RECOGNITION OF DFS TREES: SEQUENTIAL AND PARALLEL ALGORITHMS WITH REFINED VERIFICATIONS

(Revised Version of TR#520)

by

E. Korach and Z. Ostfeld

Technical Report #582
July 1989
RECOGNITION of DFS TREES: SEQUENTIAL and PARALLEL

ALGORITHMS with REFINED VERIFICATIONS*

(Revised, July 1989)

Ephraim Korach and Zvi Ostfeld **

Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

The Depth First Search (DFS) algorithm is one of the basic techniques which is used in a very large variety of graph algorithms. Every application of the DFS involves, beside traversing the graph, constructing a special structured tree, called a DFS tree that may be used subsequently.

In a previous work we have shown that the family of graphs in which every spanning tree is a DFS tree is quite limited. Therefore the question: Given an undirected graph \( G = (V, E) \) and an undirected spanning tree \( T \), is \( T \) a DFS tree (T-DFS) in \( G \) ? was naturally raised and answered by sequential linear time algorithms. Here, we present a parallel algorithm which solves this problem in \( O(t) \) time complexity and uses \( O(\frac{|E|}{t}) \) processors, where \( t \geq \log |V| \), on a CREW PRAM. We also study the problem for directed graphs. A linear \( O(|E|) \) time algorithm for solving it in the sequential case and a parallel implementation of it, which has \( O(\log^2 |V|) \) time complexity and uses \( O(|V|^{2.376}) \) processors on a CREW PRAM, are presented.

An important feature of our algorithms, that we call refined verification, is that some of their decisions are endowed with proofs that can be verified with a better complexity than that of the algorithms themselves: In the undirected case, if the answer of the algorithm is negative then it outputs a proof for that fact that can be verified in \( O(t) \) time complexity with \( O(\frac{|V|}{t}) \) processors, where \( t \geq \log |V| \), on a CREW PRAM. In the directed case, if \( T \) is not a DFS tree in \( G \) then the sequential algorithm supplies an \( O(|V|) \) time proof for that fact and the parallel implementation supplies a proof for that fact that can be verified in \( O(t) \) time complexity with \( O(\frac{|V|}{t}) \) processors, where \( t \geq \log |V| \), on a CREW PRAM. If \( T \) is a DFS tree in \( G \) then the parallel implementation of the algorithm outputs a proof that can be verified in \( O(t) \) time complexity with \( O(\frac{|E|}{t}) \) processors, where \( t \geq \log |V| \), on a CREW PRAM. Hence, all the verifications have an optimal speed-up.

Key words: acyclic graphs, DFS trees, graph algorithms, PRAM, parallel algorithms, refined verifications.

AMS(MOS) subject classification: 05C05, 05C20, 05C38, 68Q10, 68Q25, 68R10

* A preliminary version of this work was presented at the Binaional French-Israeli Symposium on Combinatorics and Algorithms, Jerusalem (November, 1988).

** The work of the second author was performed in partial fulfillment of the requirements of the M.Sc. degree in the Computer Science Department, Technion, Haifa.
1. INTRODUCTION

The Depth First Search (DFS) algorithm is a basic technique which is used in a very large variety of graph algorithms. The history of this algorithm (in a different form) goes back to 1882 when Tremaux' algorithm for the maze problem was first published (see [BLW, page 18]). The impact of DFS grew rapidly since the Hopcroft and Tarjan version of it was published (see [Ta], [HT-a], [HT-b] and [HT-c]). This algorithm is used in many areas of computer science, and recently it also has penetrated the field of parallel and distributed algorithms (e.g. [AA], [Aw], [HY], [LMT], [Re], [Sm] and [Ti]).

Every use of the DFS, besides traversing the graph, constructs a special structured tree, called a DFS tree, that may be used subsequently. Previous results ([KO-a]) have shown that the family of graphs in which every undirected spanning tree is a DFS tree, is quite limited. Therefore the problem: Given an undirected graph \( G=(V, E) \) and an undirected spanning tree \( T \), is \( T \) a DFS tree in \( G \)? was naturally raised and answered by linear time algorithms in [KO-a] and independently in [HN]. The solution to this problem might be useful in many applications. For example when we would like to run a DFS in an undirected graph where the weights of the edges are all distinct and would like to obtain the unique minimum spanning tree as a DFS tree. DFS is also used in many algorithms in the field of Artificial Intelligence. In particular, the structure of the DFS tree is important in the constraint satisfaction problems (e.g., see [FFL]). In [FFL] it is proved that the problem of finding a special DFS tree (optimal with respect to its depth) is NP-hard. The algorithms presented here, enable us to test very efficiently whether a given tree with a desired structure is a DFS tree.

In section 3, we present a parallel algorithm for solving this problem. This algorithm has \( O(t) \) time complexity and uses \( O\left(\frac{\left|E\right|}{t}\right) \) processors, where \( t \geq \log |V| \), on a concurrent read exclusive write parallel random access machine (CREW PRAM). In addition, if the decision of the algorithm is negative then it outputs a proof that can be verified in \( O(t) \) time complexity with \( O\left(\frac{|V|}{t}\right) \) processors, where \( t \geq \log |V| \), on a CREW PRAM. The proof consists of

(i) A spanning subgraph \( G' \) of \( G \) with \( O(|V|) \) edges, supplied by the algorithm, where \( T \) is a spanning tree of \( G' \) and it is not a DFS tree in \( G' \), and
(ii) The algorithm itself.

By checking that $G'$ is a subgraph of $G$ and rerunning the same algorithm on $G'$ one can have the proof.

Both the algorithm and the verification have an optimal speed-up in the sense that the time-processor product is the time required by an optimal sequential algorithm (verification). Since the verification has a better complexity than that of the algorithm itself we call this property of the algorithm \textit{refined verification} (see section 2). Motivation for refined verifications is given in the sequel.

In sections 4 - 5 we study the analogous problem for directed graphs:

In section 4, a linear time algorithm for solving the problem in the sequential directed case is presented. In addition, if the decision of the algorithm is negative then it supplies a spanning subgraph $G'$ of $G$ with $O(|V|)$ edges, and analogously to the undirected case this constitutes an $O(|V|)$ time proof to justify the decision.

In section 5, an efficient parallel implementation of the algorithm from section 4, based on parallel implementation of matrix multiplication, is presented. By using the methods in [CW] the algorithm has $O(\log^2 |V|)$ time complexity using $O(|V|^{2.376})$ processors on a CREW PRAM. If $T$ is not a \textit{DFS} tree in $G$ then the algorithm supplies a verification for that fact that can be verified in $O(t)$ time complexity by $O\left(\frac{|V|}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM (a notable refined verification). If $T$ is a \textit{DFS} tree in $G$ then the algorithm supplies a verification for that fact that can be verified in $O(t)$ time complexity with $O\left(\frac{|E|}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM (both verifications have an optimal speed-up). A parallel algorithm for recognizing a \textit{DFS} tree in a digraph, based on similar ideas, was independently* presented in [SchVit]. In the case of a positive decision [SchVit] also includes a computation of a \textit{DFS} numbering, that we use to obtain a refined verification. However, it does not contain a refined verification for a negative decision.

In the following we present some motivations for refined verifications and a table that summarizes our results.

* A preliminary version of our paper appeared in May (1988) (see [KO-b]).
The fact that our algorithms supply verifications that can be verified in a better complexity than the complexity of the algorithms themselves is important not only from the theoretical point of view, but also in practical situations such as in the following example: Assume we have a network with a set of working stations, which are low power and busy computers, and a set of central powerful computers. Assume that the stations ask the powerful computers to solve problems and that the network is not completely reliable (i.e., errors may occur during communications). The powerful computers send back to the stations the answers together with refined proofs and the stations just have to verify the proofs to be sure that no error occurred during the communication process. In a situation where the stations are busy or not powerful enough to solve a problem, but they can afford verification of a refined proof, it will be best for them to use the central computers.

### Summary of our results:

<table>
<thead>
<tr>
<th>Graph $G = (V, E)$</th>
<th>Implementation</th>
<th>Type of Complexity</th>
<th>Recognition Algorithm</th>
<th>Negative Verification</th>
<th>Positive Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undirected</td>
<td>Sequential</td>
<td>Time</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>$O(t)$</td>
<td>$O(t)$</td>
<td>$O(t)^*^#$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processor</td>
<td>$O\left(\frac{</td>
<td>E</td>
<td>}{t}\right)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Product</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>Directed</td>
<td>Sequential</td>
<td>Time</td>
<td>$O\left(\log^2</td>
<td>V</td>
<td>\right)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processor</td>
<td>$O(</td>
<td>V</td>
<td>^{\omega})^{**}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Product</td>
<td>$O\left(</td>
<td>V</td>
<td>^{\omega} \log^2</td>
</tr>
</tbody>
</table>

# This algorithm does not appear in this paper but in [KO-a].

## For $t \geq \log |V|$.  

* The verification is in fact the algorithm itself.

** Where the product of two $N \times N$ matrices can be computed in $O(N^{\omega})$ arithmetic operations for $\omega > 2$ (see [PR, Theorem A.1]). It is known ([CW]) that $\exists \epsilon > 0$ such that $\omega = 2.376 - \epsilon$.  

& Since $\omega < 2.376$ then the product is $O(|V|^{2.376})$.

&& The model of computation is a CREW PRAM.
The fact that our algorithms supply proofs to justify negative answers which are based on an $O(|V|)$ subgraph of $G$ is important also in the following example: Consider a graph $G$ which represents a network where edges may fall at random. After we have obtained a proof that a specific spanning tree $T$ of $G$ is not a DFS tree in $G$, the proof remains valid until one of the non tree edges in it falls. If in addition $G$ is a dense graph, and we would like to wait until enough edges fall so that $T$ becomes a DFS tree in $G$, then there is a high probability that we wait for a long time until the proof is not valid anymore. Only at that moment we have to rerun the algorithm.

In section 6, some open problems are presented.

Other related characterizations and algorithms appear in [KO-c] and [Sy]. Another motivation for recognition of DFS trees in undirected graphs - a model of computation - can be found in [KO-a].

2. SOME DEFINITIONS and CONVENTIONS

Let $T$ be an undirected spanning tree in an undirected graph $G = (V, E)$ and let $s \in V$. $T_s$ is the tree $T$ with an orientation that makes $s$ the root. $T$ is called a DFS tree ($T$-DFS) in $G$ if there exists a vertex $s \in V$ such that $T_s$ is a DFS tree ($T$-DFS) in $G$ (i.e. $T_s$ can be constructed by a DFS run in $G$). Let $T$ be a directed spanning tree in a digraph $G$. $T$ is a DFS tree ($T$-DFS) in $G$ if it can be constructed by a DFS run in $G$. (Note that we use the same term for undirected graphs and digraphs.) Let $\{a, b, c, d, e, f, g, h\}$ be vertices in a directed tree $T$. If there is a directed path in $T$ from $a$ to $b$, we say that $a$ is an ancestor of $b$ and $b$ is a descendant of $a$. A vertex is an ancestor and a descendant of itself. $d$ is called second-$\pi_{c,e}$ if there is a tree edge $c \rightarrow d$ and $d$ is an ancestor of $e$ in $T$.

$f$ is called the lowest common ancestor of $g$ and $h$ if (i) $f$ is a common ancestor of $g$ and $h$ in $T$, and (ii) Any common ancestor of $g$ and $h$ in $T$ is an ancestor of $f$.

In this paper, the refined verifications are in fact deterministic algorithms. For the analysis of the verifications we add to our model the following natural basic assumptions: (i) The input of the problem is already stored in the memory and therefore we do not consider the complexity of reading the input as part of the complexity of the verification. (ii) We assume uniform cost criterion - RAM (e.g. see [AHU, p.12]). Therefore a processor can access in $O(1)$ time complexity any cell in the memory. Hence, there are some cases where the complexity of reading the input of the verification is less than the complexity of reading.
the input of the original problem and there are some cases where the time (the time - processor product in
the parallel case) complexity of the verification itself is even less than the complexity of reading the
description of the problem by the original algorithm. Another assumption (that we use in Corollary 3.10
and in Corollary 4.24) is that the time complexity required for any algorithm in order to know the descrip-
tion of a graph (or a sub graph) with \( E \) edges is \( \Omega(\log E) \).

To simplify the discussion we assume that all graphs in this paper are without loops and parallel edges.
This assumption does not affect the complexity of the sequential algorithm presented here. As for the
parallel algorithms, by using the sort algorithm from [Co], which has \( O(\log V) \) time complexity and
uses \( O(\log V) \) processors on an exclusive read exclusive write parallel random access machine (EREW
PRAM), we can eliminate loops and parallel edges in a graph \( G=(V, E) \).

We say that a parallel algorithm (verification) has an optimal speed-up if the time-processor product in it
is equal to the lower bound of the time complexity of the sequential algorithm (verification) for the same
problem.

Where no confusion may arise we use \( n \) instead of \( |V| \) and \( m \) instead of \( |E| \) for a given graph
\( G=(V, E) \).

The symbol: " □ " stands for "end of the proof" or "end of the statement and a proof is not provided".

3. PARALLEL RECOGNITION of DFS TREES in UNDIRECTED GRAPHS

Previous results ([KO-a]) have shown that the family of graphs in which every spanning tree, is a
DFS tree is quite limited. Therefore the problem: \( \text{Given an undirected graph } G=(V, E) \) and an
undirected spanning tree \( T \), is \( T \) a \( T \)-DFS in \( G \) ? was naturally raised and answered by linear time algo-
rithms in [KO-a] and independently in [HN].

In this work we present a parallel algorithm which solves this problem in \( O(t) \) time complexity and
uses \( O(\frac{|E|}{t}) \) processors, where \( t \geq \log |V| \), on a concurrent read exclusive write parallel random access
machine (CREW PRAM). In addition, if the decision is negative then the algorithm outputs a proof that
can be verified in \( O(t) \) time complexity with \( O(\frac{|V|}{t}) \) processors, where \( t \geq \log |V| \), on a CREW
PRAM. The proof consists of the following two parts:
(i) A spanning subgraph $G'$ of $G$ with $O(|V|)$ edges, supplied by the algorithm, where $T$ is a spanning tree of $G'$ and it is not a DFS tree in $G'$ (and hence, by [KO-a, Proposition 3.1] - $T$ is not a $T$-DFS in $G$).

(ii) The algorithm itself.

By checking that $G'$ is a subgraph of $G$ and rerunning the same algorithm on $G'$ one can have the proof. So in a sense, this algorithm is a "self refinement" algorithm - complexity wise.

Both the algorithm and the negative verification have an optimal speed-up.

Definition 3.1: Let $T$ be an undirected spanning tree in an undirected graph $G = (V, E)$ and let $s \in V$. $T_s$ induces a partition of $E$ into three types of edges:

(i) **Tree edges**.

(ii) **Back edges**: An edge $(a, b) \in E - T$ is a back edge with respect to $T_s$ if $a$ is either an ancestor of $b$ or a descendant of $b$ in $T_s$.

(iii) **Cross edges**: The rest of the edges in $E$.

Observation 3.2: Let $T$ be an undirected spanning tree in an undirected graph $G = (V, E)$ where $\{s, a, b\} \subseteq V$ and let $\pi_{s, a}$ and $\pi_{s, b}$ be the paths in $T$ from $s$ to $a$ and from $s$ to $b$, respectively. $(a, b)$ is a cross edge with respect to $T_s$ if and only if $b \notin \pi_{s, a}$ and $a \notin \pi_{s, b}$. □

Proposition 3.3: Let $T$ be an undirected spanning tree in an undirected graph $G = (V, E)$. Let $\{r, s, u, v\} \subseteq V$ and let $T_r$ and $T_s$ be two orientations of $T$ rooted at $r$ and $s$, respectively. A non tree edge $e = (u, v) \in E$ is a cross edge in $T_r$ if and only if one of the following conditions holds relative to $T_s$:

1. $e$ is a cross edge and $r$ is neither a descendant of $u$ nor a descendant of $v$.
2. $e$ is a back edge (assume w.l.o.g. that $u$ is an ancestor of $v$) and there is a vertex $w \in V$ such that: (i) $w$ is second-\pi_{u,v}, and (ii) $r$ is a descendant of $w$ and is not a descendant of $v$.

Proof: Follows from Observation 3.2. □

Proposition 3.4 ([Ta, Theorem 1]): Let $T$ be an undirected spanning tree in an undirected graph $G = (V, E)$ and let $s \in V$. $T_s$ is a $T$-DFS in $G$ if and only if every edge $(a, b) \in E - T$ is a back edge in $T_s$. □

Corollary 3.5: Let $T$ be an undirected spanning tree in an undirected graph $G = (V, E)$, then $T$ is a $T$-DFS in $G$ if and only if there is some $r \in V$ such that $G$ contains no cross edges in $T_r$. □
Proof: Follows directly from Proposition 3.4 and the definition of a $T$-DFS.

In the following, we present an efficient parallel algorithm for checking whether a given undirected spanning tree is a $T$-DFS in an undirected graph $G$. The algorithm is based on some ideas and techniques from [HN], [TV] and [Vi].

The parallel algorithm:

$\text{PAR\_CHECK} (G, T)$ \{Check in parallel whether $T$ is a $T$-DFS in $G$\}

\begin{itemize}
\item input: An undirected graph $G$ and an undirected spanning tree $T$ of $G$.
\item output: A decision whether $T$ is a $T$-DFS in $G$. If the decision is positive then the algorithm gives as an output the set of vertices $S = \{ s \in V : T_s$ is a DFS tree in $G \}$.
\end{itemize}

\begin{itemize}
\item variables: $f$, $g$, $\text{count}$, $\text{sum}$. \{These variables are four arrays indexed by the vertices of $V$.\}
\item $\text{count\_cross}$ \{This is a three dimensional array indexed by the edges of $E$, where each entry is partitioned into three parts: $\text{side\_1}$, $\text{side\_2}$ and $\text{root}$ where each part corresponds to a vertex and each vertex knows its appropriate sides\}.
\end{itemize}

\begin{verbatim}
begin \{ of the algorithm \}
\begin{enumerate}
\item Choose a vertex $s \in V$ and compute $T_s$ ($s$ is the root of the tree).
\item For every vertex $x \in V$ set $f(x)$ to be the father of $x$ in $T_s$ ($f(s) = \text{null}$). Set all the entries in $\text{count\_cross}$ to be 0.
\item For every edge $e \in E - T$
begin
\begin{enumerate}
\item If $e=(u,v)$ is a cross edge in $T_s$ then
begin
\begin{itemize}
\item \{The two sides of $\text{count\_cross} (e)$ correspond to $u$ and $v$.\}
\item $\text{count\_cross} (e)[u] := \text{count\_cross} (e)[u] - 1$;
\item $\text{count\_cross} (e)[v] := \text{count\_cross} (e)[v] - 1$;
\item $\text{count\_cross} (e)[s] := \text{count\_cross} (e)[s] + 1$
\end{itemize}
end \{ of (3.1) \}
\item Else \{$e=(x,y)$ is a back edge in $T_s$\} begin
\end{enumerate}
\end{verbatim}

Technion - Computer Science Department - Technical Report  CS0582 - 1989
if \( x \) is an ancestor of \( y \) then begin \( u := x \); \( v := y \) end

else \( \{ y \) is an ancestor of \( x \} \) begin \( u := y \); \( v := x \) end;

find \( w \in V \) such that \( w \) is second-\( \pi_{u,v} \).

\( \{ \text{The two sides of} \ count_{-cross} (e) \text{correspond to} v \text{and} w. \} \)

\( count_{-cross} (e)[v] := count_{-cross} (e)[v] - 1; \)

\( count_{-cross} (e)[w] := count_{-cross} (e)[w] + 1 \)

end \{ of (3.2) \}

end \{ of (3) \}.

(4) For every vertex \( x \in V \) compute \( count (x) \), which is the sum of values of \( count_{-cross} (e)[x] \) for all edges \( e \in E \).

(5) For every vertex \( x \in V \) compute \( sum (x) \), which is the sum of values of \( count (u) \) for all vertices \( u \in V \) where \( u \) is an ancestor of \( x \) in \( T_x \).

**Decision (of algorithm PAR_CHECK):** For all \( x \in V \), \( T \) rooted at \( x \) is a \( T-DFS \) in \( G \) if and only if

\( \sum (x) = 0 \).

\( T \) is a \( T-DFS \) in \( G \) if and only if there is at least one vertex \( x \in V \) such that \( \sum (x) = 0 \).

end \{ of algorithm PAR_CHECK \}.

**Theorem 3.6:** Algorithm \( PAR\_CHECK \) \( (G=(V,E), T) \) is correct.

**Proof:** Let \( e=(u,v) \in E - T \) and let \( \{ s, u, v, x \} \subseteq V \) such that \( s \) is the vertex chosen by the algorithm at step (1). Assume \( e \) is a cross edge in \( T_x \). Since we add one to \( count_{-cross} (e)[s] \) (and hence to \( count (s) \)) it is clear that if \( x \) is either a descendant of \( u \) or a descendant of \( v \) then \( \sum (x) \) is not affected by the changes in step (3.1). Otherwise \( e \) contributes one to \( \sum (x) \). Now assume that \( e \) is a back edge with respect to \( T_x \), where \( u \) is an ancestor of \( v \), and let \( w \in V \) be second-\( \pi_{u,v} \). Then the operations in step (3.2) of the algorithm affect \( \sum (x) \) if and only if \( x \) is a descendant of \( w \) and is not a descendant of \( v \). In the latter case \( e \) contributes one to \( \sum (x) \). Herere, by Proposition 3.3, for every vertex \( x \in V \) \( \sum (x) \) is the number of cross edges in \( T_x \) and by Corollary 3.5 the decision of the algorithm is correct. \( \square \)

Baruch Schieber has shown how to obtain the following lemma by a slight modification of the algorithm in [SchVis].

**Lemma 3.7** ([Sc]): Let \( T \) be a rooted tree with \( n \) vertices. It is possible to answer a set of \( q \) queries of the
form: "Which vertex is \text{second-π}_u v?" for any set of \( q \) pairs of vertices \( \{ (u, v) \}, u \) is an ancestor of \( v \) in \( T \), in \( O(t) \) time complexity using \( O(\frac{n + q}{t}) \) processors, where \( t \geq \log n \), on a CREW PRAM.

In a previous version of this paper we have shown how algorithm \( \text{PAR-CHECK}(G = (V, E), T) \) can be implemented in \( O(\log n) \) time complexity using \( O(m) \) processors, on a CREW PRAM (where \( n = |V| \) and \( m = |E| \)). Baruch Schieber has improved our result as follows:

Theorem 3.8 ([Sc]): Algorithm \( \text{PAR-CHECK}(G = (V, E), T) \) can be implemented in \( O(t) \) time complexity using \( O(\frac{m}{t}) \) processors, where \( t \geq \log n \), on a CREW PRAM.

Proof: (The main ideas of this implementation were sketched in [Sc].) Step (1) of the algorithm is computed by \( O(\frac{n}{\log n}) \) processors in \( O(\log n) \) time complexity on an EREW PRAM. We use the Euler tour technique which was presented in [TV] and in [Vi]. We will implement it, however, using the optimal parallel list-ranking algorithm in [CV]. Every tree edge is replaced by two anti-parallel edges and then an Euler circuit is created in the new graph. After we set \( s \) to be the root of \( T \) we refer to the Euler circuit as a path which begins in one of the edges emanating from \( s \). Step (2) is trivially done by \( O(n) \) processors in \( O(1) \) time complexity. Step (3) is implemented as follows: First we use the parallel implementation of the preprocessing algorithm in [SchVis], which has \( O(\log n) \) time complexity and uses \( O(\frac{n}{\log n}) \) processors on an EREW PRAM. Then, we are able to answer each LCA query in \( O(1) \) time complexity using a single processor. This way we recognize cross and back edges (note that \( e = (x, y) \) is a cross edge if and only if the lowest common ancestor of \( x \) and \( y \) is neither \( x \) nor \( y \)). The computation of \( w \ (\text{second-π}_u v) \) in Step (3.2) is done according to Lemma 3.7.

The parallel additions and subtractions in step (4) can be done in \( O(\log n) \) time complexity with \( O(\frac{m}{\log n}) \) processors on an EREW PRAM (see the algorithm in [LF]). The computation of step (5) is done in \( O(\log n) \) time complexity with \( O(\frac{n}{\log n}) \) processors on an EREW PRAM using the optimal algorithm for computing prefix sums in [LF]. We use the Euler path that is obtained from the tree \( T_s \). For every edge \( v \to f(v) \) we assign the value \( \text{count}(v) \) and for every edge \( f(v) \to v \) we assign the value \( -\text{count}(v) \). Consider the last part of the path that starts with the edge \( v \to f(v) \). Clearly, the sum of the
values of the edges along this subpath plus the value of count(s) is the desired sum(v) (and clearly, \(sum(s) = count(s)\)). Finally, the decision can be done in \(O(\log n)\) time complexity with \(O(\frac{n}{\log n})\) processors on an EREW PRAM. \(\square\)

**Theorem 3.9:** If \(T\) is not a \(T-DFS\) in an undirected graph \(G = (V, E)\) then we can modify algorithm PAR_CHECK (without affecting its complexity) so that in addition it supplies a refined verification for that fact. The verification is done in \(O(t)\) time complexity by \(O\left(\frac{|V|}{t}\right)\) processors, for \(t \geq \log |V|\), on a CREW PRAM.

**Proof:** In the following we present the modified algorithm MODIFIED_PAR_CHECK:

1. Choose a vertex \(s \in V\) and compute \(T_s\). \(G'' := G\) \((G''\) is an auxillary graph).\)

2. For every non tree edge \(e = (u, v)\) in \(G\), if \(e\) is a cross-edge in \(T\), then \(G'' := G'' - e \cup \{ (x, u), (x, v), (\hat{u}, \hat{v}) \}\), where \(x\) is the lowest common ancestor of \(u\) and \(v\) in \(T_s\), \(\hat{u} \in V\) is second-\(π_x, u\) and \(\hat{v} \in V\) is second-\(π_x, v\). The edges \((x, u), (x, v)\) and \((\hat{u}, \hat{v})\) are called new edges and the edge \(e\) is the source of those three new edges.

3. Delete all parallel edges in \(G''\).

4. For every vertex in \(G''\), choose one back edge incident to it in which the other end is closest to the root (\(s\)). Then delete all back edges that were not chosen by any vertex.

5. For every vertex in \(G''\), choose two cross edges incident to it (if there is only one edge - choose it). Then delete all cross edges that were not chosen by any vertex.

6. Replace each remaining new edge in \(G''\) by its source to obtain \(G'\).

7. Run the algorithm PAR_CHECK\((G', T)\).

Clearly, both \(G'\) and \(G''\) contain \(O(|V|)\) edges. One can see that \(T\) is a DFS tree in \(G\) iff it is a DFS tree in \(G'\) and that the implementation of Steps 1)-(6) can be done in \(O(t)\) time complexity with \(O\left(\frac{|E|}{t}\right)\) processors, for \(t \geq \log |V|\), on a CREW PRAM. By checking that \(G'\) is a subgraph of \(G\) and rerunning the algorithm on \(G'\) one can have the desired verification. \(\square\)

The results of Theorem 3.8 and Theorem 3.9 can be summarized in the following:
Corollary 3.10: Algorithm *MODIFIED_PAR_CHECK* can be implemented in $O(t)$ time complexity with $O(\frac{|E|}{t})$ processors, where $t \geq \log |V|$, on a CREW PRAM. In addition, if the decision of the algorithm is negative then it supplies a refined verification for its decision. The negative verification has $O(t)$ time complexity and uses $O(\frac{|V|}{t})$ processors, where $t \geq \log |V|$, on a CREW PRAM. Both the algorithm and the negative verification have an optimal speed-up.

Proof: In order to prove the optimality of the speed-up of the algorithm let us assume that there is at most one vertex $s \in V$ such that $T_s$ is a $T$-$DFS$ in $G$. Clearly, we cannot have a proof that $T$ is a $T$-$DFS$ in $G$, unless we go over all the edges in $G - T$ (every edge that we ignore may be a cross edge relative to $T_s$). Hence, $O(|E|)$ is an optimal speed-up for a positive verification, and therefore it is an optimal speed-up for the algorithm itself.

As for the optimality of the negative verification, observe the infinite family which contains the pairs $(G_i, T_i)$ of the following form: $T_i = \{(v_1, v_3), (v_{i-2}, v_i)\} \cup \{(v_j, v_{j+1}) : 2 \leq j \leq i - 2\}$.

$G_i = T_i \cup \{(v_1, v_4), (v_{i-3}, v_i)\} \cup \{(v_j, v_{j+2}) : 2 \leq j \leq i - 3\}$. For every graph $G_i = (V_i, E_i)$ which belongs to the above family, $T_i$ is not a $T$-$DFS$ in $G_i$. However, for every edge $e \in E_i - T_i$, $T_i$ is a $T$-$DFS$ in $G_i - e$. Hence, we cannot have a proof that $T_i$ is not a $T$-$DFS$ in $G_i$, unless we go over all the non tree edges (there are $O(|V_i|)$ such edges). Hence, $O(|V_i|)$ is an optimal speed-up for a negative verification for this family and therefore it is an optimal speed-up for a negative verification in the general case. \(\square\)

4. RECOGNITION of DFS TREES in DIGRAPHS with a REFINED VERIFICATION

In this section we present a linear time algorithm for deciding whether a given directed spanning tree $T$ is a $T$-$DFS$ in a directed graph $G = (V, E)$. If the decision of the algorithm is negative then it supplies a spanning subgraph $G'$ of $G$ with $O(|V|)$ edges, and analogously to the undirected case this constitutes an $O(|V|)$ time proof to justify the negative decision.

Definition 4.1: A directed spanning tree $T$ in a digraph $G = (V, E)$ induces a partition of $E$ into four types of edges:

(i) *Tree edges* ($\overrightarrow{T}$).

(ii) *Forward edges* ($\overrightarrow{F}$): An edge $x \rightarrow y \in E - \overrightarrow{T}$ is a forward edge if $x$ is an ancestor of $y$ in $T$.

(iii) *Back edges* ($\overrightarrow{B}$): An edge $x \rightarrow y \in E$ is a back edge if $y$ is an ancestor of $x$ in $T$. 

Technion - Computer Science Department - Technical Report CS0582 - 1989
Definition 4.2: Let $V$ be a set of vertices. We say that $V$ has an order induced by $f$ if and only if $f : V \rightarrow \{1, 2, ..., \mid V \mid \}$ is a bijection.

Definition 4.3: Let $G=(V, E)$ be a digraph where $V$ has an order induced by $f$. The order is compatible (in $G$) if for every edge $x \rightarrow y \in E : f(x) < f(y)$.

Definition 4.4: Let $T=(V, E)$ be a directed tree. An order of $V$ induced by $f$ is called $DFS-T$-order if there is a $DFS$ run on the tree such that for every vertex $v \in V$, $f(v)=i$ if and only if $v$ is the $i$-th vertex to be discovered during the $DFS$ run.

Clearly, every $DFS$ run (numbering) induces a $DFS-T$-order.

Proposition 4.5: A directed spanning tree $T$ in a digraph $G=(V, E)$ is a $DFS$ tree ($T-DFS$) if and only if $T$ has a $DFS-T$-order induced by $f$ that is compatible in $G=(V, E)$, where $\overrightarrow{T}$ are the tree edges, $\overrightarrow{F}$ are the forward edges, $\overrightarrow{B}$ are the back edges with the reverse direction, and $\overrightarrow{C}$ are the cross edges with the reverse direction.

Proof: For the only if part, see [Ev, p.63].

As for the if part, let us assume that every edge $e = u \rightarrow v$ is labeled by the pair $(B, H)$ where (i) $B=0$ if $e \in \overrightarrow{T}$ and $B=1$ otherwise. (ii) $H=f(v)$. Consider the $DFS$ algorithm with the additional freedom breaking rule: "whenever we have to choose an unused edge we choose an edge with the label which is smallest lexicographically". We denote this modified $DFS$ algorithm $M-DFS$.

The proof of the if part follows from the following claim that can be proved by induction on $\mid E \mid$, for every given $\mid V \mid$.

Claim: Let $T$ be a directed spanning tree on $V$ - a given set of vertices - and assume $T$ has a $DFS-T$-order induced by $f$. For every digraph $G=(V, E)$ that contains $T$, such that $f$ is compatible in $\widehat{G}$, the above $M-DFS$ algorithm, starting at the root of $T$, will give $T$ (as a $T-DFS$), and for each vertex $v \in V$, $f(v)=i$ if and only if $v$ is the $i$-th vertex discovered during the search.

Corollary 4.6: Let $G=(V, E')$ be a subgraph of $G=(V, E)$ and let $T$ be a spanning tree of $G'$. If $T$ is not a $DFS$ tree in $G'$ then $T$ is not a $DFS$ tree in $G$.

Proof: Assume that $T$ is a $DFS$ tree in $G$. By Lemma 4.5 there is a $DFS-T$-order in $T$ that is compati-
ble in $\hat{G}$. Since $G'$ is a subgraph of $G$ then the same $DFS-T$-order is also compatible in $\hat{G}' = (V, \vec{T} \cup \vec{F'} \cup \vec{B'} \cup \vec{C'})$ where $\vec{T}$ are the tree edges, $\vec{F'}$ are the forward edges in $G'$, $\vec{B'}$ are the back edges in $G'$ with the reverse direction and $\vec{C'}$ are the cross edges in $G'$ with the reverse direction (note that $G'$ is a subgraph of $\hat{G}$). Hence, by Lemma 4.5 - $T$ is a $DFS$ tree in $G'$, a contradiction. □

Definition 4.7: Let $T$ be a directed tree and let $x, y$ be two vertices in $T$. $T_x$ is the directed subtree induced by all the descendants of $x$ in $T$ ($x$ is the root of $T_x$). Two directed subtrees $T_x$ and $T_y$ are called brother subtrees if $x$ and $y$ are brothers in $T$ ($x$ and $y$ have a common father in $T$).

Definition 4.8: Let $T$ be a directed spanning tree in a digraph $G = (V, E)$ where $e = x \rightarrow y$ is a non tree edge in $G$ and let $z \in V$ be the lowest common ancestor of $x$ and $y$ in $T$. We define the following elementary reduction operation $\Phi_T(G, e)$:

\[(4.8.1) \text{ If } e \in \vec{T} \cup \vec{B} \text{ then } \Phi_T(G, e) = (V, E - e) \text{ (i.e. } e \text{ is deleted).} \]

\[(4.8.2) \text{ If } e \in \vec{C} \text{ then } \Phi_T(G, e) = (V, E - e \cup \{\hat{x} \rightarrow \hat{y}\}) \text{, where } \hat{x} \in V \text{ is second} - \pi_{x,y} \text{ and } \hat{y} \in V \text{ is second} - \pi_{x,y} \text{ (i.e. } e \text{ is replaced by another cross edge } \hat{x} \rightarrow \hat{y} \text{ where } x \text{ and } y \text{ are in the brother subtrees } T_x \text{ and } T_y, \text{ respectively).} \]

Definition 4.9: Let $T$ be a directed spanning tree in a digraph $G = (V, E)$. We define the following set $\Phi_T^*(G)$:

(i) $G \in \Phi_T^*(G)$.

(ii) If $G' \in \Phi_T^*(G)$ and $e$ is a non tree edge in $G'$ then $\Phi_T(G', e) \in \Phi_T^*(G)$.

Definition 4.10: $G'$ is a minor digraph of $(G, T)$ if $G' \in \Phi_T^*(G)$.

Lemma 4.11: Let $T$ be a directed spanning tree of a digraph $G$. $T$ is a $DFS$ tree in $G = (V, E)$ if and only if it is a $DFS$ tree in every minor digraph of $(G, T)$.

Proof: One can see that any single implementation of $\Phi_T$ does not change the compatibility of a $DFS-T$-order induced by $f$. By Lemma 4.5 the proof is completed. □

Definition 4.12: A digraph $G$ is irreducible relative to a spanning tree $T$ if $\Phi_T^*(G) = \{G\}$.

Definition 4.13: Let $T$ be a directed spanning tree of a digraph $G$. A minor digraph $G'$ of $(G, T)$ is a minimal minor if $G'$ is irreducible relative to $T$. 

Observation 4.14: Let $G=(V, E)$ be a digraph which is irreducible relative to $T$. Then $G$ contains neither forward edges of $T$ nor back edges of $T$ and $x\to y\in E$ is a cross edge of $T$ only if $x$ and $y$ are brothers in $T$. 

Lemma 4.15: Let $T$ be a directed spanning tree of a digraph $G=(V, E)$ then the minimal minor digraph $G'=(V, E')$ of $(G, T)$ is unique and can be obtained by a finite number of elementary reduction operations.

Proof: The tree $T$ remains unchanged after every reduction operation. Therefore, in every reduction operation an edge in $\overrightarrow{B}\cup\overrightarrow{F}$ is either deleted or remains in $\overrightarrow{B}\cup\overrightarrow{F}$. For every edge $e\in \overline{C}$ we can observe three possible outcomes of every reduction: (i) $e$ is not affected. (ii) $e$ is replaced by another edge in $\overline{C}$. (iii) $e$ is deleted. It follows that in $G'$ all the edges in $\overrightarrow{B}\cup\overrightarrow{F}$ are deleted (according to 4.8.1) and every cross edge $x\to y\in E$, where $x$ and $y$ are in the brother subtrees $T_x$ and $T_y^*$, respectively, has a unique image $\hat{x}\to \hat{y}\in E'$ in the minimal minor digraph (according to 4.8.2). Hence, we can get a minimal minor digraph after performing at most $|\overrightarrow{F}| + |\overrightarrow{B}| + |\overline{C}|$ reduction operations. 

Observation 4.16: Let $G=(V, E)$ be an irreducible digraph relative to a spanning tree $T$. A directed circuit in $G$ contains only cross edges.

Proof: It is obvious that for every tree edge $x\to y$, $d(x) < d(y)$ where $d(v)$ is the distance of the vertex $v\in V$ from the root of $T$. From Observation 4.14 it follows that every non tree edge $x\to y$ in $G$ is a cross edge where $d(x)=d(y)$. Hence every circuit may not contain a tree edge. 

Lemma 4.17: Let $G=(V, E)$ be an irreducible digraph relative to a spanning tree $T$. Then $T$ is a $T$-DFS in $G$ if and only if $G$ is acyclic.

Proof: Only if: Since $T$ is a $T$-DFS in $G$ then by Proposition 4.5 - $\hat{G}=(V, \overrightarrow{T}\cup \overrightarrow{C})$ has a compatible order. Hence, $\hat{G}$ has no dicircuit and by Observation 4.16 - $G$ is acyclic.

If: Since $G$ is acyclic then the vertices can be labeled by a bijection $g: V\to \{1, 2, ..., |V|\}$ such that for every edge $x_i\to x_j \in E$, $g(x_i) > g(x_j)$ (e.g. $g$ is the result of a topological sort in $G$ ). Let us assume that every edge $e=u\to v$ is labeled by the pair $(B, H)$ where (i) $B=0$ if $e\in \overrightarrow{T}$ and $B=1$ otherwise. (ii) $H=g(v)$.

The proof of the if part follows from the following claim that can be proved by induction on $|E|$, for
every given $|V|$.

**Claim:** Let $T$ be a directed spanning tree on $V$ - a given set of vertices. Let $G=(V, E)$ be an acyclic digraph which is irreducible relative to $T$ and let $V$ have an order induced by a bijection $g$, as described above (recall that $T$ and $g$ induce a labeling of the edges in $E$). Then, the modified DFS algorithm ($M$-DFS, described in the proof of Proposition 4.5) starting at the root of $T$ will give $T$ as its DFS tree in $G$. □

The algorithm for checking whether a given directed spanning tree $T$ is a $T$-DFS in a given digraph $G$ has two phases. In phase one we build the minimal minor digraph and in phase two we check whether it is acyclic. Phase two has a linear time implementation which is based on the following observation.

**Observation 4.18:** A digraph $G$ contains a directed circuit if and only if for every DFS forest in $G$ there is a back edge.

**Proof:** The if part is obvious since any back edge in a DFS tree creates a cycle. As for the only if part, if $G$ is not acyclic then it has at least one strongly connected component $C$ with more than one vertex. By [Ta, Corollary 11] the vertices of $C$ define a subtree of a tree of every DFS forest in $G$. Hence, in every DFS forest there is at least one back edge which enters the root of the subtree defined by the vertices of $C$. □

**Corollary 4.19:** Given a digraph $G$ and an arbitrary DFS forest $F$ in $G$: $G$ contains no directed circuit if and only if there is no back edge for $F$. □

The structure of the algorithm is as follows:

**DI_CHECK**($G$, $T$) {Check whether $T$ is a $T$-DFS in $G$}.

**Input:** A digraph $G$ and a directed spanning tree $T$ in $G$.

**Output:** A decision whether $T$ is a $T$-DFS in $G$.

**Phase One:** $BMM$($G$, $T$) {Build Minimal Minor}

**Input:** A digraph $G$ and a directed spanning tree $T$ in $G$.

**Output:** The minimal minor digraph of ($G$, $T$).

**begin** (of phase one:)

(1) Deleting all the back and forward edges of $G$ to get $G_1$. 

Technion - Computer Science Department - Technical Report CS0582 - 1989
(2) Creating a minor digraph $G_2$ of $(G_1, T)$ by using 4.8.2 for every cross edge $e$ in $G_1$.

$G_2$ is the output of phase one \{i.e. $G_2 = BMM(G, T)$\}.

end \{of phase one.\}

PHASE TWO: CAD($G_2$) \{Check the Acyclicity of a Digraph\}

input: A digraph $G_2$ \{the output of phase one\}.

output: A decision whether $G_2$ is acyclic.

begin \{of phase two\}

(1) Build a $DFS$ forest in $G_2$.

(2) Check whether there are back edges in this forest.

Decision \{of phase two\}: $G_2$ is acyclic if and only if there are no back edges in it.

end \{of phase two.\}

Decision \{of algorithm $DI$ CHECK\}: $T$ is a $T$-$DFS$ in $G$ if and only if $G_2$ is acyclic.

Lemma 4.20: $BMM(G, T)$ computes the (unique) minimal minor digraph of $(G, T)$.

Proof: Follows from the proof of Lemma 4.15 and the fact that we have applied 4.8.2 for every cross edge in $G$. □

We now present an efficient sequential implementation of algorithm $BMM$.

Step (1) of the algorithm is done by using a $DFS$ algorithm in $G$ along $T$. In step (2) of the algorithm we want to replace each cross edge $e = x \rightarrow y$ by the cross edge $R[e] = \hat{x} \rightarrow \hat{y}$ where $x$ and $y$ are in the brother subtrees $T_x$ and $T_y$, respectively, and $R$ is an array indexed by the cross edges of $G_1$ \{$=$ the cross edges of $G$\}.

First we find the lowest common ancestors of $(x, y)$ in $T$, for every cross edge $e = x \rightarrow y$ in $G_1$ \{by using the algorithm in [Ha] or in [SchVis]\}. The results are organized in an array $LCA$ indexed by the cross edges of $G_1$.

After computing $LCA$ we use a modification of the $DFS$ algorithm for computing $R$ as follows:

$NCE(G_1, T, LCA)$ \{compute the New Cross Edges\}

input: A digraph $G_1 = (V, E)$ \{the output of step 1\}, $T$ \{a spanning tree in $G_1$\}, and an array $LCA$ \{computed as above\}.
output: An array $R$ indexed by the cross edges of $G_1$. For every cross edge $e = x \rightarrow y$ in $G_1$ where $x$ and $y$ are in the brother subtrees $T_x^*$ and $T_y^*$, respectively, $R[e] = \hat{x} \rightarrow \hat{y}$.

begin {of the algorithm}
(1) Mark all the edges of $T$ "unused"; $v := r$ where $r \in V$ is the root of $T$;
(2) If all the tree edges emanating from $v$ are used then go to (4);
(3) Choose an unused tree edge $v \rightarrow u$; Mark $e$ "used"; $f(u) := v$; $s(v) := u$; $v := u$; go to (2);
(4) For every cross edge $e$ where $v$ is either the tail of $e$ or the head of $e$ do
    begin
        $z := LCA[e]$; $\hat{v} := s(z)$; {Clearly, $s(z)$ is second-ancestor of $v$}
        If $v$ is tail($e$) then tail($R[e]$) := $\hat{v}$;
        Else {v is head($e$)} head($R[e]$) := $\hat{v}$
        end;
(5) If $v \neq r$ then $v := f(v)$ and go to (2)
        Else {v=r, all the vertices have been scanned} Halt.

$R$ is the output of algorithm $NCE$ (i.e. $R = NCE(G_1, T, LCA)$).

end {of the algorithm $NCE$}.

$R$ contains all the new cross edges where duplications may occur. After computing $R$ we compute $\vec{R}$, which is the result of eliminating duplications in $R$, and create a digraph $G_2 = (V, T \cup R)$ which is the output of phase one. For every cross edge $e$ in $G_1$ there is a cross edge in $\vec{R}$, which represents the cross edge $R[e]$. $G_2$ is the minimal minor digraph of $(G_1, T)$ (and of $(G, T)$).

Lemma 4.21: $BMM (G=(V, E), T)$ has time complexity $O(|E_1|)$.

Proof: It is obvious that the complexity of step (1) is $O(|E_1|)$. As for step (2), the computation of $LCA$ is done using the algorithm in [Ha] or in [SchVis] for finding lowest common ancestors in a static tree in an off-line model. Both algorithms get as an input a static tree and a collection of queries about lowest common ancestors and give as an output answers to those queries. The time complexity of those two algorithms is $O(|E_1|)$. The computation of $R$ (algorithm $NCE$)
is in fact a modified DFS algorithm and has time complexity $O(|E|)$. The rest of step (2) (removing duplications from $R$ and completing the creation of $G_2$) has time complexity which is linear in the number of cross edges of $G$. □

Since both phases of $DI\_CHECK$ are linear in the number of edges of $G$ we can conclude:

Corollary 4.22: Algorithm $DI\_CHECK(G=(V, E), T)$ is correct and has $O(1)$ time complexity. □

Theorem 4.23: If $T$ is a spanning tree which is not a $T$–DFS in $G=(V, E)$ then algorithm $DI\_CHECK$ can supply an $O(1)$ time complexity proof for that fact.

Proof: Let $G''=(V, \overrightarrow{T} \cup \overrightarrow{G''})$ be the minimal minor digraph of $(G, T)$. If $T$ is not a $T$–DFS in $G$ then there is a circuit in $G''$ which contains only cross edges. It is easy to modify the algorithm (without affecting its complexity) in order to find a set of edges $\{e'_1, e'_2, \ldots, e'_p\} \subseteq \overrightarrow{G''}$ which form a simple circuit in $G''$ and to recognize a set of cross edges $\overrightarrow{C'} = \{e_1, e_2, \ldots, e_p\}$ in $G$ such that $R[e_i]=e'_i$ for all $i=1, 2, \ldots, p$. Hence, $G'=(V, \overrightarrow{T} \cup \overrightarrow{C'})$ is a subgraph of $G$ with $O(1)$ edges where $T$ is a spanning tree which is not a $T$–DFS in $G'$. By checking that $G'$ is a subgraph of $G$ and rerunning the algorithm $DI\_CHECK(G', T)$ one can have an $O(1)$ time proof that $T$ is not a $T$–DFS in $G$. □

Corollary 4.24: Algorithm $DI\_CHECK(G=(V, E), T)$ has an optimal $O(1)$ time complexity. In addition, in the case of a negative answer the algorithm outputs a proof for that fact that can be verified in an optimal $O(1)$ time complexity.

Proof: Assume $G$ has at least two cross edges relative to $T$.

Clearly, we can not have a proof that $T$ is a $T$–DFS in $G$, unless we go over all the edges in $G - \overrightarrow{T}$ (every edge that we ignore may be a cross edge which causes the creation of a circuit in the minimal minor digraph of $(G, T)$). Hence, $O(1)$ is an optimal time complexity for a positive verification, and therefore it is an optimal time complexity for the algorithm itself.

As for the verification of a negative answer, consider the infinite family of pairs $(G_i, T_i)$ as follows:

\[ T_i = \{ (v_i, v_j) : 2\leq i \leq l \} \]
Similarly to the proof of Corollary 3.10, one can see that we have to scan all the non tree edges (there are $O(1V_{i1})$ such edges) to have a negative proof. □

5. PARALLEL RECOGNITION of DFS TREES in DIGRAPHS with REFINED VERIFICATIONS

In this section we describe how to implement algorithm $DI\_CHECK(G, T)$ in $O(\log^2 n)$ time complexity with $O(n^{2.376})$ processors on a CREW PRAM. In addition, the parallel implementation supplies proofs which have a better complexity than that of the algorithm (refined verifications).

In the case of a negative answer the algorithm outputs a proof for that fact that can be verified in $O(t)$ time complexity with $O(\frac{n}{t})$ processors, where $t \geq \log n$, on a CREW PRAM. In the case of a positive answer the algorithm outputs a proof for that fact that can be verified in $O(t)$ time complexity with $O(\frac{m}{t})$ processors, where $t \geq \log n$, on a CREW PRAM. In both cases the verification has an optimal speed-up.

A parallel algorithm for recognizing a DFS tree in a digraph was independently presented in [SchVit].

5.1 Parallel implementation of algorithm $DI\_CHECK$

The algorithm has two phases which are identical to the phases of the algorithm presented in section 4. In phase one we build the minimal minor digraph and in phase two we check whether it is acyclic.

Parallel implementation of phase one (algorithm $BMM$):

The implementation of phase one (algorithm $BMM$) is done as follows:

(i) Compute $z(e)$, the lowest common ancestor of $x$ and $y$ in $T$ for every non tree edge $e = x \rightarrow y$ in $G$.

(ii) Delete the back and forward edges of $G$ (step (1) of algorithm $BMM$). Note that $e = x \rightarrow y$ is a forward edge if and only if $z(e) = x$ and $e$ is a back edge if and only if $z(e) = y$.

(iii) (Step (2) of $BMM$) Replace every cross edge $x \rightarrow y$ by the cross edge $\hat{x} \rightarrow \hat{y}$ where $x$ and $y$ are in the brother subtrees $T_x^-$ and $T_y^-$, respectively.

Lemma 5.1.1: Algorithm $BMM(G, T)$ can be implemented in $O(\log n)$ time complexity where $O\left(\frac{m}{\log n}\right)$ processors are used on a CREW PRAM.

Proof: The implementation of (i) is done by using the algorithm in [SchVis]. The implementation of (ii)
is trivial and can be done in $O(1)$ time complexity with $O(m)$ processors. The implementation of (iii) follows from Lemma 3.7. □

Parallel implementation of phase two (algorithm CAD):

Phase two can be implemented as follows: Let $A$ be the adjacency matrix of a digraph $G = (V, E)$. $G$ is acyclic if and only if $A^{2^k} = 0$ for $k = \lceil \log n \rceil$. By the result of [CW] for matrix multiplication and the result in [PR, Theorem A.1] for parallel implementation of matrix multiplication, $A^2$ can be computed in $O(\log n)$ time complexity using $O(n^{2.376})$ processors on a CREW PRAM. Hence, performing matrix multiplication $k$ times we can get $A^{2^k}$ and decide whether $G$ is acyclic.

Proposition 5.1.2: Given a digraph $G$ with $n$ vertices we can check in $O(\log^2 n)$ time complexity with $O(n^{2.376})$ processors on a CREW PRAM whether $G$ is a acyclic. □

Corollary 5.1.3: Given a digraph $G$ with $n$ vertices and a directed spanning tree $T$ in $G$ we can check in $O(\log^2 n)$ time complexity with $O(n^{2.376})$ processors on a CREW PRAM whether $T$ is a $T$-DFS in $G$.

Proof: Follows immediately from Lemma 5.1.1 and Proposition 5.1.2. □

Note: Improving the complexity of checking the acyclicity of a digraph will improve the complexity of our solution.

5.2 Refined verifications of DI_CHECK

In the following part of this section we show how the parallel implementation of algorithm $\text{DI\_CHECK}(G = (V, E), T)$ can be modified, without affecting the complexity stated in Corollary 5.1.3, to supply a proof for a negative answer that can be verified in $O(t)$ time complexity with $O\left(\frac{n^2}{t}\right)$ processors, where $t \geq \log n$, on a CREW PRAM.

Let $G$ be a digraph which is irreducible relative to a spanning tree $T$. Let $B = A^{2^k}$, where $A$ is the adjacency matrix of $G$ and $k = \lceil \log |V| \rceil$. Since $T$ is not a $T$-DFS in $G$ then there is an entry in the matrix $B$, $B(i, j) > 0$ which corresponds to a path of length $2^k$ in $G$. The following algorithm finds one such path.
FP \((G=(V,E))\) \{Find Path\}

input: A digraph \(G\) which is not acyclic. \(G\) is represented by its adjacency matrix \(A\). In addition, we are given the set of matrices \(\{A^{2^l} \mid l=0, \ldots, \lfloor \log |V| \rfloor \}\) that were computed in the parallel implementation of phase two of algorithm \(DI\_CHECK\) (see the parallel implementation of phase two).

output: A path of length \(2^k\) in \(G\), where \(k=\lfloor \log |V| \rfloor\).

begin \{of the algorithm\}

(1) Find \(i, j\) such that \(A^{2^k}(i, j) > 0\);

(2) Output the path created by the execution of \(RS(k, i, j)\) \{the procedure \(RS\) is given below.\}

end \{of algorithm \(FP\)\}.

\(RS(l, i, j)\) \{Recursive Search\}

\{It is a recursive subroutine of \(FP\)\}.

input: A number \(l (0 \leq l \leq \lfloor \log |V| \rfloor)\) and two vertices \(v_i, v_j (1 \leq i, j \leq |V|)\) such that there is a path of length \(2^l\) from \(v_i\) to \(v_j\) in \(G\).

output: A path (a sequence of edges) of length \(2^l\) from \(v_i\) to \(v_j\).

begin \{of the algorithm\}

(1) If \(l=0\) then return the path which is the edge \(v_i \rightarrow v_j\);

(2) \{ \(l > 0\) \} find \(q\) such that \(A^{2^{l-1}}(i, q) > 0\) and \(A^{2^{l-1}}(q, j) > 0\);

{ There is at least one \(1 \leq q \leq |V|\) such that there are paths of length \(2^{l-1}\) from \(v_i\) to \(v_q\) and from \(v_q\) to \(v_j\).}

(3) Compute in parallel \(\pi_{l, i, q}^{-1} := RS(l-1, i, q)\) and \(\pi_{l, q, j}^{-1} := RS(l-1, q, j)\);

(4) Return the path \(\pi_{l, i, j}\) which is the concatenation of the paths \(\pi_{l, i, q}^{-1}\) and \(\pi_{l, q, j}^{-1}\) created by the executions of \(RS\) in (3)

end \{ of the subroutine \(RS\)\}.
Recall that the set of matrices $A^{2^l}$ was already computed in the parallel implementation of algorithm DI_CHECK for all $l = 0, \ldots, \lceil \log |V| \rceil$. This leads us to the following proposition.

**Proposition 5.2.1:** Algorithm FP $(G=(V,E))$ can be implemented in $O(\log^2 |V|)$ time complexity with $O\left(\frac{|V|^2}{\log |V|}\right)$ processors on a CREW PRAM.

**Proof:** It is clear that Step (1) of the algorithm can be done in $O(\log |V|)$ time complexity with $O\left(\frac{|V|^2}{\log |V|}\right)$ processors on a CREW PRAM. As for the subroutine RS one can see that the depth of the recursive calls of the subroutine to itself is $O(\log |V|)$ (since $l$ goes from $\lceil \log |V| \rceil$ down to 0). Step (2) of RS can be done in $O(\log |V|)$ time complexity with $O\left(\frac{|V|^2}{\log |V|}\right)$ processors on a CREW PRAM: Go over $|V|$ pairs of entries in the matrix of the form $(A^{2^l-1}(i,h), A^{2^l-1}(h,j))$, for $1 \leq h \leq |V|$, using one processor for a set of $O(\log |V|)$ pairs, and choose (in $O(\log |V|)$ time) one pair in which the two entries are greater than zero. Since the subroutine has no more than $O(|V|)$ executions at the same time (the exact number of parallel executions is $2^{\lceil \log |V| \rceil} - 1$ and the maximum is obtained when $l = 0$) therefore each recursive level can be done in $O(\log |V|)$ time complexity with $O\left(\frac{|V|^2}{\log |V|}\right)$ processors. Since the recursive depth is $O(\log |V|)$ the proof of the proposition is completed. □

**Theorem 5.2.2:** If $T$ is a directed spanning tree which is not a $T$-DFS in a digraph $G=(V,E)$ then the parallel implementation of algorithm DI_CHECK can supply a proof for that fact that can be verified in $O(t)$ time complexity by $O\left(\frac{|V|^2}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM.

**Proof:** In the first part of the proof we describe the extra information produced by the algorithm for the purpose of the verification. In the second part we prove the complexity of the verification (as stated in the theorem).

Let $T$ be a spanning tree which is not a DFS tree in a digraph $G=(V,E)$ and let $G''=(V,E'')$ be a minimal minor digraph of $(G, T)$. By using algorithm FP we find a sequence of cross edges (with possible repetition) $\hat{C}'' = e_1'', e_2'', \ldots, e_p''$ in $G''$ which form a path of length $p = 2^{\lceil \log |V| \rceil}$ in $G''$. The parallel implementation of the algorithm can be modified without affecting its complexity to number each
appearance of an edge according to its place in the above sequence such that for every appearance of an edge \( e_i'' \) in \( \overrightarrow{C''} \) we have \( \text{num} (e_i'') = i \). It is also easy to modify the parallel implementation of algorithm \( \text{DI}_\text{CHECK} (G=(V, E), T) \) (without affecting its complexity) in order to find a minimal subset of cross edges \( \overrightarrow{C'} = \{ e_1, e_2, \ldots, e_h \} \) in \( G \) that satisfies the following: for every \( i = 1, \ldots, p \) there exist a \( j \), \( 1 \leq j \leq h \) such that \( R[e_j] = e_i'' \). Hence, \( G'=(V, \overrightarrow{T} \cup \overrightarrow{C'}) \) is a subgraph of \( G \) with \( O(|V|) \) edges where \( T \) is a spanning tree which is not a \( T-\text{DFS} \) in \( G' \) (and therefore, by Corollary 4.6 - \( T \) is not \( T-\text{DFS} \) in \( G \)).

The verification consists of the following steps:

1. A verification that \( G' \) - as described above - is a subgraph of \( G \).
2. By using the algorithm in \([CV]\) we assign the vertices in \( T \) preorder numbering and postorder numbering.
3. A verification that for every edge \( e_i'' = \overrightarrow{x \rightarrow y} \) in \( \overrightarrow{C''} \) there exists an edge \( e_j = x \rightarrow y \) in \( G' \) such that \( R[e_j] = e_i'' \). Recall, that for every edge \( e_i'' \) the parallel implementation of the algorithm \( \text{DI}_\text{CHECK} \) supplied the edge \( e_j \) so it is left to verify only that \( R[e_j] = e_i'' \). By using the preorder numbering and postorder numbering of the vertices of \( T \) (which were calculated in the last step) we check whether all the following four conditions hold: (i) \( \overrightarrow{x} \) is an ancestor of \( x \) in \( T \). (ii) \( \overrightarrow{y} \) is an ancestor of \( y \) in \( T \). (iii) \( \overrightarrow{y} \) and \( \overrightarrow{x} \) have a common father in \( T \). (iv) \( \overrightarrow{x} \neq \overrightarrow{y} \).
4. A verification that the sequence of edges \( \overrightarrow{C''} \) is a path of length \( p \) in \( G'' \). Since the algorithm has numbered each appearance of an edge according to its place in this path the verification is easily done in the appropriate complexity.

All the four steps above can be implemented in the complexity stated in the theorem and hence we are done. \( \Box \)

**Theorem 5.2.3**: If \( T \) is a \( T-\text{DFS} \) in a digraph \( G=(V, E) \) then the parallel implementation of algorithm \( \text{DI}_\text{CHECK} \) (with a slight modification) supplies an optimal speed-up verification for that fact that can be verified in \( O(t) \) time complexity with \( O\left(\frac{|E|}{t}\right) \) processors, for \( t \geq \log |V| \), on a CREW PRAM.
In order to prove Theorem 5.2.3 we need the following proposition.

**Proposition 5.2.4:** If $T$ is a $T$-DFS in a digraph $G = (V, E)$ then the parallel implementation of algorithm $DI_{\text{CHECK}}$ (with a slight modification) finds a $DFS-T$-order in $T$ induced by $f$ which is compatible in $\hat{G} = (V, \overrightarrow{T} \cup \overrightarrow{F} \cup \overrightarrow{B} \cup \overrightarrow{C})$, where $\overrightarrow{T}, \overrightarrow{F}, \overrightarrow{B}$ and $\overrightarrow{C}$ are as described in Proposition 4.5.

**Proof:** The implementation is as follows:

1. Build $G' = (V, \overrightarrow{T} \cup \overrightarrow{C'})$ - the minimal minor digraph of $(G, T)$.

2. Find a $DFS-T$-order induced by $f$ which is compatible in $G' = (V, \overrightarrow{T} \cup \overrightarrow{C'})$ (where $\overrightarrow{C'}$ are the edges of $\overrightarrow{C'}$ in the reverse direction). By the proof of Lemma 4.11 this order is compatible in $\hat{G}$.

The implementation of (2) is taken from [SchVit] and we outline here the main steps of it: A topological sort of the vertices which are brothers in $T$ is done. The topological sort enables us to arrange the incidence list of the edges of $T$ such that by implementing the algorithm from [CV] on this list we get a $DFS-T$-order (preorder numbering) induced by $f$ which is compatible in $\hat{G}$. In addition, by the techniques in [CV] we compute a postorder numbering of the vertices of $T$ according to the arranged incidence list. This process does not affect the complexity of the parallel implementation of algorithm $DI_{\text{CHECK}}$ as stated in Corollary 5.1.3. □

**Proof of Theorem 5.2.3:** One can check that the numberings given by the algorithm are correct by rerunning the algorithm from [CV]. After this check, the compatibility of the given $DFS-T$-order (in $\hat{G}$) can be also checked in the appropriate complexity as follows:

1. For every edge check to which of the following groups $(T, F, B, C)$ it belongs. The methods are analogous to the methods that appear in the proofs of Theorem 3.8. and Lemma 5.1.1.

2. Check that there is no (cross) edge for which the order is not compatible.

Hence, by Proposition 4.5, one can have a verification that $T$ is a $T$-DFS in $G$, the complexity of which is as stated in this theorem. □

**Corollary 5.2.5:** The parallel implementation of algorithm $DI_{\text{CHECK}} (G = (V, E), T)$ has an $O(\log^2 |V|)$ time complexity and uses $O(|V|^{2.376})$ processors on a CREW PRAM. In addition, in the
case of a negative answer the algorithm outputs a proof for that fact that can be verified in $O(t)$ time complexity with $O\left(\frac{|V|}{t}\right)$ processors, for $t \geq \log |V|$, on a CREW PRAM. In the case of a positive answer the algorithm outputs a proof for that fact that can be verified in $O(t)$ time complexity with $O\left(\frac{|E|}{t}\right)$ processors, for $t \geq \log |V|$, on a CREW PRAM. Both verifications have an optimal speed-up.

Proof: By Proposition 5.2.1 and Proposition 5.2.4 the modifications of the algorithm - the addition of algorithm $FP$ in the case of a negative answer and finding a compatible order in the case of a positive answer - do not change the complexity of the algorithm. Hence, the correctness follows from Corollary 5.1.3, Theorem 5.2.2, Theorem 5.2.3 and the proof of Corollary 4.24. □

6. DISCUSSION and OPEN PROBLEMS

A parallel algorithm for recognizing an undirected $DFS$ tree in an undirected graph $G=(V, E)$ was presented. The algorithm has $O(t)$ time complexity and uses $O\left(\frac{|E|}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM. In addition, the algorithm supplies an optimal speed-up verification to justify a negative answer. This verification has $O(t)$ time complexity and uses $O\left(\frac{|V|}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM.

A linear algorithm for recognizing a directed $DFS$ tree in a digraph $G=(V, E)$ is presented. The algorithm supplies an $O(|V|)$ time proof to justify a negative answer. The algorithm has an efficient parallel implementation which has $O(\log^2 |V|)$ time complexity and uses $O(|V|^{2.376})$ processors on a CREW PRAM. This implementation supplies a proof to justify a negative answer that can be verified in $O(t)$ time complexity with $O\left(\frac{|V|}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM. If the answer is positive then this implementation supplies a proof to justify that fact that can be verified in $O(t)$ time complexity with $O\left(\frac{|E|}{t}\right)$ processors, where $t \geq \log |V|$, on a CREW PRAM. Both proofs are optimal in the sense that we can not improve the time-processor product.

The sequential algorithm for the directed case has an optimal ($O(|E|)$) time complexity and the parallel algorithm for the undirected case has an optimal speed-up. However, the time-processor product
in the parallel algorithm for the directed case is $O(|V|^{2.376})$ (see table). It is left open to decide whether there is an NC algorithm which improves the speed-up of the parallel solution for digraphs.

A common generalization of the problems presented here and in [KO-a] is to decide whether a given spanning tree $T$ in a mixed graph $G$ is a DFS tree. A trivial solution is to use algorithm $DI_{CHECK}$ for every possible orientation of $T$ that makes $T$ a directed tree (where every non directed edge in $G$ is replaced by two anti parallel edges). It is left open to find more efficient algorithms (sequential and parallel) to perform this task.

ACKNOWLEDGMENT

We thank Baruch Schieber for providing Lemma 3.7 and for improving the implementation complexity of algorithm $PAR_{CHECK}$ that we got in a previous manuscript, as stated in Theorem 3.8. This also enables us to simplify the optimal negative verification we had for this algorithm.

We thank Michael Kaminski for pointing out some references on the complexity of matrix multiplication.

This research was partially supported by Technion V.P.R. Fund - Coleman Cohen Research Fund.

REFERENCES


