A fully-pipelined systolic algorithm for finding bridges on an undirected connected graph

Su-Chu Hsu, Hsien-Fen Hsieh and Shing-Tsaan Huang

Institute of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC

Received 7 January 1991
Revised 13 August 1991

Abstract

Hsu, S.-C., H.-F. Hsieh and S.-T. Huang, A fully pipelined systolic algorithm for finding bridges on an undirected connected graph, Parallel Computing 18 (1992) 377–391.

A new fully-pipelined systolic algorithm for finding all the bridges of an undirected connected graph is proposed. Given a graph of n vertices and m edges, the proposed algorithm uses $(2n-2)$ systolic cells and runs in $(m+3n-3)$ systolic cycles. This improves a previous result. The use of fully-pipelined cells and the uniformity of the operations in each cell make the proposed algorithm distinctive.

Keywords. Undirected connected graph; spanning trees; bridges; systolic algorithm.

1. Introduction

This paper proposes a fully-pipelined systolic algorithm for finding all the bridges of an undirected connected graph. A *bridge* of such a graph is an edge whose removal disconnects the graph. Let $G=(V, E)$ be an undirected connected graph with $|V|=n$ and $|E|=m$. An *ordered spanning tree* of G according to an edge sequence can be constructed as follows. Let us maintain a graph G' initially with a vertex set V and an empty edge set. Then, attach each edge e of E one by one to G' according to the edge sequence. If a cycle is formed in G' because of the attachment of e, edge e is discarded. Otherwise, edge e is appended to G'. After all the edges of E are considered, the resulting G' will be an ordered spanning tree of G. An ordered spanning tree constructed by this way depends very much on the edge sequence considered. For example, the first edge in the edge sequence is always one of the edges of the spanning tree. Such a construction is used in our algorithm to determine whether an edge is a bridge or not.

Two lemmas are derived in this paper. Lemma 1 gives a way to test whether an edge e is a bridge or not. First, we construct an ordered spanning tree from an edge sequence in which edge e is considered last. If edge e is the last edge of the ordered spanning tree, then edge e

All correspondence should be addressed to: Prof. Shing-Tsaan Huang, Institute of Computer Science, National Tsing Hua University, Hsinchu 30043, Taiwan, ROC

This research was supported by National Science Council of the Republic of China under Contract NSC79-0408-E007-03.

is a bridge. Otherwise, edge e is not a bridge. Let $T=(V, E_T)$ be an arbitrary spanning tree of G. It is found that all the bridges are edges of T. By the observation, we simply test whether the edges of T are bridges rather than test all the edges of G. By Lemma 1, we only need to construct $n-1$ ordered spanning trees, each with one of the edges of T being considered last, and then we check whether the last considered edge is on the ordered spanning tree. Lemma 2 gives an efficient way to construct these $(n-1)$ trees one after another and each can be constructed from the previous one.

The proposed algorithm is based on the two lemmas. These two lemmas not only help the implementation in the systolic arrays but also decrease greatly the time complexity. The two lemmas support the use of fully-pipelined cells and the uniformity of the operations in each cell. The part that implements the construction of the ordered spanning trees on a fully-pipelined systolic array follows Huang's F-function [3]. The algorithm uses $(2n-2)$ pipelined cells and runs in $(m+3n-3)$ systolic cycles. This improves the previous algorithm proposed by Prasad and Rangan [5]. Their algorithm, based on an inverted spanning tree and a layout function L, runs in $(2m+7n-2)$ systolic cycles by using n pipelined cells. Other related systolic algorithms on graph problems can be found in [1,2,4 and 6].

The rest of the paper is organized as follows. In section 2, we sketch the theoretical foundation used in our algorithm and propose a pseudo algorithm for finding bridges. In section 3, we review the F-function and describe how to implement the proposed algorithm in a systolic array of $(2n-2)$ pipelined cells. In section 4, an example is provided to show how the algorithm works. Finally, in section 5 we analyze the time complexity of the proposed algorithm and compare it with the one proposed by Prasad and Rangan.

2. Theoretical foundation

Let $X=(x_1, x_2, \ldots, x_s)$ and $Z=(z_1, z_2, \ldots, z_t)$ be two ordered sequences. We use $X \leq Z$ to denote that $\{x_1, x_2, \ldots, x_s\} \subseteq \{z_1, z_2, \ldots, z_t\}$ and the order of the elements in X follows their order in Z. For example, $(b, c, e) \leq (a, b, c, d, e, f)$.

Also let $T(e_1, e_2, \ldots, e_{n-1})$ denote an arbitrary ordered spanning tree of G, where e_1, $e_2, \ldots, e_{n-1}$ are the edges of T. According to the order of edges of T, the edges of G can be named as $e_1, e_2, \ldots, e_m$ such that e_1, e_2, $\ldots$ and e_{n-1} are the first $(n-1)$ edges in the sequence $(e_1, e_2, \ldots, e_m)$. With this naming of the edges, we define $T_i(x_1, x_2, \ldots, x_{n-1})$, $1 \leq i \leq n-1$, to be the ordered spanning tree constructed from the edge sequence $(e_{i+1}, e_{i+2}, \ldots, e_m, e_1, e_2, \ldots, e_i)$, where e_{i+1} is the edge considered first and e_i last. It is obvious that $(x_1, x_2, \ldots, x_{n-1}) \leq (e_{i+1}, e_{i+2}, \ldots, e_m, e_1, e_2, \ldots, e_i)$. For brevity, we sometimes use T_i to denote the ordered spanning tree $T_i(x_1, x_2, \ldots, x_{n-1})$. By the above definition of T_i, we have the following Lemma 1. It gives a way to check whether an edge e_i is a bridge of G or not.

Lemma 1. *Edge e_i is a bridge of G iff e_i is the last edge of T_i.*

Proof ($\Rightarrow$) If e_i is a bridge of G, it is obvious that e_i is an edge of any spanning tree of G. Because e_i is considered last to construct T_i, e_i is the last edge of T_i. ($\Leftarrow$) By definition of T_i, e_i is last edge considered to construct T_i. If e_i is the last edge of T_i, e_i must not be on any cycle of G. In other words, e_i is a bridge of G. □

For example, suppose $T((1, 2), (1, 4), (5, 6), (2, 3), (3, 5))$ be an ordered spanning tree of G. As shown in *Fig. 1*, according to T, the edges (1, 2), (1, 4), (5, 6), (2, 3), (3, 5), (2, 4) and (3, 4) of G are named as $e_1, e_2, \ldots,$ and e_7, respectively. From the edge sequence

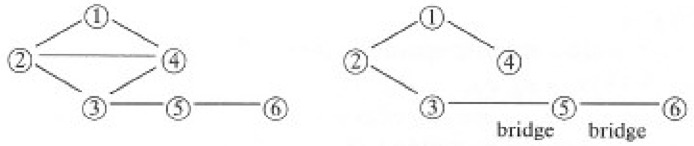


Fig. 1. (a) graph G, (b) spanning tree T.

$(e_6, e_7, e_1, e_2, e_3, e_4, e_5) = ((2, 4), (3, 4), (1, 2), (1, 4), (5, 6), (2, 3), (3, 5))$, we can construct $T_5((2, 4), (3, 4), (1, 2), (5, 6), (3, 5))$. Because $e_5 = (3, 5)$ is the last edge of T_5, $(3, 5)$ is a bridge of G.

As mentioned earlier, every bridge of G is always in any possible spanning tree of G. That is, only the edges of T are the possible candidates for bridges. Thus, in order to find all bridges, we need to check only the edges of T. By Lemma 1, for each edge e_i of T, we first construct T_i from the edge sequence $(e_{i+1}, e_{i+2}, \ldots, e_m, e_1, e_2, \ldots, e_i)$, and then detect whether e_i is the last edge of T_i. For checking all the edges of T, we need to construct $|T|(=n-1)$ ordered spanning trees. The following lemma describes an efficient way to construct these $(n-1)$ ordered spanning trees one after another, and each can be constructed from the previous one. It is useful for the implementation on the fully-pipelined systolic array. It also greatly reduces the time complexity of the proposed algorithm.

Lemma 2. *Let* $(y_1, y_2, \ldots, y_{n-1})$ *be the edge sequence of* T_{i+1}, *then* $T_i(x_1, x_2, \ldots, x_{n-1})$ *can be constructed from the edge sequence* $(e_{i+1}, y_1, y_2, \ldots, y_{n-1})$.

Proof. *Case (1).* If e_{i+1} is a bridge, by Lemma 1 e_{i+1} must be the last edge of T_{i+1}; i.e. $e_{i+1} = y_{n-1}$. Since $T_{i+1}(y_1, y_2, \ldots, y_{n-1} = e_{i+1})$ is constructed from $(e_{i+2}, \ldots, e_m, e_1, \ldots, e_{i+1})$ and $T_i(x_1, x_2, \ldots, x_{n-1})$ is constructed from $(e_{i+1}, e_{i+2}, \ldots, e_m, e_1, \ldots, e_i)$, we can get that T_i must have the same edges as T_{i+1} except that the first edge of T_i is the last edge of T_{i+1}. That is, $(x_1, x_2, \ldots, x_{n-1}) = (e_{i+1}, y_1, y_2, \ldots, y_{n-2})$. Since T_i has the edges $(e_{i+1}, y_1, y_2, \ldots, y_{n-2})$, it is obvious that T_i must be the same as the one constructed from the edge sequence $(e_{i+1}, y_1, y_2, \ldots, y_{n-2}, y_{n-1})$.

Case (2). If e_{i+1} is not a bridge, then by Lemma 1 e_{i+1} is not an edge of T_{i+1}. Hence, in constructing T_{i+1} from $(e_{i+2}, \ldots, e_m, e_1, \ldots, e_{i+1})$, the attaching e_{i+1} to G' must result in a cycle. Let the edge sequence of the cycle be $(s_1, s_2, \ldots, y_k, e_{i+1})$, where $(s_1, s_2, \ldots, y_k) \le (y_1, y_2, \ldots, y_k)$. Then, T_i must have the same edges as T_{i+1} except that e_{i+1} is the first edge of T_i and y_k is not in T_i. That is, $(x_1, x_2, \ldots, x_{n-1}) = (e_{i+1}, y_1, y_2, \ldots, y_{k-1}, y_{k+1}, \ldots, y_{n-1})$. Since T_i has the edges $(e_{i+1}, y_1, y_2, \ldots, y_{k-1}, y_{k+1}, \ldots, y_{n-1})$, it is obvious that T_i must be same as the one constructed from the edge sequence $(e_{i+1}, y_1, y_2, \ldots, y_k, \ldots, y_{n-1})$. This is because $(e_{i+1}, s_1, s_2, \ldots, y_k)(\le (e_{i+1}, y_1, y_2, \ldots, y_k))$ form a cycle, when we attach y_k to G'. Hence, y_k is discarded. □

For example, as shown in *Fig. 1*, suppose $T_5((2, 4), (3, 4), (1, 2), (5, 6), (3, 5))$ is already constructed. By Lemma 2, $T_4((3, 5), (2, 4), (3, 4), (1, 2), (5, 6))$ can then be constructed simply from the edge sequence $((e_5 = (3, 5), (2, 4), (3, 4), (1, 2), (5, 6), (3, 5))$. Similarly, $T_3((2, 3), (3, 5), (2, 4), (1, 2), (5, 6))$ can be constructed from the edge sequence $((e_4 = (2, 3), (3, 5), (2, 4), (3, 4), (1, 2), (5, 6))$.

From the above two lemmas, we outline our algorithm in a non-systolic fashion. The complete systolic algorithm is presented in the next section.

```
Algorithm. Bridge-Finding
Step 1. Construct an arbitrarily ordered spanning tree $T$ of $G$.
Step 2. Name the edges in E as $e_1, e_2, \dots, e_m$
            such that $e_1, e_2, \dots, e_{n-1}$ are the edges of $T$.
        Then construct the ordered spanning tree $T_{n-1}$ according to
            the edge sequence $(e_n, e_{n+1}, \dots, e_m, e_1, e_2, \dots, e_{n-1})$.
Step 3. For $i = n-1$ down to 2
            Check whether $e_i$ is a bridge of $G$ according to Lemma 1;
            (i.e. check whether $e_i$ is the last edge of $T_i(x_1, x_2, \dots, x_{n-1})$.)
            Construct $T_{i-1}$ from $(e_i, x_1, x_2, \dots, x_{n-1})$ according to Lemma 2.
        Endfor.
        Check whether $e_1$ is a bridge of $G$ according to Lemma 1;
        (i.e. check whether $e_1$ is the last edge of $T_1$.)
```

Note that we do not combine Step 2 and Step 3 together. This is because T_{n-1} and T_i, $1 \le i \le n-2$, are constructed by different ways. T_{n-1} is constructed from the edge sequence $(e_n, \dots, e_m, e_1, \dots, e_{n-1})$. Whereas, T_i, $1 \le i \le n-2$, is constructed from the edge sequence $(e_{i+1}, y_1, y_2, \dots, y_{n-1})$, where y_1, y_2, $\dots$ and y_{n-1} are the edges of T_{i+1}. Note that in the construction of T_{n-1}, the non-tree edges $(e_n, \dots, e_m)$ of $G - T$ must be considered before the tree edges $(e_1, \dots, e_{n-1})$ of T. It seems rather difficult, since initially T is constructed from an arbitrary edge sequence rather than from the edge sequence $(e_n, \dots, e_m, e_1, \dots, e_{n-1})$. We will solve this problem in section 3. In Step 3, there are two operations for edges of T. First each edge is examined to see whether it is a bridge, and then used to construct the next ordered spanning tree. Hence, the identification of the bridges can be carried out as the same time with the construction of the ordered spanning trees. The details of implementation will be given in section 3.

3. Implementation on a fully-pipelined systolic array

In this section, we first review the F-function [3], then describe how to implement the three steps of the pseudo bridge-finding algorithm presented in section 2. We use a fully-pipelined systolic array of $(2n-2)$ cells numbered from 1 to $2n-2$. We suppose that each cell knows its own cell number. The first $(n-1)$ cells are used to implement the construction of T in Step 1. Constructing of T_{n-1} in Step 2 is implemented in the last $(n-1)$ cells. Constructing T_i, $1 \le i \le n-2$, and finding of all bridges in Step 3 are also implemented in the last $(n-1)$ cells. Two systolic algorithms will be given to describe the operations of the first $(n-1)$ cells and the last $(n-1)$ cells, respectively.

3.1. F-function

The F-function is used to detect whether a cycle exists during the construction of an ordered spanning tree in a pipelined systolic array. F-function is mainly based on the mapping function $\langle x, y \rangle$ defined as below.

$$\langle x, y \rangle(z) = \begin{cases} \min(x, y) & \text{if } z = \max(x, y); \\ z & \text{otherwise.} \end{cases}$$

We let $\langle x, y \rangle(u, v)$ denote $(\langle x, y \rangle(u), \langle x, y \rangle(v))$. For example, $\langle 4, 5 \rangle(1, 5) = (\langle 4, 5 \rangle(1), \langle 4, 5 \rangle(5)) = (1, 4)$.

For each edge (u_i, v_i) (i.e. edge e_i), we define its f-value (x_i, y_i) recursively. We let $(x_1, y_1) = (u_1, v_1)$ and $(x_i, y_i) = \langle x_{i-1}, y_{i-1} \rangle \cdots \langle x_1, y_1 \rangle(u_i, v_i)$. F-function is then defined as follows: F_0 is the identity function and F_i is the complete function $\langle x_{i-1}, y_{i-1}\rangle \cdots \langle x_1, y_1\rangle$. By definition, $(x_i, y_i) = (F_{i-1}(u_i), F_{i-1}(v_i)) = \langle x_{i-1}, y_{i-1}\rangle \cdots \langle x_1, y_1\rangle(u_i, v_i)$.

For example, in *Fig. 1(a)*, suppose the edge sequence is ((1, 2), (1, 4), (5, 6), (2, 4), (2, 3), (3, 5), (3, 4)); i.e. $(u_1, v_1) = (1, 2)$, $(u_2, v_2) = (1, 4)$, $(u_3, v_3) = (5, 6)$, $(u_4, v_4) = (2, 4)$, $(u_5, v_5) = (2, 3)$, $(u_6, v_6) = (3, 5)$ and $(u_7, v_7) = (3, 4)$. We have

$$\begin{aligned}
(x_1, y_1) &= (u_1, v_1) = (1, 2)\\
(x_2, y_2) &= (F_1(u_2), F_1(v_2)) = \langle x_1, y_1\rangle(u_2, v_2) = \langle 1, 2\rangle(1, 4) = (1, 4)\\
(x_3, y_3) &= (F_2(u_3), F_2(v_3)) = \langle x_2, y_2\rangle\langle x_1, y_1\rangle(u_3, v_3)\\
&= \langle 1, 4\rangle\langle 1, 2\rangle(5, 6) = (5, 6)\\
(x_4, y_4) &= (F_3(u_4), F_3(v_4)) = \langle 5, 6\rangle\langle 1, 4\rangle\langle 1, 2\rangle(2, 4) = \langle 5, 6\rangle\langle 1, 4\rangle(1, 4)\\
&= \langle 5, 6\rangle\langle 1, 1\rangle = (1, 1)\\
(x_5, y_5) &= (F_4(u_5), F_4(v_5)) = \langle 1, 1\rangle\langle 5, 6\rangle\langle 1, 4\rangle\langle 1, 2\rangle(2, 3) = (1, 3)\\
(x_6, y_6) &= (F_5(u_6), F_5(v_6)) = \langle 1, 3\rangle\langle 1, 1\rangle\langle 5, 6\rangle\langle 1, 4\rangle\langle 1, 2\rangle(3, 5) = (1, 5)\\
(x_7, y_7) &= (F_6(u_7), F_6(v_7)) = \langle 1, 5\rangle\langle 1, 3\rangle\langle 1, 1\rangle\langle 5, 6\rangle\langle 1, 4\rangle\langle 1, 2\rangle(3, 4)\\
&= (1, 1)
\end{aligned}$$

It has been shown [3] that if x_i is equal to y_i, then attaching edge (u_i, v_i) to G' must create a cycle and the edge must be discarded. From above, attaching edges (u_4, v_4) and (u_7, v_7) to G' will create cycles, since $x_4 = y_4$ and $x_7 = y_7$. Therefore, we discard (u_4, v_4) and (u_7, v_7) and finally get the ordered spanning tree T as shown in *Fig. 1(b)*.

3.2. Implementation

In order to implement the proposed algorithm in a fully-systolic array, we use two tuples in each cell, *U-tuple* and *L-tuple*. *U*-tuples record the input data and flow into the cells. *L*-tuples record the tree edges appended to G' and are stored in the cells. When the *U*-tuple flows to cells, the cells execute the *F*-function recursively to compute the *f*-value. By *f*-value, we can detect whether an edge is a tree-edge or not. If it is, then we copy the *L*-tuple from the *U*-tuple and store the *L*-tuple in the cell. If it is not, then the *U*-tuple just flows over the cell. The maintenance of *U*-tuples and *L*-tuples in a fully-pipelined implementation is the main role of our algorithms.

Because T, T_{n-1} and T_i, $1 \le i \le n-2$, are constructed from different edge sequences, we use different inputs for the *U*-tuples and the *L*-tuples for these three different categories of trees. Moreover, we suppose that the cycle time of the fully-pipelined systolic array is the maximal cycle time for these three different constructions.

3.2.1 Construction of T

The first $(n-1)$ cells is used to construct T. We use $U.(u, v, x, y, \text{STATE}, \text{TYPE})$ and $L.(u, v, x, y)$ to denote the *U*-tuple and the *L*-tuple, respectively. (u, v) and (x, y) indicate the flowing edge and its *f*-value. TYPE is used to indicate the type of an edge when it is processed among cells. It may be an edge, a tree edge or a non-tree edges, denoted by '$_{\text{edge}}$', 'T_{edge}' or '$\neg\text{T}_{\text{edge}}$'. STATE is used to indicate the processing state of the edges. Initially, each edge (u, v) of E is assigned a *U*-tuple: $U.(u, v, x, y, \text{STATE}, \text{TYPE}) := U.(u, v, u, v, `_{\text{initial}}', `_{\text{edge}}')$, and flows into the cells one by one in an arbitrary edge sequence. Here, '$_{\text{initial}}$' and '$_{\text{edge}}$' mean that the edge just flows into the systolic array and is only known to be an edge of E. There are three cases to process the flowing *U*-tuple in each cell.

(1) If U.STATE = ‘$_{\text{initial}}$’ and the cell has no L-tuple, it implies that the flowing edge (u, v) is a tree edge and must be appended to G'. For this case, there is a L-tuple copied from U-tuple and stored in the cell: $L.(u, v, x, y) := U.(u, v, x, y)$. At the same time, the U-tuple must be modified: $U.(u, v, x, y, x', y'$, STATE, TYPE$) := (u, v, u, v, u, v$, ‘$_{\text{bypass}}1$’, ‘T_{edge}’). Here, ‘T_{edge}’ means edges (u, v) has been determined to be a tree edge, and ‘$_{\text{bypass}}1$’ indicates we need not do anything for edge (u, v) in the following cells of the first $(n-1)$ cells. Note that the newly modified U-tuple is prepared as the input data for constructing T_{n-1}. The details will be discussed in the following subsection.

(2) If U.STATE = ‘$_{\text{initial}}$’ and the cell has an L-tuple, it implies that we need to compute the f-value $(U.x, U.y)$ by the application of F-function: $(U.x, U.y) := \langle L.x, L.y\rangle(U.x, U.y)$. If $U.x = U.y$, it means that attaching the flowing edge (u, v) into G' will create a cycle and we must discard the non-tree edge (u, v). Thus, in this case we need not store the L-tuple and just modify the U-tuple: $U(u, v, x, y, x', y'$, STATE, TYPE$) := U(u, v, u, v, u, v$, ‘$_{\text{bypass}}1$’, ‘$\neg\text{T}_{\text{edge}}$’). Here, ‘$\neg\text{T}_{\text{edge}}$’ means that edge (u, v) is determined to be a non-tree edge. If $U.x \neq U.y$, then the U-tuple just flows over the cell.

(3) If U.STATE = ‘$_{\text{bypass}}1$’, we just propagate the U-tuple to the next cell. In other words, the edge has been determined to be a tree edge or a non-tree edge of T and we need not do anything for it.

3.2.2. *Construction of T_{n-1}*

We use the last $(n-1)$ cells to construct T_{n-1}. As mentioned in section 2, by the definition, T_{n-1} is constructed from the edge sequence $(e_n, \ldots, e_m, e_1, \ldots, e_{n-1})$, where the non-tree edges $(e_n, \ldots, e_m)$ of $G - T$ are considered before the tree edges $(e_1, \ldots, e_{n-1})$ of T. It seems rather difficult, since initially T is constructed from an arbitrary edge sequence rather than from the edge sequence $(e_n, \ldots, e_m, e_1, \ldots, e_{n-1})$. In the solution we use ‘$_{\text{insert}}$’ to mark STATE. If the edge in the U-tuple flowing to the cell is a non-tree edge and the edge in the L-tuple stored in the cell is a tree edge, we must change the U.STATE to ‘$_{\text{insert}}$’. ‘$_{\text{insert}}$’ indicates that the non-tree edge must be inserted in cells before the tree edges. Since we need to know whether an edge stored in the L-tuple is a tree edge or a non-tree edge, we use $L.(u, v, x, y$, TYPE) to denote L-tuples. We also use $U.(u, v, x, y, x', y'$, STATE, TYPE) to denote U-tuples which are prepared in the first $(n-1)$ cells. Here (x', y') is used to denote the f-value of the inserted edge.

In the last $(n-1)$ cells, initially we must check whether the flowing U-tuple needs to be marked ‘$_{\text{insert}}$’ or not. After the checking, there are four cases to process the flowing U-tuple in each cell.

(1) If U.STATE = ‘$_{\text{bypass}}1$’ and the cell has no L-tuple, we do: $L.(u, v, x, y$, TYPE$) := U.(u, v, x, y$, TYPE) and U.STATE := ‘$_{\text{bypass}}2$’. That means that edge (u, v) flows into the cell and is detected to be a tree edge, then we append it to G'. Here, ‘$_{\text{bypass}}2$’ indicates that the flowing edge has been determined to be a tree edge for constructing T_{n-1} and we need not do anything for the edge in the following cells.

(2) If U.STATE = ‘$_{\text{bypass}}1$’ and the cell has an L-tuple, then we need to compute the f-value $(U.x, U.y)$ by the application of F-function: $(U.x, U.y) := \langle L.x, L.y\rangle(U.x, U.y)$; $(U.x', U.y') := \langle L.x, L.y\rangle(U.x', U.y')$. If $U.x = U.y$, it means that attaching edge (u, v) into G' will create a cycle and we must discard edge (u, v). For this case, we need not store the L-tuple and simply modify the U-tuple by doing U.STATE := ‘$_{\text{bypass}}2$’. Here, ‘$_{\text{bypass}}2$’ means that the flowing edge has been known to be a non-tree edge and we just propagate the edge in the following cells. If $U.x \neq U.y$, then the edge just flows over the cell.

(3) If U.STATE = '$_{\text{insert}}$', it means that the flowing edge in the U-tuple must be inserted into the cell. For doing so, we just swap $U.(u, v, x, y, \text{TYPE})$ with $L.(u, v, x, y, \text{TYPE})$ in the cell. Besides, in order to make the f-values of the new U-tuple and L-tuple correct after the swapping, we must modify the f-values of L-tuple and U-tuple before the swapping: $(Tx, Ty) := (L.x, L.y)$, $(L.x, L.y) := \langle x', y' \rangle (L.x, L.y)$ and $(x', y') := \langle Tx, Ty \rangle (x', y')$. Here, (Tx, Ty) is a temporary variable. After the swapping, if $U.x = U.y$, it means that the edge of the new U-tuple is a non-tree edge and we need to modify the U-tuple: U.STATE := '$_{\text{bypass}}$2'. After the swapping, if $U.x \neq U.y$, the new U-tuple just flows over the cell.

(4) If U.STATE = '$_{\text{bypass}}$2', we simply propagate the U-tuple to the next cell.

3.2.3. Identification of bridges

After constructing T and T_{n-1}, the edges of $T(e_1, e_2, \ldots, e_{n-1})$ are stored in the L.tuples in the first $(n-1)$ cells and the edges of $T_{n-1}(z_1, z_2, \ldots, z_{n-1})$ are stored in the L.tuples in the last $(n-1)$ cells. Subsequently, we need to identify whether the edges $e_{n-1}, e_{n-2}, \ldots$ and e_1, which are stored in the L.tuples in the first $(n-1)$ cells, are bridges or not. Thus, those edges must be triggered from the first $(n-1)$ cells to the last $(n-1)$ cells. As mentioned in Step 3 of section 2, there are two operations for each edge e_i of T. First each e_i must be identified whether it is a bridge by checking whether it is the last edge of the currently constructed tree $T_i(x_1, x_2, \ldots, x_{n-1})$. Then it is also used to construct the next tree T_{i-1}. Because T_{i-1} is constructed from the edge sequence $(e_i, x_1, x_2, \ldots, x_{n-1})$, e_i must be inserted before $x_1, x_2, \ldots$ and x_{n-1}. For doing so, we use '$_{\text{insert}}$' to mark STATE.

In order to trigger the flow of the edges $e_1, e_2, \ldots$ and e_{n-1} from the first $(n-1)$ cells into the last $(n-1)$ cells, we introduce $(n-1)$ extra U-tuples as the input data. We use $U.(u, v, x, y, x', y', u', v', \text{STATE}, \text{TYPE})$ to denote those $(n-1)$ U-tuples. (u', v') is used to denote the edge which is currently identified whether it is a bridge. The initial values of those $(n-1)$ U-tuples are assigned as $U.(-, -, -, -, -, -, -, -,$ '$_{\text{insert}}$', 'i'$)$, $i = n-1, n-2, \ldots, 1$, respectively. 'i' indicates the number of cell. When such a U-tuple flows into cells and if U.TYPE equals to the cell number, we copy the U-tuple from the L-tuple: U-tuple := L-tuple. Note that those U-tuple are prepared in the first $(n-1)$ cells. The following illustrates how we prepare the U-tuples corresponding to the edges of T in the first $(n-1)$ cells.

(0) If U.TYPE = '$_{\text{cellno}}$', we copy U-tuple from l-tuple: $U.(u, v, x, y) := U.(x', y', u', v') := L.(u, v, u, v)$, where '$_{\text{cellno}}$' is the number of the cell.

When each extra U-tuple flows into the last $(n-1)$ cells, first we must check whether it is a bridge. By Lemma 1, we can detect whether edge e_i in the flowing U-tuple is a bridge by simply checking in cell $(2n-2)$ whether e_i is the last edge of $T_i(x_1, x_2, \ldots, x_{n-1})$ or not. However, for the sake of uniformity, we let the detection be carried out in each of the last $(n-1)$ cells. Note that the checking operation must be done before the application of F-function, because after the application of F-function, the edge of the L-tuple belongs to T_{i-1} rather than T_i. The following illustrates how the bridge is detected.

(1) If the condition $U.(u', v') = L.(u, v)$ is detected, it means the currently identified edge (u', v') is equal to the last tree edge of the currently constructed tree. In this case, we change L-TYPE to '$_{\text{bridge}}$'. This means that the edge of L-tuple is detected to be a bridge. In fact, this case only appears in the cell $(2n-2)$.

After the checking, the flowing U-tuple must be inserted in the cell to construct its corresponding ordered spanning tree.

(2) If U.STATE = '$_{\text{insert}}$', it means that the flowing edge in the U-tuple must be inserted into the cell. This case is the same as the third case of section 3.2.2.

'$_{\text{insert}}$' is used so that each edge e_i of T be inserted in cell $(n-1)+1$. Then the L-tuples originally stored in the last $(n-1)$ cell will be shifted forward or be detected to be a non-tree edge of T_{i-1} by the application of F-function. It is worth mentioning that when each e_i of T flows into the last $(n-1)$ cells, in addition it will be identified whether it is a bridge in T_i and is also to construct T_{i-1}. The overlaps between the construction of the ordered spanning trees and the identification of bridges reduce much of the time complexity.

Now, we propose the two fully-pipelined systolic algorithms implemented in the first $(n-1)$ cells and the last $(n-1)$ cells.

Algorithm 1 (*for the first n − 1 cells*)

```
[  (U.STATE = '$_{\text{initial}}$') $\wedge$ ($L$ part is empty) $\rightarrow$ /* Append */
      Let $L.(u, v, x, y) := U.(u, v, x, y)$;
      Let $U$.(STATE, TYPE) := ('$_{\text{bypass}}$1', '$T_{\text{edge}}$');
      /* Prepare the input data $U.(u, v, x, y, x', y'$, STATE, TYPE)
      for constructing $T_{n-1}$ */
      Let $U.(x, y, x', y') := U.(u, v, u, v)$;
 □($U$.STATE = '$_{\text{initial}}$') $\wedge$ ($L$ part is not empty) $\rightarrow$ /* Compute $f$-value */
    Let $(U.x, U.y) := \langle L.x, L.y\rangle(U.x, U.y)$;
      $(U.x = U.y) \rightarrow$ /* Discard */
```

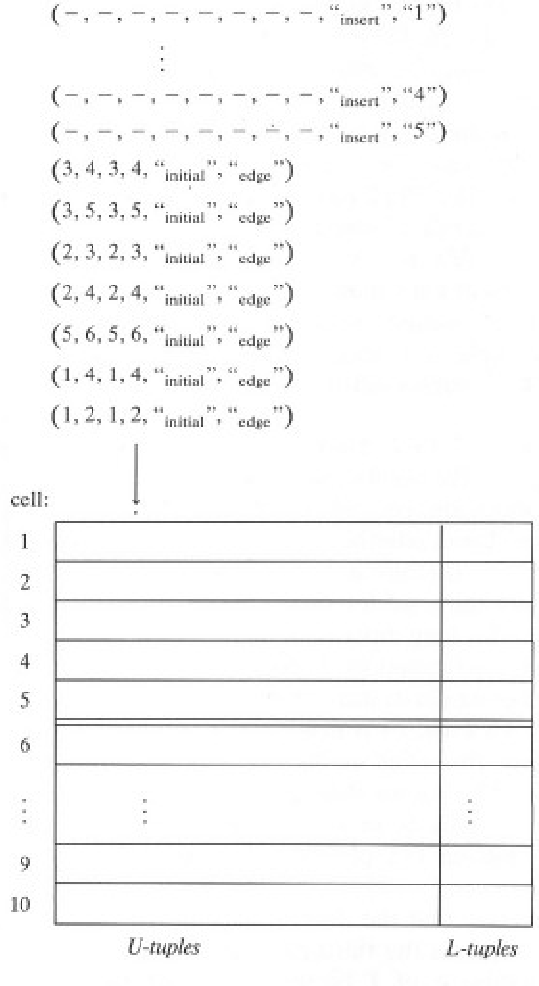
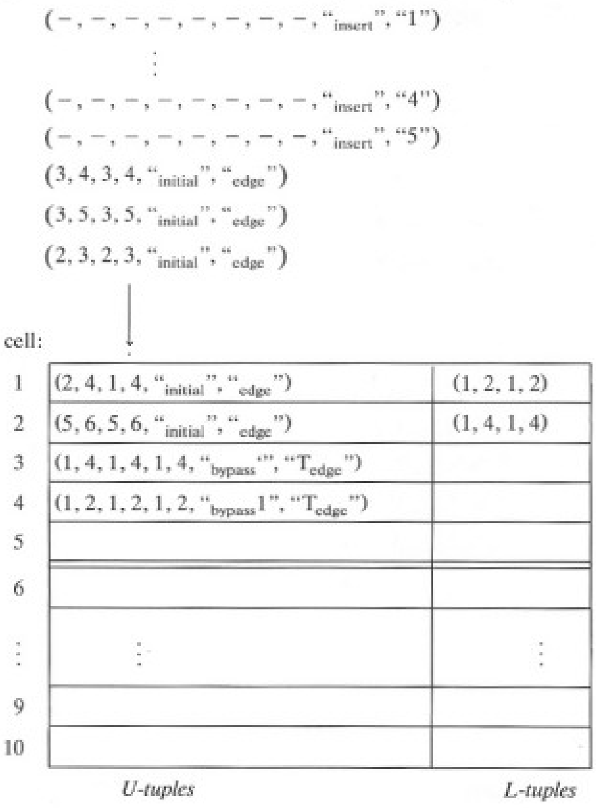

(−, −, −, −, −, −, −, −, "insert", "1")
⋮
(−, −, −, −, −, −, −, −, "insert", "4")
(−, −, −, −, −, −, −, −, "insert", "5")
(3, 4, 3, 4, "initial", "edge")
(3, 5, 3, 5, "initial", "edge")
(2, 3, 2, 3, "initial", "edge")
(2, 4, 2, 4, "initial", "edge")
(5, 6, 5, 6, "initial", "edge")
(1, 4, 1, 4, "initial", "edge")
(1, 2, 1, 2, "initial", "edge")
cell:
1
2
3
4
5
6
⋮
9
10
U-tuples
L-tuples

Fig. 2(a).

```
        Let $U.$(STATE, TYPE) := (‘$_{\text{bypass}}1$’, ‘$\neg \mathrm{T}_{\text{edge}}$’);
        /* Prepare the input data $U.(u, v, x, y, x', y'$, STATE, TYPE)
        for constructing $T_{n-1}$ */
        Let $U.(x, y, x', y') := U.(u, v, u, v)$;
  □($U.$STATE = ‘$_{\text{bypass}}1$’) → /* Propagate */
      Do nothing;
  □($U.$TYPE = ‘$_{\text{cellno}}$’) → /* Prepare the input data
        $U.(u, v, x, y, x', y', u', v'$, STATE, TYPE) for identifying bridges */
      /* ‘$_{\text{cellno}}$’ is the number of the cell */
      Let $U.(u, v, x, y) := U.(x', y', u', v') := L.(u, v, u, v)$;
]
```

Algorithm 2 (*for the last n − 1 cells*)

```
[  ($U.(u', v') = L.(u, v)$ → /* Identify bridges */
      Let $L.$TYPE := ‘$_{\text{bridge}}$’;
  □($U.$STATE = ‘$_{\text{bypass}}1$’) ∧ ($U.$TYPE = ‘$\neg \mathrm{T}_{\text{edge}}$’) ∧ ($L.$TYPE = ‘$\mathrm{T}_{\text{edge}}$’) →
      Let $U.$STATE := ‘$_{\text{insert}}$’; /* Mark ‘$_{\text{insert}}$’ */
]
[  ($U.$STATE = ‘$_{\text{bypass}}1$’) ∧ ($L$ part is empty) → /* The first case */
      Let $L.(u, v, x, y$, TYPE) := $U.(u, v, x, y$, TYPE); /* Append */
      Let $U.$STATE := ‘$_{\text{bypass}}2$’;
  □($U.$STATE = ‘$_{\text{bypass}}1$’) ∧ ($L$ part is not empty) → /* The second case */
      Let $(U.x, U.y) := \langle L.x, L.y\rangle(U.x, U.y)$; /* Compute $f$-value */
```



(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “1”)

⋮

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “4”)

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “5”)

(3, 4, 3, 4, “$_{\text{initial}}$”, “$_{\text{edge}}$”)

(3, 5, 3, 5, “$_{\text{initial}}$”, “$_{\text{edge}}$”)

(2, 3, 2, 3, “$_{\text{initial}}$”, “$_{\text{edge}}$”)

cell:	U-tuples	L-tuples
1	(2, 4, 1, 4, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(1, 2, 1, 2)
2	(5, 6, 5, 6, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(1, 4, 1, 4)
3	(1, 4, 1, 4, 1, 4, “$_{\text{bypass}}$‘”, “T_{edge}”)	
4	(1, 2, 1, 2, 1, 2, “$_{\text{bypass}}1$”, “T_{edge}”)	
5		
6		
⋮	⋮	⋮
9		
10		

Fig. 2(b).

```
        Let $(U.x', U.y') := \langle L.x, L.y\rangle(U.x', U.y')$;
        $(U.x = U.y) \rightarrow$ Let $U$.STATE $:=$ '$_{\text{bypass}}2$'; /* Discard */
  □$(U$.STATE $=$ '$_{\text{insert}}$'$) \rightarrow$ /* The third case */
        /* Modify the $f$-values */
        Let $(Tx, Ty) := (L.x, L.y)$;
        Let $(L.x, L.y) := \langle U.x', U.y'\rangle(L.x, L.y)$;
        Let $(U.x', U.y') := \langle Tx, Ty\rangle(U.x', U.y')$;
        Swap $U.(u, v, x, y$, TYPE) and $L.(u, v, x, y$, TYPE); /* Swap */
        $(U.x = U.y) \rightarrow$ Let $U$.STATE $:=$ '$_{\text{bypass}}2$'; /* Discard */
  □$(U$.STATE $=$ '$_{\text{bypass}}2$'$) \rightarrow$ /* The fourth case */
        Do nothing; /* Propagate */
]
```

3.4. An example

Now, let us consider the graph G in *Fig. 1(a)* to illustrate the proposed systolic algorithms. Since there are six vertices in G, a ten-cell array is used. We assumed that the edges flow into cells in the following order: (1, 2), (1, 4), (5, 6), (2, 4), (2, 3), (3, 5), (3, 4). The results shown in the example were obtained from actual simulation. The following gave the essential parts of the simulation. *Figure 2(a)* shows the initial input data including $m(=7)$ edges of E and $(n-1)(=5)$ extra edges.

After four cycles, the edges (1, 2), (1, 4), (5, 6) and (2, 4) flow into cells. After F-function is applied, the contents of cells are given in *Fig. 2(b)*. In the fourth cycle, by the application of

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “1”)

⋮

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “4”)

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “5”)

(3, 4, 3, 4, “$_{\text{initial}}$”, “$_{\text{edge}}$”)

(3, 5, 3, 5, “$_{\text{initial}}$”, “$_{\text{edge}}$”)

cell:	U-tuples	L-tuples
1	(2, 3, 1, 3, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(1, 2, 1, 2)
2	(2, 4, 2, 4, 2, 4, “$_{\text{bypass}}1$”, “$\neg T_{\text{edge}}$”)	(1, 4, 1, 4)
3	(5, 6, 5, 6, 5, 6, “$_{\text{bypass}}1$“”, “T_{edge}”)	(5, 6, 5, 6)
4	(1, 4, 1, 4, 1, 4, “$_{\text{bypass}}1$”, “T_{edge}”)	
5	(1, 2, 1, 2, 1, 2, ,“$_{\text{bypass}}1$”, “T_{edge}”)	
6		
⋮	⋮	⋮
9		
10		

Fig. 2(c).

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “1”)

⋮

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “4”)

↓

cell:	*U-tuples*	*L-tuples*
1	(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “5”)	(1, 2, 1, 2)
2	(3, 4, 3, 1, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(1, 4, 1, 4)
3	(3, 5, 3, 5, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(5, 6, 5, 6)
4	(2, 3, 2, 3, 2, 3, “$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(2, 3, 1, 3)
5	(2, 4, 2, 4, 2, 4, ,“$_{\text{bypass}}$1”, “¬T$_{\text{edge}}$”)	
6	(5, 6, 5, 6, 5, 6, “$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(1, 2, 1, 2, “T$_{\text{edge}}$”)
7	(1, 4, 1, 4, 1, 4, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	(1, 4, 1, 4, “T$_{\text{edge}}$”)
8	(1, 2, 1, 2, 1, 2, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	
9		
10		

Fig. 2(d).

F-function, $(U.x, U.y)$ in cell 1 is changed: $(U.x, U.y) := \langle L.x, L.y\rangle(U.x, U.y) = \langle 1, 2\rangle(2, 4) = (1, 4)$.

In the fifth cycle, edge (2, 3) flows into cell 1. By the application of F-function, $(U.x, U.y)$ in cell 1 is changed: $(U.x, U.y) := \langle 1, 2\rangle(2, 3) = (1, 3)$. At the same time by the application of

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “1”)
(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “2”)
(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “3”)

↓

cell:	*U-tuples*	*L-tuples*
1	(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “4”)	(1, 2, 1, 2)
2	(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “5”)	(1, 4, 1, 4)
3	(3, 4, 3, 1, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(5, 6, 5, 6)
4	(3, 5, 1, 5, “$_{\text{initial}}$”, “$_{\text{edge}}$”)	(2, 3, 1, 3)
5	(2, 3, 2, 3, 2, 3, ,“$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	
6	(1, 2, 1, 2, 1, 4, “$_{\text{insert}}$”, “T$_{\text{edge}}$”)	(2, 4, 2, 4, “¬T$_{\text{edge}}$”)
7	(5, 6, 5, 6, 5, 6, “$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(1, 4, 1, 4, “T$_{\text{edge}}$”)
8	(1, 4, 1, 4, 1, 4, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	
9	(1, 2, 1, 2, 1, 2, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	
10		

Fig. 2(e).

(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “1”)
(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “2”)

↓

cell:	U-tuples	L-tuples
1	(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “3”)	(1, 2, 1, 2)
2	(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “4”)	(1, 4, 1, 4)
3	(−, −, −, −, −, −, −, −, “$_{\text{insert}}$”, “5”)	(5, 6, 5, 6)
4	(3, 4, 3, 4, 3, 4, “$_{\text{bypass}}$1”, “$\neg$T$_{\text{edge}}$”)	(2, 3, 1, 3)
5	(3, 5, 3, 5, 3, 5, ,“$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(3, 5, 1, 5)
6	(2, 3, 2, 3, 2, 3, “$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(2, 4, 2, 4, “$\neg$T$_{\text{edge}}$”)
7	(1, 4, 1, 1, 1, 1, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	(1, 2, 1, 2, “T$_{\text{edge}}$”)
8	(5, 6, 5, 6, 5, 6, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	(5, 6, 5, 6, “T$_{\text{edge}}$”)
9	(1, 4, 1, 4, 1, 4, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	
10	(1, 2, 1, 2, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	

Fig. 2(f).

F-function, the U-tuple in cell 2 is changed from (2, 4, 1, 4, ‘$_{\text{initial}}$’, ‘$_{\text{edge}}$’) to (2, 4, 1, 1, ‘$_{\text{initial}}$’, ‘$_{\text{edge}}$’). Because $U.x = U.y$, edge (2, 4) is detected to be a non-tree edge, finally the U-tuple is modified as (2, 4, 2, 4, 2, 4, ‘$_{\text{bypass}}$1’, ‘$\neg$T$_{\text{edge}}$’). *Figure 2(c)* shows the result.

After eight cycles, we obtain the result shown in *Fig. 2(d)* . At the same time $T_{n-1}(= T_5)$ is being constructed in cell 6 ~ 10. In the 8th cycle, edge (1, 4) flowing into cell 7, the initial U-tuple is (1, 4, 1, 4, 1, 4, ‘$_{\text{bypass}}$1’, ‘T$_{\text{edge}}$’). Because initially in the cycle U.STATE = ‘$_{\text{bypass}}$1’ of cell 7 and the cell has no L-tuple, edge (1, 4) is detected to be a tree edge of T_{n-1}. It must be appended into the cell and the U-tuple must be modified: L.tuple := (1, 4, 1, 4, ‘T$_{\text{edge}}$’); U-tuple := (1, 4, 1, 4, 1, 4, ‘$_{\text{bypass}}$2’, ‘T$_{\text{edge}}$’).

↓

cell:	U-tuples	L-tuples
1	(1, 2, 1, 2, 1, 2, 1, 2, “$_{\text{insert}}$”, “1”)	(1, 2, 1, 2)
2	(1, 4, 1, 4, 1, 4, 1, 4, “$_{\text{insert}}$”, “2”)	(1, 4, 1, 4)
3	(5, 6, 5, 6, 5, 6, 5, 6, “$_{\text{insert}}$”, “3”)	(5, 6, 5, 6)
4	(2, 3, 2, 3, 2, 3, 2, 3,“$_{\text{insert}}$”, “4”)	(2, 3, 1, 3)
5	(3, 5, 3, 5, 3, 5, 3, 5,“$_{\text{insert}}$”, “5”)	(3, 5, 1, 5)
6	(3, 4, 3, 2, 3, 2, “$_{\text{bypass}}$1”, “$\neg$T$_{\text{edge}}$”)	(2, 4, 2, 4, “$\neg$T$_{\text{edge}}$”)
7	(3, 5, 3, 5, 3, 5, “$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(1, 2, 1, 2, “T$_{\text{edge}}$”)
8	(2, 3, 1, 3, 1, 3, “$_{\text{bypass}}$1”, “T$_{\text{edge}}$”)	(5, 6, 5, 6, “T$_{\text{edge}}$”)
9	(1, 4, 1, 1, 1, 1, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	
10	(5, 6, 5, 6, 5, 6, “$_{\text{bypass}}$2”, “T$_{\text{edge}}$”)	

Fig. 2(g).

j:6	U.(3, 5, 3, 5, 3, 5, 3, 5, “$_{\text{insert}}$”, “5”)	U.(2, 4, 2, 4, 3, 5, 3, 5, “$_{\text{insert}}$”, “$\neg T_{\text{edge}}$”)
	L.(2, 4, 2, 4, “$\neg T_{\text{edge}}$”)	L.(3, 5, 3, 5, “5”)
j:7	U.(2, 4, 2, 4, 3, 5, 3, 5, “$_{\text{insert}}$”, “$\neg T_{\text{edge}}$”)	U.(3, 4, 3, 2, 2, 5, 3, 5, “$_{\text{insert}}$”, “$\neg T_{\text{edge}}$”)
	L.(3, 4, 3, 2, “$\neg T_{\text{edge}}$”)	L(2, 4, 2, 4, “$\neg T_{\text{edge}}$”)
j:8	U.(3, 4, 3, 2, 2, 5, 3, 5, “$_{\text{insert}}$”, “$\neg T_{\text{edge}}$”)	U.(1, 2, 1, 2, 1, 5, 3, 5, “$_{\text{insert}}$”, “T_{edge}”)
	L.(1, 2, 1, 2, “T_{edge}”)	L.(3, 4, 3, 2, “$\neg T_{\text{edge}}$”)
j:9	U.(1, 2, 1, 2, 1, 5, 3, 5, “$_{\text{insert}}$”, “T_{edge}”)	U.(5, 6, 1, 6, 1, 5, 3, 5, “$_{\text{insert}}$”, “T_{edge}”)
	L.(5, 6, 5, 6, “T_{edge}”)	L.(1, 2, 1, 2, “T_{edge}”)
j:10	U.(5, 6, 1, 6, 1, 5,3, 5, “$_{\text{insert}}$”, “T_{edge}”)	U.(3, 5, 1, 1, 1, 1, 3, 5, “$_{\text{bypass}}$2”, “$_{\text{bridge}}$”)
	L.(3, 5, 1, 5, “T_{edge}”)	L.(5, 6, 1, 6, “T_{edge}”)

Fig. 2(h).

In the ninth cycle, edge (2, 4) flows into cell 6 and the initial U-tuple of cell 6 is (2, 4, 2, 4, 2, 4, ‘$_{\text{bypass}}$1’, ‘$\neg \mathrm{T}_{\text{edge}}$’) and the L-tuple is (1, 2, 1, 2, ‘T_{edge}’). Because edge (2, 4) in the U-tuple is a non-tree edge and edge (1, 2) in the L-tuple is a tree edge in cell 6, first we must mark U.STATE as ‘$_{\text{insert}}$’. When U.STATE = ‘$_{\text{insert}}$’ which means that we must swap the U-tuple and the L-tuple in the cell. Before swapping the U-tuple and the L-tuple in cell 6, we must modify the f-values in cell 6: $(L.x, L.y) := \langle 2, 4\rangle(1, 2) = (1, 2)$; $(U.x', U.y') := \langle 1, 2\rangle(2, 4) = (1, 4)$. After the modification of f-values, we swap the U-tuple and the L-tuple. *Figure 2(e)* shows the result.

In the tenth cycle, the initial U-tuple of cell 7 is (1, 2, 1, 2, 1, 4, ‘$_{\text{insert}}$’, ‘T_{edge}’). Because U.STATE = ‘$_{\text{insert}}$’ in cell 7, we must first modify the f-values of the U-tuple and the L-tuple in the cell and then swap the U-tuple and the L-tuple. The new U-tuple is (1, 4, 1, 1, 1, 1, ‘$_{\text{insert}}$’, ‘T_{edge}’) and the new L-tuple is (1, 2, 1, 2, ‘T_{edge}’). Because $U.x = U.y$ edge (1, 4) is detected to be a non-tree edge and must be discarded. The U.STATE must be changed to be ‘$_{\text{bypass}}$2’. *Figure 2(f)* shows the result. From the L-tuples of the first 5 cells, we can get T((1, 2), (1, 4), (5, 6), (2, 3), (3, 5)).

After twelve cycles, the 5 extra U-tuples for edges of T are copied from the L-tuples in the first 5 cells and will flow into the last 5 cells. They are prepared as the input data for identifying bridges in the last 5 cells. *Figure 2(g)* shows the result.

j:6	U.(2, 3, 2, 3, 2, 3, 2, 3, “$_{\text{insert}}$”, “4”)	U.(3, 5, 2, 5, 2, 3, 2, 3, “$_{\text{insert}}$”, “5”)
	L.(3, 5, 3, 5, “5”)	L.(2, 3, 2, 3, “4”)
j:7	U.(3, 5, 2, 5, 2, 3, 2, 3, “$_{\text{insert}}$”, “5”)	U.(2, 4, 2, 4, 2, 3, 2, 3, “$_{\text{insert}}$”, “$\neg T_{\text{edge}}$”)
	L.(2, 4, 2, 4, “$\neg T_{\text{edge}}$”)	L(3, 5, 2, 5, “5”)
j:8	U.(2, 4, 2, 4, 2, 3, 2, 3, “$_{\text{insert}}$”, $\neg T_{\text{edge}}$”)	U.(3, 4, 2, 2, 2, 2, 2, 3, “$_{\text{bypass}}$2”, $\neg T_{\text{edge}}$”)
	L.(3, 4, 3, 2, “$\neg T_{\text{edge}}$”)	L.(2, 4, 2, 4, “$\neg T_{\text{edge}}$”)
j:9	U.(3, 4, 2, 2, 2, 2, 2, 3, “$_{\text{bypass}}$2”, “$\neg T_{\text{edge}}$”)	U.(3, 4, 2, 2, 2, 2, 2, 3, “$_{\text{bypass}}$2”, “$\neg T_{\text{edge}}$”)
	L.(1, 2, 1, 2, “T_{edge}”)	L.(1, 2, 1, 2, “T_{edge}”)
j:10	U.(3, 4, 2, 2, 2,2, 2, 3, “$_{\text{bypass}}$2”, “$\neg T_{\text{edge}}$”)	U.(3, 4, 2, 2, 2, 2, 2, 3, “$_{\text{bypass}}$2”, “$\neg T_{\text{edge}}$”)
	L.(5, 6, 1, 6, “T_{edge}”)	L.(5, 6, 1, 6, “T_{edge}”)

Fig. 2(i).

Let us now consider the identification of bridges. As stated earlier, the U-tuples in cell 5, 4, ..., 1 serially flow into the last 5 cells to construct T_i, $1 \le i \le 4$ and to be identified whether they are bridges. Let us first consider the edge $e_5 = (3, 5)$. In *Fig. 2(h)*, the left column describes the snapshot of cell j at the beginning and the right column describes the snapshot of cell j after the application of F-function for one systolic cycle. Cycle by cycle, *Fig. 2(h)* shows the snapshots of cell j.

In cell 10 of the left column of *Fig. 2(h)*, since $U.(u', v') = L.(u, v) = (3, 5)$, edge (3, 5) is detected to be a bridge. First, L.TYPE must be changed to '$_{\text{bridge}}$'. Then because U.STATE = '$_{\text{insert}}$', we must modify the f-values of the U-tuple and L-tuple and swap the U-tuple and the L-tuple in cell 10. After the swapping, because $U.x = U.y$, U.STATE is changed to '$_{\text{bypass}}$2'. Note that after edge $e_5 = (3, 5)$ is detected to be a bridge in cell 10, T_4((3, 5), (2, 4), (3, 4), (1, 2), (5, 6)) is also constructed in cell 6–10 on cycle 13 ~ 17.

For edge $e_4 = (2, 3)$ flowing into the last 5 cells, the snapshots of cell j are shown as in *Fig. 2(i)* cycle by cycle. In cell 8, after the application of F-function, because $U.x = U.y$, edge (3, 4) must be discarded and U.STATE must be changed to '$_{\text{bypass}}$2'. Edge (2, 3) is not a bridge since in cell 10 $U.(u', v') \ne L.(u, v)$ before the application of F-function. For other edges of T, the operations are similar. Finally, the two edges (3, 5) and (5, 6) are identified to be bridges of G as shown in *Fig. 1(b)*.

4. Conclusions

In the proposed algorithms, there are a lot of overlaps between the construction of the trees and the identification of the bridges. There are $m + (n - 1)$ input data including the m edges of E for constructing trees T and T_{n-1} and the $(n - 1)$ extra data for identifying the bridges. The following discusses the execution time of the constructions of T and T_{n-1} and the identification of the bridges.

(1) Constructing T: From the first edge of E flowing into the first $(n - 1)$ cells to the last edge of E flowing out the first $(n - 1)$ cells, it takes $m + (n - 1)$ cycles.

(2) Constructing T_{n-1}: It is obvious that after $(n - 1)$ cycles the first edge of E will flow into the first cell of the last $(n - 1)$ cells. From the first edge of E flowing into the last $(n - 1)$ cells to the last edge of E flowing out the last $(n - 1)$ cells, it also takes $m + (n - 1)$ cycles.

(3) Identifying the bridges: We can find that after $m + (n - 1)$ cycles the first edge of those extra edges flows into the first cell in the last $(n - 1)$ cells. It takes $2(n - 1)$ cycles from the first edge of those extra edges flowing into the last $(n - 1)$ cells to the last edge of those extra edges flowing out the last $(n - 1)$ cells.

The total execution time of the proposed algorithm is $(m + 3n - 3)$ systolic cycles. We use the following figure to illustrate the execution time of our algorithm. It is worthy to mention that in our algorithm, each cell maintains uniform data and data always moves forward. Therefore all steps can be fully pipelined (*Fig. 3*).

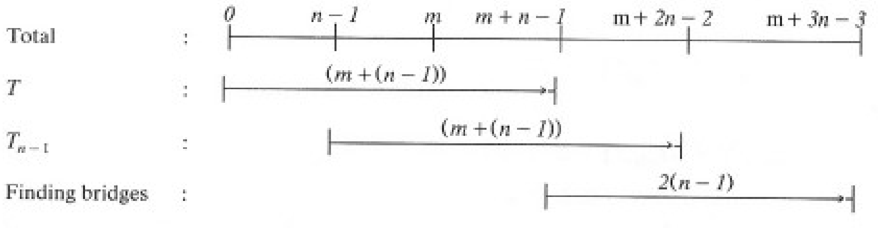


Fig. 3.

Let us briefly review Prasad and Rangan's algorithm [5]. Their algorithm requires a linear systolic array of n cells. Each cell is used to represent a vertex. Their algorithm is based on an inverted spanning tree and a layout function L. If the edges of a directed spanning tree of an undicted graph are all reversed (i.e. we direct the edges from a vertex to its father vertex in the tree), the resulting graph is called an inverted spanning tree. A layout function L is used to denote whether or not an edge is stored in the cells. They need four phases. First, an algorithm is developed to find a spanning tree. Second, three complicated steps are proposed to give directions to the edges obtained from first phase so that the spanning tree becomes an inverted spanning tree. In this phase, the data may need to travel back to the left end of the array. Third, two steps are used to find a layout function L and then each edge is put in its proper cell according to the function. Fourth, the bridges of the graph are identified by marking out those edges in the inverted spanning tree which are not bridges. Their first phase needs $(m + n - 1)$ systolic cycles and the second phase needs at least $3n$ systolic cycles. For the third phase, it needs at least $2n$ systolic cycles. In the fourth phase, they have to test each edge of a graph. Thus, it is accomplished in $(m + n - 1)$ systolic cycles. Therefore, total execution time of their algorithm requires at least $(2m + 7n - 2)$ systolic cycles.

The total execution time of the proposed algorithm is expected to be less than that proposed by Prasad and Rangan, because their algorithm needs more execution cycles and has non-uniform and rather complicated phases. The use of fully-pipelined cells, and the simplicity and uniformity of the operations in each cell are nontrivial. Two lemmas provided in the paper support our design to have the distinctive properties.

References

[1] S. Ashtaputre and C.D. Savage, Systolic arrays with embedded tree structures for connectivity problems, *IEEE Trans. Comput.* 34 (5) (May 1985) 483–484.

[2] K. Doshi and P. Varman, Determining biconnectivity on a systolic array, *Proc. Internat. Conf. on Parallel Processing* (1987) 848–850.

[3] S.T. Huang, A fully-pipelined minimum-cost-spanning-tree constructor, *J. Parallel Distributed Comput.* 9, (1) (May 1990) 55–62.

[4] S.T. Huang and M.S. Tsai, A linear systolic algorithm for the connected component problem, *BIT* 29 (1989) 217–226.

[5] B.K. Prasad and C.P. Rangan, Inverted spanning tree paradigm on systolic arrays, *Proc. Internat. Workshop on Systolic Arrays*, University of Oxford (Jul. 1986).

[6] C.A. Savage, Systolic design for connectivity problems, *IEEE Trans. Comput.* 33 (1) (Jan. 1984) 99–104.