculating the arithmetic mean $f(a_1, ..., a_n) = (a_1 + ... + a_n)/n$, the minimum aggregate extension is $g(a_1, ..., a_n) = (a_1 + ... + a_n, n)$; $f = \pi_1 g/\pi_2 g$;

• for the function of calculating the geometric mean $f(a_1, ..., a_n) = \sqrt[n]{(a_1 \cdot ... \cdot a_n)}$, the minimum aggregate extension is $g(a_1, ..., a_n) = (a_1 \cdot ... \cdot a_n, n)$; $f = \sqrt[\pi_2 g]{\pi_1 g}$;

• for the function of calculating the root mean square $f(a_1, ..., a_n) = \sqrt{((a_1^2 + ... + a_n^2)/n)}$, the minimum aggregate extension is $g(a_1, ..., a_n) = (a_1^2 + ... + a_n^2, n)$; $f = \sqrt{(\pi_1 g/\pi_2 g)}$;

Thus, every function $f: X^- \longrightarrow A$ can be represented as $f(a) = h(g(a))$ and $\forall b, c \in X^- g(b \cup c) = e(g(b), g(c))$. By a *partition* of a multiset $b \in X^-$ is meant a collection $b_1, ..., b_r$ of its subsets whose union coincides with b: $b = b_1 \cup ... \cup b_r$. We will say that an *embedded partition* of a multiset $b \in X^-$ is defined if its partition $b_1, ... b_r$ is defined and, for every nonsingleton (containing more than one element) multiset b_i, also an embedded partition is defined. Then, if an embedded partition of a multiset $a \in X^-$ is defined, then the function $f(a)$ can be calculated as follows. First, one calculates the values of $g(x)$ for ever $x \in a$ (without regard to multiplicity). Next, taking into account Remark 4.1, for every element $b = b_1 \cup ... \cup b_r$ of the embedded partition, one calculates the value of $g(b) = e_r(g(b_1), ..., g(b_r))$. In this case, the function e_r itself can be calculated iteratively with the use of the function e: for $r > 2$, we have $e_r(g(b_1), ..., g(b_r)) = e(e_r - 1(g(b_1), ..., g(b_r - 1)), g(b_r))$. After the value of $g(a)$ is obtained, we calculate the required result $f(a) = h(g(a))$.

5. DESCRIPTION OF THE PULSATION ALGORITHM AND AN ESTIMATE FOR ITS OPERATING TIME

The purpose of the pulsation algorithm is to calculate a value of a given function of a multiset $a \in X^-$ of values recorded at the vertices of a graph. We will assume that a value $x(i)$ with unit multiplicity is

recorded at each vertex i. The pulsation algorithm uses the marking of the graph made by the traversal algorithm described above. This marking includes direct and back spanning trees of the graph: at each vertex, direct arcs and (if this is not the root) one back arc are marked. In addition, a value of the *counter of incoming back arcs* is set at each vertex.

5.1. Description of the Pulsation Algorithm

The pulsation algorithm uses two types of messages: *Question* and *Answer*. First, a *Question* message containing an indication of three functions h, e and g arrives at the root automaton from outside. After the completion of the algorithm, the root sends outside an *Answer* message with parameter $f(a)$.

The *Question* message is distributed from the root to all vertices over the direct spanning tree. Having received this message, the root stores the parameters h, e, and g and sends the *Question* message along each outgoing arc with parameters e and g. In addition, each nonroot vertex, having received the *Question* message, stores the parameters e and g and forwards the *Question* message along each outgoing direct arc.

The *Answer* message is sent from all vertices to the root over the back spanning tree. The back spanning tree defines an embedded partition of the multiset a.

A leaf vertex i of the back spanning tree (at this vertex, the *counter of incoming back arcs* is equal to zero), having received a *Question* message, calculates $g(x(i))$ and sends it along a back arc as a parameter of the *Answer* message.

An internal (nonleaf) vertex i of the back spanning tree corresponds to an element $b_i = b_{i1} \cup \ldots \cup b_{ir}$ of the embedded partition, where $r - 1$ is equal to the number of back arcs incoming to this vertex. The task of this vertex is to calculate the value of $g(b_i)$; in this case, for $j = 1, \ldots, r - 1$, the value of $g(b_{ij})$ will be obtained along the jth incoming back arc, and $b_{ir} = g(x(i))$. Having received a *Question* message, the automaton of such a vertex calculates $g(x(i))$ and stores it as an *intermediate result* $y(i)$ and copies the *counter of incoming back arcs* into the *counter of answers*. Further, when obtaining an *Answer* message along an incoming back arc j with parameter $g(b_{ij})$, the *intermediate result* $y(i) := e(g(b_{ij}), y(i))$ is changed, and the *counter of answers* decreases by 1.

If i is a nonroot vertex, then, when the *counter of answers* is zeroed, an *Answer* message with parameter $y(i) = g(b_i)$ is sent along an outgoing back arc. If a vertex i is the root, then, when the *counter of answers* is zeroed, the final value of $f(a) = h(y(i))$ is calculated and is sent outside as a parameter of the *Answer* message.

5.2. Estimate for the Operating time of the Pulsation Algorithm

We can always consider that messages are sent from vertex in the order of decreasing priorities: (1) *Question* and (2) *Answer*.

Question messages are distributed over the direct spanning tree, and each message passes a simple path of length of at most D. Only *Answer* messages propagating along back arcs can be sent simultaneously with the *Question* messages, and some arcs can be direct and back at the same time. Therefore, a *Question* message can wait at a vertex for the release of an arc occupied with the *Answer* message. However, since *Question* has higher priority than *Answer*, the waiting time does not exceed one time step. Hence, the distribution time of all *Question* messages is not greater than $2D$ time steps.

Consider the motion of *Answer* messages after all *Question* messages cease to propagate. Each *Answer* message moves along back arcs to the root and, thus, passes a simple path of length of at most D. Since the *Answer* messages are summed at each vertex i in the form of an *intermediate result* $y(i)$ and there are no other types, the propagation time of all *Answer* messages is not greater than D time steps.

Thus, the operating time of the pulsation algorithm does not exceed $3D = O(D)$.

6. CONCLUSIONS

The pulsation algorithm does not only calculate the value of the function $f(a)$, where a is a multiset of values recorded at the vertices of the graph, but also changes the state of the automata of the vertices: at each vertex i, there remains, as an *intermediate result*, the value of $g(b_i)$, where b_i is a multiset of values recorded at the vertices of the maximum subtree of the back spanning tree with a root at vertex i. It is clear that the pulsation algorithm can be used for installing the vertex automata in some other, for example, identical, states.

It is also possible to set up a problem of calculating a function of a sequence of values recorded at the vertices of the graph, rather than a function of a multiset. For this purpose, one should define a certain linear order of vertices. At the same time, we already have a numbering of arcs outgoing from each vertex. Such numbering defines a natural partial order of vertices in the form of a direct spanning tree. Such partial order can be used as a basis of the corresponding linear order. However, one can set up a problem of calculating a function with regard to this natural order, rather than a linear order. This means that the value of a function may depend on the order of the values at vertices, which are comparable in partial order, but should not depend on the order of values at the vertices, which are not comparable in partial order.

However, first, the traversal algorithm described above not necessarily constructs the direct spanning tree that is defined by the numbering of arcs, and, second, the pulsation algorithm performs calculations on the back, rather than the direct, spanning tree. It is this problem that is the focus of the development of an algorithm for calculating a function for given partial or linear order of vertices of the graph. The solution of this problem can be the subject of future study.

REFERENCES

1. Skiena, S.S., *The Algorithm Design Manual*, New York: Springer, 1997.
2. 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 [*Program. Comput. Software* (Engl. Transl.), vol. 29, no. 5, pp. 245–258].
3. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant algorithms for traversing directed graphs: The nondeterministic case, *Programmirovanie,* 2004, no. 1, pp. 59–69 [*Program. Comput. Software* (Engl. Transl.), vol. 30, no. 1, pp. 2–17].
4. Rabin, M.O., *Maze Threading Automata: An unpublished lecture presented at MIT and UC*, Berkeley, 1967.
5. Bourdonov, I.B., Traversal of an unknown directed graph by a finite robot, *Programmirovanie*, 2004, vol. 30, no. 4, pp. 11–34 [*Program. Comput. Software* (Engl. Transl.), vol. 30, no. 4, pp. 188–203].
6. Bourdonov, I.B., Backtracking problem in the traversal of an unknown directed graph by a finite robot, *Programmirovanie*, 2004, no. 6, pp. 6–29 [*Program. Comput. Software* (Engl. Transl.), vol. 30, no. 6, pp. 305–322].
7. Bourdonov, I.B., and Kossatchev A.S., Traversal of an unknown graph by a collective of automata), *Proc. Int. Supercomput. Conf. "Scientific Service in Internet: All Facets of Parallelism,*|2013, pp. 228–232 (in Russian).
8. Bourdonov, I.B., and Kossatchev A.S., Traversal of an unknown Graph by a Collective of Automata, *Tr. Inst. Syst. Program.*, 2014, no. 27, pp. 43–86.
9. Kushnerenko, A.G., and Lebedev, G.V., *Programmirovanie dlya matematikov* (Programming for Mathematicians), Moscow: Nauka, 1988.

Translated by I. Nikitin

SPELL: OK