Bull. Nov. Comp. Center, Comp. Science, 33 (2012), 43–57

A technique for finding the second simple shortest paths using associative parallel processors

A. S. Nepomniaschaya

Abstract. This paper proposes a technique to build the data structure for finding the second simple shortest paths on a model of associative parallel processors (the STAR-machine). It includes three associative parallel algorithms represented on the STAR-machine as the corresponding procedures. We prove the correctness of these procedures and evaluate their time complexity. Among these algorithms, the new version of the Dijkstra algorithm and the associative parallel algorithm for finding the matrix of tree paths can be used to solve other graph problems on the STAR-machine.

1. Introduction

The problem of finding k shortest paths is a natural generalization of the shortest paths problem when several shortest paths are to be determined. Given a graph G with n vertices and m edges, two vertices s and t, and an integer k, one has to enumerate k shortest paths from s to t in the order of increasing their length.

For the problem of finding k simple (loopless) shortest paths, the fastest $O(k(m+n\log n))$ time algorithm for undirected graphs was proposed by Katoh et al. [5] and the best $O(kn(m + n\log n))$ time algorithms for directed graphs were proposed independently by Yen [12] and Lawler [6]. Such estimations are obtained using the modern data structures for implementing the classical Dijkstra algorithm [1]. For unweighted directed graphs and for graphs with small integer weights, Roditty and Zwick [11] proposed a randomized algorithm for finding k simple shortest paths that runs in $O(km\sqrt{n}\log n)$ time. This estimation is obtained due to the use of their replacement paths algorithm. The authors also reduced the problem of finding k simple shortest paths to $O(k)$ computations of the second simple shortest path each time in a different subgraph of G. Gotthilf and Lewenstein [4] presented an $O(k(mn + n^2\log\log n))$ time algorithm for finding k simple shortest paths from s to t in weighted directed graphs using the efficient solution of the all-pairs shortest paths problem. In the case of directed graphs, Eppstein [2] proposed an efficient $O(m + n\log n + kn)$ time algorithm for finding k shortest paths (allowing cycles) from s to each vertex of G. This algorithm builds an implicit representation of paths. The edges in any path can be explicitly listed in the time proportional to the number of edges.

In [10], we proposed an efficient associative algorithm for finding the second simple shortest paths from s to all vertices of a directed weighted graph. Our model of computation (the STAR-machine) simulates running associative (content addressable) parallel systems of the SIMD type with vertical processing. Following Foster [3], we assume that each elementary operation of the model (its microstep) takes one unit of time. On the STAR-machine, the associative algorithm for finding the second simple shortest paths from s to all graph vertices was implemented as procedure `SecondPaths`, whose correctness was proved. The procedure uses the graph representation as a list of triples (edge endpoints and the weight) and the shortest paths tree as a bit-column that saves positions of the tree edges. The procedure `SecondPaths` returns a matrix `TPaths`[2], whose every ith column saves positions of edges belonging to the second simple shortest path from s to i. We obtain that it takes $O(r(\log n + \deg^+(G)))$ time, where r is the number of non-tree edges that are really used for finding the second simple shortest paths and $\deg^+(G)$ is the maximum number of edges outgoing from graph vertices.

In this paper, we propose special tools for finding the second simple shortest paths on the STAR-machine. They include a new implementation of the Dijkstra algorithm on the STAR-machine, the use of the Eppstein function, and building a matrix of tree paths. We provide the corresponding procedures and prove their correctness.

2. An associative parallel machine model

Our model is defined as a STAR-machine of the SIMD type with bit-serial (vertical) processing and simple single-bit processing elements. Its description is given in [7]. Here, we first provide a few remarks and then recall a group of operations and basic procedures to be used.

The input binary data are given in the form of two-dimensional tables, where each data item occupies an individual row and is updated by a dedicated processing element. In every matrix, the rows are numbered from top to bottom and the columns — from left to right.

To simulate data processing in the matrix memory, one uses data types **word**, **slice**, and **table**. 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 tabular data. For simplicity, let us call *slice* any variable of the type **slice**.

Recall the main operations for slices.

Let X, Y be two slices 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 a value of the ith 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 $'0'$.

To carry out the data parallelism, the following bitwise Boolean operations are introduced in the usual way: X *and* Y, X *or* Y, *not* Y, X *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 slices are also performed for the type **word**. Moreover, for two variables v and w of the type **word** the following operations are used:

ADD(v, w) performs addition of the binary strings v and w having the same length. Its result is the arithmetical sum of v and w.

SUBT(v, w) performs subtraction of the binary string w from the binary string v for the case when $v > w$ and they have the same length. Its result is the difference of these strings.

TRIM(i, j, w) cuts a substring of the string w from the ith through the jth bits, where $1 \leq i < j \leq |w|$.

Recall a group of basic procedures. They utilize a given slice X to indicate with the bit $'1'$ the row positions used in the corresponding procedure. In [7], we have shown that these procedures take $O(l)$ time each, where l is the number of bit columns in the corresponding matrix.

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

The procedure MIN(T, X, Z) simultaneously defines positions of the matrix T rows, where minimum entries are located. It returns the slice Z, where $Z(i) = '1'$ if and only if ROW(i, T) is the minimum entry of the matrix T and $X(i) = '1'$. To return the minimum entry of T, we define the position of the first $'1'$ in Z and then select the corresponding row of T.

The procedure SETMIN(T, F, X, Y) simultaneously defines the positions of the matrix T rows being less than those 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 ADDC(T, X, v, F) simultaneously adds the binary word v to the matrix T rows selected by $'1'$ in X, and writes down the result into

the corresponding rows of the matrix F. The rows of F, which are selected by $'0'$ in X, consist of zeros.

The procedure ADDV(T, F, X, R) simultaneously writes the result of adding rows of the matrices T and F selected by ones in the slice X into the corresponding rows of the matrix R.

The procedure TMERGE(T, X, F) writes the matrix T rows selected by ones in X into the matrix F. The rows of the matrix F selected by zeros in X are not changed.

The procedure WCOPY(v, X, F) writes the binary word v into the matrix F rows selected by ones in X. The rows of the matrix F selected by zeros in X consist of zeros.

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

3. Preliminaries

Let $G = (V, E)$ denote a *digraph* with n vertices and m directed edges (arcs). We assume that $V = \{1, 2, \ldots, n\}$. Let $wt(e)$ denote a function that assigns a weight to every edge e. We assume that $wt(u, v) = \infty$ if $(u, v) \notin E$. We also assume that the arcs belonging to the set E have a nonnegative weight.

An arc e directed from u to v is denoted by $e = (u, v)$, where $u = \text{tail}(e)$ and $v = \text{head}(e)$.

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

An *adjacency matrix* A for G is an $n \times n$ Boolean matrix, where $a_{ij} = 1$ if $(v_i, v_j) \in E$ and $a_{ij} = 0$, otherwise.

The *shortest path* from s to v_k is a finite sequence of vertices $s = v_1, v_2, \ldots, v_k$, where $(v_i, v_{i+1}) \in E$ $(1 \leq i < k)$, and the sum of weights of the corresponding arcs is minimal. Let $\text{dist}(s, v_k)$ denote the *length* of the shortest path from s to v_k. If there is no path between these vertices, then $\text{dist}(s, v_k) = \infty$.

The *shortest paths tree* F with a root s is a connected acyclic subgraph of G which includes all graph vertices, and for every vertex v_j there is a unique shortest path from s. A *leaf* in a tree is a vertex that has no outgoing arcs. The *height* of a tree is the number of arcs in the longest path from the root to a leaf.

The arcs of G that do not belong to F are called *non-tree* edges. For any path p, let *sidetracks*(p) be a sequence of non-tree edges that belong to p.

For every arc (u, v) in G, Epstein [2] defines a function $\delta(u, v) = wt(u, v) + \text{dist}(s, u) - \text{dist}(s, v)$. Informally, $\delta(u, v)$ shows how much a distance is lost if, instead of taking the shortest path from s to v, we first use the shortest path from s to u and then take the arc (u, v). Clearly, for every

$e \in G$, $\delta(e) \geq 0$, and for every $e \in T$, $\delta(e) = 0$. If $\delta(e)$ is considered to be a weight function on the arcs of G, then the weight of every path p will be equal to the sum of weights of the non-tree edges that appear in this path. Therefore the problem of finding k shortest paths p can be stated as the problem of computing k smallest values of $\sum_{(u,v)\in \text{sidetracks}(p)} \delta(u,v)$.

4. Data structure

In this section, we explain how to obtain the data structure used for finding the second simple shortest paths from s to all vertices of a digraph on the STAR-machine. We will use the following data structure:

- an $n \times hn$ matrix `Weight` that contains the arc weights as entries. It consists of n fields having h bits each. Every ith field of this matrix saves the weights of arcs *outgoing* from the vertex i;
- an $n \times hn$ matrix `Weight1` that contains the arc weights as entries. It consists of n fields having h bits each. Every jth field of this matrix saves the weights of arcs *entering* the vertex j;
- an $n \times \log n$ matrix `Code`, whose every ith row saves the binary representation of the vertex i;
- an $n \times h$ matrix `Dist`, whose every ith row saves the shortest distance from the vertex s to the vertex i;
- an association of $m \times \log n$ matrices `Left` and `Right` that is built from the matrix `Weight` as follows: we first write a group of arcs outgoing from vertex 1, then we write a group of arcs outgoing from vertex 2, and so on;
- a slice `Tree` that saves positions of arcs that belong to the shortest paths tree;
- an $m \times n$ matrix `TPaths`, whose every ith column saves with bit $'1'$ the positions of arcs that belong to the shortest path from s to the vertex v_i;
- an $m \times h$ matrix `Cost` that saves the value of the function $\delta(u,v)$ for every arc (u,v).

Note that initially a graph G is given as a matrix `Weight` that consists of n fields. Such a representation is necessary both to find the matrix `Dist` in the implementation of the Dijkstra algorithm on the STAR-machine and to compute the matrix `Cost` that consists of m rows. Knowing the matrix `Weight`, one can easily build the matrices `Left` and `Right` by means of the method presented above. As soon as we obtain the matrices `Dist` and `Cost`, we will further use the matrix `Cost` instead of the matrix `Weight`. Therefore the associative algorithm for finding the second simple shortest

paths from s to all vertices of G will use a graph representation as association of the matrices `Left`, `Right`, and `Cost`, where every arc (u, v) occupies an individual row, $u \in$ `Left`, $v \in$ `Right`, and $\delta(u, v) \in$ `Cost`.

5. Special tools for finding the second simple shortest paths on the STAR-machine

In this section, we first provide a special implementation of the classical Dijkstra algorithm on the STAR-machine that allows us to simultaneously obtain the distances and the shortest paths tree (SPT) given as a slice `Tree`. Then we explain how to build the matrix `Cost` that computes the function $\delta(u, v)$ for every arc (u, v). Finally, we build a matrix of tree paths `TPaths`, whose every ith column saves the positions of arcs belonging to the shortest path from s to the vertex i.

5.1. A new implementation of the Dijkstra algorithm on the STAR-machine. The Dijkstra algorithm [1] assigns temporal labels $l(v)$ to each vertex $v \in V$ so that $l(v) \geq \mathrm{dist}(s, v)$. These labels are constantly decreased by means of a certain iteration procedure, and at each step only a unique temporal label becomes invariant. The algorithm constructs a set of vertices $F \subseteq V$ in such a way that the current shortest path from s to any vertex of F passes only through vertices in F. Initially, $F = \{s\}$, $l(s) = 0$ and $\forall v \notin F\ l(v) = \infty$. Let F consist of k vertices ($1 \leq k < n$) and u be the last vertex included into F. Then the $(k+1)$th vertex for the set F is defined as follows.

We first define all arcs (u, v_i), where $v_i \notin F$. Then for every vertex $v_i \notin F$, we determine the label $l(v_i) = \mathrm{dist}(s, u) + wt(u, v_i)$. After that, among the vertices $v_i \notin F$ being adjacent with a vertex from F, we select such a vertex v whose label has the minimum value and include it into the set F. The label $l(v)$ is minimized as follows. If $l(u) + wt(u, v) < l(v)$, then $l(v) := l(u) + wt(u, v)$. On terminating the algorithm, $l(v_i)$ is the weight of the shortest path from s to v_i for all $v_i \in V$.

To select *simultaneously* the distances and the shortest paths tree given as the slice `Tree`, we make use of the following idea.

By means of the method used in the procedure `DistPaths` [8], we first define the current vertex v_k, which is included into SPT F, and the distance from s to v_k. Then we define such a vertex v_i from F, which is next to the last one in the shortest path from s to v_k. Further, we determine the *position* of the arc (v_i, v_k) and include it into the slice `Tree`.

The associative parallel algorithm is given as procedure `TreeDist`, which uses the following parameters: the matrices `Left`, `Right`, `Weight`, `Weight1`, and `Code`, the source vertex s, the number of graph vertices n, the binary representation of infinity **inf**, and the number of bits h for representing

infinity. The procedure returns both the matrix `Dist` and the slice `Tree`.

The associative parallel algorithm performs the following steps:

Step 1. Define the *position* of the current vertex v_k, which is included into SPT F, and the distance from s to v_k.

Step 2. By means of a slice, say X, save *positions* of those vertices v_j from F, for which there is an arc entering v_k. Then compute in parallel the weights of paths from s to v_k selected with the bit $'1'$ in the slice X.

Step 3. Select the *position* of such a vertex v_i from F, for which $\mathrm{dist}(s, v_i) + wt(v_i, v_k) = \min_j\{\mathrm{dist}(s, v_j) + wt(v_j, v_k)\}$. After that determine the *position* of the arc (v_i, v_k) and include it into the slice `Tree`.

Note that the procedure `TreeDist` uses the auxiliary procedure `Adj` presented in [8] that returns the adjacency matrix A for the specified matrix `Weight1`.

Consider the procedure `TreeDist`.

```
procedure TreeDist(Left,Right: table; Weight,Weight1: table;
          Code: table; s,h,n: integer; inf: word(Weight);
          var Tree: slice(Left); var Dist: table);
var i,j,k: integer;
    U,X,Z: slice(Weight); Y,Y1,Y2: slice(Left);
    v: word(Dist); v1,v2: word(Code);
    A,R1,R2: table;
 1. Begin Adj(Weight1,h,n,inf,A);
 2.   SET(Y); CLR(Tree); SET(U); U(s):='0';
 3.   WCOPY(inf,U,Dist); k:=s;
   /* Here k saves the last vertex included into F. */
 4.   while SOME(U) do
 5.     begin
   /* The first stage. */
 6.       TCOPY1(Weight,k,h,R1);
 7.       MATCH(R1,U,inf,X);
 8.       X:=X xor U;
   /* Positions of v_i ∉ F forming an arc v_k → v_i are marked with '1'
      in the slice X. */
 9.       v:=ROW(k,Dist);
10.       ADDC(R1,X,v,R2);
   /* Here, l(v_k) + wt(v_k, v_i) is written in every row of R2 marked with '1'
      in the slice X. */
11.       SETMIN(R2,Dist,X,Z);
12.       TMERGE(R2,Z,Dist);
```

```
    /* Here l(v_i) is decreased to l(v_k) + wt(v_k, v_i) in every ith row of Dist
       marked by '1' in Z. */
13.       MIN(Dist,U,X); k:=FND(X);
14.       U(k):='0';
    /* A new vertex is included into F. */
    /* The second stage. */
15.       X:=COL(k,A);
16.       X:=X and (not U);
    /* Positions of vertices from F, for which there is an arc entering v_k,
       are marked with '1' in X. */
17.       X(k):='0';
18.       TCOPY1(Weight1,k,h,R1);
    /* The kth field of Weight1 is stored in R1. */
19.       ADDV(R1,Dist,X,R2);
    /* The weights of paths from s to v_k, selected by '1' in X, are saved
       in the corresponding rows of R2. */
    /* The third stage. */
20.       MIN(R2,X,Z); i:=FND(Z);
    /* The minimum weight of arcs included into F is attained for the vertex v_i. */
21.       v1:=ROW(i,Code); v2:=ROW(k,Code);
22.       MATCH(Left,Y,v1,Y1);
23.       MATCH(Right,Y1,v2,Y2);
24.       j:=FND(Y2);
25.       Tree(j):='1';
26.     end;
27. End;
```

Theorem 1. *Let the matrices* `Left`, `Right`, `Weight`, `Weight1`, *and* `Code` *be given. Let the integers* s, h, n *and the binary word* ***inf*** *be also given. Then the procedure* `TreeDist` *returns the slice* `Tree` *that saves the positions of arcs belonging to the shortest paths tree, and the matrix* `Dist`, *whose every* i*th row saves the distance from* s *to* v_i. *It takes* $O(n \max(h, \log n))$ *time.*

Since the procedure `TreeDist` determines the matrix `Dist` in the same manner as the procedure `DistPaths` [8], we only have to check that the procedure `TreeDist` returns the slice `Tree`.

Proof. (Sketch.) We prove this by induction on the number of arcs q included into the SPT F.

Basis is proved for $q = 1$. After performing the initialization (lines 1–3), the slice `Tree` consists of zeros, and the root s is included into the SPT F. After performing lines 6–12, the weights of arcs outgoing from the

root s are written in the corresponding rows of the matrix `Dist`. After performing lines 13–14, we determine the head v_k of the arc outgoing from s with the minimum weight and include v_k into the SPT F. After performing lines 15–17, there is a single bit $'1'$ in the slice X, that is, $X(s) = '1'$. After performing lines 18–19, we obtain the weight of the arc (s, k) because $\texttt{Dist}(s) = 0$. After performing lines 21–25 of the third stage, we determine the *position* of the first arc (v_i, v_k) and include it into the slice `Tree`.

Step of induction. Let the assertion be true for $q \geq 1$. We will prove this when the positions of $q + 1$ arcs will be included into the slice `Tree`.

By the inductive assumption after including the first q vertices into the shortest paths tree F, the positions of the corresponding arcs will be marked with $'1'$ in the slice `Tree`. Let k be the last vertex included into F. Since the slice U is non-empty, we start to perform the first stage. Further we use the same reasoning as for the basis. After performing the first stage, we determine the $(q + 1)$th vertex for including into F. Its position is marked with $'0'$ in the slice U. After performing the second stage, we determine the weights of different paths from the root to the $(q + 1)$th vertex and write them in the corresponding rows of the matrix $R2$. Really, the tails of arcs entering the $(q + 1)$th vertex have been already included into the shortest paths tree F. Therefore the matrix `Dist` saves the corresponding weights of the shortest paths from the root. After performing line 20 of the third stage, we determine the tail of the arc entering the $(q+1)$th vertex and having the minimum weight. After fulfilling lines 21–25, we determine the position of this arc being the last one included into the slice `Tree`. □

Let us evaluate the time complexity of this procedure. We first observe that the execution of lines 1–3 takes $O(hn)$ time in view of the auxiliary procedure `Adj`. Inside the cycle, the basic procedures `MATCH(Left,Y,v1,Y1)` and `MATCH(Right,Y1,v2,Y2)` (lines 22–23) take $O(\log n)$ time each, while the other basic procedures take $O(h)$ time each. Since the cycle is executed n times, the procedure `TreeDist` takes $O(n \max(h, \log n))$ time.

5.2. Finding new weights using the Eppstein function. Here, we provide an associative parallel algorithm for representing the Eppstein function and its implementation on the STAR-machine. Note that the Eppstein function must be implemented after performing the procedure `TreeDist`.

Let the matrices `Left`, `Right`, `Weight`, `Dist`, and `Code`, and the slice `Tree` be given. The associative parallel algorithm performs the following steps:

Step 1. Define the *position* l of the current non-tree arc, say γ, in the association of the matrices `Left` and `Right`. Determine the endpoints of γ. Let $\gamma = (i, j)$, where $i = \text{tail}(\gamma)$ and $j = \text{head}(\gamma)$.

Step 2. Save the weight of γ and distances from the root to its endpoints. Let $w_1 = wt(\gamma)$, $s_1 = \mathrm{dist}(s, i)$ and $s_2 = \mathrm{dist}(s, j)$.

Step 3. Compute $\delta(i, j) = w_1 + s_1 - s_2$ and write the result into the lth row of the matrix `Cost`.

Step 4. Simultaneously write a string consisting of h zeros in the rows of the matrix `Cost` that correspond to positions of the bit $'1'$ in the slice `Tree`.

Consider the implementation of this algorithm on the STAR-machine.

```
   procedure Recount(Left,Right: table; Weight: table;
             Code: table; Dist: table; Tree: slice(Left);
             h: integer; var Cost: table);
   var X,X1: slice(Left); Y,Z: slice(Code);
       w: word(Code); s1,s2,v1,w1,w2,w3: word(Dist);
       v: word(Weight); i,j,l: integer;
 1. Begin SET(Y); CLR(w3); X:= not Tree;
 2.   while SOME(X) do
 3.     begin l:=STEP(X);
   /* We select the position l of the non-tree arc in the association of
      the matrices Left and Right. */
 4.       w:=ROW(l,Left);
 5.       MATCH(Code,Y,w,Z); i:=FND(Z);
   /* We determine the tail of the non-tree arc from the lth row of
      the graph representation. */
 6.       w:=ROW(l,Right);
 7.       MATCH(Code,Y,w,Z); j:=FND(Z);
   /* We determine the head of the non-tree arc from the lth row of
      the graph representation. */
 8.       s1:=ROW(i,Dist); s2:=ROW(j,Dist);
   /* Here s1 (respectively, s2) saves the distance from the root
      to the vertex i (respectively, j). */
 9.       v:=ROW(j,Weight);
10.       w1:=TRIM(1+(i-1)h,ih,v);
   /* Here w1 saves the weight of the arc (i, j). */
11.       v1:=ADD(s1,w1); w2:=SUBT(v1,s2);
12.       ROW(l,Cost):=w2;
   /* We write δ(i, j) in the lth row of Cost. */
13.     end;
14.   WMERGE(w3,Tree,Cost);
```

```
    /* In the matrix Cost, we set to zero the weights of arcs from
       the shortest paths tree. */
15. End;
```

Theorem 2. *Let the matrices* `Left`, `Right`, `Weight`, `Dist`, *and* `Code`, *and the slice* `Tree` *be given. Then the procedure* `Recount` *returns the matrix* `Cost`, *which saves the arc weights obtained by means of the Eppstein function.*

Proof. (Sketch.) We prove this by contradiction. Let all assumptions of Theorem 2 be true. However, there is such an arc, say γ, whose new weight is not equal to $\delta(\gamma)$ after performing the procedure `Recount`. We will prove that this contradicts the execution of our procedure.

To this end, we first analyze the update of the non-tree arc γ. Initially, after performing line 1 of the procedure `Recount`, the slice X saves *positions* of all non-tree arcs in the association of the matrices `Left` and `Right`. Since the position of any current non-tree arc is determined by means of the operation STEP(X), we will consider the case when the position of γ is deleted from X. After performing line 3, we determine the *position* l, where the arc γ is located in the association of matrices `Left` and `Right`. After performing lines 4–7, we obtain $\gamma = (i, j)$, that is, $i = \text{tail}(\gamma)$ and $j = \text{head}(\gamma)$. Knowing the endpoints of γ and the matrix `Dist`, we determine the distances $s1$ and $s2$ to the vertices i and j, respectively (line 8). After performing lines 9–10, we determine the weight of the non-tree arc γ, that is, $wt(i, j)$. Finally, after performing lines 11–12, we compute the Eppstein function $\delta(i, j)$ and write it in the corresponding row of the matrix `Cost`. This contradicts our assumption.

Consider the case when $\gamma \in T$. Then after performing line 14, $\delta(\gamma)$ consisting of zeros is written in the corresponding rows of the matrix `Cost`. This also contradicts our assumption. □

Let us evaluate the time complexity of the procedure `Recount`. This procedure runs in $O(r \max(h, \log n))$ time because the cycle `while SOME(X) do` (line 2) is performed r times, where r is the number of non-tree arcs, the basic procedure MATCH takes $O(\log n)$ time and the basic procedure WMERGE takes $O(h)$ time.

5.3. Finding the matrix of tree paths. Here, we first present an associative parallel algorithm for finding the matrix of tree paths `TPaths`, whose every ith column saves the positions of tree edges belonging to the shortest path from the root to the vertex i. Then we propose the procedure `TreePaths` to implement this algorithm on the STAR-machine. Finally, we prove the correctness of this procedure and evaluate its time complexity.

Let a graph be given as association of matrices `Left` and `Right`. Let its shortest paths tree be given as a slice `Tree`. Since $V = \{1, 2, \ldots, n\}$, let us agree that the vertex v_1 is the root of the tree.

Let $P(r)$ denote the shortest path from v_1 to the vertex r. The associative parallel algorithm uses the following idea proposed in [9]. Assume we know *positions* of arcs included into $P(l)$. Then we construct a tree path for such a vertex v_k which is the head of the arc (v_l, v_k) in the shortest paths tree, and $P(k)$ has not been defined yet. The shortest path $P(k)$ is obtained by including the position of the arc (v_l, v_k) into $P(l)$.

The associative parallel algorithm performs the following steps:

Step 1. Write zeros in the first column of the matrix `TPaths`.

Step 2. By means of a slice, say X, save *positions* of tree edges outgoing from the root.

Step 3. While the slice X is nonempty, perform the following actions:

- select the *position* i of the uppermost tree edge, say γ, and mark it with zero in the slice X;
- determine the endpoints of γ. Let $k = \mathrm{tail}(\gamma)$ and $j = \mathrm{head}(\gamma)$;
- by means of a slice, say Z, save the kth column of the matrix `TPaths`, then include the *position* of the tree edge γ into Z;
- write this result into the jth column of the matrix `TPaths`;
- mark the *position* of the tree edge γ as an updated one in the slice `Tree`;
- determine *positions* of tree edges outgoing from $\mathrm{head}(\gamma)$ and include them into the slice X.

Consider the procedure `TreePaths`.

```
procedure TreePaths(Left,Right: table; Code: table;
          Tree: slice(Left); var TPaths: table);
var U,U1: slice(Code); X,Y,Z: slice(Left);
    node1,node2: word(Code);
    i,j,k: integer;
1. Begin SET(U); Y:=Tree;
2.   CLR(X); COL(1,TPaths):=X;
  /* We write zeros in the first column of TPaths. */
3.   node1:=ROW(1,Code);
  /* Here node1 saves the binary code of the root. */
4.   MATCH(Left,Y,node1,Z); X:=Z;
  /* By means of the slice X, we save positions of tree edges
     outgoing from the root. */
```

```
 5.   while SOME(X) do
 6.     begin i:=STEP(X);
 7.       node1:=ROW(i,Left);
    /* Here, node1 saves the binary code of the vertex
       for which the tree path has been obtained. */
 8.       node2:=ROW(i,Right);
    /* Here node2 saves the binary code of the vertex
       for which the tree path has not been obtained yet. */
 9.       MATCH(Code,U,node1,U1); k:=FND(U1);
10.       MATCH(Code,U,node2,U1); j:=FND(U1);
11.       Z:=COL(k,TPaths); Z(i):='1';
12.       COL(j,TPaths):=Z;
    /* We have obtained the tree path to the vertex v_j. */
13.       Y(i):='0';
    /* We mark the edge from the ith position of the tree as an updated one. */
14.       MATCH(Left,Y,node2,Z); X:=X or Z;
15.     end;
16. End;
```

Theorem 3. *Let the matrices* `Left`*,* `Right`*, and* `Code` *and the slice* `Tree` *be given. Then the procedure* `TreePaths` *returns the matrix* `TPaths`*, whose every ith column saves the positions of edges belonging to the tree path from* v_1 *to the vertex* v_i*.*

Proof. (Sketch.) We prove this by induction on the height r of the shortest paths tree.

Basis is proved for $r = 1$, that is, the shortest paths tree only consists of arcs outgoing from v_1. Let us consider the execution of this procedure. After performing lines 1–4, the first column of the matrix `TPaths` consists of zeros, and the slice X saves *positions* of tree edges outgoing from v_1. After performing line 6, we determine the position of the uppermost arc, say γ, outgoing from v_1 in the association of the matrices `Left` and `Right`. After fulfilling lines 7–10, we determine end-points of γ. Since *node*1 is the binary code of v_1 (line 3), we obtain $\gamma = (1, j)$. After performing lines 11–12, the jth column of `TPaths` saves the position of γ in the association of matrices `Left` and `Right`, that is, the shortest path to the vertex j. Then the position of γ in the shortest paths tree is replaced with zero (line 13). After fulfilling line 14, none of new arcs is included into the slice X. In the same manner, we update other tree arcs outgoing from v_1. As soon as the slice X consists of zeros, we go to the end of the procedure.

Step of induction. Let the assertion be true for the shortest paths trees of the height $r \geq 1$. We prove this for the trees of the height $r+1$. Consider the update of a sequence of tree edges $\gamma_1\gamma_2 \ldots \gamma_r\gamma_{r+1}$. By the inductive

assumption, after updating every edge γ_i $(1 \leq i \leq r)$ of this sequence, the column of the matrix `TPaths`, whose number is head(γ_i), saves the *positions* of all edges belonging to the shortest path from v_1 to head(γ_i).

Consider the update of the tree edge γ_{r+1}. In this case, the uppermost bit $'1'$ in the slice X corresponds to the position of γ_{r+1} in the association of the matrices `Left` and `Right`. Here, we make use of the same reasoning as in the case of the basis. After performing lines 6–10, we first determine the position i of the tree edge γ_{r+1} and replace it with zero. Then we define its end-points, that is, $\gamma_{r+1} = (k, j)$, where $k = \text{tail}(\gamma_{r+1})$ and $j = \text{head}(\gamma_{r+1})$. Since the shortest path from v_1 to k consists of r edges, the vertex k belongs to the shortest paths tree of the height not exceeding r. By the inductive assumption, the kth column of the matrix `TPaths` saves the *positions* of edges belonging to the shortest path from v_1 to k. After performing lines 11–12, we first include the position of the edge γ_{r+1} into the shortest path from v_1 to k and then write the result into the jth column of the matrix `TPaths`. After fulfilling lines 13–14, we first mark the edge γ_{r+1} as an updated one in the shortest paths tree. Then the positions of edges outgoing from j in the current shortest paths tree are included into the slice X. Since the jth column of the matrix `TPaths` saves the shortest path from v_1 to j, in future one can determine the shortest path from v_1 to the head of every edge outgoing from j in the current shortest paths tree. □

Obviously, the procedure `TreePaths` runs in $O(n \log n)$ time.

6. Conclusion

We have proposed a technique to build the data structure for finding the second simple shortest paths on associative parallel processors. This data structure can also be used for finding k simple shortest paths from the root to other graph vertices. Unlike the previous implementations of the Dijkstra algorithm on the STAR-machine, the proposed one uses a graph representation as a list of edges. The corresponding procedure returns the shortest paths tree in the form of a slice that saves the positions of tree edges. This version of the Dijkstra algorithm along with the associative algorithm for finding the matrix of tree paths can be used, in particular, to solve the replacement paths problem on the STAR-machine.

We are planning to present the associative parallel algorithm for finding k simple shortest paths from the root to other graph vertices using the technique proposed in this paper.

References

[1] Dijkstra E.W. A note on two problems in connection with graphs // Numerische Mathematik. — 1959. — No. 1. — P. 269–271.

[2] Eppstein D. Finding the k shortest paths // SIAM J. of Computing. — 1998. — Vol. 28. — P. 652–673.

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

[4] Gotthilf Z., Lewenstein M. Improved algorithms for the k simple shortest paths and the replacement paths problems // Information Processing Letters. — Elsevier, 2009. — Vol. 109. — P. 352–355.

[5] Katoh N., Ibaraki T., Mine H. An efficient algorithm for k shortest simple paths // Networks. — 1982. — Vol. 12. — P. 411–427.

[6] Lawler E.L. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem // Management Science. — 1972. — Vol. 18. — P. 401–405.

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

[8] Nepomniaschaya A.S. Simultaneous finding the shortest paths and distances in directed graphs using associative parallel processors // Proc. Intern. Conf. "Information Visualization". — Los Alamitos: IEEE Computer Society, 2003. — P. 665–670.

[9] Nepomniaschaya A.S. Associative parallel algorithms for computing functions defined on paths in trees // Proc. Intern. Conf. on Parallel Computing in Electrical Engineering (PARELEC 2002). — Los Alamitos: IEEE Computer Society, 2002. — P. 399–404.

[10] Nepomniaschaya A.S. Efficient associative algorithm for finding the second simple shortest paths in a digraph // Proc. 11th Intern. Conf. on Parallel Computing Technologies (PaCT-2011). — Berlin: Springer, 2011. — P. 182–191. — (LNCS; 6873).

[11] Roditty L., Zwick U. Replacement paths and k simple shortest paths in unweighted directed graphs // Proc. Intern. Conf. on Automata, Languages and Programming (ICALP 2005). — Berlin: Springer, 2005. — P. 249–260. — (LNCS; 3580).

[12] Yen J.Y. Finding the k shortest loopless paths in a network // Management Science. — 1971. — Vol. 17, No. 11. — P. 712–716.