Computing and Informatics, Vol. 32, 2013, 331–354

EFFICIENT PARALLEL IMPLEMENTATION OF THE RAMALINGAM DECREMENTAL ALGORITHM FOR UPDATING THE SHORTEST PATHS SUBGRAPH

Anna Nepomniaschaya

Institute of Computational Mathematics and Mathematical Geophysics
Siberian Division of the Russian Academy of Sciences
pr. Lavrentieva 6
630090 Novosibirsk, Russia
e-mail: anep@ssd.sscc.ru

Communicated by Isabel Campos Plasencia

Abstract. We propose an efficient parallel implementation of the Ramalingam algorithm for dynamic updating the shortest paths subgraph of a directed weighted graph with a sink after deletion of an edge. To this end, a model of associative (content addressable) parallel systems with vertical processing (the STAR-machine) is used. On the STAR-machine, the Ramalingam decremental algorithm for dynamic updating the shortest paths subgraph is represented as the main procedure DeleteArc that uses a group of auxiliary procedures. We provide the DeleteArc procedure along with the auxiliary procedures, prove correctness of these procedures and evaluate the time complexity. We also consider an example of implementing the DeleteArc procedure on the STAR-machine.

Keywords: Directed weighted graph, subgraph of the shortest paths, adjacency matrix, decremental algorithm, associative parallel processor, access data by contents, time complexity

Mathematics Subject Classification 2010: 05C20, 05C38, 05C85

1 INTRODUCTION

In many applications, graphs are subject to discrete changes, such as insertions and deletions of edges or vertices. The objective of a dynamic algorithm is to efficiently update the solution to 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* (*semi-*) *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 problem of finding the shortest paths in a directed weighted graph arises in practice in different application settings. In particular, if a graph represents a communication or transport network, then an insertion or deletion of an arc reflects such real changes in the network as insertion or deletion of connections during its existence. There are two versions of this problem: finding the single source shortest paths and finding the all-pairs shortest paths. The most general types of update operations for the single source shortest paths problem include insertions and deletions of edges, update operations on the weight of edges, insertions or deletions of isolated vertices [7]. The typical operations for the all-pairs shortest paths problem include update operations on weights, finding the shortest distance and finding the shortest path between two vertices, if any.

In the case of positive edge weights, several solutions have been proposed for the dynamic maintenance of the shortest paths. Ausiello et al. [1] propose an efficient solution for the all-pairs incremental problem assuming that edge weights are restricted in the range of integers $[1..C]$. Chaudhuri and Zaroliagis [2] devise efficient solutions for the all-pairs shortest paths problem for bounded treewidth graphs when the weight of edges changes. Klein et al. [9] propose a fully dynamic solution to maintain all-pairs shortest paths for planar graphs with unrestricted edge weights. Franciosa et al. [5] devise fast algorithms that maintain a single source shortest paths tree (*sp-tree*) of a general directed graph with integer edge weights in the range of integers $[1..C]$ during a sequence of edge deletions or a sequence of edge insertions.

In the case of arbitrary real edge weights, Ramalingam and Reps [17, 18] devise fully dynamic algorithms for updating the single source shortest paths using the output bounded model. In this model, the running time of an algorithm is analyzed in terms of the output change rather than the input size. The authors assume that the graph has no negative-length cycles before and after input update. Frigioni et al. [7] study the semi-dynamic single source shortest paths problem for both directed and undirected graphs with positive real edge weights in terms of the output complexity. The decremental solution works only for planar graphs, while the incremental solution works for any graph and its complexity depends on the existence of a k-bounded accounting function for the graph. Frigioni et al. [6] propose fully dynamic algorithms for updating the distances and an sp-tree in either a directed or an undirected graph with positive real edge weights under arbitrary sequences of edge updates. The cost of the update operations is given as a function of the number of output updates by using the notion of k-bounded accounting function. For general graphs with n vertices and m edges the algorithms require $O(\sqrt{m}\log n)$ worst case time per output update. Frigioni et al. [8] propose the fully dynamic solution for the problem of updating the shortest paths from a given source in a directed graph with arbitrary edge weights. The authors devise a new algorithm for performing edge deletions and weight increases that explicitly deals with zero-length cycles. They also propose an algorithm for handling edge insertions and weight decreases that explicitly deals with negative-length cycles. The cost of the update operations is evaluated as a function of the structural property of the graph and of the number of output updates. Algorithms from [5-8, 17, 18] use the dynamic version of the Dijkstra algorithm [3]. Narváez et al. [11] study a group of algorithms for dynamic maintaining an sp-tree after performing the update operations on the edge weights. The authors propose two incremental methods to transform the well-known static algorithms of Dijkstra and Bellman-Ford into new dynamic algorithms. In [16], we propose an associative version of the Ramalingam decremental algorithm for the dynamic update of the shortest paths subgraph $SP(G)$ [17] that consists of all shortest paths from every vertex to the sink. We describe the associative algorithm by means of the STAR-machine that simulates the run of associative (content addressable) parallel systems of the SIMD type with bit-serial (vertical) processing and simple single-bit processing elements. Following [4], we assume that each elementary operation of our model (its microstep) takes one unit of time. We measure the *time complexity* of an associative algorithm by counting all elementary operations performed in the worst case. The associative version of the Ramalingam decremental algorithm is given as a group of algorithms that provide the execution of different parts of the Ramalingam decremental algorithm on the STAR-machine. Moreover, we present the main advantages of the associative version of the Ramalingam decremental algorithm [17].

The main objective of this paper is to provide an efficient parallel implementation on the STAR-machine of the Ramalingam decremental algorithm mentioned above. The associative version is represented as the main procedure DeleteArc that makes use of a group of auxiliary procedures. We prove correctness of the DeleteArc procedure and all its parts. We obtain that this procedure takes $O(hk)$ time, where h is the number of bits required for coding the maximal weight of the shortest paths to the sink and k is the number of vertices, whose shortest paths to the sink change in $SP(G)$ after deleting an edge from the given graph G. We also provide an example of implementing the DeleteArc procedure on the STAR-machine.

2 MODEL OF ASSOCIATIVE PARALLEL MACHINE

Here, we propose a brief description of our model which is based on a Staran-like associative parallel processor [4, 10]. It is defined as an abstract STAR-machine of the SIMD type with vertical data processing [12]. In [14], we compare different models for vertical processing systems. Our model consists of the following components (Figure 1):

- 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. Activation of a PE depends on the data.

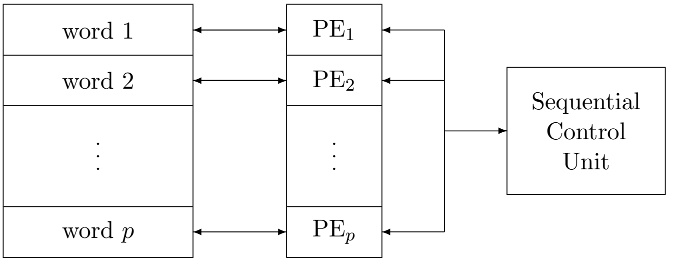

Fig. 1. The STAR-machine

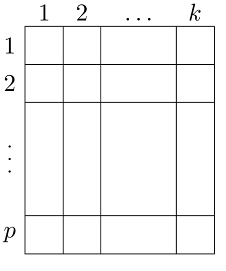

Fig. 2. Data array

Input binary data are loaded to the matrix memory in the form of 2D tables, where each data item occupies an individual row and is updated by a dedicated processing element (Figure 2). We assume that the number of PEs is not less than the number of rows in an input table. 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 to the matrix memory.

An associative processing unit is represented as h ($h \geq 4$) vertical registers each consisting of p bits (Figure 3). A vertical register can be regarded as a one-column array. The STAR-machine runs as follows. The bit columns of the tabular data are stored in the registers which perform the necessary Boolean operations.

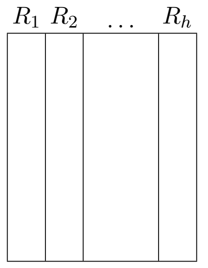

Fig. 3. Associative processing unit

To simulate data processing in the matrix memory, we use data types **word**, **slice**, and **table**. Constants for the **slice** and **word** types are represented as a sequence of symbols of $\{0, 1\}$ in single quotation marks. The **slice** and **word** types are used for the bit column access and the bit row access, respectively, and the **table** type 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 **slice** type.

Let us present the main operations for slices.

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

- SET(Y) simultaneously sets all components of Y to '1';
- CLR(Y) simultaneously sets all components of Y to '0';
- $Y(i)$ selects the value of the i^{th} component of Y;
- FND(Y) returns the ordinal number i of the first (the uppermost) bit '1' of Y, $i \geq 0$;
- STEP(Y) returns the same result as FND(Y) and then resets the first found '1' to '0';
- CONVERT(Y) returns a row, whose every i^{th} bit coincides with $Y(i)$. It is applied when a row of one matrix is used as a slice for another matrix.

The operations FND(Y), STEP(Y), and CONVERT(Y) are used only as the right part of the assignment statement, while the operation $Y(i)$ is used as both the right part and the left part of the assignment statement.

To carry out data parallelism, we introduce in the usual way the bitwise Boolean operations: X and Y, X or Y, $\text{not}\, Y$, $X \,\text{xor}\, Y$. We also use the predicate SOME(Y) that results in **true** if there is at least a single bit '1' in the slice Y. For simplicity, the notation $Y \neq \emptyset$ denotes that the predicate SOME(Y) results in **true**.

Note that the predicate SOME(Y) and all operations for the **slice** type are also performed for the **word** type. We will also employ the bitwise Boolean operations between a variable w of the **word** type and a variable Y of the **slice** type, where the number of bits in w coincides with the number of bits in Y.

Let T be a variable of the **table** type. 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.

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

Now, we recall a group of basic procedures [13, 15] implemented on the STAR-machine. These procedures use the given global slice X to indicate with bit '1' the row positions used in the corresponding procedure.

The procedure MATCH(T, X, v, Z) defines positions of those rows of the given matrix T which coincide with the given pattern v (Figure 4). It returns the slice Z, where $Z(i) =$ '1' if and only if ROW$(i, T) = v$ and $X(i) =$ '1'.

v				
	1	0	1	1

T				X	Z
1	0	1	0	1	0
0	0	1	1	0	0
1	0	1	1	1	1
1	0	1	1	0	0
1	0	1	1	1	1
1	0	1	0	1	0

Fig. 4. Testing $v \in T$

The procedure MIN(T, X, Z) defines positions of those rows of the given matrix T where the minimal element is located. It returns the slice Z, where $Z(i) =$ '1' if and only if ROW(i, T) is the minimal element in the matrix T and $X(i) =$ '1'.

The procedure SETMIN(T, F, X, Z) defines positions of the given matrix T rows that are less than the corresponding rows of the matrix F. It returns the slice Z, where $Z(j) =$ '1' if and only if ROW$(j, T) <$ ROW(j, F) and $X(j) =$ '1'.

The procedure TCOPY1(T, j, h, F) writes h columns from the given matrix T, starting from the $(1 + (j - 1)h)^{\text{th}}$ column, into the resulting matrix F, where $j \geq 1$.

The procedure ADDV(T, F, X, R) writes into the matrix R the result of parallel addition of the corresponding rows of matrices T and F, whose positions are selected with bit '1' in the given slice X. This algorithm uses table 5.1 from [4].

The procedure ADDC(T, X, v, F) adds the binary word v to the rows of the matrix T selected with bit '1' in the slice X, and writes the result into the corresponding rows of the matrix F. Other rows of the matrix F are set to zeros.

The procedure TMERGE(T, X, F) writes the rows of the matrix T, selected with bit '1' in the slice X, in the corresponding rows of the matrix F. Other rows of the matrix F do not change.

In [13, 15], we have shown that the basic procedures take $O(k)$ time each, where k is the number of bit columns in the corresponding matrix.

3 PRELIMINARIES

Let $G = (V, E)$ be a *directed weighted graph* with n vertices and m directed edges (arcs). We assume that $V = \{1, 2, \ldots, n\}$. Let wt denote a function that assigns a weight to every edge. We will consider graphs with a distinguished vertex s called *sink*.

An *adjacency matrix* $Adj = [a_{ij}]$ of a directed graph G is an $n \times n$ Boolean matrix, where $a_{ij} = 1$ if and only if there is an arc from the vertex i to the vertex j in the set E.

An arc e directed from i to j is denoted by $e = (i, j)$, where $i = tail(e)$ and $j = head(e)$. Also, if $(i, j) \in E$, then j is said to be *adjacent* to i. We assume that all arcs have non-negative weights and $wt(u, v) = \infty$, if $(u, v) \notin E$.

The infinity is implemented by the value $\sum_{i=1}^{n} c_i$, where c_i is the maximal weight of arcs outgoing from the vertex i. Let h be the number of bits for coding this sum.

A *path* from u to s in G is a finite sequence of vertices $u = v_1, v_2, \ldots, v_k = s$, where $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$ and $k > 0$. The *shortest path* from u to s is the path of the minimal sum of weights of its edges.

Let $dist(u)$ denote the *length* (*weight*) of the shortest path from u to s and $SP(G)$ denote the *subgraph* of the shortest paths from all vertices of G to the sink.

By analogy with Ramalingam, we introduce the following notations.

We denote by $outdegree(v)$ the number of arcs outgoing from (leaving) the vertex v in $SP(G)$. Let an arc (i, j) be deleted from $SP(G)$.

We denote by *AffectedV* the set of all vertices u in $SP(G)$ such that all paths from u to the sink include the deleted arc (i, j).

An arc (x, y) is called *affected* by deleting the arc (i, j) in $SP(G)$ if there is no such path from x to s in the new graph that uses the arc (x, y) and the weight of the path is equal to $dist_{old}(x)$.

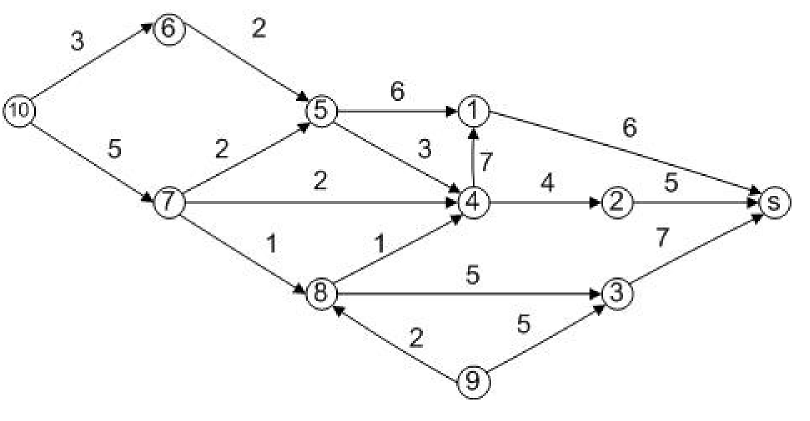

Fig. 5. Graph G

Now we provide an example. Let a graph G (Figure 5) and the shortest paths subgraph $SP(G)$ (Figure 6) be given.

We observe that in $SP(G)$ there is a single shortest path to the sink from the vertices 1, 2, 3, 4, and 8, while there are two different shortest paths to the sink from other vertices.

Let the arc $(4, 2)$ be deleted from $SP(G)$. Then vertices 4, 7, 8, and 10 become *affected* because there is no a path from them to the sink.

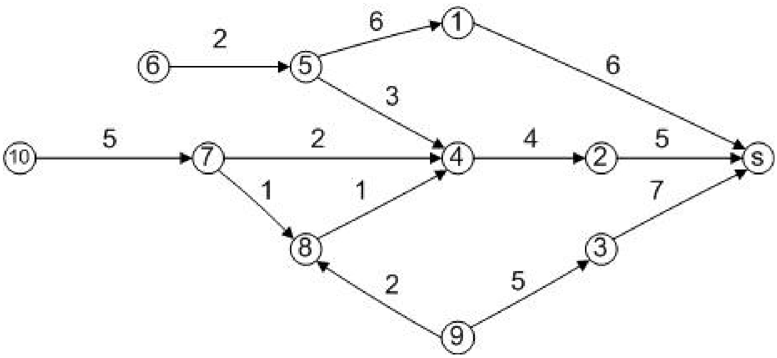

Fig. 6. The shortest paths subgraph $SP(G)$

4 THE RAMALINGAM DECREMENTAL ALGORITHM FOR THE SINGLE-SINK SHORTEST PATHS PROBLEM

Let an arc (i, j) be deleted from $SP(G)$ and *outdegree*$(i) = 0$.

The Ramalingam decremental algorithm for dynamic updating of the single-sink shortest paths subgraph consists of the following two stages.

At the *first stage*, the set *AffectedV* and all affected arcs obtained after deleting the arc (i, j) from $SP(G)$ are determined. Then affected arcs are deleted from $SP(G)$. At the *second stage*, for every affected vertex v_i, a new shortest path from v_i to s in G is computed and $SP(G)$ is updated.

The *first stage* is performed as follows.

Initially, *AffectedV* $= \emptyset$. To construct it, an auxiliary set of vertices *WorkSet* is used. Initially, *WorkSet* $= \{i\}$ because *outdegree*$(i) = 0$ after deleting the arc (i, j) from $SP(G)$. Vertices in *WorkSet* are sequentially updated. The current updated vertex u is deleted from *WorkSet* and is included into the set *AffectedV*. Then every arc (x, u) is deleted from $SP(G)$ and *outdegree*(x) is decreased by one. If *outdegree*$(x) = 0$, the vertex x is included into *WorkSet*.

To perform the *second stage*, a heap *PriorityQueue* is used, whose elements are affected vertices with a key. At this stage, first such a new shortest path to the sink is determined for every affected vertex u that does not include other affected vertices. The value of $dist(u)$ is its current key in the heap. After that $SP(G)$ is updated as follows.

At every iteration, a vertex with the minimum key in the heap (say a) is deleted from the set *PriorityQueue*. Then those arcs (a, b) are determined that belong to an alternative path from the vertex a to the sink and $dist_{new}(a) = wt(a, b) + dist_{old}(b)$. Such arcs are included into $SP(G)$. Further all arcs (c, a) are analyzed. If a new path from the vertex c to the sink includes the arc (c, a) and $dist_{new}(c) < dist_{old}(c)$, the current value $dist(c)$ is equal to $dist_{new}(c)$ and this value is the new key for the vertex c in *PriorityQueue*. If $c \in PriorityQueue$, the previous key of c receives a new value. Otherwise, the vertex c is included into the heap with the key $dist_{new}(c)$.

The process is completed after updating all vertices in the heap.

Let n_1 denote the number of modified or affected vertices and n_2 denote the number of modified or affected arcs and vertices. Then the Ramalingam decremental algorithm [17] takes $O(n_2 + n_1 \log n_1)$ time.

5 ASSOCIATIVE VERSION OF THE RAMALINGAM DECREMENTAL ALGORITHM

To design an associative version of the Ramalingam decremental algorithm for the dynamic update of the shortest paths subgraph, we employ the following data structure:

- an $n \times n$ adjacency matrix G, whose every i^{th} column saves with '1' the heads of arcs outgoing from the vertex i;
- an $n \times n$ adjacency matrix SP, whose every i^{th} column saves with '1' the heads of arcs outgoing from the vertex i that belong to the shortest paths subgraph;
- an $n \times hn$ matrix $Weight$ that contains the arc weights as elements. It consists of n fields having h bits each. The weight of an arc (i, j) is written in the j^{th} row of the i^{th} field;
- an $n \times hn$ matrix $Cost$ that contains the arc weights as elements. It consists of n fields having h bits each. The weight of an arc (i, j) is written in the i^{th} row of the j^{th} field;
- an $n \times h$ matrix $Dist$, whose every i^{th} row saves the shortest distance from the vertex i to the sink;
- a slice $AffectedV$ that saves with '1' positions of all affected vertices.

We observe that the i^{th} field of the matrix $Weight$ saves the weights of arcs *outgoing* from the vertex i, while the i^{th} field of the matrix $Cost$ saves the weights of arcs *entering* the vertex i. Moreover, every j^{th} row of the matrices G and SP saves with '1' the tails of arcs entering the vertex j.

We first explain some reasons why this data structure is used.

Knowing an affected vertex k, the k^{th} field of the matrix *Weight*, the matrix *Dist*, and positions of other affected vertices, in particular, we can perform the following actions of the Ramalingam decremental algorithm on the STAR-machine:

- simultaneously determine the weight of every path from k to the sink that does not include other affected vertices;
- simultaneously determine positions of arcs outgoing from the vertex k that belong to different shortest paths from k to the sink.

If we use the k^{th} field of the matrix *Cost* instead of the matrix *Weight*, we can simultaneously determine positions of arcs, entering the vertex k, whose new distance to the sink is decreased.

Let an arc (i, j) be deleted from G and $SP(G)$.

We first provide an associative parallel algorithm (say Algorithm A) for selecting the set of affected vertices and arcs. This algorithm makes use of the slices WS and *AffectedV* and performs the following steps.

1. Set zeros into the slices *AffectedV* and WS. Check whether there is an arc outgoing from the vertex i in SP. If it is true, go to exit. Otherwise, include the vertex i into WS.
2. While $WS \neq \emptyset$, perform the following actions:
   - delete the position of the first bit '1' (say k) from the slice WS. Include the vertex k into the slice *AffectedV*;
   - delete all arcs from SP that enter the vertex k;
   - for every deleted arc (r, k), include the vertex r into the slice WS if and only if there is no arc entering r in SP.

On the STAR-machine, this algorithm is implemented as the FindAffectedVert procedure.

An associative parallel algorithm for computing new distances to the sink from all affected vertices (say Algorithm B) runs as follows.

While *AffectedV* $\neq \emptyset$, determine the new distance to the sink from every affected vertex by means of the following steps.

1. Select the position of the current affected vertex k in the slice *AffectedV* and mark it with zero.
2. Compute in parallel the weight of every path in the matrix G from the vertex k to the sink that begins with an arc (k, r), where $r \notin$ *AffectedV*.
3. Select the minimal distance from k to s and write it down into the k^{th} row of the matrix *Dist*.

On the STAR-machine, this algorithm is implemented as the ComputeNewDist procedure.

An associative parallel algorithm for updating arcs outgoing from an affected vertex k (say Algorithm C) performs the following steps.

1. By means of a slice (say Z), save the positions of all arcs outgoing from the vertex k in the graph G.
2. Determine *in parallel* the weights of different paths from the vertex k to the sink in the graph G that include an arc marked with bit '1' in Z.
3. By means of a slice (say Y), save positions of those arcs (k, l) for which $dist(k) = wt(k, l) + dist(l)$.
4. Include positions of arcs marked with '1' in the slice Y into SP.

On the STAR-machine, this algorithm is implemented as the UpdateOutgoing-Arcs procedure.

An associative parallel algorithm for updating arcs entering an affected vertex k (say Algorithm D) performs the following steps.

1. By means of a slice (say Z), save the tails of arcs entering the vertex k in G.
2. For all vertices l marked with '1' in the slice Z, determine *in parallel* the weight of every path from k to the sink that starts with the arc (l, k).
3. By means of a slice (say Y), save *positions* of those vertices r, marked with '1' in the slice Z, for which $dist_{new}(r) < dist_{old}(r)$. Then write $dist_{new}(r)$ in the corresponding rows of the matrix *Dist*.

On the STAR-machine, this algorithm is implemented as the UpdateIncomingArcs procedure.

Now, we provide an associative parallel algorithm for dynamic updating the shortest paths subgraph after deleting the arc (i, j) from the graph G. It performs the following steps.

1. Delete the *position* of the arc (i, j) from the matrix G. If $(i, j) \notin SP$, then go to exit. Otherwise, delete the *position* of this arc from the matrix SP.
2. By means of Algorithm A, construct the slice *AffectedV* and delete positions of the affected arcs from SP. Save a copy of the slice *AffectedV* in another slice (say X).
3. By means of Algorithm B, determine new distances to the sink in the matrix G for all affected vertices and write them in the corresponding rows of the matrix *Dist*.
4. While $AffectedV \neq \emptyset$, update affected vertices taking into account their new distances to the sink as follows:

   - knowing the slice *AffectedV* and the matrix *Dist*, determine the position of an affected vertex q having the minimum distance to the sink and delete q from the slice *AffectedV*;

- by means of Algorithm C, determine *in parallel* positions of arcs (q, l) in the matrix G, for which $dist_{new}(q) = wt(q, l) + dist_{old}(l)$ and include these positions into SP;
- by means of Algorithm D, determine *in parallel* positions of arcs (r, q) in the matrix G, for which $dist_{new}(r) < dist_{old}(r)$, and write $dist_{new}(r)$ in the corresponding rows of the matrix *Dist*.

On the STAR-machine, this algorithm is given as the DeleteArc procedure.

6 IMPLEMENTATION OF THE RAMALINGAM DECREMENTAL ALGORITHM ON THE STAR-MACHINE

In this section, we first provide four auxiliary procedures and prove their correctness. Then we propose the DeleteArc procedure.

The FindAffectedVert procedure determines all affected vertices and affected arcs obtained after deleting the arc (i, j) from $SP(G)$. It uses an auxiliary slice WS. The procedure returns the updated matrix SP and a slice *AffectedV*, where positions of all affected vertices are marked with bit '1'.

```
procedure FindAffectedVert(i: integer; var SP: table;
  var AffectedV: slice(SP));
/* The arc (i,j) has been deleted from the matrices G and SP. */
var X,WS: slice(SP);
  v,v1: word(SP);
  k,r: integer;
Begin CLR(WS); CLR(AffectedV); CLR(v1);
  X:=COL(i,SP);
  if not SOME(X) then
/* There was a single arc outgoing from i in SP(G). */
    begin WS(i):='1';
      while SOME(WS) do
/* The cycle for selecting affected vertices. */
        begin k:=STEP(WS);
          AffectedV(k):='1';
/* The vertex k is saved in the slice AffectedV. */
          v:=ROW(k,SP);
/* The row v saves the tails of arcs entering k. */
          ROW(k,SP):=v1;
/* We delete from SP(G) all arcs entering k. */
          while SOME(v) do
/* The cycle for updating the tails of arcs entering k. */
            begin r:=STEP(v);
              X:=COL(r,SP);
              if not SOME(X) then WS(r):='1';
```

```
14.             end;
15.         end;
16.     end;
17. End.
```

Lemma 1. Let an arc (i, j) be deleted from the shortest paths subgraph $SP(G)$. Then the FindAffectedVert procedure returns the slice AffectedV, where positions of affected vertices are marked with '1'. Moreover, it deletes from the matrix SP positions of all arcs that enter every affected vertex.

Proof. [Sketch.] We prove this by induction in terms of the number of vertices to be included into the slice *AffectedV*.

Basis is checked for $l = 1$, that is, only the vertex i is an affected one after deleting the edge (i, j) from $SP(G)$.

After performing lines 1–2, the row $v1$ and the slices WS and *AffectedV* consist of zeros and the slice X saves the i^{th} column of the matrix SP. Since the edge (i, j) has been deleted from SP and the vertex i is affected, then the slice X consists of zeros. After performing lines 4–8, $k = i$, the slice WS consists of zeros again, the i^{th} bit of the slice *AffectedV* is equal to '1', and the variable v saves the tails of edges entering the vertex i.

After performing line 9, all edges entering the vertex i are deleted from the matrix SP. Since there is a single affected vertex after deleting the edge (i, j) from $SP(G)$, for every vertex r marked with '1' in v, there is at least one outgoing edge that differs from (r, i). Therefore after execution of the cycle for updating the tails of arcs entering the veretx i (lines 10–14), the slice WS consists of zeros and we go to the procedure end.

Step of induction. Let the assertion be true for $l \geq 1$ vertices included into the slice $AffectedV$. We prove this for $l + 1$ vertices. By the inductive assumption, after including the first l vertices into the slice $AffectedV$, all edges entering every affected vertex are deleted from the matrix SP. Moreover, the tails of the deleted edges, for which there is no path to the sink, are included into the slice WS.

After including the l^{th} vertex into the slice $AffectedV$, the slice WS saves the position of the $(l+1)^{\text{th}}$ affected vertex. Therefore the cycle for selecting affected vertices (line 5) is performed. In this cycle, after performing lines 6–7, we first delete the single vertex from the slice WS and $WS = \emptyset$. Then we include this vertex into the slice $AffectedV$. After performing line 8, the variable v saves the tails of edges entering the $(l + 1)^{\text{th}}$ affected vertex. After performing line 9, all edges entering this vertex are deleted from the matrix SP. After fulfilling the cycle for updating the tails of arcs entering an affected vertex (lines 10–14), none new vertex is included into the slice WS because there are only $l + 1$ affected vertices after deleting the edge (i, j) from $SP(G)$. Therefore the cycle for selecting affected vertices (lines 6–16) is finished, and we go to the procedure end.

This completes the proof. □

Let us consider the ComputeNewDist procedure. It determines new distances to the sink from all affected vertices. The procedure uses the slice *AffectedV* and the matrices G, *Weight*, and *Dist*. It returns the updated matrix *Dist*.

```
procedure ComputeNewDist(h: integer; G: table; Weight: table;
  AffectedV: slice(G); var Dist: table);
var k,r: integer;
  X,Z,Z1: slice(G);
  v: word(Dist);
  W1,W2: table;
 1.  Begin X:=AffectedV;
 2.  while SOME(X) do
 3.    begin k:=FND(X); Z:=COL(k,G);
 4.      Z1:=Z and ( not X);
/* The slice Z1 saves the heads of arcs outgoing from k
  that are not affected. */
 5.      TCOPY1(Weight,k,h,W1);
/* The matrix W1 saves the k^th field of the matrix Weight. */
 6.      ADDV(W1,Dist,Z1,W2);
/* The matrix W2 saves the weights of paths from k to s. */
 7.      MIN(W2,Z1,Z);
/* In the slice Z, we mark with '1' positions of the
  rows in W2, where the minimal element is located. */
 8.      r:=FND(Z); v:=ROW(r,W2);
 9.      ROW(k,Dist):=v;
/* The new distance from k to s is saved in ROW(k,Dist). */
10.      X(k):='0';
11.    end;
12.  End;
```

Lemma 2. Let the number of bits h for coding the infinity, the slice AffectedV and the current matrices G, Weight, and Dist be given. Then the ComputeNewDist procedure returns the updated matrix Dist that saves new distances to the sink from all affected vertices.

Proof. We prove by induction in terms of the number of affected vertices l.

Basis is checked for $l = 1$. After performing lines 1–3, the slice X is a copy of the slice *AffectedV*, $k = i$, and the slice Z saves positions of arcs outgoing from the vertex i in G. After performing lines 4–6, the matrix $W2$ saves the weights of different paths from the vertex i to the sink that are starts from an arc (i, l), where $l \notin \mathit{AffectedV}$. After performing lines 7–9, we first determine the vertex r that belongs to the new shortest path from i to the sink, then we write the new distance from i to the sink in the i^{th} row of the matrix $Dist$. After fulfilling line 10, $X = \emptyset$, and we go to the exit.

Step of induction. Let the assertion be true for $l \geq 1$ affected vertices. We prove this for $l + 1$ vertices. By the inductive assumption, after updating the first l affected vertices, their new distances to the sink are written in the corresponding rows of the matrix $Dist$, and there is only a single affected vertex in the slice X. Further we reason by analogy with the basis.

This completes the proof. □

Now, we proceed to the UpdateOutgoingArcs procedure. Knowing the current updated vertex k, the number of bits h for coding the infinity, and the current matrices G, $Weight$, $Dist$, and SP, the procedure returns the updated matrix SP.

```
procedure UpdateOutgoingArcs(h,k: integer; G: table;
  Weight: table; Dist: table; var SP: table);
var W1,W2: table;
  v: word(Dist);
  Y,Z: slice(G);
1. Begin Z:=COL(k,G);
2.   TCOPY1(Weight,k,h,W1);
3.   ADDV(W1,Dist,Z,W2);
/* The matrix W2 saves different distances from
  the vertex k to the sink. */
4.   v:=ROW(k,Dist);
/* The variable v saves the shortest distance from
the vertex k to the sink. */
5.   MATCH(W2,Z,v,Y);
/* In the slic Y, we mark with '1' the vertices l
  for which dist(k) = wt(k,l) + dist(l). */
6.   COL(k,SP):=Y;
/* Positions of arcs (k,l) are included into SP. */
7. End;
```

Lemma 3. Let h be the number of bits for coding the infinity and k be the current updated vertex. Let the current matrices G, Weight, Dist, and SP be given. Then, after performing the UpdateOutgoingArcs procedure, positions of all edges (k, l) for which $\mathit{dist}(k) = \mathit{wt}(k, l) + \mathit{dist}(l)$ are included into the matrix SP.

Proof. We prove this by contradiction. Let an arc (k, r) belong to G and $\mathit{dist}(k) = \mathit{wt}(k, r) + \mathit{dist}(r)$. However, after performing the UpdateOutgoingArcs procedure, the position of the arc (k, r) does not belong to the matrix SP. We prove that this contradicts the execution of UpdateOutgoingArcs.

Really, since $(k, r) \in G$, then after performing line 1 $Z(r) =$ '1'. After performing lines 2–3, the weight of the shortest path from vertex k to the sink that includes the edge (k, r) is written unto the r^{th} row of the matrix $W2$. By assumption, $dist(k) = wt(k, r) + dist(r)$. Therefore $Y(r) =$ '1' after fulfilling the basic MATCH procedure. Hence, after performing line 6, the edge (k, r) is included into the matrix SP. This contradicts our assumption. □

Finally, we propose the UpdateIncomingArcs procedure. Knowing the current updated vertex k, the number of bits h for coding the infinity, and the current matrices G, *Cost*, and *Dist*, the procedure returns the updated matrix *Dist*.

```
procedure UpdateIncomingArcs(h,k: integer; G: table;
  Cost: table; var Dist: table);
var Y,Z: slice(G);
  v: word(G);
  v1: word(Dist);
  W,W1: table;
1. Begin v:=ROW(k,G); Z:=CONVERT(v);
/* The slice Z saves the tails of arcs entering k. */
2.  v1:=ROW(k,Dist);
  /* The row v1 saves the shortest distance from k to s. */
3.  TCOPY1(Cost,k,h,W1);
/* The k^th field of the matrix Cost is written into
  the matrix W1. */
4.  ADDC(W1,Z,v1,W);
/* In every l^th row of W that corresponds to '1' in Z,
  the new distance from l to s is written. */
5.  SETMIN(W,Dist,Z,Y);
/* In the slice Y, we mark with '1' positions of vertices,
  whose new distances to the sink are decreased. */
6.  TMERGE(W,Y,Dist);
/* In every l^th row of the matrix Dist, a new distance
  to the sink is written if and only if Y(l) = '1'. */
7. End;
```

Lemma 4. Let h be the number of bits for coding the infinity and k be the current updated vertex. Let the current matrices G, Cost, and *Dist* be given. Then the UpdateIncomingArcs procedure maintains the matrix *Dist*, where new distances to the sink are written for the tails r of arcs entering the vertex k whose $dist_{new}(r)$ is decreased.

Proof. We prove this by contradiction. Let an arc (r, k) belong to the graph G and $dist_{new}(r) < dist_{old}(r)$. However, after performing the UpdateIncomingArcs procedure, the r^{th} row of the matrix *Dist* does not change. We prove that this cotradicts the execution of the UpdateIncomingArcs procedure.

Really, the k^{th} row of G saves the tails of edges entering the vertex k. Therefore after performing line 1, these tails are marked with '1' in slice Z. Since $(r,k) \in G$, then $Z(r)$ = '1'. After performing line 2, the row $v1$ saves the shortest distance from vertex k to the sink. After fulfilling line 3, the r^{th} row of the matrix $W1$ saves the weight of the arc (r,k). Obviously, after performing lines 4–5, the r^{th} row of the matrix W saves the new distance from the vertex r to the sink. After performing the basic SETMIN procedure (line 6), we obtain that $Y(r)$ = '1' because by the assumption $dist_{new}(r) < dist_{old}(r)$. Hence, after performing line 7, $\mathrm{ROW}(r, Dist) = dist_{new}(r)$. This contradicts our assumption.

This completes the proof. □

Let us proceed to the DeleteArc procedure. Knowing the deleted arc (i,j) and the current matrices G, *Weight*, *Cost*, *Dist*, and SP, the procedure returns the updated matrices G, SP, and *Dist* with the use of the above auxiliary procedures.

```
procedure DeleteArc(i,j,h: integer; Weight,Cost: table;
  var G,SP: table; var Dist: table);
/* The arc (i,j) will be deleted from G and SP. */
var k: integer;
  AffectedV,X,Y: slice(G);
  label 1;
1. Begin X:=COL(i,G); X(j):='0';
2.  COL(i,G):=X;
/* The arc (i,j) is deleted from G. */
3.  X:=COL(i,SP);
4.  if X(j)='0' then goto 1;
5.  X(j):='0'; COL(i,SP):=X;
/* The arc (i,j) is deleted from SP(G). */
6.  FindAffectedVert(i,SP,AffectedV);
/* This procedure returns the updated matrix SP
  and the slice AffectedV. */
7.  ComputeNewDist(h,G,Weight,AffectedV,Dist);
/* This procedure returns the updated matrix Dist. */
8.  while SOME(AffectedV) do
/* The cycle for updating affected vertices. */
9.    begin MIN(Dist,AffectedV,Z);
10.       k:=FND(Z); AffectedV(k):='0';
11.       UpdateOutgoingArcs(h,k,G,Weight,Dist,SP);
/* We include into SP those arcs (k,r), for which
  dist(k) = wt(k,r) + dist(r). */
12.       UpdateIncomingArcs(h,k,G,Cost,Dist);
/* We write dist_new(l) into ROW(l,Dist) if dist_new(l) < dist_old(l)
  and the path from l to s starts from the arc (l,k). */
13.   end;
14. 1: End;
```

Theorem 1. Let a directed weighted graph be given as an adjacency matrix G and a matrix Weight. Let matrices *Cost*, SP, and *Dist* and the number of bits h for coding the infinity be given. Let an arc(i, j) be deleted from the graph. Then after performing the DeleteArc procedure, this arc is deleted from the matrices G and SP. Moreover, matrices SP and Dist are updated according to the algorithms A, B, C, and D.

Proof. [Sketch.] We prove this by induction in terms of the number q of affected vertices that appear after deleting the arc (i, j) from the shortest paths subgraph $SP(G)$.

Basis is proved for $q = 1$. It can be checked immediately that after performing lines 1–5, the position of the arc (i, j) is deleted from the matrices G and SP. After performing line 6, in view of Lemma 1, the slice *AffectedV* saves the position of the affected vertex i and positions of all arcs, entering this vertex, are deleted from SP. After performing line 7, in view of Lemma 2, *AffectedV*(i) = '1' and the new distance from i to the sink is written in the i^{th} row of the matrix *Dist*. Since *AffectedV* $\neq \emptyset$, we perform the cycle for updating affected vertices (9–13). Here, after fulfilling lines 9-10, we have $k = i$ and *AffectedV* $= \emptyset$. After performing line 11, in view of Lemma 3, we include into SP positions of the arcs (i, r) for which $dist_{new}(i) = wt(i, r) + dist_{old}(r)$. By the assumption, there is a single affected vertex in SP. It means that there is an alternative path to the sink from every vertex l, being the tail of any arc (l, i) in the matrix SP. Therefore after performing line 12, the matrix *Dist* does not change.

Hence, after performing the DeleteArc procedure, the position of the arc (i, j) is deleted from matrices G and SP, $dist_{new}(i)$ is written into the i^{th} row of the matrix *Dist*, and positions of all arcs (i, r), for which $dist_{new}(i) = wt(i, r) + dist_{old}(r)$, are included into SP.

Step of induction. Let the assertion be true when $q \geq 1$ affected vertices are updated in the given graph. We prove the assertion for $q + 1$ affected vertices.

One can immediately verify that, after performing lines 1–7, the position of the arc (i, j) is deleted from the matrices G and SP, the slice *AffectedV* saves positions of $q + 1$ affected vertices, positions of all affected arcs are deleted from SP, and the new distances to the sink from all affected vertices are written in the corresponding rows of the matrix *Dist*. Since *AffectedV* $\neq \emptyset$, we carry out line 8.

After performing lines 9–10, we determine the position of the affected vertex k having the minimal new distance to the sink and mark it with '0' in the slice *AffectedV*. By analogy with the basis, after performing line 11, we include into SP the positions of arcs (k, r), for which $dist_{new}(k) = wt(k, r) + dist_{old}(r)$. Further, after performing line 12, for every affected vertex r, for which $dist_{new}(r) < dist_{old}(r)$, we write $dist_{new}(r)$ into the r^{th} row of the matrix *Dist*.

Now, there are only q affected vertices, whose positions are marked with '1' in the slice *AffectedV*. By the inductive assumption, after updating q affected vertices, all alternative shortest paths from every affected vertex r to the sink are included into SP and the distance from r to s is written in the r^{th} row of the matrix *Dist*. Hence, the assertion is true for $q+1$ affected vertices.

This completes the proof. □

Let us evaluate the time complexity of the DeleteArc procedure. To this end, we first evaluate the time complexity of the auxiliary procedures. Let k be the number of affected vertices that appear in the matrix SP after deleting the arc (i,j). The auxiliary FindAffectedVert procedure takes $O(k)$ time because the cycle for updating the tails of arcs entering an affected vertex takes $O(1)$ time which is not greater than the maximum number of bits '1' in the rows of the matrix SP. The auxiliary ComputeNewDist procedure takes $O(kh)$ time because the cycle `while SOME(X) do` (lines 2–11) is performed k times and inside this cycle, the basic procedures require $O(h)$ time each. The auxiliary procedures UpdateOutgoingArcs and UpdateIncomingArcs take $O(h)$ time each. In the DeleteArc procedure, the cycle for updating an affected vertex (lines 9–13) takes $O(kh)$ time because inside this cycle, the basic procedure and two auxiliary procedures require $O(h)$ time each. Hence, the DeleteArc procedure takes $O(kh)$ time.

In [16], we presented in detail the main advantages of the associative version of the Ramalingam decremental algorithm. Briefly speaking, these advantages appear due to the use of a natural two-dimensional data structure, the data access by contents, and the use of a group of basic procedures that permits us to update in parallel both the arcs outgoing from every affected vertex and the arcs entering this vertex.

7 EXAMPLE

In this section, we provide the dynamic update of the shortest paths subgraph $SP(G)$ (Figure 6) after deleting the arc $(4,2)$ from the graph G (Figure 5).

For simplicity, we will provide the changes of matrices *Dist* and SP during the execution of the DeleteArc procedure. For our example, $s=11$, the infinity is chosen as $\mathit{inf} = (50)_{10} = (110010)_2$ and $h=6$ according to the formula given in Preliminaries.

Initially, the matrices *Dist* and SP have the form depicted in Table 1.

During the execution of the DeleteArc procedure, we first delete the arc $(4,2)$ from $SP(G)$ as shown in Figure 7. This corresponds to performing the following operations of the STAR-machine: `X:=COL(4,SP); X(2):='0'; COL(4,SP):=X`. Obviously, after performing these operations, the second row of the matrix SP consists of zeros.

Further, we execute the auxiliary FindAffectedVert procedure. Here, we first simultaneously delete the positions of the arcs $(5,4)$, $(7,4)$, and $(8,4)$. Then we simultaneously delete the positions of the arcs $(7,8)$ and $(9,8)$. Finally, we delete the position of the arc $(10,7)$ and obtain the result depicted in Figure 8.

	The matrix Dist					
1	0	0	0	1	1	0
2	0	0	0	1	0	1
3	0	0	0	1	1	1
4	0	0	1	0	0	1
5	0	0	1	1	0	0
6	0	0	1	1	1	0
7	0	0	1	0	1	1
8	0	0	1	0	1	0
9	0	0	1	1	0	0
10	0	1	0	0	0	0
11	0	0	0	0	0	0

	The matrix SP										
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	1	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	1	0	1	1	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	1	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0
11	1	1	1	0	0	0	0	0	0	0	0

Table 1. The initial distances and the shortest paths

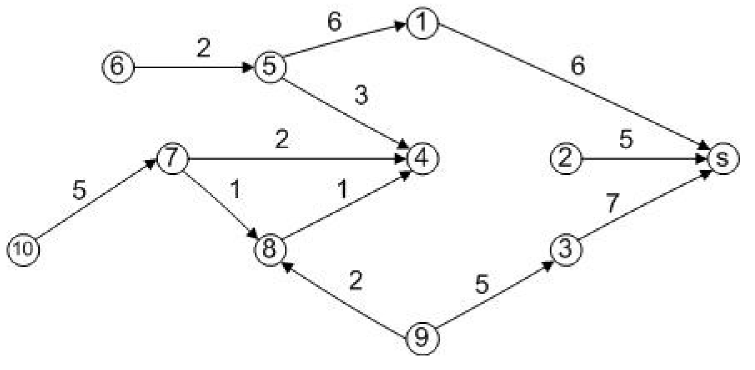

Fig. 7. SP(G) after deletion of the arc $(4, 2)$

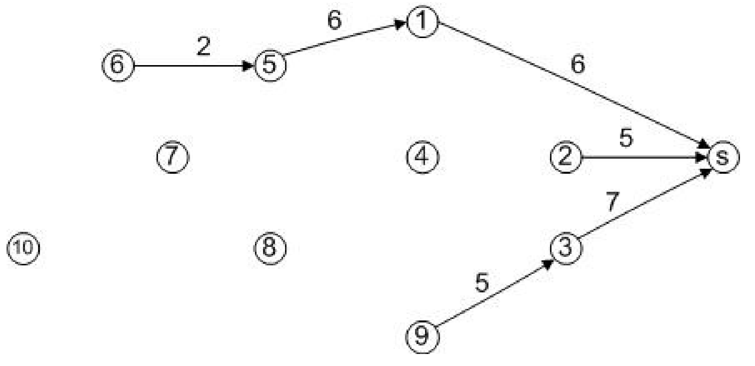

Fig. 8. SP(G) after executing FindAffectedVert

Hence, after performing the auxiliary FindAffectedVert procedure, we obtain the slice *AffectedV*, where positions of vertices 4, 7, 8, and 10 are marked with bit '1'. Moreover, the corresponding rows in the matrix SP consist of zeros.

Now, we execute the auxiliary ComputeNewDist procedure and determine a new distance from every affected vertex to the sink. We obtain that $dist(4) = 001101$, $dist(7) = 001110$, $dist(8) = 001100$, and $dist(10) = 010001$. These values are written in the corresponding rows of the matrix *Dist*.

	The matrix Dist					
1	0	0	0	1	1	0
2	0	0	0	1	0	1
3	0	0	0	1	1	1
4	0	0	1	1	0	1
5	0	0	1	1	0	0
6	0	0	1	1	1	0
7	0	0	1	1	0	1
8	0	0	1	1	0	0
9	0	0	1	1	0	0
10	0	1	0	0	0	1
11	0	0	0	0	0	0

	The matrix SP										
	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	1	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0
11	1	1	1	0	0	0	0	0	0	0	0

Table 2. The final distances and the shortest paths

During the execution of the cycle for updating the affected vertices (lines 9–13) in the DeleteArc procedure, we first update the vertex 8 with the minimal distance to the sink. As a result, the position of the arc $(8, 3)$ is included into the matrix SP. Moreover, we obtain that $dist_{new}(7) = 001101$ and $dist_{new}(7) < dist_{old}(7)$. Therefore we write the value 001101 into the seventh row of the matrix *Dist*. Then we update the vertex 4 and include the position of the arc $(4, 1)$ in the matrix SP. Further we update the vertex 7 and include the position of the arc $(7, 8)$ in SP.

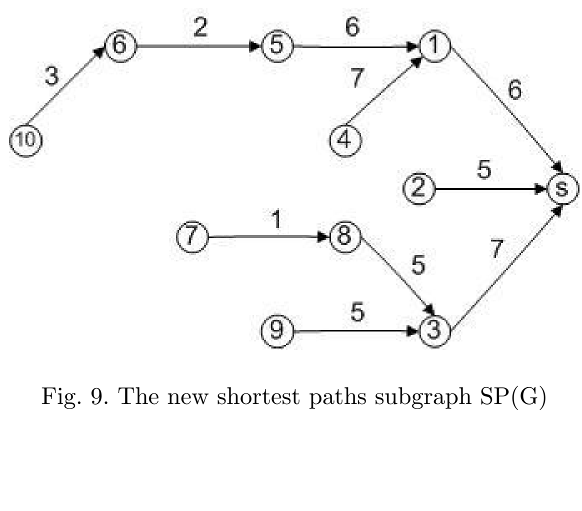

Fig. 9. The new shortest paths subgraph SP(G)

After updating the last affected vertex, we include the position of the arc $(10, 6)$ in the matrix SP. The result of performing the DeleteArc procedure is given in Table 2.

We observe that Table 2 corresponds to the following subgraph of the shortest paths depicted in Figure 9.

8 CONCLUSIONS

We have proposed the efficient implementation of the Ramalingam decremental algorithm for updating the shortest-paths subgraph on the STAR-machine having not less than n PEs. The associative version of the Ramalingam decremental algorithm is represented as the DeleteArc procedure that includes a group of auxiliary procedures for performing different parts of this algorithm. We have proved correctness of the auxiliary procedures and the DeleteArc procedure and evaluated the time complexity. We have obtained that the DeleteArc procedure takes $O(kh)$ time per a deletion, where h is the number of bits for coding the infinity and k is the number of affected vertices that appear in $SP(G)$ after deleting an arc. It is assumed that each microstep of the STAR-machine takes one unit of time. We have also considered an example of implementing the Ramalingam decremental algorithm on the STAR-machine.

The proposed data structure and the proposed technique for updating the shortest paths on associative parallel processors can be used for solving other tasks, such as implementation of the Ramalingam incremental algorithm for the dynamic update of the shortest paths subgraph after insertion of an arc into the given graph and for dynamic update of the shortest paths tree after deletion or insertion of an arc.

REFERENCES

[1] Ausiello, G.—Italiano, G. F.—Marchetti-Spaccamela, A.—Nanni, U.: Incremental Algorithms for Minimal Length Paths. Journal of Algorithms. Vol. 12, 1991, pp. 615–638.

[2] Chaudhuri, S.—Zaroliagis, C. D.: Shortest Path Queries in Digraphs of Small Treewidth. In: Proc. 1995 International Colloquiumon Automata Languages, and Programming. Lecture Notes in Computer Science, Berlin: Springer-Verlag, Vol. 944, 1995, pp. 244–255.

[3] Dijkstra, E. W.: A Note on two Problems in Connection With Graphs, Numerische Mathematik. Vol. 1, 1959, pp. 269–271.

[4] Foster, C. C.: Content Addressable Parallel Processors, New York: Van Nostrand Reinhold Company, 1976.

[5] Franciosa, P. G.—Frigioni, D.—Giaccio, R.: Semi-Dynamic Shortest Paths and Breadth-First Searsch in Digraphs. In: Proc. the 14$^{\text{th}}$ Annual Symposium on

Theoretical Aspects of Computer Science. Lecture Notes in Computer Science, Berlin: Springer-Verlag, Vol. 1200, 1997, pp. 33–46.

[6] Frigioni, D.—Marchetti-Spaccamela, A.—Nanni, U.: Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. Journal of Algorithms, Academic Press. Vol. 34, 2000, pp. 351–381.

[7] Frigioni, D.—Marchetti-Spaccamela, A.—Nanni, U.: Semi-Dynamic Algorithms for Maintaining Single Source Shortest Paths Trees. Algorithmica, Berlin: Springer-Verlag, Vol. 25, 1998, pp. 250–274.

[8] Frigioni, D.—Marchetti-Spaccamela, A.—Nanni, U.: Fully Dynamic Shortest Paths in Digraphs With Arbitrary arc Weights. Journal of Algorithms, Elsevier Science, Vol. 49, 2003, pp. 86–113.

[9] Klein, P. N.—Rao, S.—Rauch, M.—Subramanian, S.: Faster Shortest Path Algorithms for Planar Graphs. In: Proc. ACM Symposium on Theory of Computing. Montreal, Quebec, Canada, 1994, pp. 27–37.

[10] Mirenkov, N.: The Siberian Approach for an Open-System High-Performance Computing Architecture. Computing and Control Engineering Journal. Vol. 3, 1992, No. 3, pp. 137–142.

[11] Narvàez, P.—Siu, K.-Y.—Tzeng, H.-Y.: New Dynamic Algorithms for Shortest Paths Tree Computation. IEEE/ACM Trans. Networking, Vol. 8, 2000, pp. 734–746.

[12] Nepomniaschaya, A. S.: Language STAR for Associative and Parallel Computation with Vertical Data Processing. In: Proc. of the International Conference "Parallel Computing Technologies", World Scientific, Singapore, 1991, pp. 258–265.

[13] Nepomniaschaya, A. S.: Solution of Path Problems Using Associative Parallel Processors. In: Proc. of the Intern. Conf. on Parallel and Distributed Systems, ICPADS '97, Korea, Seoul, IEEE Computer Society Press, 1997, pp. 610–617.

[14] Nepomniaschaya, A. S.—Vladyko, M. A.: Comparison of Models for Associative Parallel Computations. Programming, Moscow: Nauka, No. 6, 1997, pp. 41–50 (in Russian).

[15] Nepomniaschaya, A. S.–Dvoskina, M. A.: A Simple Implementation of Dijkstra's Shortest Path Algorithm on Associative Parallel Processors. Fundamenta Informaticae, IOS Press, Vol. 43, 2000, pp. 227–243.

[16] Nepomniaschaya, A. S.: Associative Version of the Ramalingam Decremental Algorithm for Dynamic Updating the Single-Sink Shortest Paths Subgraph. In: Proc. of the 10^{th} International Conference on Parallel Computing Technologies, PaCT-2009, Novosibirsk, Russia. Lecture Notes in Computer Science, Berlin: Springer-Verlag, Vol. 5698, 2009, pp. 257–268.

[17] Ramalingam, G.: Bounded Incremental Computation. Lecture Notes in Computer Science, Berlin: Springer-Verlag, Vol. 1089, 1996, pp. 30–51.

[18] Ramalingam, G.—Reps, T.: An Incremental Algorithm for a Generalization of the Shortest Paths Problem. Journal of Algorithms, Academic Press, Vol. 21, 1996, pp. 267–305.

Anna Shmilevna Nepomniaschaya graduated from the Chernovitsky State University in 1967 and worked in the Novosibirsk Institute of Mathematics (Siberian Division of the USSR Academy of Sciences). In 1981, she got her Ph. D. degree in computer science from the Novosibirsk Computing Center (the new name is Institute of Computational Mathematics and Mathematical Geophysics, Siberian Division of the Russian Academy of Sciences). Now, she is a Senior Researcher in the Laboratory of Parallel Algorithms and Structures of the Institute of Computational Mathematics and Mathematical Geophysics. She published 117 papers in automata theory, theory of formal grammars and languages, theory of parallel algorithms and parallel processing. Her current research interests include associative processing in fine-grained parallel systems for such applications as graph algorithms and relational databases, and techniques for specification and analysis of associative parallel processors.