Joint NCC & IIS Bull., Comp. Science, 7 (1997), 21–33

Associative algorithms for graphs represented as an adjacency matrix*

A. S. Nepomniaschaya, O. V. Taborskaya

In this paper by means of an abstract model (the STAR–machine) we study a group of associative algorithms for unweighted graphs given as an adjacency matrix. These algorithms are written as the corresponding STAR procedures whose correctness is proved and time complexity is evaluated.

1. Introduction

The revived interest in the associative (or content-addressable) architecture results from remarkable advances in the VLSI technology [1]. A class of associative parallel processors belonging to the fine-grained SIMD systems with bit-serial (or vertical) processing and simple single-bit processing elements (PEs) is of special interest. This class of parallel computers includes the well-known systems STARAN, DAP, MPP and Connection Machine (CM). In these systems input data are physically loaded in a matrix memory in such a way that each data item occupies an individual row and is processed by its own processing element. These systems provide a massively parallel search by contents and processing of unordered data [2, 3]. Such an architecture is primarily oriented to solving the non-numerical problems.

Our prime interest is in application of associative systems with vertical processing to solving graph theoretical problems. In [3–8], the problem of finding a minimal spanning tree is studied for different graph representation forms using different algorithms and different formal models. In [4] problems of finding connected components, transitive closure of a graph, verifying an articulation point and verifying a bridge are considered on the orthogonal machine for unweighted graphs represented as an adjacency matrix. In [8] the same problems are studied for undirected weighted graphs represented on the STAR-machine as a list of triples (edge vertices and the weight). In [5] algorithms for the solution of two shortest path problems are considered on the associative array processor LUCAS.

In this paper, we focus our primary attention on a group of problems using the graph representation as an adjacency matrix. It includes finding the transitive closure both for undirected and directed graphs, and the following

*Partially supported by the Russian Foundation for Basic Research under Grant 96-01-01704.

problems for undirected graphs: finding the shortest path between two given vertices, finding a connected component, verifying a bridge and an articulation point. We solve these problems on the STAR-machine by performing comparatively simple manipulations of the adjacency matrix. Simplicity is the main property of any algorithm intended for a simple realization on a vertical processing system. The algorithms are represented as corresponding procedures written in the language STAR being an extension of Pascal. It is shown that on the STAR-machine the Warshall algorithm for finding the transitive closure of a directed graph takes $0(n^2)$ time, while on conventional sequential computers it takes $0(n^3)$ time, where n is the number of graph vertices. We also reveal that on the STAR-machine the Dijkstra algorithm of finding the shortest path between two vertices takes $0(n)$ time, while on sequential computers it takes $0(n^2)$ time and on the associative array processor LUCAS it requires $0(ln)$, where l is the shortest path length. We show that on the STAR-machine any of the other problems considered requires $0(n)$ time, while on sequential computers it takes $0(n + m)$ time, where m is the number of graph edges [9].

Hence, the use of vertical processing systems for solving the above-mentioned problems gives an evident time improvement over the sequential computers.

2. Model of associative parallel processing

In the paper the model is defined as an abstract STAR-machine of the SIMD type with vertical data processing. It consists of the following components:

- a sequential control unit where programs and scalar constants are stored;
- an associative processing unit consisting of m single–bit PEs;
- a matrix memory for the associative processing unit.

The matrix memory consists of cells each storing one bit. Input binary data are loaded in the matrix memory in the form of two-dimensional tables in which each data item occupies an individual row, and it is processed by a dedicated processing element. Both a row and a column can be easily accessed.

The associative processing unit is represented as h vertical registers each consisting of m bits. Vertical registers can be regarded as a one-column array. The STAR-machine runs as follows. The bit columns of the tabular data are stored in the registers which perform the necessary Boolean operations and record the search results.

Let us briefly consider the STAR constructions described in [10] and needed in the sequel. To simulate data processing in the matrix memory

new data types **word, slice** and **table** are introduced in the same manner as in Pascal. The types **slice** and **word** are used for bit column access and 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 m components which belong to $\{0,1\}$[1].

Consider operations, predicates and functions for slices.

Let X, Y be variables of the type **slice** and i, j be variables of the type **integer**. We define the following operations:

SET(Y)	sets all components of Y to **1**;
CLR(Y)	sets all components of Y to **0**;
$Y(i)$	selects the i-th component of Y;
FND(Y)	returns the ordinal number i of the first component **1** of Y, $i \geq 0$;
STEP(Y)	returns the same result as FND(Y) and then resets the first component **1**;
NUMB(Y)	returns the number i of components **1** of Y, $i \geq 0$;

In the usual way we introduce the predicates ZERO(Y) and SOME(Y) and the following bitwise Boolean operations: X *and* Y is conjunction, X *or* Y is disjunction, *not* Y is negation, X *xor* Y is exclusive *'or'*.

Recall that all the operations for the type **slice** are also performed for the type **word**.

For a variable T of the type **table** we employ the following two operations:

ROW(i, T)	returns the i-th row of the matrix T $(1 \leq i \leq m)$;
COL(i, T)	returns the i-th column of the matrix T.

Remark 1. Note that STAR statements resemble those of Pascal.

Remark 2. When we define a variable of the type **slice** we put in brackets the name of the matrix which uses it. Therefore if the matrix consists of n rows, where $n < m$, then only the first n components of the corresponding slice will be used in the vertical processing.

3. Preliminary notions

Let us recall some notions used in the paper.

Let $G = (V, E)$ be a graph with the vertex set $V = \{1, 2, \ldots, n\}$ and the edge set $E \subseteq V \times V$.

At first, consider some definitions for undirected graphs.

A path in G from a vertex i to a vertex j is a sequence of the edges $(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$, where $i_1 = i$, $i_n = j$ and $n \geq 2$.

[1] For simplicity let us call *slice* any variable of the type **slice**.

The **shortest path** is a minimum-length path from a vertex i to a vertex j.

A **connected component** is a maximal connected subgraph.

An **articulation point** (respectively, a **bridge**) is a vertex (respectively, an edge) whose deletion from the graph increases the number of its connected components.

Following [9], the **transitive closure** of an undirected graph G is defined as a graph G^* whose any vertex is a connected component of G.

Now, recall some definitions for directed graphs.

An **adjacency matrix** $A = [a_{ij}]$ of a directed graph is the $n \times n$ Boolean matrix in which $a_{ij} = 1$, if in the set E there is an arc from the vertex i to the vertex j, otherwise $a_{ij} = 0$.

A **path matrix** $P = [p_{ij}]$ of a directed graph is the $n \times n$ Boolean matrix in which $p_{ij} = 1$ if there exists a path from the vertex i to the vertex j, otherwise $p_{ij} = 0$, where $i, j \in V$.

The **transitive closure** of a directed graph G is a graph G^* consisting of all the arcs of G together with all the arcs of the form (i, k) such that there is a path of positive length from the vertex i to the vertex k.

Following [4], assume that any elementary operation of the STAR-machine needs one unit of time. Therefore, we will measure **time complexity** of an algorithm by counting all the elementary operations performed in the worst case.

4. Transitive closure of a directed graph

Here, by means of the STAR-machine we analyze the Warshall algorithm [11] for determining the existence of a path between all the pairs of vertices of a graph. For a given adjacency matrix A of a directed graph $G = (V, E)$ it computes the path matrix P for the transitive closure G^*.

Note that in the STAR-machine matrix memory any graph is directly mapped onto the two-dimensional table.

Denote by $a_{i,*}$ the i-th row in the matrix A. Following [12], we will write the Warshall algorithm:

$$a_{i,*}^{(k)} = a_{i,*}^{(k-1)}, \qquad \text{if} \quad a_{i,k}^{(k-1)} = 0,$$

$$a_{i,*}^{(k)} = a_{i,*}^{(k-1)} \vee a_{k,*}^{(k-1)}, \qquad \text{if} \quad a_{i,k}^{(k-1)} = 1.$$

This algorithm runs as follows. It scans down a k-th column, and when it finds a bit **1** in a position i it performs the disjunction between the i-th and the k-th rows of the matrix $A^{(k-1)}$ and writes the result in the i-th row. It is assumed that $A^{(0)} = A$.

For representing the Warshall algorithm on the STAR-machine we will employ a slice X for storing the k-th processing column of the current matrix $A^{(k-1)}$, an integer i for saving the position of the first bit **1** in X and two variables v, w for storing the i-th and the k-th rows of $A^{(k-1)}$.

Now, let us consider the following procedure.

```
proc WARSHALL(n: integer; var P: table);
/*Here n is the number of graph vertices. */
var X: slice(P); v,w: word; i,k: integer;
begin for k := 1 to n do
  begin X :=COL(k,P); w :=ROW(k,P);
    while SOME(X) do
      begin i :=STEP(X); v :=ROW(i,P);
        v := v or w; ROW(i,P) := v
      end;
  end;
end;
```

Claim 1. *Let G be a given directed graph with n vertices and A be its adjacency matrix. Then the procedure WARSHALL(n, A) returns the matrix $A^{(n)}$ being the path matrix for the transitive closure of G.*

For proving the claim it is sufficient to verify that at any k-th iteration, $(1 \leq k \leq n)$, the procedure WARSHALL(n, A) sequentially processes only the rows of the current matrix $A^{(k-1)}$ having bit **1** in the k-th column of $A^{(k-1)}$. This can be easily verified by induction on k.

It is obvious that the procedure WARSHALL takes $0(n^2)$ time for a graph with n vertices. More precisely, it takes time which is proportional to the sum of the number of bits **1** in the slice X at any external loop.

In [12] Warren presents a modification of the Warshall algorithm to scan by rows. This algorithm executes faster for sparse matrices on most sequential computers[1], particularly, in a paging environment. However, on associative parallel processors the Warshall algorithm can be also efficiently applied to sparse matrices since it processes only non-empty columns.

5. Finding the shortest path between two vertices

Let G be a connected, undirected, unweighted graph. Without loss of generality one assumes that the weight of any edge is equal to one. In [5], for finding the shortest path from the source vertex s to the final vertex f the following special case of the Dijkstra algorithm [13] is applied. Beginning with the vertex s one constructs a subgraph by viewing the initial graph

[1] Entries of a sparse matrix are mostly zeros.

in width. At any i-th step one stores a set S_i of those vertices which are reachable from the vertex s in i steps, each time verifying whether the vertex f belongs to S_i. The process is finished as soon as the vertex f belongs to some set S_j. It is obvious that such a step j always exists and the length of the shortest path $P_{s,f}$ from the vertex s to the vertex f is equal to j. To restore this path it is necessary to construct a path from f to s employing the sets $S_{j-1}, S_{j-2}, \ldots, S_2$.

On the associative array processor LUCAS the corresponding procedure is represented by means of a high-level microprogramming language because it is impossible to write this in Pascal/L. This procedure updates the adjacency matrix G in turn by columns (in odd steps) and by rows (in even steps) taking $0(ln)$ time, where l is the shortest path length.

By means of the STAR-machine we present the procedure ShortPath which takes $0(n)$ time. To this end we improve the implementation of the Dijkstra algorithm as follows. At any i-th step we store a set S_i' consisting of those *new* vertices which are reachable for the first time from the source vertex s in i steps. Such a simplification allows us to decrease the number of iterations.

Before considering the procedure ShortPath let us explain the use of the variables A, X, Y, Z of the type **slice** and the variable R of the type **table**.

At *the first stage* in the slice A we accumulate positions of all the vertices which are reachable from s in j steps. The slice Z is used for storing positions of new vertices which are reachable from the vertex s in the j-th step. For any vertex whose position is indicated by **1** in the slice Z we accumulate positions of vertices being reachable in one step by means of the slice Y. We use the slice X for selecting any current column of the matrix G. In any i-th column of the matrix R we store positions of vertices belonging to the set S_i'.

At *the second stage* the slice Z is used for selecting the current j-th column of the matrix R, and the slice X is employed for selecting that column of the matrix G which corresponds to the current vertex included in the result slice.

Consider the procedure ShortPath.

```
proc ShortPath(G: table; s, f: integer;
  var l: integer; B: slice(G));
/* Here G is the adjacency matrix, s is the source vertex
  and f is the final vertex. */
var A, X, Y, Z: slice(G); i, j: integer; R: table;
/* The first stage. Finding the shortest path length. */
begin j := 0; CLR(A); CLR(B); CLR(Z);
  A(s) := 1; Z(s) := 1; B(s) := 1; B(f) := 1;
  while Z(f) =' 0' do
```

```
    begin CLR(Y);
      while SOME(Z) do
        begin i :=STEP(Z); X :=COL(i, G); Y := Y or X
        end;
          Z := Y and(not A); j := j + 1;
/* In the slice Z we store positions of those new vertices from
   the slice Y which do not belong to the slice A. */
          COL(j, R) := Z; A := A or Y
/* In the slice A we accumulate positions of the vertices which
   are reachable from the vertex s in j steps. */
    end;
      l := j;
/* The second stage. Finding the shortest path. */
      X :=COL(f, G);
      while j > 1 do
        begin j := j - 1; Z :=COL(j, R); Z := Z and X;
          i :=FND(Z); B(i) := 1; X :=COL(i, G)
        end;
end;
```

Claim 2. *Let an undirected graph G be given as an adjacency matrix. Let s and f be two graph vertices. Then the procedure ShortPath (G, s, f, l, B) returns the length l of the shortest path between s and f, and the slice B in which we indicate by* **1** *positions of vertices included in the shortest path.*

Sketch of the proof. The claim is proved by induction on the length k of the shortest path. Basis is immediately verified for $k = 1$. For proving the step of induction it is necessary to show that at the second stage the intersection of the f-th column of the matrix G (the slice X) and the k-th column of the matrix R (the slice Z) is non-empty. It means that one can define the position of the next to the last vertex belonging to the shortest path between s and f. In fact, since the position of the vertex f is indicated by **1** in the $(k+1)$-th column of R, there exists such a vertex q which forms an edge with the vertex f and whose position is indicated by **1** in Z. Clearly, the vertex q appears at the k-th step for the first time. On the other hand, in the slice X positions of all the vertices which form an edge with f are indicated by **1**. Hence, there is a position of the vertex q among them. Therefore, intersection of the slices X and Z is non-empty. The positions of other vertices belonging to the shortest path are obtained by inductive assumption. □

Let us evaluate time complexity. The first stage requires at most $0(n)$ time, since in the loop **while** SOME(Z) **do** we analyze only different vertices which are reachable from the vertex s. The second stage takes $0(l)$ time. Hence, the procedure ShortPath takes $0(n)$ time.

Remark 3. It should be emphasized that the STAR procedure ShortPath returns both the shortest path between two vertices and its length. Moreover, owing to the vertical processing the shortest path is represented in a natural way by indicating positions of vertices which belong to it.

6. Algorithms based on selection of connected components

Here, at first we present an algorithm for finding a connected component. Then, we consider associative algorithms for finding the transitive closure of an undirected graph and for verifying a bridge and an articulation point.

6.1. Finding connected components

To find a connected component including a given vertex v we employ the algorithm from [14]. The idea underlying this algorithm is as follows. At first, we generate a set S_1 of all the vertices which form an edge with the vertex v. Among the vertices not included in S_1 we make up a set S_2 consisting of those vertices any of which forms an edge with a vertex from S_1. We continue the process until an empty set S_n is obtained.

Consider the following procedure.

```
proc COMP (G: table; v: integer; var R: slice(G));
/* In the slice R we indicate by 1 positions of vertices which
   belong to a connected component generated from the vertex v. */
var X, Y, Z: slice(G); i: integer;
begin CLR(Z); CLR(R); R(v) := 1; Z(v) := 1;
  while SOME(Z) do
    begin CLR(Y);
      while SOME(Z) do
        begin i :=STEP(Z); X :=COL(i, G); Y := Y or X
        end;
/* In the slice Y we accumulate positions of vertices which
   are reachable at one step in the current iteration. */
        Z := Y and (not R); R := R or Z
/* In the slice Z we save positions of those new vertices
   from Y which do not belong to the slice R. */
    end;
end;
```

Claim 3. *Let an undirected graph G be given as an adjacency matrix and let v be a selected vertex. Then the procedure COMP(G, v, R) returns the slice R in which we indicate by* **1** *positions of vertices belonging to the connected component generated from v.*

Proof. We will prove the claim by induction on the number of steps required for generating a connected component from v.

Basis is immediately verified.

Step of induction. Assume the assumption is true when a connected component has at least one vertex which is reachable from v in k steps, where $k \geq 1$. We will prove the claim for the case when in G there is at least one vertex reachable from v in $k+1$ steps. In view of inductive assumption after performing the first k iterations positions of vertices, which are reachable from v in k steps, are indicated by **1** in the slice R. It is immediately verified that after executing the $(k+1)$-th iteration the positions of vertices reachable from v in $k+1$ steps are indicated by **1** in the slice Z, and these positions are added to the slice R by using the statement $R := R\ or\ Z$. Since there is at least one component **1** in Z, we perform the external loop **while** SOME(Z) **do**. After executing the internal loop **while** SOME(Z) **do** none of new vertices will be added to the slice R, because in G there is no any vertex reachable from v in $(k+2)$ steps. Hence, as a result of performing the statement $Z := Y\ and\ (not\ R)$ the slice Z will consist of components **0** and we jump to the procedure end.

This completes the proof. □

Remark 4. In [4], another procedure is presented for solving the same problem. Unlike it, the considered STAR procedure COMP is based on Tutte's serial algorithm from [14]. This permits one to perform the vertical processing in a natural and robust way.

6.2. Transitive closure of an undirected graph

For finding the transitive closure of an undirected graph G it is necessary to determine all its connected components. To this end, using the procedure COMP, we sequentially select each connected component in G and store it in the current column of the result matrix F.

Consider the following procedure.

```
proc TRANS(G: table; var F: table);
var X,S: slice(G); i,j: integer;
begin j := 0; SET(X);
  while SOME(X) do
    begin i := FND(X); COMP(G,i,S);
      j := j + 1; COL(j,F):= S;
 * The current connected component is written into the j-th column
 of the matrix F. */
      X := X and (not S)
 * We delete from the slice X positions of vertices included into
 the current connected component. /*
```

```
    end;
end;
```

Claim 4. *Let an undirected graph G be given as an adjacency matrix. Then the procedure TRANS(G, F) returns the matrix F in any column of which we indicate by* **1** *positions of vertices belonging to the same connected component.*

This claim is verified by induction on the number of connected components.

6.3. Verifying a bridge and an articulation point

To verify whether a given edge (i, j) is a bridge in a given undirected graph G, we employ the following simple algorithm. We delete the occurrence of this edge from the graph G. Then we generate a connected component beginning with the vertex i. We conclude that the edge (i, j) is a bridge if the vertex j does not belong to this connected component.

Consider the following procedure.

```
proc BRIDGE(G: table; i, j: integer; var B: boolean);
/* The Boolean variable B takes the value true if the edge (i, j)
   is a bridge in the graph G. */
var X, S: slice(G);
begin X := COL(i, G); X(j) := 0; COL(i, G) := X;
   X := COL(j, G); X(i) := 0; COL(j, G) := X;
/* The occurrence of the edge (i, j) is deleted from the graph G. */
   COMP(G, i, S);
/* The connected component is generated beginning
   with the i-th vertex . */
   if S(j) = 1 then B := false else B := true
end;
```

The following claim is easily verified.

Claim 5. *Let an undirected graph G be given as an adjacency matrix and let (i, j) be its edge. Then the procedure BRIDGE(G, i, j, B) returns the value* **true** *if and only if the edge (i, j) is a bridge in G.*

To verify whether a given vertex i is an articulation point in the given undirected connected graph G, we check whether the deletion of the vertex i from G causes the appearance of a new connected component.

Consider the following procedure.

```
proc ARTIC(G: table; i, n: integer; var R: boolean);
  /* Here n is the number of graph vertices and i is a selected vertex. */
  var S, X : slice; w : word; k: integer;
  begin CLR(X); CLR(w); COL(i, G) := X; ROW(i, G) := w;
```

```
/* The occurrence of the vertex v is deleted from G. */
  if i < n then k := i + 1 else k := i − 1;
/* We define the number of the source vertex for generating
  a connected component. */
    COMP(G, k, S); X := not S;
      if NUMB(X) > 1 then R := true else R := false
end;
```

Claim 6. *Let an undirected connected graph G, having n vertices, be given as an adjacency matrix and let i be its selected vertex. Then the procedure ARTIC(G, i, n, B) returns the value* **true** *if and only if the vertex i is an articulation point in G.*

This claim is directly verified.

Remark 5. In the general case it is necessary to determine whether the number of connected components of G will be increased after deleting the occurrence of the vertex i from G. It can be also easily done.

Now, let us evaluate time complexity of the procedure COMP. One can immediately verify that it requires $0(n)$ time, where n is the number of graph vertices. It is obvious that any of the procedures TRANS, BRIDGE and ARTIC takes the same time. Note that on sequential computers each of them takes $0(n + m)$ time, where m is the number of graph edges [9].

7. Conclusions

In this paper for the unweighted graphs represented as an adjacency matrix by means of the STAR-machine we have studied a group of associative algorithms employing the vertical processing. It includes finding the transitive closure both for undirected and for directed graphs, and the following problems for undirected graphs: finding the shortest path between two given vertices, finding a connected component, verifying a bridge and an articulation point. On the one hand, these problems have been considered because their solutions can be obtained by rather simple manipulations of the adjacency matrix. On the other hand, the corresponding procedures can be represented on the STAR-machine in a natural and robust way. We have shown that both Warshall's serial algorithm for finding the transitive closure of a directed graph and Tutte's serial algorithm for finding a connected component are efficiently implemented on the STAR-machine without any modification. It has been obtained that the Warshall algorithm takes $0(n^2)$ time on the STAR-machine having n processing elements, while on sequential computers it requires $0(n^3)$ time. It has been also shown that on the STAR-machine the Dijkstra algorithm for finding the shortest path between

two vertices takes $0(n)$ time, while on sequential computers it takes $0(n^2)$ time and on the associative array processor LUCAS it requires $0(ln)$, where l is is the shortest path length. We have obtained that on the STAR-machine any of other considered problems requires $0(n)$ time, while on sequential computers it takes $0(n+m)$ time, where m is the number of graph edges [9].

Hence, for a group of problems using the graph representation as an adjacency matrix we have studied the efficient implementation on vertical processing systems in a natural way.

Acknowledgments

We would like to thank Prof. V. A. Evstigneev for useful comments.

References

[1] K. E. Grosspietsch, *Associative processors and memories: A survey*, IEEE, Micro, June, 1992, 12–19.

[2] J. Potter, J. Baker, A. Bansal, S.Scott, C. Leangsuksun, C. Asthagiri, *ASC – an associative computing paradigm*, Computer: Special Issue on Associative Processing, **27**, No. 11, 1994, 19–24.

[3] J. L. Potter, *Associative Computing: A Programming Paradigm for Massively Parallel Computers*, Kent State University, Plenum Press, New York and London, 1992.

[4] B. Otrubova, O. Sykora, *Orthogonal computer and its application to some graph problems*, Parcella'86, Berlin, Academie Verlag, 1986, 259–266.

[5] C. Fernstrom, J. Kruzela, B. Svensson, *LUCAS Associative Array Processor. Design, Programming and Application Studies*, Lecture Notes in Computer Science, Berlin: Springer–Verlag, **216**, 1986.

[6] A. S. Nepomniaschaya, *Comparison of two MST algorithms for associative parallel processors*, Proc. of the 3-d Intern. Conf. "Parallel Computing Technologies", PaCT–95, St. Petersburg, Russia, Lecture Notes in Computer Science, **964**, 1995, 85–93.

[7] A.S. Nepomniaschaya, *Representations of the Prim–Dijkstra algorithm on associative parallel processors*, Proc. of VII Intern. Workshop on Parallel Processing by Cellular Automata and Arrays, Parcella'96, Academie Verlag, Berlin, 1996, 184–194.

[8] A.S. Nepomniaschaya, *An associative version of the Prim–Dijkstra algorithm and its application to some graph problems*, Proc. of Andrei Ershov Second Intern. Memorial Conf. "Perspectives of System Informatics", Lecture Notes in Computer Science, Berlin: Springer–Verlag, **1181**, 1996, 203–213.

[9] E.M. Reingold, J. Nievergelt, M. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice–Hall, Inc., Englewood Cliffs, New Jersey, 1977.

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

[11] S. Warshall, *A theorem on Boolean martices*, J. ACM, **9**, No. 1, 1962, 11–12.

[12] H.S. Warren, *A modification of Warshall's algorithm for the transitive closure of binary relations*, Comm. ACM, **18**, No. 4, 1975, 218–220.

[13] E.W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Math., **1**, 1959, 269–271.

[14] W.T. Tutte, *Graph Theory*, Addison–Wesley, 1984.