Bull. Nov. Comp. Center, Comp. Science, 39 (2016), 37–49
© 2016 NCC Publisher

An associative version of the Ramalingam decremental algorithm for the dynamic all-pairs shortest-path problem

A.S. Nepomniaschaya

Abstract. This paper proposes an efficient parallel representation of the Ramalingam algorithm for the dynamic update of the all-pairs shortest paths of a directed weighted graph after deleting an edge. To this end, a model of associative parallel systems with vertical processing (the STAR-machine) is used. The associative version is given as a group of algorithms that provide an efficient parallel execution on the STAR-machine of different parts of the Ramalingam decremental algorithm. We also present the main advantages of the associative version of the Ramalingam decremental algorithm for the dynamic update of the all-pairs shortest paths. In the Appendix, we propose a special technique being used in the associative parallel algorithm for finding affected vertices after the edge deletion.

1. Introduction

The associative processing is a completely different way of storing, manipulating, and retreiving data as compared to traditional computation techniques. Associative (content addressable) parallel processors of the SIMD type with bit-serial (vertical) processing and simple processing elements (PEs) perform the massively parallel search by contents and use 2D tables as the basic data structure. In particular, such an architecture is best suited for natural and efficient implementation of graph algorithms. In [6], we propose an abstract model of the SIMD type (the STAR-machine) that simulates the run of such systems at the micro level. Associative parallel algorithms are represented as corresponding procedures for the STAR-machine. In [7], we present the basic associative parallel algorithms that are used to design different associative algorithms for different applications. Of special interest is implementation of the dynamic graph algorithms on the STAR-machine. The aim of the dynamic algorithm is to support queries about some properties of a graph and update the solution of a problem after dynamic changes faster than to compute the entire graph from scratch each time with the fastest static algorithm. Let us enumerate a group of dynamic graph algorithms implemented on the STAR-machine. In [8], we propose two associative parallel algorithms for the dynamic edge update of a minimum spanning tree (MST) of an undirected graph. In [9, 10], we propose associative parallel algorithms for the dynamic reconstruction of an MST after deleting and after inserting a new vertex along with its incident edges. In [11], we present associative versions of the Italiano algorithms for the dynamic update of the transitive closure of a directed graph after inserting and after deleting an edge. In [12, 13], we propose associative versions of the Ramalingam algorithms for updating the shortest paths subgraph with a sink after inserting and after deleting an edge.

In this paper, we study the dynamic update of the all-pairs shortest paths (APSP) by means of the STAR-machine. The commonly accepted types of update operations for the APSP problem include insertions and deletions of edges, update operations on edge weights, finding the shortest distance and finding the shortest path between two vertices if any. An algorithm is called *fully dynamic* if the update operations include both insertions and deletions of edges. A partially dynamic algorithm is called *incremental* if it supports only insertions, while it is called *decremental* if only deletions are supported.

In the case of positive edge weights, several solutions have been proposed for the dynamic maintenance of the all-pairs shortest paths. Ausiello et al. [1] propose an efficient solution for the incremental APSP problem assuming that edge weights are restricted in the range of integers $[1, C]$. Chaudhuri and Zaroliagis [2] devise efficient solutions for the APSP 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. King [4] proposes fully dynamic algorithms for updating the all-pairs shortest paths in directed graphs with positive integer weights less than C. Ramalingam [14] proposes fully dynamic algorithms for updating the all-pairs shortest paths in directed graphs with *positive* edge weights. Demetrescu and Italiano [3] devise fully dynamic algorithms for the dynamic maintenance of the all-pairs shortest paths in directed graphs with nonnegative real edge weights. Let us note that the algorithms by Ramalingam [14, 15] are described by means of the output bounded model in which the running time of an algorithm is analyzed in terms of the output change rather than of the input size. Complexity of other above-enumerated algorithms is measured in terms of the amortized time, that is, the total worst-case time over a sequence of operations.

Here, we construct an associative version of the Ramalingam decremental algorithm for the dynamic update of the all-pairs shortest paths in a directed graph G. This algorithm is built as a modification of its previous algorithm for updating the shortest paths subgraph with a sink after deleting an edge from G. The associative version is given as a group of algorithms that provide an efficient parallel execution of different parts of the Ramalingam decremental algorithm on the STAR-machine. We also consider the main advantages of the associative version of the Ramalingam decremental algorithm for updating the all-pairs shortest paths. In Appendix, we propose a special technique that is used in the associative parallel algorithm for finding affected vertices after the edge deletion.

2. Preliminaries

Let $G = (V, E)$ be a *directed weighted graph* with n vertices and m directed edges (arcs). We assume that $V = \{1, 2, \dots, n\}$. Let *wt* denote a function that assigns a weight to every edge.

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 $i = \text{tail}(e)$ and $j = \text{head}(e)$. 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 $\text{wt}(u, v) = \infty$ if $(u, v) \notin E$.

The *shortest path* from u to w is the path of the minimum sum of weights of its edges. Let $\text{dist}(u, z)$ denote the *weight* of the shortest path from vertex u to the distinguished vertex z called *sink*. Let `Dist` be a matrix whose every entry $\text{dist}[i, j]$ represents the weight of the shortest path in G from the vertex i to the vertex j.

By analogy with Ramalingam, we introduce the following notions.

Let an arc (i, j) be deleted from G. Let `AffectedV` denote the set of all vertices u in G such that all paths from u to the selected sink z include the deleted arc (i, j).

A predicate $\text{SP}(a, b, c)$ is defined as follows:

$$\text{SP}(a, b, c) \equiv (\text{dist}(a, c) = \text{wt}(a, b) + \text{dist}(b, c)) \wedge (\text{dist}(a, c) \neq \infty).$$

This predicate verifies whether the arc (a, b) belongs to the shortest path from the vertex a to the selected sink c.

3. The Ramalingam decremental algorithm for updating the all-pairs shortest paths

In [14], Ramalingam represents the problem of the dynamic update of the all-pairs shortest paths after edge deletion using the dynamic single-sink shortest path problem after edge deletion as a subroutine but with a different sink vertex at each call. Following Ramalingam, a vertex y is called *an affected sink* if there exists such a vertex x that $\text{dist}(x, y)$ changes after the edge deletion. As shown in [14], the set of affected sink vertices after deleting an edge $i \to j$ coincides with the set of all affected vertices for the single-source shortest path problem with the source vertex i. Therefore the decremental algorithm for the dynamic update of the all-pairs shortest paths first determines the set of all *affected sink vertices* and then it applies a *modification* of the decremental algorithm for the dynamic update of the single-sink shortest paths for every affected sink vertex.

Let an arc (i, j) be deleted from G, and z be a sink.

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

At the *first stage*, one determines the set `AffectedV` of all affected vertices obtained after deleting the arc (i, j) from G. At the *second stage*, for every affected vertex v_i, one computes a new distance to the sink z.

The first stage is performed as follows. At first, `AffectedV` $= \emptyset$. To construct it, an auxiliary set of vertices `WorkSet` is used. Initially, all arcs (i, x) are analyzed. For every *head* x, one checks whether the predicate $\mathrm{SP}(i, x, z)$ is not performed. If it is true, then `WorkSet` $:= \{i\}$. Vertices in `WorkSet` are sequentially updated. The current updated vertex u is deleted from `WorkSet` and is included into the set `AffectedV`. Then all arcs (x, u) are analyzed. For every *tail* x for which the predicate $\mathrm{SP}(x, u, z)$ is true, one checks whether there does not exist such a non-affected head y of an arc (x, y) for which the predicate $\mathrm{SP}(x, y, z)$ is true. In this case, 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. Initially `PriorityQueue` $= \emptyset$. To build it, for every affected vertex, one first computes a current key. To this end, for every affected vertex k, one defines non-affected heads r of the arcs outgoing from the vertex k for which there is a path to z. Then for every vertex r, one computes the sum $\mathrm{wt}(k, r) + \mathrm{dist}(r, z)$. The current key of k in the heap is defined as the *minimum* value of such sums.

After that, one updates the heap `PriorityQueue` as follows. At every iteration, a vertex with the minimum key in the heap (say a) is deleted from the set `PriorityQueue`. Further all the arcs (c, a) are analyzed. For every tail c, for which $\mathrm{wt}(c, a) + \mathrm{dist}(a, z) < \mathrm{dist}(c, z)$, the current value $\mathrm{dist}(c)$ is equal to $\mathrm{dist}_{\mathrm{new}}(c)$ and this value is a 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 $\mathrm{dist}_{\mathrm{new}}(c)$. The process is completed after updating all vertices in the heap.

A modification of the decremental algorithm for updating distances to the sink z is given as a function DeleteUpdate that returns a set of affected vertices.

Finally, the Ramalingam decremental algorithm for updating the all-pairs shortest paths runs as follows. At first, the arc $i \to j$ is deleted from G. Then by means of the function DeleteUpdate, one defines a set of affected sink vertices. After that, the function DeleteUpdate is applied to every affected sink vertex.

4. The associative version of the decremental algorithm for updating the all-pairs shortest paths

As shown in [14], the decremental algorithm for the dynamic update of the all-pairs shortest paths is based on a modification of the decremental algorithm for updating the single-sink shortest paths. To represent this modification on the STAR-machine, we first propose a special data structure and build associative algorithms to perform the predicates $\mathrm{SP}(u, x, z)$ and $\mathrm{SP}(y, u, z)$ for a given vertex u and a sink z. Then we construct associative algorithms for finding affected vertices and new distances from them to the sink z.

4.1. The data structure. We will use the following data structure:

- an $n \times n$ adjacency matrix `Adj`, whose every ith column saves with bits $'1'$ the heads of arcs outgoing from the vertex i;
- an $n \times n$ adjacency matrix `Adj1`, whose every ith column saves with bits $'1'$ the vertices for which there is the shortest path from the vertex i;
- an $n \times hn$ matrix `Weight` that consists of n fields having h bits each. The weight of an arc (i, j) is written in the jth row of the ith field;
- an $n \times hn$ matrix `Cost` that consists of n fields having h bits each. The weight of an arc (i, j) is written in the ith row of the jth field;
- an $n \times hn$ matrix `Dist` that consists of n fields having h bits each. The distance from the vertex i to the vertex j is written in the jth row of the ith field;
- an $n \times hn$ matrix `Dist1` that consists of n fields having h bits each. The distance from the vertex i to the vertex j is written in the ith row of the jth field;
- a slice `AffectedV` that saves with bits $'1'$ the positions of all affected vertices.

Let us note that the ith field of the matrix `Weight` saves the weights of arcs *outgoing* from the vertex i, while the ith field of the matrix `Cost` saves the weights of arcs *entering* the vertex i. Moreover, every jth row of the matrix `Adj` saves with bits $'1'$ the tails of arcs entering the vertex j, while every jth row of the matrix `Adj1` saves the vertices from which there is the shortest path entering the vertex j.

4.2. Associative algorithms for performing predicates $\mathbf{SP}(u, x, z)$ and $\mathbf{SP}(y, u, z)$. Here, we present two associative algorithms for concurrent execution of predicates both for the heads of arcs outgoing from a given vertex and for the tails of arcs entering it. Observe that these algorithms are used for finding affected vertices.

Let an arc (i, j) be deleted from G. Let z be a sink.

We first propose an associative algorithm (say, Algorithm $A1$) for parallel execution of a group of predicates $\mathrm{SP}(u, x, z)$ for the heads of arcs outgoing from the vertex u. This algorithm uses the matrices `Adj`, `Adj1`, `Weight`, `Dist`, and `Dist1`. It performs the following steps.

Step 1. By means of a slice, say X, save non-affected heads of arcs (u, l) from which there is the shortest path to z.

Step 2. Save the uth field of the matrix `Weight` by means of a matrix, say R_1.

Step 3. Save the zth field of the matrix `Dist1` by means of a matrix, say R_2.

Step 4. Knowing the matrices R_1 and R_2 and the slice X, compute the weights of different paths from the vertex u to z. Save the results in a matrix, say R_3.

Step 5. Knowing the matrix `Dist`, save the distance from u to z by means of a variable, say v.

Step 6. By means of the basic procedure $\mathrm{MATCH}(R_3, v, X, Y)$, define the heads of the arcs outgoing from u for which the corresponding predicates $\mathrm{SP}(u, x, z)$ are true.

On the STAR-machine, the algorithm is given as the procedure Compute-Pred1. It returns a slice to save by bits $'1'$ non-affected heads of the arcs (u, x) for which the corresponding predicates are true.

Now, we present an associative algorithm (say, Algorithm $A2$) for parallel execution of a group of predicates $\mathrm{SP}(x, u, z)$ for the *tails* of arcs entering the vertex u. This algorithm uses the matrices `Cost`, `Adj`, `Dist`, and `Dist1`. It performs the following steps.

Step 1. Save the tails of arcs entering the vertex u in the matrix `Adj` by means of a slice, say Y.

Step 2. Save the uth field of the matrix `Cost` by means of a matrix, say R_1.

Step 3. Knowing the matrix `Dist`, save the distance from u to z by means of a variable, say v.

Step 4. Knowing the matrix R_1 and the variable v, compute the weights of different paths to z starting from the vertices saved in the slice Y. Write the results in a matrix, say R_2.

Step 5. Save the zth field of the matrix `Dist1` by means of a matrix, say R_3.

Step 6. By means of the basic procedure $\mathrm{HIT}(R_2, R_3, Y, Y_1)$, define the tails of arcs entering u for which the corresponding predicates $\mathrm{SP}(x, u, z)$ are true.

On the STAR-machine, the algorithm is given as the procedure ComputePred2. It returns a slice that saves tails of the arcs (x, u) for which the corresponding predicates are true.

4.3. Associative algorithms for finding affected vertices and new distances to a sink. Here, we first provide an associative parallel algorithm (say Algorithm A) for selecting a set of affected vertices obtained after deleting the arc (i, j). This algorithm uses, in particular, the slices `WS` and `AffectedV`. It performs the following steps.

Step 1. Set zeros into the slices `AffectedV` and `WS`. Let a slice Y save the heads of arcs outgoing from the vertex i that start the shortest paths to the sink z.

Step 2. Update the predicates $\mathrm{SP}(i, x, z)$ for the heads of the arcs (i, x) belonging to the slice Y. Let the procedure ComputePred1 return a slice, say Y_1.

Step 3. If $Y_1 \neq \emptyset$, go to exit. Otherwise, include the vertex i into `WS`.

Step 4. 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`;

– update the predicates $\mathrm{SP}(x, k, z)$ for the tails of the arcs (x, k). Let a slice Y_2 save the result of the procedure ComputePred2;

– update every vertex u from the slice Y_2 in the following way. At first, perform the procedure ComputePred1. If it returns an empty slice, insert the vertex u into the slice `WS`.

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

Now we propose an associative parallel algorithm (say Algorithm B) for finding initial distances from all the affected vertices to the sink z. This algorithm uses, in particular, the slice `AffectedV` and the matrices `Weight` and `Dist1`. It constructs a matrix `Queue` whose every ith row saves the initial distance from the affected vertex i to z. The algorithm runs as follows.

While `AffectedV` $\neq \emptyset$, perform the following steps.

Step 1. Select the uppermost vertex, say k, in the slice `AffectedV` and update it as follows.

Step 2. By analogy with Step 1 of Algorithm A, define a slice, say X, to save the non-affected heads of the arcs outgoing from the vertex k that belong to different paths from k to z.

Step 3. Knowing the slice X, the kth field of the matrix `Weight` and the zth field of the matrix `Dist1`, compute in parallel the weights of different paths from k to z and save them in a matrix, say F.

Step 4. Define the initial distance from k to z as the weight of the minimum value in the matrix F. Write this value in the kth row of the matrix `Queue`.

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

Let us consider an associative parallel algorithm (say Algorithm C) for finding a *new* distance from every affected vertex to z. This algorithm uses, in particular, the slice `AffectedV` and the matrices `Queue`, `Cost`, `Dist`, and `Dist1`. It runs as follows.

While `AffectedV` $\neq \emptyset$, perform the following steps.

Step 1. Knowing the current slice `AffectedV`, define the position of the matrix `Queue` row, say k, where a minimum weight γ is written. The new distance from the affected vertex k to z is defined as γ. Write this value both in the matrix `Dist` and in the matrix `Dist1`. Delete the vertex k from the slice `AffectedV`.

Step 2. By means of a slice, say X, save tails of the arcs entering the vertex k. Knowing the kth field of the matrix `Cost` and the new distance from k to z, define in parallel the weights of paths to z each starting with an arc (l, k), where l belongs to the slice X. Save these weights in a matrix, say R_1.

Step 3. Knowing the current affected vertices, find positions of the matrix R_1 rows, where $\mathrm{ROW}(i, R_1) < \mathrm{ROW}(i, \mathtt{Queue})$. Then replace the matrix *Queue* rows with the corresponding rows of the matrix R_1.

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

Remark. The implementation of the algorithms A, B, and C will be given in the full paper. In the Appendix, we propose the implementation of associative algorithms for performing the predicates $SP(u, x, z)$ and $SP(y, u, z)$ on the STAR-machine.

Let us represent the *modification* of the decremental algorithm for updating the single-sink shortest paths on the STAR-machine. It performs in succession the above algorithms A, B, and C. On the STAR-machine, it is given as procedure DeleteUpdate that returns the slice `AffectedV` and the current matrices `Dist` and `Dist1`.

```
procedure DeleteUpdate(i,h,z: integer; Adj,Adj1: table;
          Weight,Cost: table; var Dist,Dist1: table;
          var AffectedV: slice(Adj));
/* Here, h is the number of bits for coding the infinity, z is the sink,
   and (i,j) is the arc to be deleted from G. */
Begin
  FindAffectedVert(i,z,Adj,Adj1,AffectedV);
/* The procedure returns the slice AffectedV that saves affected vertices
   obtained after deleting (i,j) from G. */
  FindCurrentDist(h,z,Adj,Adj1,AffectedV,Weight,Dist1,Queue);
/* The procedure returns the matrix Queue whose every kth row saves
   a current distance from the affected vertex k to z. */
  FindNewDist(h,z,Adj,Adj1,AffectedV,Cost,Queue,Dist,Dist1);
/* The procedure computes a new distance from every affected vertex to z
   and saves it in matrices Dist and Dist1. */
End;
```

The associative version of the Ramalingam decremental algorithm for updating the all-pairs shortest paths runs as follows. At first, the arc $i \to j$ is deleted from G. Then by means of the Algorithm A, one defines the set of *affected sink* vertices. After that the procedure DeleteUpdate is applied to every affected sink vertex.

```
procedure DeleteEdge(i,j,h: integer; Weight,Cost: table;
          var Adj,Adj1: table; var Dist,Dist1: table);
/* The arc (i,j) will be deleted from the graph G, h is the number of bits
   for coding the infinity. */
  var X,AffectedSinks: slice(Adj); z: integer;
Begin X:=COL(i,Adj); X(j):='0'; COL(i,Adj):=X;
/* The arc (i,j) is deleted from G. */
  FindAffectedVert(i,j,Adj,Adj1,AffectedV);
  AffectedSinks:=AffectedV;
  while SOME(AffectedSinks) do.
    begin z:=STEP(AffectedSinks);
      DeleteUpdate(i,h,z,Adj,Adj1,Weight,Cost,Dist,
                   Dist1,AffectedV);
    end;
End;
```

Let us enumerate the main advantages of the associative version of the Ramalingam decremental algorithm for updating the all-pairs shortest paths:

1. For every affected vertex u and the sink z, the associative version *simultaneously* computes the predicates $\mathrm{SP}(u, x, z)$ for all heads of arcs outgoing from the vertex u and the predicates $\mathrm{SP}(y, u, z)$ for all tails of arcs entering the vertex u.
2. For every affected vertex u and the sink z, the associative version *simultaneously* determines the weights of paths from u to z and writes the minimum value of these weights in the corresponding row of the matrix `Queue`.

3. After selecting a new distance from the current affected vertex k to z, the associative version *simultaneously* computes the weights of paths to z each starting from the arc entering the vertex k and saves them in a matrix, say R_1. After that one *simultaneously* replaces the matrix `Queue` rows, where $\text{ROW}(i, R_1) < \text{ROW}(i, \texttt{Queue})$, with the corresponding rows of the matrix R_1.

5. Conclusions

We have proposed a parallel representation of the Ramalingam decremental algorithm for updating the all-pairs shortest paths on the STAR-machine having no less than n PEs. We have also selected the main advantages of the associative version of the Ramalingam decremental algorithm. In the Appendix, we have presented a special technique that allows one to execute in parallel some parts of the Ramalingam decremental algorithm.

In the full paper, we will present the implementation on the STAR-machine of the associative version of the Ramalingam decremental algorithm for updating the all-pairs shortest paths.

We are planning to design an associative version of the Ramalingam incremental algorithm for updating the all-pairs shortest paths.

References

[1] Ausiello G., Italiano G.F., Marchetti-Spaccamela A., Nanni U. Incremental algorithms for minimal length paths // Journal of Algorithms. — 1991. — Vol. 12, No. 4. — P. 615–638.

[2] Chaudhuri S., Zaroliagis C.D. Shortest path queries in digraphs of small treewidth // Proc. Int. Colloquium on Automata Languages, and Programming. Szeged, Hunhary, July 10–14, 1995. — Springer, 1995. — P. 244–255. — (Lect. Notes Comp. Sci.; 944)

[3] Demetrescu C., Italiano G.F. Fully dynamic all-pairs shortest paths with real edge weights // Proc. 42nd IEEE Annual symposium on foundations of computer science (FOCS'01), Las Vegas, Nevada. — 2001. — P. 260–267.

[4] King V. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs // Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99). — 1999. — P. 81–99.

[5] Klein P.N., Rao S., Rauch M., Subramanian S. Faster shortest path algorithms for planar graphs // Proc. ACM Symposium on Theory of Computing. Montreal, Quebec, Canada, May 23–25. — 1994. — P. 27–37.

[6] 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.

[7] Nepomniaschaya A.S. Basic associative parallel algorithms for vertical processing systems // Bull. Novosibirsk Comp. Center. Ser. Computer Science.— Novosibirsk, 2009.—IIS Special Iss. 29.—P. 63–77.

[8] Nepomniaschaya A.S. Associative parallel algorithms for dynamic edge update of minimum spanning trees // Proc. 7th Int. Conf. PaCT 2003.—Springer, 2003.—P. 141–150.—(Lect. Notes Comp. Sci.; 2763).

[9] Nepomniaschaya A.S. Associative parallel algorithm for dynamic reconstructing a minimum spanning tree after deletion of a vertex. // Proc. 8th Int. Conf. PaCT, 2005.—Springer, 2005.—P. 151–173.—(Lect. Notes Comp. Sci.; 3606).

[10] Nepomniaschaya A.S. Associative parallel algorithm for the dynamic update of a minimum spanning tree after insertion of a new vertex // Cybernetics and System Analysis.—Kiev: Naukova Dumka, 2006.—No. 1.—P. 19–31 (In Russian). (English translation by Plenum Press).

[11] Nepomniaschaya A.S. Efficient implementation of the Italiano algorithms for updating the transitive closure on associative parallel processors // Fundamenta Informaticae.—IOS Press, 2008.—Vol. 89.—No. 2–3.—P. 313–329.

[12] Nepomniaschaya A.S. Associative version of the Ramalingam decremental algorithm for dynamic updating the single-sink shortest paths subgraph // Proc. 10th Int. Conf. on Parallel Computing Technologies, PaCT-2009, Novosibirsk, Russia.—Springer, 2009.—P. 257–268.—(Lect. Notes Comp. Sci.; 5698).

[13] Nepomniaschaya A.S. Associative version of the Ramalingam algorithm for the dynamic update of the shortest paths subgraph after inserting a new edge // Cybernetics and System Analysis.—Kiev: Naukova Dumka, 2012.—No. 3.— P. 45–57 (In Russian). (English translation by Springer).

[14] Ramalingam G. Bounded Incremental Computation.—Springer, 1996.—(Lect. Notes Comp. Sci.; 1089).

Appendix

We provide two auxiliary procedures ComputePred1 and ComputePred2 that are used for finding affected vertices after the edge deletion from G.

The procedure ComputePred1 simultaneously defines the non-affected *heads* of arcs outgoing from a given vertex u for which the predicates $\mathrm{SP}(u,x,z)$ are true. It uses, in particular, the matrices `Adj`, `Adj1`, `Weight`, `Dist`, and `Dist1`. It returns a slice to save by bits $'1'$ the above-mentioned heads of arcs.

Let us briefly explain the main idea of this procedure. We first define positions of arcs (u,x) that belong to different paths from u to z. Then we define the weights of these paths. After selecting the distance from u to z, we save positions of those arcs that belong to the shortest path from u to z.

```
procedure ComputePred1(h,u,z:  integer; AffectedV: slice(Adj);
          Adj,Adj1: table; Weight,Dist,Dist1: table;
          var Y: slice(Adj));
  var l1,l2: integer; X,X1,X2: slice(Adj); R1,R2,R3: table;
      v: word(Dist); v1: word(Adj); w1: word(R1);
Begin X1:=COL(u,Adj);
/* The slice $X1$ saves the heads of arcs outgoing from $u$. */
  v1:=ROW(z,Adj1); X2:=CONVERT(v1);
/* The slice $X2$ saves the vertices from which there is the shortest path to $z$. */
  X:=X1 and X2;
  X:=X and (not AffectedV);
/* The slice $X$ saves non-affected heads of arcs outgoing from $u$ that belong to
   different paths from $u$ to $z$. */
  TCOPY1(Weight,u,h,R1);
  TCOPY1(Dist1,z,h,R2);
  ADDV(R1,R2,X.R3);
/* The matrix $R3$ saves the weights of different paths from $u$ to $z$. */
  v1:=ROW(z,Dist1);
  l1:=1+(u-1)h; l2:=uh;
  w1:=TRIM(l1,l2,v1);
/* The row $w1$ saves the distance from $u$ to $z$. */
  MATCH(R3,w1,X,Y);
/* The slice $Y$ saves positions of the matrix $R3$ rows that coincide with
   the distance from $u$ to $z$. */
End;
```

The procedure ComputePred2 simultaneously defines the *tails* of arcs entering a given vertex u for which the predicates $\mathrm{SP}(y, u, z)$ are true. It uses, in particular, the matrices `Adj`, `Adj1`, `Weight`, `Dist`, and `Dist1`. It returns a slice to save by bits $'1'$ the mentioned above tails of arcs.

Explain briefly the main idea of this procedure. We first define the weights of arcs entering the vertex u. After selecting the distance from u to z, we define the weights of different paths to z, that start from the tails of arcs entering u. Then we select the different distances to z starting from the tails of arcs entering u. Finally, we compare these distances with the weights of the corresponding paths to z.

```
procedure ComputePred2(h,u,z: integer; Adj: table;
          Cost,Dist,Dist1: table; var Y: slice(Adj));
  var l1,l2: integer; X: slice(Adj); R1,R2,R3: table;
      v: word(Dist); v1: word(Adj); v2: word(R1);
Begin v1:=ROW(u,Adj); X:=CONVERT(v1);
```

```
/* The slice X saves the tails of arcs entering u. */
  TCOPY1(Cost,u,h,R1);
/* The matrix R1 saves the weights of arcs entering u. */
  v:=ROW(z,Dist);
  l1:=1+(u-1)h; l2:=uh;
  v2:=TRIM(l1,l2,v);
/* The row v2 saves the distance from u to z. */
  ADDC(R1,v2,X,R2);
/* The matrix R2 saves the weights of different paths to z starting from
   the tails of arcs entering u. */
  TCOPY1(Dist1,z,h,R3);
/* The matrix R3 saves different distances to z. */
  HIT(R2,R3,X,Y);
/* The slice Y saves positions of the matrix R2 rows that coincide with
   the corresponding distances to z. */
End;
```

The correctness of these procedures is proved by contradiction. Let us note that these procedures take $O(h)$ time each because they apply the basic procedures [7].