RECOGNITION OF DFS TREES: SEQUENTIAL AND PARALLEL ALGORITHMS WITH REFINED VERIFICATIONS
(Extended Version of Sects. 4, 5, 6 of TR#508)

by

E. Korach and Z. Ostfeld

Technical Report #520
August 1988
RECOGNITION of DFS TREES: SEQUENTIAL and PARALLEL
ALGORITHMS with REFINED VERIFICATIONS

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, a parallel algorithm which solves this problem in \( O(\log |V|) \) time complexity and uses \( O(|E|) \) processors on a CREW PRAM, is presented. 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 positive then it outputs a proof for that fact that can be verified in \( O(t) \) time complexity with \( O(\frac{|E|}{t}) \) 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 \( O(|V|^3) \) time proof for that fact and the parallel implementation supplies a proof for that fact that can be verified in \( O(\log |V|) \) time complexity with \( O(|V|^1) \) processors 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.

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

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

The Depth First Search (DFS) algorithm is one of the basic techniques 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, beside 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.

In section 3, we present a parallel algorithm for solving this problem. This algorithm has \( O(\log |V|) \) time complexity and uses \( O(|E|) \) processors on a concurrent read exclusive write parallel random access machine (CREW PRAM). In addition, if the decision of the algorithm is positive then it outputs a proof 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. The speed-up of this verification is optimal in the sense that the time-processor product is \( O(|E|) \), which is the time required by an optimal sequential verification (any verification must go over all the edges). Since the verification has a better complexity than that of the algorithm itself we call this property of the algorithm refined verification.

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 assumption: 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. 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 case is presented. In addition, if $T$ is not a DFS tree in $G$ then the algorithm supplies an $O(|V|)$ time proof for that fact. 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 rerun the same algorithm on $G'$ one can have an $O(|V|)$ time proof. So in a sense, this algorithm is a "self refinement" algorithm - complexity wise - in the same spirit of [LP].

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 DFS tree in $G$ then the algorithm supplies a verification for that fact that can be verified in $O(log |V|)$ time complexity by $O(|V|)$ processors on a CREW PRAM (a notable refined verification). If $T$ is a DFS tree in $G$ then the algorithm supplies a verification for that fact that can be verified in $O(t)$ time complexity with $O(\frac{|E|}{t})$ processors, where $t \geq log |V|$, on a CREW PRAM (a verification with an optimal speed-up).

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

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 proof to be sure that no error occurred during the communication process. In a situation where a station is busy or not powerful enough to solve the problem, but it can afford verification of a refined proof, it will be best for it to use the central computers.

The fact that our algorithm supplies a proof to justify a negative answer to the problem in the directed case, which is 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.

### 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(\log</td>
<td>V</td>
<td>) )</td>
</tr>
<tr>
<td>Undirected</td>
<td>Parallel</td>
<td>Processor</td>
<td>( O(</td>
<td>E</td>
<td>) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Product</td>
<td>( O(</td>
<td>E</td>
<td>\log</td>
</tr>
<tr>
<td>Directed</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(\log ^2</td>
<td>V</td>
<td>) )</td>
</tr>
<tr>
<td>Directed</td>
<td>Parallel</td>
<td>Processor</td>
<td>( O(</td>
<td>V</td>
<td>^\omega) )**</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Product</td>
<td>( O(</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 \( \exists \varepsilon > 0 \) such that \( \omega < 2.376 - \varepsilon \).

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

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 $O(\log |E|)$ time complexity and uses $O(|E|)$ 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: 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 algorithms in [KO a] and independently in [HN].

In this work we present a parallel algorithm which solves this problem in $O(\log |V|)$ time complexity and uses $O(|E|)$ processors on a concurrent read exclusive write parallel random access machine (CREW PRAM). In addition, if the decision is positive then 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. This proof has 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 a spanning tree in $G = (V,E)$. 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:

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

input: An undirected graph $G$ and an undirected spanning tree $T$ in $G$.

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 $G \}$.

variables: $f$, $count$, $sum$. {These variables are three arrays indexed by the vertices of $V$.}

begin { of the algorithm}

(1) Choose a vertex $s \in V$ and compute $T_s$.

(2) For every vertex $x \in V$ set $f(x)$ to be the father of $x$ in $T_s$ ($f(s) =$ null) and set $count(x)$ to be 0.

(3) For every edge $e \in E - T$

begin

(3.1) If $e = (u, v)$ is a cross edge in $T_s$ then

begin

$\text{count}(u) := \text{count}(u) - 1$;

$\text{count}(v) := \text{count}(v) - 1$;

$\text{count}(s) := \text{count}(s) + 1$

end { of (3.1)}

(3.2) Else $(e = (x, y)$ is a back edge in $T_s$)
begin
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-$\pi_{u,v}$.
$\text{count}(v) := \text{count}(v) - 1$;
$\text{count}(w) := \text{count}(w) + 1$
end { of (3.2) }
end { of (3) }.

(4) For every vertex $x \in V$ compute $\text{sum}(x)$ which is the sum of values of $\text{count}(u)$ 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
$\text{sum}(x) = 0$. $T$ is a $T$-DFS in $G$ if and only if there
is at least one vertex $x \in V$ such that $\text{sum}(x) = 0$.

end { of algorithm PAR_CHECK }.

**Theorem 3.6:** Algorithm $\text{PAR CHECK} (G = (V, E), T)$ 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 that $e$ is a cross edge in $T_s$. Since we add one to $\text{count}(s)$ it is clear that if $x$ is either a
descendant of $u$ or a descendant of $v$ then $\text{sum}(x)$ is not affected by the changes in step (3.1). Otherwise
$e$ contributes one to $\text{sum}(x)$. Now assume that $e$ is a back edge with respect to $T_s$, 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 $\text{sum}(x)$
only if $x$ is a descendant of $w$ and is not a descendant of $v$. In the latter case $e$ contributes one to $\text{sum}(x)$.
Hence, by Proposition 3.3, for every vertex $x \in V$ $\text{sum}(x)$ is the number of cross edges in $T_x$ and by
Corollary 3.5 the decision of the algorithm is correct. □

**Theorem 3.7:** 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 $m = |E|$).

**Proof:** Step (1) of the algorithm is computed by $O(n)$ processors in $O(\log n)$ time complexity using the
Euler tour techniques as presented in [TV]. The algorithm in [TV] replaces every tree edge by two anti
parallel edges and then creates an Euler circuit 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. The techniques in [TV] with the same time and processor complexity are used in step (3), as shown in [Vi], to form a data-structure which enables the retrieval of the lowest common ancestor of any pair of vertices in \( O(\log n) \) time by 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 \( y \)). For a back edge \((u, v)\) in step (3.2) of the algorithm, it is easy to find the vertex \( w \in V \) which is \( \text{second-} \pi_{u,v} \) by using the following modification of the algorithm in [Vi]: Consider the above Euler path that is obtained from the tree \( T_s \). Consider the last part of that path that starts with the edge \( v \to f(v) \) (where \( f(v) \) is the father of \( v \) in \( T_s \)) and find the first time that \( u \) appears in this subpath. It is clear that in this case \( u \) is the head of the edge \( w \to u \), and so we find \( w \) as required. This modification does not change the complexity of the algorithm in [Vi]. The parallel additions and subtractions are easily done in \( O(\log n) \) time complexity with \( O(m) \) processors (we compute the value of \( \text{count} \) which is an array of length \( n \); the number of items to be added and subtracted is \( O(m) \)). The computation of step (4) is done in \( O(\log n) \) time complexity with \( O(n) \) processors by using a "doubling" technique [Wy] in the following way. For each vertex \( v \in V \), we initialize \( \text{sum}(v) := \text{count}(v) \) and then repeat the following step, in parallel on all the vertices of \( V \), until all of the \( f \) values are null: If \( f(v) \) is not null then set \( \text{sum}(v) \) to \( \text{sum}(v) + \text{sum}(f(v)) \) and replace \( f(v) \) by \( f(f(v)) \). It is obvious that we repeat this step no more than \( \lceil \log n \rceil \) times (since the depth of \( T_s \) is at most \( n-1 \)). Finally, the decision can be done in \( O(\log n) \) time complexity with \( O(n) \) processors on a CREW PRAM.

\[ \square \]

**Theorem 3.8:** If \( T \) is a \( T-\text{DFS} \) in a graph \( G = (V, E) \) then algorithm \( \text{PAR_CHECK} \) outputs a proof for that fact that can be verified in \( O(t) \) time complexity with \( O(\lceil E \rceil \log V) \) processors, for \( t \geq \log |V| \), on a CREW PRAM.

**Proof:** If the decision of the algorithm is positive then the algorithm can output a vertex \( x \in V \) such that \( T_x \) is a \( T-\text{DFS} \) in \( G \). The fact that \( T_x \) is a \( T-\text{DFS} \) in \( G \) can be verified as follows:

(1) We give the vertices in \( T_x \) a preorder numbering and a postorder numbering. This is done in \( O(\log |V|) \) time complexity with \( O(|V|) \) processors on a CREW PRAM as shown in [TV].
(2) For every non tree edge \((u,v)\) check whether it is a cross edge in \(T_x\). Note that \(u\) is an ancestor of \(v\) if and only if \(\text{preorder}(u) < \text{preorder}(v)\) and \(\text{postorder}(u) > \text{postorder}(v)\). Hence, we can perform this check in \(O(t')\) time complexity with \(O(\frac{|E|}{t'})\) processors, for every \(t' \geq 1\), on a CREW PRAM.

By (1) and (2) we can check in the complexity stated in the theorem that there are no cross edges in \(T_x\) and by Proposition 3.4 we have a proof that \(T_x\) is a \(T-DFS\) in \(G\). □

The results of Theorem 3.7 and Theorem 3.8 can be summarized in the following:

**Corollary 3.9:** Algorithm \(\text{PAR}_\text{CHECK}\) can be implemented in \(O(\log |V|)\) time complexity with \(O(E)\) processors on a CREW PRAM. In addition, if the decision of the algorithm is positive then it supplies an optimal speed-up verification for that fact that can be verified in \(O(t)\) time complexity with \(O(\frac{|E|}{t})\) processors, where \(t \geq \log |V|\), on a CREW PRAM. □

### 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 an \(O(|V|)\) time proof to justify its decision. 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 Corollary 4.6 - \(T\) is not a \(T-DFS\) in \(G\)).

(ii) The algorithm itself.

By checking that \(G'\) is a subgraph of \(G\) and rerun the same algorithm on \(G'\) one can have an \(O(|V|)\) time proof. So in a sense, this algorithm is a "self refinement" algorithm - complexity wise - in the same spirit of [LP].

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

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

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

(iv) **Cross edges** ($\mathcal{C}$): The rest of the edges in $E$.

**Definition 4.2:** Let $V$ be a set of vertices. We say that $V$ has an order induced by $f$ if and only if $f : V \rightarrow \{1, 2, ..., |V|\}$ 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 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 $\hat{G}=(V, \mathcal{T} \cup \mathcal{F} \cup \mathcal{B} \cup \mathcal{C})$, where $\mathcal{T}$ are the tree edges, $\mathcal{F}$ are the forward edges, $\mathcal{B}$ are the back edges with the reverse direction and $\mathcal{C}$ 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 $e \in \mathcal{T}$ 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 following claim is proved by induction on $|E|$, for every given $|V|$.

**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 $\hat{G}$, 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.

**Proof of the claim:** Clearly, since $G$ contains $T$ then $|E| \geq |V|-1$. It is easy to see that the claim is true for $|E|=|V|-1$. Assume that the claim is true for all digraphs with $|E| \leq m$ for a given
Let $T$ be a spanning tree in a graph $G$ with $m+1$ edges labeled as above, and let $e_{m+1} = u \rightarrow v$ be an edge with the label which is largest lexicographically (obviously, it is not a tree edge). Let $G' = G \setminus e_{m+1}$ (the result of the deletion of $e_{m+1}$ from $G$ i.e. $G' = (V,E - e_{m+1})$). By the induction hypothesis, the $M-DFS$ algorithm in $G'$ gives $T$ as its DFS tree and the vertices are discovered in the appropriate order. Now consider a run of the $M-DFS$ in $G$ and consider the first time the edge $e_{m+1}$ is used. Until that moment, the run is identical to the run in $G'$. We have two possibilities:

1. $f(u) > f(v)$. By the induction hypothesis $v$ has already been discovered.
2. $f(u) < f(v)$. In this case, since $f$ induces a compatible order in $\hat{G}$ then $e_{m+1} \in \hat{T} \cup \hat{F}$. At this moment, since $e_{m+1}$ has the largest label in $G$ and since $e_{m+1} \notin \hat{T}$ it is clear that all the subtrees rooted at $u$ have already been visited. Hence all the descendants of $u$ have already been discovered (in particularly $v$).

In both cases $v$ has already been discovered and therefore $e_{m+1}$ is marked "used", the center of activity remains in $u$ and the rest of the execution is the same as in $G'$. This implies that the DFS tree will be the same as in $G'$ and the vertices are discovered in the same order.

This completes the proof of the claim and hence the proof of the proposition. □

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 compatible in $\hat{G}$. Since $G'$ is a subgraph of $G$ then the same $DFS-T-order$ is also compatible in $\hat{G}' = (V,\hat{T} \cup \hat{F}' \cup \hat{B}' \cup \hat{C}')$ where $\hat{T}$ are the tree edges, $\hat{F}'$ are the forward edges in $G'$, $\hat{B}'$ are the back edges in $G'$ with the reverse direction and $\hat{C}'$ are the cross edges in $G'$ with the reverse direction (note that $\hat{G}'$ 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 sub trees $T_x$ 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 operations $\Phi_T(G)$:

(4.8.1) If $e \in \overline{F} \cup B$ then $\Phi_1(G, T, e) = (V, E - e) \in \Phi_T(G)$ (i.e. $e$ is deleted).

(4.8.2) If $e \in \overline{C}$ then $\Phi_2(G, T, e) = (V, E - e \cup \{\hat{x} \rightarrow \hat{y}\}) \in \Phi_T(G)$, where $\hat{x} \in V$ is second-ancestor and $\hat{y} \in V$ is second-ancestor (i.e. $e$ is replaced by another cross edge $\hat{x} \rightarrow \hat{y}$ where $x$ and $y$ are in brother subtrees $T_x$ and $T_y$, 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 $G' \in \Phi_T^+(G)$ then $\Phi_T(G') \subseteq \Phi_T^+(G)$.

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-tree 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 $x \rightarrow y \in E$ 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 $\overline{B} \cup \overline{F}$ is either deleted or remains in $\overline{B} \cup \overline{F}$. 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 $E'$ are deleted (according to 4.8.1) and every cross edge $x \rightarrow y \in E$, where $x$ and $y$ are in the brother sub trees $T_i$ and $T_j$, respectively, has a unique image $x \rightarrow y \in E'$ in the minimal minor digraph (according to 4.8.2). Hence we can get a minimal minor digraph after performing at most $|E'| + |E| + |C|$ 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 $x \rightarrow y$, $d(x) < d(y)$ 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 $x \rightarrow y$ 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 - $\hat{G} = (V, \hat{T} \cup \hat{C})$ has 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 $g : V \rightarrow \{1,2,\ldots,|V|\}$ such that for every edge $x_i \rightarrow x_j \in E$, $g(x_i) > g(x_j)$ (e.g. $g$ is the result of a topological sort in $G$). Let us assume that every edge $e = u \rightarrow v$ is labeled by the pair $(B, H)$ where (i) $B = 0$ if $e \in \hat{T}$ and $B = 1$ otherwise. (ii) $H = g (v)$.

We can prove the following claim 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. 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 $G$.

proof of the claim: Let $T$ be a directed tree on $V$ vertices. Clearly, since $G$ contains $T$ then $|E| \geq |V| - 1$. It is easy to see that the claim is true for $|E| = |V| - 1$. Assume that the claim is true for all digraphs with $|E| \leq m$, for a given $m \geq |V| - 1$. Let $G$ be an acyclic digraph, which is irreducible relative to $T$ and let $g$ be a bijection as described above. Assume $G$ has $m+1$ edges labeled as above. Let $e_{m+1} = x_i \rightarrow x_j$ be an edge with the label which is largest lexicographically (obviously, it is not a tree edge),
and let $G' = G \setminus e_{m+1}$. Clearly, $G'$ is an acyclic digraph which is irreducible relative to $T$ and for every edge $x_i \rightarrow x_j$ in $G'$, $g(x_i) > g(x_j)$. Hence, by the induction hypothesis - the $M-DFS$ algorithm in $G'$ gives $T$ as its $DFS$ tree.

Now consider a run of the $M-DFS$ in $G$ and consider the first time the edge $e_{m+1}$ is used. Until that moment, the run is identical to the run in $G'$. The edge $x_i \rightarrow x_j$ is chosen after $x_i$ has already been discovered. Since $G$ is irreducible relative to $T$ then - by Observation 4.14 - $x_i$ and $x_j$ have a common father $z$ in $T$. By the induction hypothesis we know that $x_i$ was discovered by a tree edge $z \rightarrow x_i$ and hence $z$ was discovered as well. Since the tree edge $z \rightarrow x_i$ was chosen by the $M-DFS$ algorithm and since $g(x_j) < g(x_i)$ we know that the tree edge $z \rightarrow x_j$ had been chosen before the tree edge $z \rightarrow x_i$. This implies that $x_j$ has already been discovered and therefore $x_i \rightarrow x_j$ is a cross edge in the $M-DFS$ run in $G$.

The rest of the run is identical to that in $G'$ and hence we get $T$ as a $DFS$ tree in $G$.  

The algorithm for checking whether a given directed spanning tree $T$ is a $T-DFS$ 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 $G$ is acyclic if and only if for every $DFS$ tree in $G$ there is no back edge.

**Proof:** The *only if* part is obvious since any back edge in a $DFS$ tree creates a cycle. As for the *if* part, since $G$ is cyclic 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 every $DFS$ tree in $G$. Hence, in every $DFS$ tree there is at least one back edge which enters the root of the subtree defined by the vertices of $C$.  

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 \(G_1\).

(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. \(G_2 = \text{BMM}(G, T)\)).

end (of phase one.)

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

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 \(\text{DFS}\) tree in \(G\).

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

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 \(\text{DI\_CHECK}\)): \(T\) is a \(T\text{-DFS}\) in \(G\) if and only if \(G_2\) is acyclic.

Lemma 4.19: \(\text{BMM}(G, T)\) 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 \(G_1\) (i.e., the cross edges of \(G\)).

We now present an efficient sequential implementation of algorithm \(\text{BMM}\).

Step (1) of the algorithm is done by using a \(\text{DFS}\) 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 sub trees \(T_x\) and \(T_y\), respectively, and \(R\) is an array indexed by the cross edges of \(G_1\) (i.e., 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 [AHU]). The results are organized in an array \(\text{LCA}\) indexed by the cross edges of \(G_1\).

After computing \(\text{LCA}\) we use a modification of the \(\text{DFS}\) algorithm for computing \(R\) as follows:
\[ NCE(G_1, T, LCA) \quad \text{[compute the New Cross Edges]} \]

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 sub trees \( T_x \) 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 or the head of \( e \) do

begin

\( z := LCA[e]; \hat{v} := s(z); \) \{Clearly, \( s(z) \) is second-ancestor \}

If \( v \) is tail(e) then \( \text{tail}(R[e]) := \hat{v}; \)

Else \{\( v \) is head(e)\} \( \text{head}(R[e]) := \hat{v} \)

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 \( \bar{R} \), which is the result of eliminating duplications in \( R \), and create a digraph \( G_2 = (V, \bar{T} \cup \bar{R}) \) which is the output of phase two. For every cross edge \( e \) in \( G_1 \) there is a cross edge in \( \bar{R} \), which represents the cross edge \( R[e] \). \( G_2 \) is the minimal minor digraph of \((G_1, T)\) (and of \((G, T)\)).

Lemma 4.20: \( BMM(G=(V,E), T) \) has time complexity \( O(|E|) \).

Proof: It is obvious that the complexity of step (1) is \( O(|E|) \).

As for step (2), the computation of \( LCA \) is done using the algorithm of [AHU] for finding lowest common ancestors in a static tree in an off line model. This algorithm with the improvement of [GT] has time
complexity $O(|E|)$ as stated in [GT]. 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 4.21:** Algorithm $DI\_CHECK (G=(V,E), T)$ is correct and has $O(|E|)$ time complexity. 

**Theorem 4.22:** If $T$ is a spanning tree which is not a $T\,-DFS$ in $G=(V,E)$ then algorithm $DI\_CHECK$ can supply an $O(|V|)$ time complexity proof for that fact.

**Proof:** Let $G''=(V,\overrightarrow{T}\cup\overrightarrow{C''})$ 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 in order to find a set of edges $\{e''_1,e''_2,\ldots,e''_p\} \subseteq \overrightarrow{C''}$ 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 $O(|V|)$ 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 rerun the algorithm $DI\_CHECK (G',T)$ one can have an $O(|V|)$ time proof that $T$ is not a $T\,-DFS$ in $G$. 

**Corollary 4.23:** Algorithm $DI\_CHECK (G=(V,E),T)$ has an optimal ($O(|E|)$) 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 ($O(|V|)$) 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, $O(|E|)$ is an optimal time complexity for a positive verification, and hence 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:

$G_i = (V_i, E_i)$

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 $G_i \setminus e$. Hence, we can not 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 time complexity for a negative verification for this family and therefore it is an optimal time complexity for a negative verification in the general case.

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 $O(\log n)$ time complexity with $O(n)$ processors 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 $O(m/t)$ processors, where $t \geq \log n$, on a CREW PRAM. The latter proof has an optimal speed-up.

A parallel algorithm for recognizing a DFS tree in a digraph was independently presented in [SV]. However, [SV] does not contain refined verifications.

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 described here. Some more explanation of phase one can be found in the proof of Lemma 5.1.1.

(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.

* A preliminary version of our paper appeared in May (1988) (see [KO b]).
Lemma 5.1.1: The implementation of algorithm $BMM(G,T)$ has $O(\log n)$ time complexity where $O(m)$ processors are used on a CREW PRAM.

Proof: (Is similar to the proof of Theorem 3.7). We use the algorithm from [Vi] as part of $BMM$. This algorithm uses $O(n)$ processors in $O(\log n)$ time complexity to form a data-structure which enables the retrieval of the lowest common ancestor of any pair of vertices in $O(\log n)$ time complexity by a single processor. This is used in (ii) above to recognize forward, back and cross edges. The computation of $\hat{x}, \hat{y} \in V$ for every cross edge $e=x\rightarrow y$, where $x, y$ are in the brother sub trees $T_x$ and $T_y$, respectively, is a slight modification of the algorithm from [Vi]: Let $z$ be the lowest common ancestor of $x$ and $y$ in $T$. Consider the Euler path that is obtained from the tree $T$ by the algorithms in [TV] and in [Vi] (as was described in the proof of Theorem 3.7) and assume w.l.o.g. that $x$ is discovered in this path before $y$ (i.e. $\text{preorder}(x) < \text{preorder}(y)$). Consider the subpath of the above Euler path that starts with the edge $x \rightarrow f(x)$ (where $f(x)$ is the father of $x$ in $T$) and ends with the edge $f(y) \rightarrow y$. Find the first appearance of $z$ in the subpath. Clearly, it precedes by the edge $\hat{x} \rightarrow z$. Now find the last appearance of $z$ in the subpath. Clearly, it follows by the edge $z \rightarrow \hat{y}$. By this modification we can find $\hat{x}$ and $\hat{y}$ without affecting the complexity of the algorithm in [Vi]. □

Parallel implementation of phase two (algorithm $CAD$):

Phase two can be implemented efficiently 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 $k = \lfloor \log n \rfloor$. 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. 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 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 vérifications of DI_CHECK

In the following part of this section we show how the parallel implementation of algorithm
$DI\_CHECK(G=(V,E),T)$ 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 $O(\log n)$ time complexity with $O(n)$ pro-
cessors 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 $2k$ 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 {$A^{2^l}$ for $l=0, \ldots, \lceil \log |V| \rceil$ } 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 $2k$ in $G$, where $k=\lceil \log |V| \rceil$.

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 $l$ ($0 \leq l \leq \lceil \log |V| \rceil$) 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 of length $2^l$ from $v_i$ to $v_j$ (the edges of that path are printed in the same
order as they appear in the path).
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 $A^{2l-1}(i, q) > 0$ and $A^{2l-1}(q, j) > 0$;

{ 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 $\pi_{i,q}^{l-1} = RS(l-1, i, q)$ and $\pi_{q,j}^{l-1} = RS(l-1, q, j)$;

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

end { of the subroutine $RS$ }.

Recall that the set of matrices $A^{2l}$ 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 $FP$ can be implemented in $O(\log^2 |V|)$ time complexity with $O(|V|^2)$ 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 $O(|V|^2)$ 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 with $O(|V|)$ processors on a CREW PRAM: Go over $|V|$ pairs of entries in the matrix of the form $(A^{2l-1}(i, q), A^{2l-1}(q, j))$, for $1 \leq q \leq |V|$ using one processor for each pair 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 $2^{\lceil \log |V| \rceil - l}$ and the maximum is obtained when $l = 0$) therefore each recursive level can be done in $O(\log |V|)$ time with $O(|V|^2)$ processors. Since the recursive depth is $O(\log |V|)$ the proof of the proposition is completed. $\square$

**Theorem 5.2.2:** If $T$ is a spanning tree which is not a $T\_DFS$ in $G = (V, E)$ then the parallel implementation of algorithm $DI\_CHECK$ can supply a proof for that fact that can be verified in $O(\log |V|)$ time complexity by $O(|V|)$ processors.

**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) $\mathcal{C}'' = e_1'', e_2'', \ldots, e_p''$ in $G''$ which form a path of length $p = 2^\lceil \log |V| \rceil$ in $G''$. We denote by $E(\mathcal{C}'')$ the set of edges in $\mathcal{C}''$. It is easy to modify the parallel implementation of algorithm $DI_{\text{CHECK}} (G = (V, E), T)$ in order to find a subset of cross edges $\mathcal{C}' = \{e_1, e_2, \ldots, e_h\}$ in $G$ such that for all $i = 1, \ldots, p$ there exist a $j$ such that $1 \leq j \leq h$ and $R[e_j] = e''_i$. Hence $G' = (V, \mathcal{T} \cup \mathcal{C}')$ is a subgraph of $G$ with $O(|V|)$ edges where $T$ is a spanning tree which is not a $T$-DFS in $G'$. By running the algorithm $BMM (G', T)$ we obtain the minimal minor digraph of $(G', T)$, the edges of which are $\mathcal{T} \cup E(\mathcal{C}'')$.

One can easily check in the above complexity that $G'$ - as describes above - is a subgraph of $G$. By Lemma 5.1.1 the above complexity is also the complexity of algorithm $BMM (G', T)$. The last part of the proof is to present the sequence of edges $\mathcal{C}'' = e_1'', e_2'', \ldots, e_p''$ in $G''$ which form a path of length $p = 2^\lceil \log |V| \rceil$ in $G''$. One can check in the complexity stated in this theorem the correctness of the following claims: (i) $\mathcal{C}''$ are edges of the minimal minor digraph of $(G', T)$. (ii) For every $1 \leq i \leq p - 1$, head ($e_i$) = tail ($e_{i+1}$). (iii) $p \geq |V|$. \qed

**Theorem 5.2.3:** 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) supplies an optimal speed-up verification for that fact that can be verified in $O(t)$ time complexity with $O(\frac{|E|}{t})$ 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 $\mathcal{G} = (V, \mathcal{T} \cup \mathcal{F} \cup \mathcal{B} \cup \mathcal{C})$, where $\mathcal{T}, \mathcal{F}, \mathcal{B}$ and $\mathcal{C}$ are as described in Proposition 4.5.

**Proof:** The implementation is as follows:
(1) Build \( G' = (V, \bar{T} \cup \bar{C'}) \) - the minimal minor of \((G, T)\).

(2) Find a DFS-\( T \)-order induced by \( f \) which is compatible in \( \hat{G} = (V, \bar{T} \cup \bar{C'}) \) (where \( \bar{C'} \) are the edges of \( \bar{C'} \) 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 [SV] 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 adjacency list of the edges of \( T \) such that by implementing the algorithm from [TV] on this list we get a DFS-\( T \)-order (preorder numbering) induced by \( f \) which is compatible in \( \hat{G} \). The implementation does not affect the complexity of algorithm DI_CHECK stated in Corollary 5.1.3. □

Proof of Theorem 5.2.3: One can check that the order given by the algorithm is in fact a DFS-\( T \)-order by reruning the algorithm from [TV] on the adjacency list of the tree, that was created as described in Proposition 5.2.4. This check is done in \( O(\log |V|) \) time complexity with \( O(|V|) \) processors on a CREW PRAM. After this step of the verification all the vertices in \( T \) have a preorder numbering (the above DFS-\( T \)-order) and a postorder numbering. The compatibility of this DFS-\( T \)-order can be check in \( O(t') \) time complexity with \( O(\frac{|E|}{t'}) \) processors, for every \( t' \geq 1 \), on a CREW PRAM as follows:

1. For every edge check to which of the following groups \((\bar{T}, \bar{F}, \bar{B}, \bar{C'})\) it belongs. The methods are analogous to the methods that appear in the proof of Theorem 3.8.

2. Check that there is no 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_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 \( O(\log |V|) \) time complexity with \( O(|V|) \) processors 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 \( O(\frac{|E|}{t}) \) processors, for \( t \geq \log |V| \), on a CREW PRAM.
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 and Theorem 5.2.3. □

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 $O(\log |V|)$ time complexity and uses $O(|E|)$ processors on a CREW PRAM. In addition, the algorithm supplies an optimal speed-up verification to justify a positive answer. This verification can be verified in $O(t)$ time complexity with $O(\frac{|E|}{t})$ processors, for $t \geq \log |V|$, on a CREW PRAM. A sequential algorithm [KO a] performs this task and supplies an $O(|V|)$ time verification to justify a negative proof. It would be interesting to find an algorithm with the same time and processor complexity presented here, that in addition can supply a verification for a negative answer that can be verified in $O(\log |V|)$ time complexity with a better processor complexity.

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 $O(\log |V|)$ time complexity with $O(|V|)$ processors 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 $O(\frac{|E|}{t})$ processors, for $t \geq \log |V|$, on a CREW PRAM. The proof of a positive answer is optimal in the sense that we can not improve the time-processor product. It is left open to decide whether there is an $NC$ proof for a negative answer with a better time-processor product. If there is one, it would be interesting to find an $NC$ algorithm that solves the decision problem and in addition supplies such a proof.

The sequential algorithm for the directed case as well as the sequential algorithm for the undirected case in [KO a] have optimal $O(|E|)$ time complexity. However, the time-processor product in the parallel algorithms presented here is $O(|E| \log |V|)$ in the undirected case and $O(|V|^{2.376})$ in the
directed case (see table). It is left open to decide whether there are $NC$ algorithms which improve the speed-up of the parallel solutions for the problems solved here.

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

This research was partially supported by Technion V.P.R. Fund No. 120-0683.

REFERENCES


