Bull. Nov. Comp. Center, Comp. Science, 28 (2008), 93–109

Parallel implementation of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest paths subgraph

A. S. Nepomniaschaya

Abstract. We propose an efficient parallel implementation of the Ramalingam algorithm for dynamic updating the single-sink shortest paths subgraph of a directed weighted graph after deletion of an edge on a model of associative (content addressable) parallel systems with vertical processing (the STAR-machine). The associative version of the Ramalingam decremental algorithm for updating the shortest paths subgraph is given as procedure DeleteEdge, whose correctness is proved and the time complexity is evaluated. We compare implementations of the Ramalingam decremental algorithm and its associative version and present the main advantages of the associative version.

1. Introduction

Finding the shortest paths in a weighted graph is a fundamental and well studied problem in computer science. Such a problem arises in practice in different application settings. There are two versions of this problem: finding the single source shortest paths and finding the all-pairs shortest paths.

The dynamic version of the single source shortest paths problem consists of maintaining the shortest paths information while the graph changes without recomputing everything from scratch after every update on the graph. In this framework, 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. When arbitrary sequences of the above operations are allowed, we refer to the *fully dynamic* problem. If we consider only insertions (deletions) of edges, we refer to the *incremental* (*decremental*) problem.

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.

However, on general graphs with arbitrary edge weights, neither a fully dynamic solution nor a decremental solution for the single source shortest paths problem is known in the standard models (the worst case and the amortized analysis) that is assymptotically better than recomputing a new solution from scratch.

In the case of arbitrary real edge weights, Ramalingam and Reps [14, 15] 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. In [14, 15] 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, 14, 15] use the dynamic version of the Dijkstra algorithm [3]. Narváez et al. [10] 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, Bellman-Ford, and D'Esopo-Pape into new dynamic algorithms.

In this paper, we deal with a single-sink directed graph G and the shortest

paths subgraph $SP(G)$ that consists of all shortest paths from every vertex to the sink. We propose an associative version of the Ramalingam algorithm [14] for the dynamic update of $SP(G)$ after deletion of an edge from G. Our model of computation (the STAR-machine) simulates the run of associative (content addressable) parallel systems of the SIMD type with bit-serial (vertical) processing and simple single-bit processing elements. Such an architecture is best suited to solve the graph problems. We first offer a simple and natural data structure that allows one to design an efficient parallel implementation of the Ramalingam algorithm on the STAR-machine. The associative algorithm is given as a procedure DeleteEdge, whose correctness is proved. We obtain that this procedure takes $O(hk)$ time, where h is the number of bits for coding the infinity and k is the number of vertices, whose shortest paths to the sink change in $SP(G)$ after deleting an edge from G. Following [4], it is assumed that each elementary operation of the STAR-machine (its microstep) takes one unit of time. We also present the main advantages of the associative version of the Ramalingam decremental algorithm [14].

2. Model of associative parallel machine

Here we propose a brief description of our model. It is defined as an abstract STAR-machine of the SIMD type with the vertical data processing [11]. 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. Activation of a PE depends on data.

Input binary data are given in the form of two-dimensional tables, where each datum occupies an individual row and is updated by a dedicated processing element. In any table, rows are numbered from top to bottom and columns – from left to right. Both a row and a column can easily be accessed. Some tables may be loaded into the 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 that 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 the

data types word, slice, and table. Constants for the types slice and word are represented as a sequence of symbols of a 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.

Let us 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) simultaneously sets all components of Y to $'1'$;

CLR(Y) simultaneously sets all components of Y to $'0'$;

$Y(i)$ selects the i-th component of Y;

FND(Y) returns the number i of the first (the uppermost) $'1'$ of Y, $i \geq 0$;

STEP(Y) returns the same result as FND(Y), then resets the first $'1'$ found to $'0'$;

CONVERT(Y) returns a row, whose every i-th component (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 the data parallelism, we introduce in the usual way the bitwise Boolean operations: $X\,and\,Y$, $X\,or\,Y, not\,Y$, $X\,xor\,Y$. We also use a predicate SOME(Y) that results in true if and only if there is at least a single component $'1'$ in the slice Y.[1]

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.

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 [12, 13] implemented on the STAR-machine which will be used later on. These procedures use the given

[1]For simplicity, the notation $Y \neq \Theta$ denotes that the predicate SOME(Y) results in true.

global slice X to indicate with $'1'$ the row positions used in the corresponding procedure.

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 matrix element and $X(i) =' 1'$.

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

The procedure WCOPY(v, X, F) writes the binary word v into those rows of the matrix F, that correspond to positions $'1'$ in the slice X. The rows of the matrix F, that correspond to positions $'0'$ in the slice X, consist of zeros.

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

The procedure HIT(T, F, X, Z) defines positions of the corresponding identical rows in the given matrices T and F using the global slice X. It returns the slice Z, where $Z(i) =' 1'$ if and only if ROW$(i, T) =$ ROW(i, F) and $X(i) =' 1'$.

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 $'1'$ in the given slice X. This algorithm uses the table 5.1 from [4].

In [12, 13], 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 $G = (V, E, w)$ be a *directed weighted graph* with the set of vertices $V = \{1, 2, \ldots, n\}$, the set of directed edges (arcs) $E \subseteq V \times V$ and the function w that assigns a weight to every edge. We assume that $|V| = n$ and $|E| = m$.

We will consider graphs with a distinguished vertex s called *sink*.

An edge e directed 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*). Also, if $(i, j) \in E$, then j is said to be *adjacent* to i. We assume that all edges have a positive weight and $w(u, v) = \infty$, if $(u, v) \notin E$. Let h be the number of bits for coding the infinity.

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 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 $outdegree_{SP(G)}(v)$ the number of edges outgoing from the vertex v in $SP(G)$.

Let an edge (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 edge (i, j). An edge in G is called *SP edge* iff it belongs to some shortest path to the sink.

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

4. The Ramalingam decremental algorithm for the single-sink shortest paths problem

Let an edge (i, j) be deleted from $SP(G)$ and $outdegree_{SP(G)}(i) = 0$.

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

At the *first stage*, one determines the set $AffectedV$ and all affected edges obtained after deleting the edge (i, j) from $SP(G)$. Then affected edges are deleted from $SP(G)$.

At the *second stage*, for every affected vertex v_i, one computes a new shortest path from v_i to s in G and updates $SP(G)$.

The *first stage* is performed as follows.

Initially, $AffectedV = \Theta$. To construct it, an auxiliary set of vertices $WorkSet$ is used. Initially, $WorkSet = \{i\}$ because $outdegree_{SP(G)}(i) = 0$ after deleting the edge (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 edge (x, u) entering the vertex u is deleted from $SP(G)$ and $outdegree_{SP(G)}(x)$ is decreased by one. If $outdegree_{SP(G)}(x) = 0$, the vertex x is included into $WorkSet$.

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

At every iteration, a vertex with the minimum key in the heap (say a) is deleted from the set $PriorityQueue$. Then one determines those edges (a, b) that belong to an alternative path from the vertex a to the sink and $dist_{new}(a) = w(a, b) + dist_{old}(b)$. Such edges are included into $SP(G)$. Further all edges (c, a) entering the vertex a are analyzed. If a new path from the vertex c to the sink includes the edge (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.

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 structures:

– an $n \times n$ adjacency matrix G, whose every i-th column saves with $'1'$ the tails of edges outgoing from the vertex i;

– an $n \times n$ adjacency matrix $SP(G)$, whose every i-th column saves with $'1'$ the tails of edges that belong to the shortest paths subgraph;

– an $n \times hn$ matrix $Weight$ that contains as elements the edge weights. It consists of n fields having h bits each. The weight of an edge (i, j) is written in the j-th row of the i-th field;

– an $n \times hn$ matrix $Cost$ that contains as elements the edge weights. It consists of n fields having h bits each. The weight of an edge (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.

Note that the i-th field of the matrix $Weight$ saves the weights of edges *outgoing* from the vertex i, while the i-th field of the matrix $Cost$ saves the weights of edges *entering* the vertex i.

Let us enumerate two properties of matrices G and $SP(G)$.

Property 1. Every i-th column of the matrices G and $SP(G)$ saves with $'1'$ the tails of edges outgoing from the vertex i.

Property 2. Every i-th row of the matrices G and $SP(G)$ saves with $'1'$ the heads of edges entering the vertex i.

Let an edge (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 edges. This algorithm uses slices WS and $AffectedV$ and performs the following steps.

Step 1. Set zeros into slices $AffectedV$ and WS. Check whether there is an edge outgoing from the vertex i in $SP(G)$. If it is true, go to exit. Otherwise, include the vertex i into WS.

Step 2. While $WS \neq \Theta$, perform the following actions:

– delete the position of the first $'1'$ (say k) from the slice WS. Include the vertex k into the slice $AffectedV$;

– delete all edges from $SP(G)$ that enter the vertex k;

– for every deleted edge (r, k), include the vertex r into the slice WS if and only if there is no edge entering r in $SP(G)$.

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

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

Step 1. By means of a slice (say Z) save positions of all edges outgoing from the vertex k in the graph G.

Step 2. Determine *in parallel* the distances from the vertex k to the sink for different paths that include an edge marked with $'1'$ in the slice Z.

Step 3. By means of a slice (say Y), save positions of those edges (k, l) for which $dist(k) = w(k, l) + dist(l)$.

Step 4. Include positions of edges marked with $'1'$ in the slice Y into $SP(G)$.

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

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

Step 1. By means of a slice (say Z), save *the heads* of edges entering the vertex k in G.

Step 2. For all vertices l marked with $'1'$ in the slice Z, determine *in parallel* the distances to the sink of every path starting with the edge (l, k).

Step 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 procedure UpdateIncomingEdges.

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

Step 1. Delete the *position* of the edge (i, j) from the matrix G. If $(i, j) \notin SP(G)$, then go to exit. Otherwise, delete the *position* of this edge from the matrix $SP(G)$.

Step 2. By means of the *Algorithm A*, construct the slice $AffectedV$ and delete affected edges from $SP(G)$. Save a copy of the slice $AffectedV$ in another slice (say X).

Step 3. While $AffectedV \neq \Theta$, determine new distances to the sink for all affected vertices as follows:

– select the *position* of a current affected vertex k in the slice $AffectedV$ and mark it with $'0'$;

– compute *in parallel* the distances from the vertex k to s for every path beginning with an edge (k, r), where $r \notin AffectedV$ and this path does not

include affected vertices;

– select the minimum distance from k to s and write it down into the k-th row of the matrix $Dist$.

Step 4. While $X \neq \Theta$, update affected vertices taking into account their new distances to the sink as follows:

– knowing the slice X 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 X;

– by means of the *Algorithm B*, determine *in parallel* the positions of edges (q, l) for which $dist_{new}(q) = w(q, l) + dist_{old}(l)$ and include these positions into $SP(G)$;

– by means of the *Algorithm C*, determine *in parallel* the positions of edges (r, q), 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 procedure DeleteEdge.

6. Implementation of the associative version of the Ramalingam decremental algorithm on the STAR-machine

In this section, we first provide three auxiliary procedures, whose correctness will be proved in the full paper. Then we propose the procedure DeleteEdge.

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

```
procedure FindAffectedVert(i,j: integer; var SP: table;
  var AffectedV: slice(SP));
/* The edge (i,j) is deleted from the matrices G and SP. */
var X,WS: slice(SP);
  v,v1: word(SP);
  k,r: integer;
1. Begin CLR(WS); CLR(AffectedV); CLR(v1);
2.  X:=COL(i,SP);
3.  if ZERO(X) then
/* There was a single edge outgoing from i in SP(G). */
4.    begin WS(i):='1';
5.      while SOME(WS) do
6.        begin k:=STEP(WS);
7.          AffectedV(k):='1';
```

```
 /* The vertex k is written into the slice AffectedV. */
 8.          v:=ROW(k,SP);
 /* The row v saves the heads of edges entering k. */
 9.          ROW(k,SP):=v1;
 /* We delete from SP(G) all edges entering k. */
10.          while SOME(v) do
11.            begin r:=STEP(v);
12.              X:=COL(r,SP);
13.              if ZERO(X) then WS(r):='1';
14.            end;
15.        end;
16.    end;
17. End.
```

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

This claim is proved by induction in terms of the number of vertices to be included into the slice $AffectedV$.

Now we proceed to the procedure UpdateOutgoingEdges. Knowing the current updated vertex k and the current matrices G, $Weight$, $Dist$, and $SP(G)$, the procedure returns the updated matrix $SP(G)$.

```
procedure UpdateOutgoingEdges(h,k: integer; G: table;
  Weight: table; Dist: table; var SP: table);
/* Here h is the number of bits for coding the infinity,
  and k is the updated vertex. */
var W1,W2: table;
  v: word(Dist);
  Y,Z: slice(G);
1. Begin Z:=COL(k,G);
2.  TCOPY1(Weight,h,k,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); WCOPY(v,Z,W1);
/* The shortest distance from the vertex k to the sink
  is written in the rows of the matrix W1 that are marked
  with '1' in the slice Z. */
5.  HIT(W1,W2,Z,Y);
/* In the slic Y, we mark with '1' the vertices l
```

```
    for which dist(k) = w(k,l) + dist(l). */
6.  COL(k,SP):=Y;
/* Positions of edges (k,l) are included into SP(G). */
7. End;
```

Claim 2. *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(G) be given. Then after performing the procedure* UpdateOutgoingEdges *all edges (k,l) for which dist(k)=w(k,l)+dist(l) are included into the matrix SP(G).*

This claim is proved by contradiction. Let an arc (k,r) belong to G and $dist(k) = w(k,r) + dist(r)$. However, after performing the procedure UpdateOutgoingEdges, the arc (k,r) does not belong to $SP(G)$. We prove that it contradicts the execution of this procedure.

Finally, we propose the procedure UpdateIncomingEdges. 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 UpdateIncomingEdges(h,k: integer; G: table;
  Cost: table; var Dist: table);
var Y,Z: slice(G);
  v: word(G);
  v1: word(Dist);
  W,W1,W2: table;
1. Begin v:=ROW(k,G); Z:=CONVERT(v);
/* The slice Z saves the heads of edges entering k. */
2.  v1:=ROW(k,Dist);
  /* The row v1 saves the shortest distance from k to s. */
3.  WCOPY(v1,Z,W1);
4.  TCOPY1(Cost,k,h,W2);
/* The k-th field of the matrix Cost is written into
  the matrix W2. */
5.  ADDV(W1,W2,Z,W);
/* In every l-th row of W that corresponds to '1' in Z,
  the new distance from l to s is written. */
6.  SETMIN(W,Dist,Z,Y);
/* In the slice Y, we mark with '1' positions of vertices,
  whose new distances to the sink are decreased. */
7.  TMERGE(W,Y,Dist);
/* In every l-th row of the matrix Dist, a new value of dist(l)
  is written if and only if Y(l) =' 1'. */
8. End;
```

Claim 3. *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 procedure* UpdateIncomingEdges *maintains the matrix $Dist$, where new distances to the sink are written for the heads r of edges entering the vertex k whose $dist_{new}(r)$ is decreased.*

This claim is proved by contradiction. Let an arc (r, k) belong to G and $dist_{new}(r) < dist_{old}(r)$. However, after performing the procedure UpdateIncomingEdges, the r-th row in the matrix $Dist$ does not changed. We prove that this contradicts the execution of the procedure UpdateIncomingEdges.

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

```
procedure DeleteEdge(i,j,h: integer; Weight,Cost: table;
  var G,SP: table; var Dist: table);
/* The edge (i,j) is deleted from the matrices G and SP. */
var k,r: integer;
  AffectedV,X,Y,Z,Z1: slice(G);
  v: word(Dist);
  W1,W2: table;
  v2: word(G);
  label 1;
1. Begin X:=COL(i,G); X(j):='0';
2.  COL(i,G):=X;
/* The edge (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 edge (i,j) is deleted from SP(G). */
6.  FindAffectedVert(i,j,SP,AffectedV);
/* This procedure returns the updated matrix SP(G)
  and the slice AffectedV. */
7.  X:=AffectedV;
8.  while SOME(AffectedV) do
9.    begin k:=STEP(AffectedV);
10.     Z:=COL(k,G); Z1:=Z and (not X);
/* The slice Z1 saves the sons of k that are not affected. */
11.     TCOPY1(Weight,h,k,W1);
/* The matrix W1 saves the k-th field of the matrix Weight
  consisting of h bits. */
12.     ADDV(W1,Dist,Z1,W2);
```

```
 /* The matrix W2 saves different distances from k to s. */
13.      MIN(W2,Z1,Y);
 /* In the slice Y, we mark with '1' positions of those
   vertices r of G, for which dist(k) = w(k,r) + dist(r). */
14.      r:=FND(Y); v:=ROW(r,W2);
15.      ROW(k,Dist):=v;
 /* The new distance from the vertex k to s is saved
   in the k-th row of the matrix Dist. */
16.    end;
17.    while SOME(X) do
18.      begin MIN(Dist,X,Z);
19.        k:=FND(Z); X(k):='0';
20.        UpdateOutgoingEdges(h,k,G,Weight,Dist,SP);
 /* We include into SP those edges (k,r), for which
   dist(k) = w(k,r) + dist(r). */
21.        UpdateIncomingEdges(h,k,G,Cost,Dist);
 /* We write dist_new(l) into the l-th row of the matrix Dist
   if and only if dist_new(l) < dist_old(l) and the edge (l,k)
   belongs to the path from l to s. */
22.      end;
23. 1: End;
```

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

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

Basis is proved for $q = 1$. One can immediately check that after performing lines 1–5, the edge (i, j) is deleted from the matrices G and $SP(G)$. After performing the procedure FinfAffectedVert (line 6), the slice $AffectedV$ saves the vertex i and all edges, entering the vertex i, are deleted from $SP(G)$. After performing line 7, we have $X(i) =' 1'$. One can easily check that after fulfilling lines 9–12, we have $k = i$, $AffectedV = \Theta$, and the matrix $W2$ saves different distances from the vertex i to the sink. After performing lines 13–16, the new minimum distance from i to s is written in the i-th row of the matrix $Dist$. Since $AffectedV = \Theta$, we carry out line 17. After performing lines 17–19, we have $X = \Theta$ and $k = i$ because initially the slice X is the copy of $AffectedV$. Further, after perform-

ing the procedure UpdateOutgoingEdges (line 20), all edges (i, r) for which $dist_{new}(i) = w(i, r) + dist_{old}(r)$ are included into $SP(G)$.

By assumption, there is a single affected vertex in $SP(G)$. It means that there is an alternative path to the sink for every vertex l, being the head of any edge (l, i) in $SP(G)$. Therefore, after performing the procedure UpdateIncomingEdges (line 21), the matrix $Dist$ does not change.

Hence, after performing the procedure DeleteEdge, the edge (i, j) is deleted from the matrices G and $SP(G)$, $dist_{new}(i)$ is written into the i-th row of the matrix $Dist$, and all edges (i, r), for which $dist_{new}(i) = w(i, r) + dist_{old}(r)$, are included into $SP(G)$.

Step of induction. Let the assertion be true when no more than $q \geq 1$ affected vertices are updated in the given graph. We will prove the assertion for $q + 1$ affected vertices.

One can immediately verify that, after performing lines 1–7, the edge (i, j) is deleted from G and $SP(G)$, the slice $AffectedV$ saves positions of $q + 1$ affected vertices, affected edges are deleted from $SP(G)$, and the slice X is a copy of $AffectedV$. After performing line 9, the position of the first (or uppermost) affected vertex k is determined. By analogy with the basis, after performing lines 10–16, the new distance from k to s is written into the k-th row of the matrix $Dist$. Now, only q affected vertices are marked with $'1'$ in the slice $AffectedV$. By the inductive assumption, after execution of the cycle `while SOME(AffectedV) do` (line 8), new distances from every affected vertex to s will be written in the corresponding rows of the matrix $Dist$.

Since $AffectedV = \Theta$, we carry out the cycle `while SOME(X) do` (line 17). After performing lines 18–19, we determine the position of the affected vertex k having the minimum new distance to s and mark it with $'0'$ in the slice X. After performing the procedure UpdateOutgoingEdges (line 20), we include into $SP(G)$ the positions of edges (k, r), for which $dist_{new}(k) = w(k, r) + dist_{old}(r)$. Further, after performing the procedure UpdateIncomingEdges (line 21), 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 X. By the inductive assumption, after updating q affected vertices, all alternative paths from every affected vertex r to the sink are included into $SP(G)$ and the minimun 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 procedure DeleteEdge. To this end, we first evaluate the time complexity of three auxiliary procedures. Let h be the number of bits for coding the infinity and k be the

number of affected vertices that appear in $SP(G)$ after deleting the edge (i, j). The auxiliary procedure FindAffectedVert takes $O(k)$ time because the cycle from line 10 takes $O(1)$ time which is no greater than the maximum number of bits $'1'$ in the rows of the matrix $SP(G)$. The auxiliary procedures UpdateOutgoingEdges and UpdateIncomingEdges take $O(h)$ time each. In the procedure DeleteEdge, the cycle `while SOME(AffectedV) do` (lines 10–16) and the cycle `while SOME(X) do` (lines 18–22) take $O(kh)$ time each. Hence, the procedure DeleteEdge takes $O(kh)$ time.

Now we compare implementations of the Ramalingam decremental algorithm and its associative version:

– the Ramalingam decremental algorithm uses *a heap of vertices*, where the distance from any affected vertex to the sink is its current key in the heap. The associative version saves the current distance from any affected vertex r to s in the r-th row of the matrix $Dist$;

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession every affected edge entering r. The associative version *simultaneously* determines all affected edges entering r;

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession different distances from r to s and assignes the minimum distance among them to the current key for the vertex r in the heap. The associative version *simultaneously* determines different distances from the affected vertex r to s and writes the minimum distance from r to s into the r-th row of the matrix $Dist$;

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession those edges (r, l), for which $dist(r) = w(r, l) + dist(l, s)$. The associative version *simultaneously* determines *positions* of such edges (r, l);

– for every affected vertex r, the Ramalingam decremental algorithm determines in succession those edges (q, r), for which the shortest path from q to s includes the edge (q, r) and $dist_{new}(q) < dist_{old}(q)$, then assigns $dist_{new}(q)$ to the current key of q in the heap. The associative version *simultaneously* determines *positions* of such heads of edges (q, r), then *simultaneously* writes the new distances from these heads to the sink into the corresponding rows of the matrix $Dist$.

7. Conclusions

We have proposed a new data structure for efficient implementation of the Ramalingam decremental algorithm on the STAR-machine having no less than n PEs. The associative version of the Ramalingam decremental algorithm is represented as the procedure DeleteEdge, whose correctness is proved. We have obtained that this 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 the edge (i, j). It is assumed that each microstep of the STAR-machine takes one unit of time. We have also compared the implementations of the Ramalingam decremental algorithm and its associative version and present the main advantages of the associative version.

We are planning to design an associative version of the Ramalingam incremental algorithm for the dynamic update of the shortest paths subgraph after insertion of an edge into the given graph.

References

[1] Ausiello G., Italiano G.F., Marchetti-Spaccamela A., and Nanni U. Incremental algorithms for minimal length paths // J. of Algorithms. – 1991. – Vol. 12, N 4. – P. 615–638.

[2] Chaudhuri S., Zaroliagis C.D. Shortest path queries in digraphs of small treewidth // Proc. of Intern. Colloquiumon Automata Languages, and Programming. Szeged, Hunhary, July 10–14, 1995. – Lect. Notes Comput. Sci. – Berlin: Springer–Verlag, 1995. – Vol. 944. – P. 244–255.

[3] Dijkstra E. W. A Note on Two Problems in Connection with Graphs // Numerische Mathematik. – 1959. – Vol. 1. – P. 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 // Proc. of 14-th Annual Symp. on Theoretical Aspects of Computer Science. Lubeck, Germany, February/March, 1997. – Lect. Notes Comput. Sci. – Berlin: Springer-Verlag, 1997. – Vol. 1200. – P. 33–46.

[6] Frigioni D., Marchetti-Spaccamela A., Nanni U. Fully dynamic algorithms for maintaining shortest paths trees // J. of Algorithms, Academic Press. – 2000. – Vol. 34, N 2. — P. 351–381.

[7] Frigioni D., Marchetti-Spaccamela A., Nanni U. Semi-dynamic algorithms for maintaining single source shortest paths trees // Algorithmica, Berlin: Springer-Verlag. - 1998. – Vol. 25, N 3. – P. 250–274.

[8] Frigioni D., Marchetti-Spaccamela A., Nanni U. Fully dynamic shortest paths in digraphs with arbitrary arc weights // J. of Algorithms, Elsevier Science. – 2003. – Vol. 49, N 1. – P. 86–113.

[9] Klein P.N., Rao S., Rauch M., Subramanian S. Faster shortest path algorithms for planar graphs // Proc. ACM Symp. on Theory of Computing. Montreal, Quebec, Canada, May 23–25. – 1994. – P. 27–377.

[10] Narváez P., Siu K.-Y., Tzeng H.-Y. New dynamic algorithms for shortest paths tree computation // IEEE/ACM Trans. Networking. – 2000. – Vol. 8, N 6. – P. 734–746.

[11] Nepomniaschaya A. S. Language STAR for associative and parallel computation with vertical data processing // Proc. of the Intern. Conf. "Parallel Computing Technologies". – World Scientific, Singapore, 1991. – P. 258–265.

[12] Nepomniaschaya A. S. Solution of path problems using associative parallel processors // Proc. of the Intern. Conf. on Parallel and Distributed Systems, ICPADS'97, Korea, Seoul. – IEEE Computer Society Press, 1997. – P. 610–617.

[13] Nepomniaschaya A. S., Dvoskina M. A. A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors // Fundamenta Informaticae. – IOS Press, 2000. – Vol. 43. – P. 227–243.

[14] Ramalingam G. Bounded incremental computation // Lect. Notes Comput. Sci. – Berlin: Springer-Verlag, 1996. – Vol. 1089.

[15] Ramalingam G., Reps T. An incremental algorithm for a generalization of the shortest paths problem // J. of Algorithms. – Academic Press, 1996. – Vol. 21. – P. 267–305.