UDC 519.172

PARALLEL IMPLEMENTATION OF ITALIANO'S INCREMENTAL ALGORITHM FOR DYNAMIC UPDATING THE TRANSITIVE CLOSURE

A.S. Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics,
Siberian Division of Russian Academy of Sciences,
630090, Novosibirsk, Russia, pr. Lavrentieva, 6,
Fax: (383) 330 8783; Phone: (383) 330 8994.
E-mail: anep@ssd.sscc.ru

The transitive closure (or reachability) problem in a directed graph consists in finding whether there is a path between any two vertices. In this paper, we first study the problem of parallelization of Italiano's algorithm for dynamic updating the transitive closure after inserting a new arc into the graph represented as a list of arcs. To this end, by means of the data structure first proposed in [9], Italiano's incremental algorithm is represented in a natural way on a model of an associative parallel processor with vertical processing (the STAR-machine). Associative version of Italiano's incremental algorithm is given as procedure InsertArc for the STAR-machine. We prove correctness of this procedure and evaluate its time complexity. We also compare implementations of Italiano's incremental algorithm and its associative version and present the main advantages of the associative version.

Проблема транзитивного замыкания (или достижимости) в ориентированном графе состоит в определении того, существует ли путь между любыми двумя вершинами. В данной статье впервые исследуется задача параллельной реализации алгоритма Итальяно для динамической обработки транзитивного замыкания после добавления к графу новой дуги для случая, когда граф задается в виде списка дуг. С этой целью с помощью структуры данных, впервые предложенной в работе [9], инкрементальный алгоритм Итальяно естественным образом представляется на модели ассоциативного параллельного процессора с вертикальной обработкой данных (STAR- машине). Ассоциативная версия инкрементального алгоритма Итальяно задается в виде процедуры InsertArc для STAR- машины. Доказывается корректность этой процедуры и оценивается ее временная сложность. Также проводится сравнение выполнения инкрементального алгоритма Итальяно и его ассоциативной версии и приводятся основные преимущества ассоциативной версии.

Introduction

In many applications, graphs are subject to discrete changes, such as insertions and deletions of edges or vertices. The goal of a dynamic algorithm is to update efficiently the solution of a problem after dynamic changes rather than to recompute the entire graph from scratch each time. An algorithm is called *fully dynamic* if the update operations include both insertions and deletions of edges or vertices, and it is called *partially dynamic* if only one type of an update, either insertions or deletions, is allowed. A partially dynamic algorithm is called *incremental* if it supports only insertions, while it is called *decremental* if it supports only deletions.

The transitive closure problem in a directed graph G with n vertices and m edges consists in finding whether there is a path between any two vertices in G. In the fully dynamic transitive closure problem a directed graph is updated under an intermixed sequence of edge insertions, edge deletions, and two types of queries: a *Boolean* query for vertices i and j that returns *yes* if there is a path from i to j and *no* otherwise, and a *path* query that returns an actual path from i to j if it exists.

We focus on incremental algorithms for the transitive closure problem. The first incremental algorithm was given by Ibaraki and Katoh [1]. Their algorithm takes $O(n^3)$ time over any sequence of insertions. For a sequence of m insertions, Italiano [2] and La Poutr'e and Leeuwen [3] improved this estimation to $O(mn)$, time, where m is the number of edges in the final graph. In [3], Yellin proposed an incremental algorithm for bounded degree graphs which requires $O(dm^*)$ time for m insertions, where d is the maximum outdegree of the final graph and m^* is the number of edges in the final transitive closure graph. All of these algorithms perform a Boolean query in $O(1)$ time. The incremental algorithm of La Poutr'e and Leeuwen [3] does not support a path query but other above-mentioned algorithms perform a path query in time proportional to the length of the path.

In [4], Frigioni et al. presented an experimental study of a group of dynamic algorithms for the transitive closure. In particular, the authors proposed a variant of Italiano's algorithms [2, 5], called *Ital-Gen*, whose decremental part applies to a general graph and any sequence of edge deletions takes $O(m^2)$ worst-case time. As shown in [4], in the case of path queries, Italiano's incremental algorithm was practically always the fastest among the dynamic algorithms of Yellin, La Poutr'e and Leeuwen, Ital-Gen, and a randomized algorithm of Henzinger and King [6], while for dense directed acyclic graphs (DAGs) Italiano's decremental algorithm was better than the other algorithms. For sparse DAGs, the other algorithms including Ital-Gen are faster than Italiano's decremental algorithm.

In [7], we proposed a data structure for implementing in a natural way Italiano's decremental algorithm for updating the transitive closure on associative (or content addressable) parallel processors. Such an architecture is mainly oriented to solve non-numerical problems. We simulate the run of associative parallel systems with vertical processing

by means of the STAR-machine [8]. Following Foster [9], *time complexity* of an algorithm is measured by counting all elementary operations of the STAR-machine (its microsteps) performed in the worst case.

In this paper, we provide an associative version of Italiano's incremental algorithm for dynamic updating the transitive closure. The associative version of Italiano's incremental algorithm is given as a procedure **InsertArc**, whose correctness is proved. We show that on the STAR-machine, this procedure takes $O(n \log n)$ time per an insertion. We also obtain that the associative algorithm performs Boolean and path queries in the same time as Italiano's incremental algorithm. Finally, we compare implementations of Italiano's incremental algorithm and its associative version and enumerate the main advantages of the associative version.

1. A model of associative parallel machine

Here, we propose a short description of the model. It is defined as an abstract STAR-machine of the SIMD type with the vertical data processing [8]. It consists of the following components:

- a sequential control unit (CU), where programs and scalar constants are stored;
- an associative processing unit consisting of p single-bit processing elements (PEs);
- a matrix memory for the associative processing unit.

The CU passes an instruction to all PEs in one unit of time. All active PEs execute it in parallel while inactive PEs do not perform it. An activation of a PE depends on the data.

Input binary data are loaded in the matrix memory in the form of two-dimensional tables in which each data item occupies an individual row and it is updated by a dedicated processing element. The rows are numbered from top to bottom and the columns – from left to right. Both a row and a column can be easily accessed. Some tables may be loaded in the matrix memory.

An associative processing unit is represented as h vertical registers each consisting of p bits. Vertical registers can be regarded as a one-column array. The bit columns of the tabular data are stored in the registers which perform the necessary Boolean operations.

Its run is described by means of the language STAR being an extension of Pascal. Let us briefly consider the STAR constructions needed for the paper. To simulate the data processing in the matrix memory, we use data types **word**, **slice**, and **table**. Constants for the types **slice** and **word** are represented as a sequence of symbols of the set {0, 1} enclosed within single quotation marks. The types **slice** and **word** are used for the bit column access and the bit row access, respectively, and the type **table** is used for defining the tabular data. Assume that any variable of the type **slice** consists of p components which belong to {0, 1}. For simplicity let us call *slice* any variable of the type **slice**.

Now we present some elementary operations and a predicate for slices.

Let X, Y be variables of the type **slice** and i be a variable of the type **integer**. We use the following operations:

SET(Y) sets all components of Y to '1';
CLR(Y) sets all components of Y to '0';
$Y(i)$ selects the i-th component of Y;
FND(Y) returns the ordinal number i of the first (or the uppermost) '1' of Y;
STEP(Y) returns the same result as FND(Y) and then resets the first found '1' to '0'.

It should be noted that operations SET(Y) and CLR(Y) are used as a separate statement. The operations FND(Y) and STEP(Y) are used as the right part of the assignment statement while the operation $Y(i)$ can be used both in the left part and in the right part of the assignment statement.

In the usual way, we introduce the predicate SOME(Y) and the bitwise Boolean operations: X and Y, X or Y, not Y, X xor Y.

Note that the predicate SOME(Y) and all operations for the type **slice** are also performed for the type **word**.

We will also employ the bitwise Boolean operations between a variable w of the type **word** and a variable Y of the type **slice,** where the number of bits in w coincides with the number of bits in Y.

Let T be a variable of the type **table**. We employ the following elementary operations:

ROW(i,T) returns the i-th row of the matrix T;
COL(i,T) returns its i-th column.

Remark 1.

Note that the STAR statements are defined in the same manner as for |Pascal. We will use them later for presenting our procedures.

2. Preliminaries

Let us present some notions being used in the paper.

Let $G = (V, E)$ be a *directed graph* (digraph) with the set of vertices $V = \{1, 2, ..., n\}$ and the set of directed edges (arcs) E. We assume that $|V| = n$, and $|E| = m$.

An arc e from i to j is denoted by $e = (i, j)$, where the vertex i is the *head* of e (or *father*) and the vertex j is its *tail* (or *son*).

A sequence of arcs $e_1, e_2, ..., e_k$ is a *path* from the head of e_1 to the tail of e_k, if the tail of e_i is the head of e_{i+1} for $1 \le i \le k$-1.

A vertex v is *reachable* from u if there is a directed path from u to v (u-v path). In such a case u is called an *ancestor* of v and v is called a *descendant* of u.

The *transitive closure* of a directed graph $G = (V, E)$ is a directed graph $G^* = (V, E^*)$ such that an arc $(u,v) \in E^*$ if and only if the vertex v is reachable from u in G.

A *spanning tree* T_u is a connected acyclic subgraph of G with the root vertex u that cointains all descendants of u.

3. Italiano's incremental algorithm for updating the transitive closure

We first recall the data structure proposed by Italiano [2] to support the efficient insertion of arcs in a digraph and the Boolean and the path queries.

For every vertex $u \in V$, $Desc[u]$ is a spanning tree with the root u. The transitive closure of a graph G is represented as a set of all $Desc[u]$. In addition, an $n \times n$ matrix of pointers *Index* is maintained which allows fast access to vertices in these trees. This matrix is defined as follows. Its every component $Index[i, j]$ points to the vertex j in the spanning tree $Desc[i]$ if $j \in Desc[i]$ and it is a *Null* pointer otherwise.

Now we explain the main idea of Italiano's incremental algorithm.

Let a new arc $\gamma = (i, j)$ be added to a digraph G. The data structure is updated only if there is no previous path from i to j. Insertion of an arc may create new paths from any ancestor r of the vertex i to any descendant of the vertex j if there was no previous path from r to j in G. In this case, the spanning tree $Desc[r]$ is maintained taking into account the descendants of j and the r-th row of the matrix *Index*. Namely, the common vertices in the trees $Desc[r]$ and $Desc[j]$ are deleted from the copy of $Desc[j]$. Then the pruned copy of $Desc[j]$ is linked to the vertex i in $Desc[r]$.

A Boolean query for vertices i and j is performed in $O(1)$ time by checking $Index[i, j]$. If every vertex in each spanning tree is provided with an additional pointer to the parent, then a path query is carried out by means of a bottom-up traversal in $Desc[i]$ from j to the root i and it takes $O(l)$ time, where l is the length of i - j path.

4. Associative version of Italiano's incremental algorithm

In this section, a graph is represented as association of matrices *Left* and *Right*, where every arc (u, v) occupies an individual row, and $u \in Left$ and $v \in Right$.

To design the associative version of Italiano's incremental algorithm, we use the following data structure first proposed in [7]:

- an association of matrices *Left* and *Right* and a global slice *X*, where positions of arcs belonging to G are marked with '1';
- an $n \times \log n$ matrix *Code*, whose every i-th row saves the binary representation of the vertex i;
- an $m \times n$ Boolean matrix *Trans*, whose every i-th column saves by '1' the positions of arcs belonging to the spanning tree T_i;
- an $n \times n$ Boolean matrix *Nodes*, whose every i-th column saves by '1' the positions of vertices that belong to the spanning tree T_i.

Let us enumerate the following two properties of matrices *Nodes* and *Trans*.

Fact 1. In every i-th row of the matrix *Nodes*, the roots of spanning trees that include the vertex i, are marked with '1'.

Fact 2. In every i-th row of the matrix *Trans*, the roots of all spanning trees that include the arc written in the i-th row of the graph representation are marked with '1'.

Let an arc (i, j) be added to the graph G. Let a spanning tree T_r include the vertex i and not include the vertex j. We first present the associative parallel algorithm that updates the spanning tree T_r after adding the arc (i, j) to the graph G. It performs the following steps.

Step 1. By means of a slice, say Z, save positions of vertices from the spanning tree T_j that do not belong to T_r. Then add these vertices to the r-th column of the matrix *Nodes*.

Step 2. For every vertex $p \ne j$ selected by '1' in the slice Z, determine the position of an arc from T_j entering this vertex and include this arc into T_r.

On the STAR-machine, this algorithm is implemented as a procedure **ChangeTree**.

Now we propose the associative parallel algorithm that updates the graph after adding the arc (i, j). It runs as follows.

Step 1. Include the position of the arc (i, j) into the association of matrices *Left* and *Right*.

Step 2. Determine the roots of trees that include the vertex i and do not include the vertex j. Let such roots be marked with '1' in a row, say w.

Step 3. Include the position of the arc (i,j) into those spanning trees of the matrix *Trans* whose roots correspond to '1' in the row w.

Step 4. While *w* does not consist of zeros, save the position *r* of its leftmost bit '1'. Then set '0' in the *r*-th bit of *w*. Further update the spanning tree T_r by means of the associative algorithm proposed above.

On the STAR-machine, this algorithm is implemented as a procedure **InsertArc**.

5. Implementation of the associative version of Italiano's incremental algorithm on the STAR-machine

In this section, we present the procedures **ChangeTree** and **InsertArc** and prove their correctness.

We first consider the procedure **ChangeTree** that maintains a spanning tree after inserting a new arc to the graph.

Now we propose the following procedure.

```
procedure ChangeTree(Right: table; Code: table; r,j: integer; var Nodes: table;
    var Trans: table);
/* The spanning tree Tr will be updated after inserting the arc (i,j) into the graph. */
var X1,X2,Y: slice(Left);
     Z,Z1,Z2,Z3: slice(Nodes);
     q,p: integer;
     v: word(Code);
1. Begin Z1:= COL(r,Nodes); Z2:= COL(j,Nodes);
2.       Z3:= Z1 and Z2;
   /* The slice Z3 saves the vertices belonging to the spanning tree Tr and the spanning tree Tj. */
3.       Z:= Z2 and (not Z3);
   /* The slice Z saves vertices from Tj that will be included into the spanning tree Tr. */
4.       Z1:= Z1 or Z;
5.       COL(r,Nodes):= Z1;
   /* The new vertices for the spanning tree Tr are added to the matrix Nodes. */
6.       Z(j):='0';
   /* The vertex j is deleted from the slice Z. */
7.       Y:= COL(r,Trans);
   /* The slice Y saves positions of arcs from Tr */
8.       X1:= COL(j,Trans);
   /* The slice X1 saves positions of arcs from Tj. */
9.       while SOME(Z) do
10.           begin q:=STEP(Z); v:= ROW(q,Code);
11.               MATCH(Right,X1,v,X2);
12.               p:= FND(X2);
13.               Y(p):='1';
   /* We include into the slice Y the arc from the p-th position of the graph representation
      that enters the vertex q. */
14.           end;
15.      COL(r,Trans):= Y;
16.  End;
```

Proposition 1. Let a directed graph G be given as association of matrices Left and Right along with the global slice X, and its transitive closure be given as the matrix Trans. Let matrices Code and Nodes be also given. Let an arc (i,j) be added to the spanning tree T_r. Let Z be a slice that saves by '1' vertices from the spanning tree T_j that do not belong to T_r. Then after performing the procedure ChangeTree, the vertices from the slice Z are added to the r-th column of the matrix Nodes and positions of arcs from T_j entering vertices from Z are added to the r-th column of the matrix Trans.

Proof (**Sketch**.) We prove this by contradiction. Let all conditions of proposition 1 be performed. However, there is such a vertex $s \in T_j$ that $s \notin T_r$ and after execution of the procedure **ChangeTree** we obtain the following two properties:

1) the spanning tree T_r does not include the vertex *s*, that is, the *s*-th bit of the *r*-th column in the matrix *Nodes* is equal to '0' ;

2) the arc, entering the vertex *s*, does not belong to T_r, that is, the position of the arc from T_j entering the vertex *s* is marked with '0' in the *r*-th column of the matrix *Trans*.

We will prove that these properties contradict to execution of the procedure **ChangeTree**.

Let us assume that the first property is correct. One can immediately check that after performing lines 1–2, the slice *Z*3 saves by '1' the vertices that belong to T_r and T_j. Since $s \notin T_r$, we obtain that *Z*3(*s*) = '0'. After performing line 3, the vertices from T_j not belonging to T_r will be marked with '1' in the slice *Z*. Since by the assumption $s \in T_j$ and $s \notin T_r$, we obtain that *Z*(*s*) = '1'. Therefore after fulfilling line 4, we obtain that *Z1*(*s*) = *'1'*. Hence after performing line 5, the *s*-th bit in the *r*-th column of the matrix *Nodes* is equal to '1'. Since the execution of lines 6-16 does not change the

matrix *Nodes*, we obtain the contradiction with the assumption that $s \notin T_r$ after execution of the procedure **ChangeTree**.

Now we assume that the second property is correct. After performing lines 7–8, the slice *Y* saves the spanning tree T_r and the slice *X*1 saves the spanning tree T_j. Let us analyze the execution of the cycle in line 9 for $q=s$. After performing lines 10-12, we determine the position *p* of an arc, say γ, entering the vertex *s* in T_j. Since every vertex in a tree has a unique father, we mark the position of γ with '1' in the slice *Y*. Obviously, after fulfilling line 15, the position of γ entering the vertex *s* in T_j will be marked with '1' in the spanning tree T_r. It contradicts to the second property.

This completes the proof.

If an arc (*i*, *j*) is inserted into the graph, we maintain all spanning trees that include the vertex *i* and do not include the vertex *j*.

Now, we provide the following procedure.

```
procedure InsertArc(Code: table; i, j: integer; var Left, Right: table; var X: slice(Left);
  var Trance: table; var Nodes: table);
/* Here, the arc (i,j) will be included into the given graph. */
var w,w1,w2: word(Nodes);
    v1,v2: word(Code);
    r,k: integer;
1. Begin v1:=ROW(i,Code); v2:= ROW(j,Code);
2.    k:=FND(notX); X(k):='1';
3.    ROW(k,Left):=v1; ROW(k,Right):= v2;
    /* The arc  (i,j)  is written in the  k-th row of matrices Left and Right. */
4.    w1:=ROW(i,Nodes); w2:= ROW(j,Nodes);
5.    w:=w1and(notw2);
    /* The word  w  saves by  '1'  the roots of trees that will be changed after inserting
      the arc (i,j) into the graph. */
6.    ROW(k,Trans):= w;
    /* The arc (i,j) is simultaneously included into all trees marked with '1' in w. */
7.    while SOME(w) do
8.        begin r:= STEP(w);
9.          ChangeTree(Right,Code,r,j,Nodes,Trans);
10.       end;
11.End;
```

Proposition 2. Let a directed graph G be given as association of matrices Left and Right along with the global slice X, and its transitive closure be given as the matrix Trans. Let matrices Code and Nodes be also given. Let an arc γ =(i,j) be inserted into the graph G. Then, after performing the procedure InsertArc, the position of the arc γ is marked with '1' in the slice X. Moreover, every spanning tree that includes the vertex i and does not include the vertex j is updated as shown in Proposition 1.

Proof (**Sketch**.) We prove this by induction on the number of spanning trees *l* in the matrix *Trans* being changed after insertion of the arc γ into the graph *G*.

Basis is checked for $l = 1$. After performing lines 1 – 3, we determine the position of the row in the graph representation, where γ will be written, and mark it with '1' in the slice *X*.

In view of Fact 1, after fulfilling lines 4-5, the row *w* saves the roots of spanning trees that will be changed after including γ in *G*. In view of Fact 2, after performing line 6, the arc γ is simultaneously included into all spanning trees whose roots are marked with '1' in *w*. Since $l = 1$, the cycle in line 7 performs only once. Here, we first determine the root *r* of a spanning tree that will be updated (line 8). Since the position of the arc γ has been included into T_r, we can apply the procedure **ChangeTree**. After its execution, the updated spanning tree T_r will be written in the *r*-th column of the matrix *Trans* and the updated set of its vertices will be written in the *r*-th column of the matrix *Nodes*.

Step of induction. Let the assertion be true for $l \geq 1$ spanning trees being changed after inserting γ into G. We will prove it for $l+1$ spanning trees. By analogy with the basis after performing lines 1 – 6, the arc (*i*, *j*) has been included into the graph representation, its position has been marked with '1' in the slice *X*, the row *w* saves roots of trees being changed after inserting γ into *G*, and the arc γ has been included into all spanning trees whose roots are marked with '1' in *w*. By the inductive assumption, after maintaining the first *l* spanning trees, whose roots are marked with '1' in *w*, the changed *l* trees will be written in the corresponding columns of the matrix *Trans* and the changed sets of their vertices will be written in the corresponding columns of the matrix *Nodes*. Since there is a single bit '1' in *w*, we determine the root of the last $(l + 1)$-th spanning tree and maintain it by means of the procedure **ChangeTree**.

This completes the proof.

Now we evaluate time complexity of the procedure **InsertArc**.

To this end, we have to determine the total number of vertices being updated after inserting an arc to the transitive closure. In view of performing the procedure **ChangeTree**, at most all vertices of a subtree rooted at the tail of the inserted arc are updated. Therefore the procedure **InsertArc** takes $O(nlog\ n)$ time per an insertion, where the

factor *log n* appears due to the use of the basic procedure **MATCH**. One can check that space complexity of the procedure **InsertArc** is $O(mn)$ bits.

On the STAR-machine, a Boolean query for vertices *i* and *j* is carried out in $O(1)$ time by checking the *j*-th bit of the *i*-th column in the matrix *Nodes*. A path query for vertices *i* and *j* is performed by means of a bottom-up traversal in the spanning tree T_i from the vertex *j* to the root *i* using the procedure **MATCH**. It takes $O(llogn)$ time, where *l* is the length of the path.

Let us compare two implementations.

- Italiano's incremental algorithm checks all vertices in each spanning tree to determine whether it includes the vertex *i* and does not include the vertex *j*. The associative version simultaneously determines the roots of trees that include the vertex *i* and do not include the vertex *j*.
- To determine the vertices from the spanning tree T_j that should be added to T_r, Italiano's incremental algorithm checks whether any vertex from T_j belongs to T_r. The associative version simultaneously determines those vertices from T_j that should be added to T_r.
- To perform a path query, for every vertex *j* in every spanning tree, Italiano's incremental algorithm uses an additional pointer to its parent. The associative version determines the parent of any vertex by means of the basic procedure **MATCH**.
- To link the pruned copy of *Desk*[*j*] and the vertex *i* in *Desk*[*r*], Italiano's incremental algorithm updates the spanning tree *Desk*[*r*] and the matrix *Index*. The associative version determines the position of any arc from the pruned copy of T_j in the graph representation and includes it into the r-th column of the matrix *Trans*. Moreover, the positions of the new vertices are included into the p-th column of the matrix *Nodes*.

Conclusions

We have proposed a natural and efficient implementation of Italiano's incremental algorithm for dynamic updating the transitive closure on the STAR-machine having no less than *m* PEs. The associative version of Italiano's incremental algorithm is represented as procedure **InsertArc** whose correctness is proved. We have obtained that this procedure takes $O(nlogn)$ time per an insertion assuming that each microstep of the STAR-machine takes one unit of time. Space complexity of this procedure is $O(mn)$ bits.

We have also compared implementations of Italiano's incremental algorithm and its associative version and enumerated the main advantages of the associative version.

We are planning to design associative versions of both Italiano's algorithms for dynamic updating the transitive closure for the case when the given graph is represented as an adjacency matrix.

1. *Ibaraki T., Katoh N*. On–line computation of transitive closure for graphs // Information Processing Letters. – 1983. – V. 16. – P. 95–97.
2. *G. F. Italiano*. Amortized efficiency of a path retrieval data structure // Theoretical Computer Science. – 1986. – V. 48 (2–3). – P. 273–281.
3. *La Poutr'e J.A., van Leeuwen J.* Maintenance of transitive closure and transitive reduction of graphs // Proc. Workshop on Graph–Theoretic Conceps in Computer Science. Lecture Notes in Computer Science, Springer–Verlag, Berlin. – 1988. – Vol. 314. – P. 106–120.
4. *Frigioni D., Miller T, Nanni U. i ar*. An experimental study of dynamic algorithms for directed graphs // Proc. of the European Symp. on Algorithms, Algorithms–ESA'98, Lecture Notes in Computer Science. – 1998. – V. 1461. – P. 368–380.
5. *Italiano G.F.* Finding paths and deleting edges in directed acyclic graphs // Information Processing Letters. – 1988. – Vol. 28. – P. 5–11.
6. *Henzinger M.R., King V.* Fully dynamic biconnectivity and transitive closure // Proc. 36th IEEE Symposium on Foundations of Computer Science (FOCS'95). – 1995. – P. 664–672.
7. *Nepomniaschaya A.S.* Associative version of Italiano's decremental algorithm for the transitive closure problem // Proc. of 9–th Intern. Conf. PaCT 2007, Pereslavl–Zalessky, Russia, September 3–7, 2007. Lecture Notes in Computer Science, Springer–Verlag, Berlin. – 2007. – Vol. 4671. – P. 442–452.
8. *Nepomniaschaya A.S.* Language STAR for associative and parallel computation with vertical data processing // Proc. of the International Conference "Parallel Computing Technologies". World Scientific, Singapure. –1991. – P. 258 – 265.
9. *Foster C.C*. Content Addressable Parallel Processors. – New York: Van Nostrand Reinhold Company, 1976.