COMBINATORICA
Akadémiai Kiadó — Springer-Verlag

COMBINATORICA 8 (1) (1988) 1—12

# A RANDOM $\mathcal{NC}$ ALGORITHM FOR DEPTH FIRST SEARCH

A. AGGARWAL[1] and R. J. ANDERSON[2]

*Received August 10, 1986*

In this paper we present a fast parallel algorithm for constructing a depth first search tree for an undirected graph. The algorithm is an $\mathcal{RNC}$ algorithm, meaning that it is a probabilistic algorithm that runs in polylog time using a polynomial number of processors on a $P$-RAM. The run time of the algorithm is $O(T_{MM}(n) \log^3 n)$, and the number of processors used is $P_{MM}(n)$ where $T_{MM}(n)$ and $P_{MM}(n)$ are the time and number of processors needed to find a minimum weight perfect matching on an $n$ vertex graph with maximum edge weight $n$.

## 1. Introduction

In this paper we present a fast parallel algorithm for constructing a depth first search tree of an undirected graph. This is the first $\mathcal{RNC}$ algorithm for the problem. The problem of performing depth first search in parallel has been considered by a number of authors [13], [5], [14], [1] and has been conjectured to be inherently sequential. The question as to whether depth first search could be performed by a fast parallel algorithm was raised by Wyllie [17] in his ground breaking thesis.

Depth first search is a very important technique for sequential computation. It has been used in the construction of a large number of efficient sequential algorithms [16]. However, depth first search seems to be sequential in nature. It is known that computing the lexicographically first depth first search tree is P-complete [14]. This is the depth first search tree found by the "natural" greedy algorithm. However, there are a number of problems, such as maximal independent set [9], [4], where computing the lexicographically minimal solution is P-complete, while a solution can be found in $\mathcal{NC}$ or $\mathcal{RNC}$. The main question that we investigate in this paper is whether the process of depth first search is inherently sequential or is it possible to perform a depth first search with a fast parallel algorithm.

The only cases where $\mathcal{NC}$ algorithms are known for depth first search are for restricted classes of graphs. For example, depth first search can be done in planar graphs [15] and directed acyclic graphs [12], [7] with $\mathcal{NC}$ algorithms. A problem that is related to depth first search that has investigated previously by the second

[1] This research was done while the first author was visiting the Mathematical Research Institute in Berkeley. Research supported in part by NSF grant 8120790.
[2] Supported by Air Force Grant AFOSR-85-0203A.
AMS subject classifications (1980): 68 Q 10, 05 C 99.

author is the problem of finding a branch of a depth first search tree. This problem is known as the maximal path problem. The maximal path problem can be solved by a rather complicated $\mathcal{RNC}$ algorithm [1]. This is another example of a problem where computing the lexicographically first solution is P-complete [3], but a different approach gives a fast parallel algorithm.

The algorithm presented in this paper is a substantial improvement over previously known parallel algorithms for depth first search. The best previously published result was an $O(n^{1/2})$ algorithm [1]. That result had been improved to $O(2^{2\sqrt{\log n}})$ [2] prior to this result. Similar techniques are used in all three of these algorithms. In particular, an important step in the algorithms is to construct sets of disjoint paths. The paths are constructed by using network flow techniques which in turn use matching. The result that made these algorithms possible was the $\mathcal{RNC}$ algorithm for matching [8]. The only use of randomness in the algorithms has been the reliance on probabilistic subroutines for matching. If matching could be solved in $\mathcal{NC}$, then depth first search could also be solved in $\mathcal{NC}$.

The depth first search problem is: given a graph $G=(V, E)$ and a vertex $r$, construct a tree $T$ that corresponds to a depth first search of the graph starting from the vertex $r$. There are a number of different ways to characterize a depth first search tree. One of them is: $T$ is a depth first search tree if and only if for all non-tree edges $(u, v)$, $u$ and $v$ lie on the same branch of the tree.

In this paper, we only address the problem of depth first search for undirected graphs. It is unclear whether or not our methods can be generalized to directed graphs.

## 2. Notational conventions

In our algorithm we work extensively with paths and vertices. We often use a set of vertices to denote an induced subgraph wherein the edges are inherited from the graph $G=(V, E)$. A path is an ordered set of distinct vertices $p=p_1, \ldots, p_k$ with edges $(p_i, p_{i+1}) \in E$ for $1 \leqq i \leqq k$. Paths are sometimes viewed as being directed. A lower segment of $p$ is a subpath $p_1, \ldots, p_j$ and an upper segment is a subpath $p_j, \ldots, p_k$. The algorithm maintains several sets of vertex disjoint paths. For a set of paths $Q=\{q_1, \ldots, q_m\}$, we use $|Q|$ to denote the number of paths. We occasionally mix notation and refer to sets of vertices and paths in the same expression. In particular, if $p$ is a path and $Q$ is a set of paths, $V-p$ denotes the induced subgraph after all vertices on $p$ have been removed, and $V-Q$ denotes the induced subgraph after all vertices contained in paths in $Q$ have been removed.

## 3. Overview of the algorithm

Our depth first search algorithm is a divide and conquer algorithm. A portion of a depth first search tree is constructed which allows the problem to be reduced to finding depth first search trees in graphs of less than half the original size. The depth of recursion is $\log n$.

An *Initial Segment* is a rooted subtree $T'$ that can be extended to some depth first search tree $T$. We give an algorithm which constructs an initial segment $T'$ with

the property that the largest connected component of $V-T'$ has size at most $n/2$. The initial segment has root $r$.

An initial segment $T'$ can be extended to a depth first search tree in the following manner. Let $C$ be a connected component of $V-T'$. There is a unique vertex $x \in T'$ of greatest depth that is adjacent to some vertex of $C$. Let $y \in C$ be adjacent to $x$. Construct a depth first search tree for $C$ rooted at $y$ and then connect it to $T'$ by an edge from $x$ to $y$. This construction can be performed independently for each connected component of $V-T'$.

Since the problem size is reduced by at least half at each stage, the depth of recursion is at most $\log n$. The time for a stage is dominated by the time to construct an initial segment. Then run time for the full algorithm is thus $\log n$ times the time to construct an initial segment.

The algorithm that constructs an initial segment consists of two parts. A set $Q$ of vertex disjoint paths is said to be a *separator* if the largest connected component of $V-Q$ has size at most $n/2$. This follows the standard graph theoretic use of the term [6]. The first part of the algorithm is to construct a separator $Q$, where the number of paths in $Q$ is bounded by a fixed constant. The second part is to construct an initial segment from the separator $Q$. The first part, constructing a small separator is the most significant part. The second part arises primarily from technical considerations, and is an adaptation of a routine from the earlier depth first search algorithm [1]. The next three sections cover the construction of the small separator and then the following section describes how the initial segment is built from the separator.

## 4. Constructing a separator

The central part of the algorithm is to construct a set of vertex disjoint paths $Q=\{q_1, \ldots, q_k\}$, where $k$ is less than a fixed constant and the largest component of $V-Q$ has size at most $n/2$. The algorithm to construct the separator relies on a routine *Reduce*$(Q)$ which reduces the number of paths in $Q$ while retaining the separator property. Each call to *Reduce* reduces the number of paths by a factor of 1/12.

Initially, $Q$ consists of all the vertices of $V$, each as a path of length 0. The separator property holds trivially at the beginning since everything is in $Q$. Since there is a constant fraction reduction in the number of paths in $Q$ by each call to *Reduce*, $O(\log n)$ calls suffice to reduce the size of $Q$ to at most 11. At that point the second part of the algorithm, that of constructing the initial segment, is performed. It would have been desirable to reduce the separator to just a single path, and then use it as a branch of the depth first search tree, but that has difficulties for a couple of reasons. First, the routine *Reduce* only guarantees a reduction if it contains a sufficiently large number of paths. Second, in order to use a single path in the depth first search tree, it would be necessary to have the vertex $r$ as one of the endpoints of the path. It is possible to change the endpoint of a single path that is a separator, but the method is essentially the same as the routine to construct the initial segment.

## 5. Reducing the number of paths

We now describe the main routine *Reduce*. The basic idea is to join the paths of $Q$ by finding vertex disjoint paths between them. This allows paths of $Q$ to be combined in pairs so that their number is reduced. The disjoint paths are found by using a parallel subroutine for matching. Our routine *Reduce* satisfies the following specification:

*Reduce*($Q$)
**input**
A set $Q$ of vertex disjoint paths such that the largest connected component of $V-Q$ has size at most $n/2$. $|Q| \geqq 12$.
**output**
A set $Q'$ of vertex disjoint paths such that the largest connected component of $V-Q'$ has size at most $n/2$. $|Q'| \leqq (11/12)|Q|$.

The general situation in *Reduce* is to have a set of vertex disjoint paths $Q$ that is a separator for the graph. $Q$ is divided into two sets of paths, $L$ and $S$. A set of vertex disjoint paths $P=\{p_1, \ldots, p_\alpha\}$ is found between the paths of $L$ and the paths of $S$. Each path $p_i$ has one of its endpoints a vertex of some path in $L$ and its other endpoint a vertex of some path in $S$, and interior vertices from $V-Q$. Each path of $Q$ contains the endpoint of at most one path of $P$. Suppose the path $p$ joins paths $l \in L$ and $s \in S$, $p$ has endpoints $x$ and $y$, $l=l'xl''$ and $s=s'ys''$. Then the paths

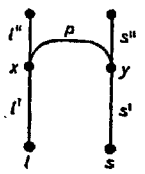


*Fig. 1*

$l$ and $s$ can be joined to form $l'ps'$. The question is what to do with the unused segments $l''$ and $s''$. If $l''$ and $s''$ are kept as paths, then there are three paths instead of the original two. On the other hand, if $l''$ and $s''$ are discarded and no longer considered part of $Q$, then components of $V-Q$ could merge and the separator property could be lost. The key idea in the routine *Reduce* is how to deal with the unused segments in a manner that maintains the separator property without increasing the number of paths.

The paths of $Q$ are divided into the two sets $L$ and $S$. $L$ is thought of as the long paths and $S$ is thought of as the short paths. The idea is to find the disjoint paths and extend the paths of $L$ using the paths of $S$. The short paths have their lengths decreased by this process. Let $P=\{p_1, \ldots, p_\alpha\}$ be a set of vertex disjoint paths joining paths of $L$ to paths of $S$. Suppose the path $p$ joins the path $l \in L$ and $s \in S$. Let $l=l'xl''$ and $s=s'ys''$ where $x$ and $y$ are the endpoints of $p$. If $s'$ is at least as long as $s''$, then $l$ is replaced by $l'ps'$, $s$ is replaced by $s''$ and $l''$ is discarded. Otherwise, if $s''$ is longer, $l$ is replaced by $l'ps''$, $s$ is replaced by $s'$ and $l''$ is discarded. In either case the path $s$ is reduced in length by half. This is done for each path

$p \in P$ and the pair of paths $p$ joins. When $L$ and $S$ are joined by a set of paths $P$, $\hat{L}$ and $\hat{S}$ denote the paths of $L$ and $S$ that are joined. The upper segments of the paths $\hat{L}$ that are discarded are denoted $L^*$. Note that this operation of joining paths does not increase the number of paths. The number of paths on $L$ remains the same. The number of paths in $S$ can decrease. If a path of $S$ is joined at an endpoint, the entire path is added to a path of $L$.

Initially, suppose $|Q| = K$. The paths of $Q$ are divided so that $K/4$ paths are placed in $L$ and the remaining paths are placed in $S$. A maximum cardinality set of disjoint paths $P = \{p_1, \ldots, p_\alpha\}$ is found between $L$ and $S$ and then the paths are joined as described above. This step is repeated until the number of paths in $Q$ is at most $11K/12$. There are, however, two things that could go wrong:

1. Discarding the paths $L^*$ could cause components of $V - Q$ to merge. This could cause the separator property to be lost.
2. The number of paths joined might be small so little progress is made by this step. This case is said to occur when the number of paths joined is less than $K/12$.

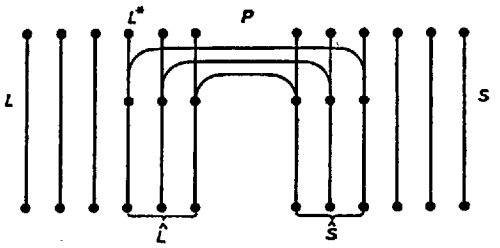


*Fig. 2*

Suppose for the time being that neither case 1 nor case 2 occurs. Then in a step the lengths of at least $K/12$ paths of $S$ are reduced by at least half. Since there are $3K/4$ paths in $S$ initially, each of length less than $n$, the maximum number of phases before $S$ is exhausted is $9 \log n$. Thus, as long as nothing goes wrong, the number of paths can be reduced by a constant fraction in $O(\log n)$ phases. Phases are repeated until the number of paths is reduced to $11K/12$. Thus, we have the following lemma.

**Lemma 1.** *If neither case* 1 *nor case* 2 *arises, the number of paths in* $Q$ *can be reduced to* $11K/12$ *in* $9 \log n$ *phases of joining paths.*

To complete the description of the algorithm, we describe what to do if either case 1 or 2 arises.

The main danger in joining paths is that discarding segments of $L$ may cause the separator property to be violated. In order to deal with this potential problem we add a restriction to the set of vertex disjoint paths that is constructed. Suppose the maximum number of vertex disjoint paths between $L$ and $S$ is $\alpha$, with $\alpha \geq K/12$. Let $P = \{p_1, \ldots, p_\alpha\}$ be a maximum set of disjoint paths from $L$ to $S$. For the path $p_l$ from $l$ to $s$ we assign a cost equal to the length of the segment cut off. That is if $l = l' x l''$, with $x$ an endpoint of $p_l$ we assign it a cost of $|l''|$. The set of vertex disjoint

paths that we find is the one that minimizes the total cost. In the next section we show that the problem of finding a mincost set of paths can be reduced to a matching problem and consequently solved by an $\mathcal{RNC}$ algorithm.

Now we show how to deal with the case where discarding the paths causes a large connected component to form. Suppose that $P=\{p_1, ..., p_\alpha\}$ is a mincost maximum set of vertex disjoint paths. Let $T$ be the set of vertices not on any path, so $T=V-Q-P$. The key is the following lemma which says that if the upper segments of the paths cannot be discarded without creating a large connected component, then a different set of paths can be discarded.

**Lemma. 2** *If the largest connected component of* $T \cup L^*$ *has size at least* $n/2$, *then the largest connected component of* $T \cup (S-\hat{S})$ *has size less than* $n/2$.

**Proof.** The largest component of $T$ has size at most $n/2$ since $Q$ is assumed to be a separator for the graph. There cannot be a path from a vertex in $L^*$ to a vertex in $S-\hat{S}$ using vertices of $T$. If there was such a path then a set of paths the same size as $P$ could have been found between $L$ and $S$ with strictly less cost. The induced subgraph on $T \cup L^* \cup (S-\hat{S})$ must contain at least two connected components. Either the components containing vertices of $L^*$ or the components containing vertices of $S-\hat{S}$ must have total size at most $n/2$ since $L^*$ and $S-\hat{S}$ fall into different components. Since the largest connected component of $T \cup L^*$ is assumed to have size at least $n/2$, it follows that the size of the components containing $S-\hat{S}$ in $T \cup L^* \cup (S-\hat{S})$ is at most $n/2$. Hence, the largest connected component of $T \cup (S-\hat{S})$ has size at most $n/2$. ∎

Thus, if case 1 occurs, the paths of $S-\hat{S}$ can be added to $T$ instead of $L^*$. Discarding the paths $S-\hat{S}$ is sufficient to achieve a constant fraction reduction in the number of paths and preserves the separator property. The remaining paths are $L$, $\hat{S}$ and $P$. The number of paths in each of these sets is at most $K/4$, so the number of paths remaining is $3K/4$. Thus, when this case occurs, we just discard the path $S-\hat{S}$ and *Reduce* is done.

The other bad case is if there are few paths between $L$ and $S$. Suppose $P=\{p_1, ..., p_\alpha\}$ is a maximum set of disjoint paths and $\alpha < K/12$. Since $P$ is maximum there could not be a path from $L-\hat{L}$ to $S-\hat{S}$ using vertices of $T$. Thus $L-\hat{L}$ and $S-\hat{S}$ fall into separate connected components in the graph on $T \cup (L-\hat{L}) \cup (S-\hat{S})$. By similar reasoning to above, either the graph on $T \cup (L-\hat{L})$ or on $T \cup (S-\hat{S})$ has connected components of size at most $n/2$, so either $L-\hat{L}$ or $S-\hat{S}$ may be discarded without losing the separator property. Suppose the paths of $L-\hat{L}$ are discarded. Then the remaining paths are $\hat{L}$, $P$, and $S$. $\hat{L}$ and $P$ each have at most $K/12$ paths and $S$ has at most $3K/4$ paths so the number of paths is reduced to $11K/12$ paths. If the paths $S-\hat{S}$ are discarded, then the remaining paths are $L$, $P$, and $\hat{S}$ which have total size at most $5K/12$. In either case, a constant fraction reduction is achieved, so that the second bad case can also be dealt with.

## 6. Finding sets of disjoint paths

We now show how the problem of constructing a maximum set of vertex disjoint paths can be reduced to a matching problem. The matching problem is to construct a perfect matching of minimum weight. The edges have integer weights of at most $n$. The reduction can be done in $O(\log n)$ time using $n^2$ processors.

The maximum set of disjoint paths problem can be stated as follows: Given a graph $G'=(V', E')$ and disjoint sets of vertices $X$ and $Y$ find a maximum cardinality set of vertex disjoint paths that have one endpoint in $X$ and the other in $Y$. The weighted version of the problem is where weights are attached to the edges and a maximum set of disjoint paths with minimum total weight is sought. It is easy to see that this is the problem we need to solve to find the disjoint paths for our algorithm. Each path in $L$ is contracted to a vertex that is put in $X$, and each path in $S$ is contracted to a vertex that is put in $Y$. The edges leaving vertices of $X$ are assigned weights that correspond to the edges leaving paths of $L$. Now, the problem of finding a maximum set of disjoint paths from $X$ to $Y$ can be expressed as a flow problem with unit capacities [6] and then reduced to bipartite matching. The weighted version of the disjoint paths problem can be solved by a mincost flow problem which can be reduced to weighted matching. However, this approach uses many processors. The reason that this approach uses so many processors is that the reduction from network flow to matching substantially increases the size of the graph. We solve the problem by giving a direct reduction to minimum weight matching. We first give a reduction of the unweighted disjoint paths problem to the matching problem where the edges have weight of only zero and one. This allows us to determine the maximum number of paths. We then give a reduction that finds the minimum weight set of disjoint paths. These reductions only increase the number of vertices in the graph by a constant fraction.

**Lemma 3.** *The problem of finding a maximum set of disjoint paths can be reduced to that of finding the minimum weight perfect matching in some graph $G''$ in which every edge has a weight of zero or one only.*

**Proof.** We transform the problem of finding a maximum set of disjoint paths in a graph $G'=(V', E')$ to the problem of finding a perfect matching of minimum weight in a graph $G''=(V'', E'')$. For convenience, we assume that $|X|=|Y|$. This can be achieved by adding dummy vertices to the smaller of the two sets.

The graph $G''$ has vertices $v_{in}$ and $v_{out}$ for each $v\in V'-X-Y$ with an edge $(v_{in}, v_{out})\in E''$. There are also vertices $x_1, \ldots, x_\alpha$ and $y_1, \ldots, y_\alpha$ in $V''$. We also denote these sets as $X$ and $Y$ in $V''$ and denote $V''-X-Y$ as $W$. For an edge $(v, w)\in E'$ with $v, w\in V'-X-Y$ there are edges $(v_{in}, w_{out})$, $(w_{in}, v_{out})\in E''$. For an edge $(x_i, v)\in E'$ there is an edge $(x_i, v_{in})\in E''$ and similarly for $(y_j, v)\in E'$ there is an edge $(v_{out}, y_j)\in E''$. For an edge $(x_i, y_j)\in E'$, there is an edge $(x_i, y_j)\in E''$. All the edges mentioned so far have weight zero. We refer to this as the *basic* construction. We add a complete bipartite graph with edges of weight one between $X$ and $Y$ in $G''$. This may create parallel edges between some pairs $x_i$ and $y_j$, with one edge of weight zero and the other edge of weight one.

We now show that a minimum weight perfect matching in $G''$ corresponds directly to a maximum set of disjoint paths in $G'$.

Suppose we have a set of $k$ vertex disjoint paths in $G'$. Let $x_l v_{i_1}, \ldots, v_{i_m} y_j$ be a path. We match the edges $(x_l, v_{i_1,in})$, $(v_{i_m,out}, y_j)$, and $(v_{i_l,out}, v_{i_{l+1},in})$ for $l=1 \ldots m-1$ in $G''$. The unmatched vertices of $W$ can be matched $(v_{j,in}, v_{j,out})$. The remaining unmatched vertices are $\alpha-k$ vertices of $X$ and $\alpha-k$ vertices of $Y$. These can be matched with edges of weight one. Thus we have a matching of weight $\alpha-k$.

Now suppose we have a perfect matching $M$ of weight $\alpha-k$. Let $M'$ be the set of all edges of the form $(v_{i,in}, v_{i,out})$. Consider the graph with vertices $V''$ and edges $M \oplus M'$, where $\oplus$ denotes symmetric difference. The vertices of $X$ and $Y$ all have degree one and the vertices of $W$ have degree zero or two. Thus, the graph consists of paths and cycles. The interior vertices of a path are alternately of the form $v_{i,in}$ and $v_{j,out}$, so the paths go from $X$ to $Y$. There are $\alpha$ paths in the graph. Since the matching had weight $\alpha-k$, $k$ paths have weight zero. These paths correspond directly to paths in $G'$.

We have shown a perfect matching in $G''$ corresponds directly to a set of vertex disjoint paths in $G'$. By minimizing the weight of a perfect matching we maximize the number of paths. ∎

We now show that the minimum cost set of paths used in the algorithm can be constructed by solving a weighted matching problem. Recall that the cost function for a path that leaves the path $l \in L$ from the vertex is the distance that $x$ is from the end of the path $l$. We first contract each path $s_i \in S$ to obtain a single vertex $y_i$ in $G'$. Then, we contract each path $l_i \in L$ to obtain a vertex $x_i$ in $G'$ and assign a weight of $j$ to an edge incident at $x_i$ if the corresponding edge is incident at a vertex in $l_i$ and this vertex is at a distance of $j$ from the topmost vertex in $l_i$. If the construction yields multiple edges between $x_i$ and $v$, only the one of minimum weight is retained. All edges not adjacent to a vertex corresponding to a path $l_i \in L$ have weight zero.

**Lemma 4.** *The problem of finding a minimum cost set of disjoint paths of a given size can be reduced to the problem of finding a minimum weight perfect matching in a graph with at most 2n vertices and edges of weight at most n.*

**Proof.** Suppose we want to find a set of $k$ disjoint paths of minimum weight between $X$ and $Y$. We construct a graph $G''=(V'', E'')$ in which a minimum weight perfect matching corresponds directly to a minimum weight set of $k$ disjoint paths in $G'=(V', E')$. Suppose $|X|=\alpha$ and $|Y|=\beta$. We begin with the basic construction used above. The edges inherit their weights for $G'$, so edges leaving vertices of $X$ may have non-zero weight and the other edges have weight zero. Instead of adding a complete bipartite graph between $X$ and $Y$, we add two sets of vertices $\hat{X}$ and $\hat{Y}$ where $|\hat{X}|=\alpha-k$ and $|\hat{Y}|=\beta-k$. We add complete bipartite graphs between $X$ and $\hat{X}$ and between $Y$ and $\hat{Y}$. These edges all have weight zero.

By an argument that is almost identical to the one used in Lemma 3, it can be shown that there is a direct correspondence between a set of $k$ disjoint paths in $G'$ and perfect matching in $G''$. The weight of the matching is the same as the weight of the disjoint paths, thus we can find the minimum weight set of paths by finding the minimum weight perfect matching. ∎

**Theorem 1.** *Let $P_{MM}(n)$ and $T_{MM}(n)$ denote, respectively, the number of processors and the parallel time required to compute a minimum weight perfect matching*

*of an n-node graph $G''$ that has a weight of at most n on all its edges. Then, the minimum cost set of vertex disjoint paths can be found in $O(T_{MM}(n))$ parallel time with $P_{MM}(n)$ processors.*

**Proof.** The above reductions are used in two stages. The first reduction is used to find the maximum number of disjoint paths, and then the second one is applied to find the minimum cost set of disjoint paths of that size. The time and number of processors are dominated by the cost of solving the matching problems. ∎

## 7. Constructing an initial segment

In this section we complete the description of the algorithm that constructs an initial segment $T'$ of some depth first search tree. The routine constructs the initial segment from a separating set of paths $Q$. The initial segment is rooted at $r$ and has the property that the largest connected component of $V-T'$ has size at most $n/2$, so $T'$ could also be viewed as a separator. The set $Q$ is assumed to have at most a constant number of paths (the bound is 11).

The routine to construct the initial segment is essentially a sequential algorithm; its only use of parallelism is in the low level routines that manipulate the graph. The algorithm maintains a subtree $\hat{T}$. Initially, $\hat{T}$ is just the vertex $r$. A step of the algorithm is to take one of the paths $q \in Q$ and to extend the subtree to contain at least half to the path $q$. This is done by picking the lowest vertex on $\hat{T}$ from which there is a path to $q$. Suppose the path $p$ is from $x \in \hat{T}$ to $y \in q$ where $q = q'yq''$ and $q'$ is at least as long as $q''$. The path $pq'y$ is added to $\hat{T}$ and the path $q$ is replaced by $q''$. Note that the length of the path $q$ is reduced by at least half. Since the number of paths initially in $Q$ is at most 11, the number of phases until all paths of $Q$ are used up is at most $11 \log n$. An individual phase of the algorithm can easily be done in $O(\log^2 n)$ time, so this algorithm runs in $O(\log^3 n)$ time. At the end of the algorithm, the subtree $\hat{T}$ is an initial segment such that the largest connected component of $V-\hat{T}$ has size at most $n/2$. To show that $\hat{T}$ can be extended to a depth first search tree it is sufficient to prove that there are no paths between separate branches of $\hat{T}$ that have all their interior vertices in $V-\hat{T}$. This condition holds through out the execution of the algorithm since the extensions are made from the lowest vertex possible. The largest connected component of $V-\hat{T}$ has size at most $n/2$ since $\hat{T}$ contains all vertices on paths in $Q$ and $Q$ is a separator.

## 8. Algorithm summary

The preceding sections have described our parallel algorithm for depth first search. We now give a more formal version in pidgin PASCAL to facilitate the timing analysis.

```
DFS(G, r)
  T' ← Initial Segment(G, r);
  for each connected component C of G−T' do
    recursively compute a dfs tree for C;
    add the tree for C to T';
```

*Initial Segment*$(G, r)$
  $Q \leftarrow V$;
  **while** $|Q| > 11$ **do**
    *Reduce*$(Q)$;
  Build the initial segment from $Q$;

The main routine of the algorithm is *Reduce*. In the code for the routine we use the following notation which is consistent with the notation used above. The set $Q$ is divided into two sets of paths, $L$ and $S$. A set of disjoint paths $P$ is found between $L$ and $S$. The joined paths of $L$ and $S$ are $\hat{L}$ and $\hat{S}$. The upper segments of the paths of $\hat{L}$ above the join are denoted $L^*$. The vertices not on any of the paths are $T$, so $T = V - Q - P$. For a subset $X$ of the vertices, we use $lcc(X)$ to denote the size of the largest connected component of the subgraph induced on $X$.

*Reduce*$(Q)$
  $K \leftarrow |Q|$;
  Divide $Q$ into two sets, $L$ and $S$, where $|L| = K/4$ and $|S| = 3K/4$;
  **while** $|Q| > 11K/12$ **do**
    Find mincost disjoint paths $P = \{p_1, \ldots, p_\alpha\}$ between $L$ and $S$;
    **if** $\alpha < K/12$ **then**
      **if** $lcc(T \cup (S - \hat{S})) < n/2$ **then**
        $Q \leftarrow L \cup \hat{S} \cup P$;
      **else**
        $Q \leftarrow S \cup \hat{L} \cup P$;
      **return**
    **else if** $lcc(T \cup L^*) > n/2$ **then**
      $Q \leftarrow L \cup \hat{S} \cup P$;
      **return**
    **else**
      Extend the paths of $\hat{L}$. Suppose $p$ joins $l$ and $s$, $x$ and $y$ are the endpoints of $p$ and $l = l'xl''$, $s = s'ys''$. If $|s'| \geq |s''|$ then $l \leftarrow l'ps'$ and $s \leftarrow s''$, otherwise, $l \leftarrow l'ps''$ and $s \leftarrow s'$. In both cases, $l''$ is discarded.

The algorithm has three levels of iteration or recursion: the recursive construction of dfs trees for components, the calls to *Reduce* to reduce the size of the separator, and the step of extending paths. Each of these can be executed $O(\log n)$ times, so the run time is $O(\log^3 n)$ times the time of the inner loop of *Reduce*. The run time for the inner loop is dominated by the cost of finding the disjoint paths, which is solved as a matching problem. Matching can be solved in $O(\log^2 n)$ time [11], so our algorithm runs in $O(\log^5 n)$ time. The only step of the algorithm which is expensive in terms of processors is finding the mincost set of disjoint paths. In Section 6, we showed that this problem could be reduced to the problem of finding a minimum weight perfect matching in a graph with edge weights of at most $n$. The number of vertices in the graph that we reduce the problem to is $O(n)$. Thus, the matching problem can be solved with $nM(n)$ processors [11], where $M(n)$ is the number of processors that are needed to multiply matrices.

## 9. Discussion

In this paper, we presented a fast parallel algorithm for computing a depth first search tree of an $n$-vertex graph. We showed that the depth first search problem is in $\mathscr{RNC}$ and if the problem of minimum weight maximum matching is in $\mathscr{NC}$ then so is the depth first search problem. Consequently, this paper disproves a widespread belief that the computation of a depth first search tree is an inherently sequential process. However, this paper also leaves the following problems unresolved:

1. Is the depth first search problem in $\mathscr{NC}$?
2. Does there exist a random or deterministic algorithm that has an optimal or a near optimal processor-time complexity and that computes a depth first search tree in polylogarithmic time?
3. Is there a polylogarithmic time algorithm that computes a depth first search tree without using a minimum weight matching algorithm as a subroutine? Such an algorithm may be useful in obtaining a deterministic polylogarithmic time algorithm for the depth first search problem, and it may also yield an algorithm with a smaller processor-time complexity than our algorithm.
4. Is the problem of computing depth first search tree of a directed graph in $\mathscr{NC}$ or $\mathscr{RNC}$?

## References

[1] R. J. Anderson, A parallel algorithm for the maximal path problem, *Combinatorica*, 7 (1987), 315—326.
[2] R. J. Anderson, A parallel algorithm for depth-first search, *Extended Abstract, Math. Science Research Institute* (1986).
[3] R. J. Anderson and E. Mayr, Parallelism and greedy algorithms, *Technical Report* No. STAN-CS-84-1003, *Computer Science Department, Stanford University* (1984).
[4] S. A. Cook, A taxonomy of problems with fast parallel algorithms, *Information and Control*, 64 (1985), 2—22.
[5] D. Eckstein and D. Alton, Parallel graph processing using depth first search, *Proc. of the Conf. on Theoretical Computer Science at the Univ. of Waterloo*, (1977), 21—29.
[6] S. Even and R. E. Tarjan, Network flow and testing graph connectivity, *SIAM Journal of Computing*, 4 (1975), 507—518.
[7] R. K. Ghosh and G. P. Bhattacharjee, A parallel search algorithm for directed acyclic graphs, *BIT*, 24 (1984), 134—150.
[8] R. M. Karp, E. Upfal and A. Wigderson, Constructing a maximum matching is in random *NC*, *Combinatorica*, 6 (1986), 35—48.
[9] R. M. Karp and A. Wigderson, A fast parallel algorithm for the maximal independent set problem, *Journal of ACM*, 32 (1985), 762—773.
[10] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM Journal of Applied Math.* 36 (1979), 177—189.
[11] K. Mulmuley, U. V. Vazirani and V. V. Vazirani, Matching is as easy as matrix inversion, *Combinatorica* 7 (1986), 105—114.
[12] V. Ramachandran, *Personal Communication.*
[13] E. Reghbati and D. Corneil, Parallel computations in graph theory, *SIAM Journal of Computing*, 7 (1978), 230—237.
[14] J. H. Reif, Depth-first search is inherently sequential, *Information Processing Letters*, 20 (1985), 229—234.

[15] J. R. SMITH, Parallel algorithms for depth first searches, *SIAM Journal of Computing*, 15 (1986), 814—830.
[16] R. E. TARJAN, Depth-first search and linear graph algorithms, *SIAM J. of Computing*, 1 (1972), 146—160.
[17] J. C. WYLLIE, *The Complexity of Parallel Computations*, Phd. Thesis, Department of Computer Science, Cornell University, 1979.

Alok Aggarval

*IBM T. J. Watson Center, P. O. Box 218*
*Yorktown Heights, New York 10598, USA*

Richard J. Anderson

*Dept. of Computer Science,*
*FR-35, University of Washington,*
*Seattle, Washington 98195, USA*