Parallel Algorithms for Depth-First Search

Jon Freeman
University of Pennsylvania

Parallel Algorithms For Depth-First Search

MS-CIS-91-71

Jon Freeman

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

October 1991

Parallel Algorithms for Depth-First Search

Jon Freeman
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

October 1991

Abstract

In this paper we examine parallel algorithms for performing a depth-first search (DFS) of a directed or undirected graph in sub-linear time. This subject is interesting in part because DFS seemed at first to be an inherently sequential process, and for a long time many researchers believed that no such algorithms existed. We survey three seminal papers on the subject. The first one proves that a special case of DFS is (in all likelihood) inherently sequential; the second shows that DFS for planar undirected graphs is in *NC*; and the third shows that DFS for general undirected graphs is in *RNC*. We also discuss randomized algorithms, *P*-completeness and matching, three topics that are essential for understanding and appreciating the results in these papers.

1 Introduction

In this paper we examine parallel algorithms for performing a depth-first search (DFS) of a directed or undirected graph in sublinear time. This subject is interesting in part because DFS seemed at first to be an inherently sequential process, and for a long time many researchers believed that no such algorithms existed. We survey three seminal papers on the subject. The first one proves that a special case of DFS is (in all likelihood) inherently sequential; the second shows that DFS for planar undirected graphs is in *NC*; and the third shows that DFS for general undirected graphs is in *RNC*. We also discuss randomized algorithms, *P*-completeness and matching, three topics that are essential for understanding and appreciating the results in these papers.

We begin by defining DFS, mentioning some important applications of DFS, and indicating where sub-linear DFS algorithms would be useful.

1.1 Definition of DFS

We begin with some standard graph-theoretic definitions. A graph G is a set of vertices V and a set of edges E, written $G = (V, E)$. We let $|V| = n$ and $|E| = m$. A *directed edge* of G is an element

of $V \times V$; an *undirected edge* of G is a subset of V of cardinality two. G is *directed (undirected)* if E consists entirely of directed (undirected) edges. The *indegree* of a vertex $\overline{w}$ in a directed graph G is the number of edges of the form $(v, \overline{w})$; the *outdegree* of a vertex $\overline{v}$ is the number of edges of the form $(\overline{v}, w)$. A *directed (rooted) tree* T is a directed graph in which every node except one has indegree 1; the remaining node has indegree 0 and is called the *root* of T. A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A directed tree T is a *spanning tree* of a graph G if T is a subgraph of G and T contains all the vertices of G. A spanning tree T is a *depth-first search tree* of G iff, for all non-tree edges $\{v, w\}$ (or (v, w) if G is directed), v and w lie on the same branch of T [3, 54].

The depth-first search problem is: given a graph G and a vertex r in G, construct a depth-first search tree T of G rooted at r. T is called a *search tree* because the standard way to construct it is to search G in a depth-first manner, as follows. We begin our search at r; T is initially empty, and all the vertices of G are marked unvisited. We mark the current vertex v as visited and sequentially examine all unexplored edges leaving v (in any order we wish). If there is an unvisited vertex w adjacent to v, we make w the current vertex and repeat. If there is no such vertex, we backtrack to the last vertex we visited that has at least one such vertex and repeat. We halt when we have explored every edge (and visited every vertex) in G. This algorithm has a simple recursive description [5]:

```
procedure DFS(v)
begin
    mark v as visited;
    while there is an unmarked vertex w adjacent to v do
        add (v, w) to T;
        DFS(w)
    end { while }
end { DFS }
```

The running time of DFS is easy to analyze; it is just $O(n + m)$, since we visit every vertex and explore every edge exactly once. Given what we are trying to accomplish, then, this algorithm is optimal to within a constant factor. For this reason, we will frequently equivocate between the terms "parallel DFS algorithms" and "sub-linear DFS algorithms", as only parallel DFS algorithms can run in sub-linear time.

It should be clear that the decision we make at each vertex (i.e., which edge to explore next) has a drastic effect on the decisions we make afterwards. This is why so many people conjectured that DFS was inherently sequential when the question first arose. The reason why they were mistaken is

that, loosely speaking, their understanding of DFS as a search problem was not declarative enough.

1.2 Applications of DFS

Here are some of the first significant applications of DFS, in chronological order:

- Tarjan used it to find the strongly connected components of a directed graph, and the biconnected components of an undirected graph, both in linear time [54].
- Hopcroft and Karp used it to improve the best-known algorithm for bipartite matching (which we will define below) from $O(nm)$ to $O(n^{1/2}m)$ [31].
- Tarjan used it to find dominators in a directed graph in time $O(n \log n + m)$ [55].
- Hopcroft and Tarjan used it to test the planarity of a graph in linear time [29].
- Even and Tarjan used it to test vertex connectivity in time $O(n^{1/2}m^2)$ and edge connectivity in time $O(n^{5/3}m)$ [18].

DFS is such a fundamental operation on graphs that it would be extremely difficult, if not impossible, to list all of the uses researchers have found for it.

1.3 Usefulness of Parallel DFS Algorithms

Speeding up any algorithm is always significant in its own right, but there may be certain circumstances in which we would not benefit greatly from doing so. With that in mind, we note that in order for us to benefit from sub-linear DFS algorithms, constructing the graph itself should not be more time-consuming than the time required to perform the search. This will be the case when the graph is constructed in parallel, for example, or when we need to search a given graph many times starting at different vertices.

The next three sections discuss parallel algorithms, randomized algorithms, and parallel randomized algorithms, respectively.

2 Parallel Algorithms

In this section, we explain the basic idea behind parallel algorithms, describe four popular models of parallel computation, and define the class *NC*.

2.1 The Basic Idea

The basic idea behind parallel algorithms is obvious: If we have more than one processor at our disposal, we can solve a problem more quickly by dividing it into independent sub-problems and solving them at the same time, one on each processor. The running time of the algorithm is then the longest running time of any of these processors; more specifically, given input n, the running time $T(n)$ on input n is the elapsed time from when the first processor begins executing to when the last processor stops executing.

We say that a parallel algorithm for a given problem is *optimal* if its processor bound $P(n)$ and its time bound $T(n)$ are such that $P(n)T(n) = O(S)$, where S is the running time of the best known sequential algorithm for the problem. Thus parallel algorithms should meet at least two criteria: $T(n)$ should be as small as possible, and they should be optimal.

2.2 Models of Parallel Computation

There are four popular models of parallel computation: shared memory models, boolean circuits, fixed connection networks, and parallel comparison trees [48]. The next four subsections describe each of these models in turn.

2.2.1 Shared Memory Models

A shared memory model consists of a set of synchronous processors and a shared global memory through which they communicate [19]. The processors are random access machines (RAM's), therefore memory accesses and operations on integers take constant time, but integers cannot be unreasonably large [13]. There are three conventions regarding whether to permit simultaneous reads or writes to the same memory location: EREW (exclusive read, exclusive write), CREW (concurrent read, exclusive write), and CRCW (concurrent read, concurrent write). If and when read or write conflicts do occur, some sort of priority scheme is used to resolve them. These three variations are not equally powerful, but researchers nevertheless consider all of them to be "reasonable".

2.2.2 Boolean Circuits

A Boolean circuit α with n inputs is a finite directed acyclic graph with nodes labeled as follows [14]. There are n input nodes and one output node. The input nodes are labeled with variables

and their negations; all other nodes are labelled with a Boolean operation such as $\vee$ or $\wedge$. There is at least one path from every input node to the output node. The *fan-in* of α is the largest indegree of any node, the *size* of α is the number of gates it contains, and the *depth* of α is the length of the longest path from an input node to the output node. Let Σ be a finite alphabet and consider a language $L \subseteq \Sigma^n$. We say that a Boolean circuit C accepts L if for all $w \in \Sigma^n$, C outputs 1 on $w \Leftrightarrow w \in L$. The circuit size complexity of L is the size of the smallest circuit that accepts L.

Now consider a language $L^* \subseteq \Sigma^*$. The circuit size complexity of L^* is a function f such that $f(n)$ is the circuit size complexity of $L^n = L^* \cap \Sigma^n$. In order to recognize languages in Σ^*, then, we need *families* of Boolean circuits $< \alpha_n >$ in which each circuit in the family accepts strings of a particular length.

We can easily extend this model by allowing these circuits to have multiple outputs. The resulting Boolean circuits can compute arbitrary functions from $\{0,1\}^n$ to $\{0,1\}^m$. The definitions of size and depth remain the same.

We are usually only interested in *uniform* circuit families, i.e., families $< \alpha_n >$ such that, given n, we can easily construct the specific circuit α_n. (See Cook for an justification [14].) Researchers have proposed several different uniformity conditions in the past [14], but the one in widest use today is *log-space uniformity.* A circuit family $< \alpha_n >$ satisfies this condition if, given n, a deterministic Turing machine can generate a description of α_n in space $O(\log n)$.

Boolean circuit families are due to Borodin [8]. They are appealing for several reasons. For one thing, they are realistic in the sense that ultimately all computers consist of Boolean circuits. Also, the theory of Boolean circuit complexity is interesting in its own right and has a long history [13, 14].

2.2.3 Fixed Connection Networks

A fixed connection network is a directed graph in which the vertices correspond to processors and the edges correspond to connections between pairs of processors. The degree of each node is typically either a constant or a slowly increasing function of the number of vertices [48].

2.2.4 Parallel Comparison Trees

A parallel comparison tree is an ordinary comparison tree in which we can make several comparisons simultaneously at each node. This model is due to Valiant [57], and is significantly more powerful than the other three.

2.3 The Class NC

Let NC^k be the set of all functions computable by a uniform Boolean circuit family $< \alpha_n >$ with size $n^{O(1)}$ and depth $O(\log^{O(1)} n)$, and let $NC = \bigcup_k NC^k$. Informally, NC is the class of all problems we can solve quickly with a reasonable number of processors.

NC stands for "Nick's Class", after Nicholas Pippenger, who first identified it and suggested that it contains precisely the problems we think of as having "good" parallel algorithms [45]. NC remains the same across a wide variety of machine models, although the subclasses NC^k may vary [14]. It is easy to see that $NC \subseteq P$, but whether $NC = P$ is a famous open problem (see Section 6 below).

3 Randomized Algorithms

In this section, we explain the basic idea behind randomized algorithms, describe some models of randomized computation, and explain why randomization is a good idea.

3.1 The Basic Idea

There are many algorithms, such as Quicksort, that work well assuming that the inputs to the algorithm have a certain probability distribution, i.e., that they are sufficiently random. Unfortunately, the validity of this assumption is often highly questionable [47]. Randomized algorithms, introduced independently by Rabin [47] and Solovay and Strassen [53], take this idea one step further by introducing randomization into the algorithm itself. Simply put, a randomized algorithm is one in which some of the decisions depend on the outcomes of coin flips. We can think of a randomized algorithm as a family of deterministic algorithms, each of which corresponds to a particular sequence of coin flips. The goal is to construct algorithms of this sort such that, for any input whatsoever, a large fraction of the possible deterministic algorithms will output the correct answer quickly, and hence the overall algorithm will output the correct answer quickly with high probability.

Let us formalize this idea by defining a complexity measure for randomized algorithms. We say that a resource bound is $\tilde{O}(f(n))$ if there exists a $c \in \mathcal{R}$ such that the amount of the resource used (on any input of size n) is no greater than $c\alpha f(n)$ with probability $\geq 1 - n^{-\alpha}$.

3.2 Types of Randomized Algorithms

There are two main kinds of randomized algorithms [48]. Some of them will always terminate within a certain amount of time but will output the correct answer with a certain probability; these are called *Monte Carlo algorithms.* The others will always output the correct answer, but their running time is a random variable whose average value is known; these are called *Las Vegas algorithms.*

The error of a randomized algorithm can also be of two different kinds [48]. Consider randomized algorithms whose output is either *yes* or *no* (e.g., for deciding membership in a language). An algorithm of this sort is said to have *1-sided error* if it is always correct when it answers yes, but is correct with high probability when it answers no. If it is correct with high probability in either case, it is said to have *2-sided error.*

3.3 The Class RP

We define the class RP to be the set of all problems that have randomized polynomial-time algorithms. It is clear that $P \subseteq RP$; whether $P = RP$ is another famous open problem. It is tempting to try to answer this question purely on philosophical grounds—by asserting that coin flips are ultimately of no use when we are seeking a definite yes or no, for example, or by claiming that nature is not truly random [14]. It is also worth noting that if $P \neq RP$, then randomized algorithms cannot work too well on typical computers (even though they seem to), because typical computers use pseudo-random numbers, which are generated deterministically [14].

3.4 Models of Randomized Computation

We can easily define models of randomized computation by simply extending the standard models. For the RAM model, for example, we simply create a slightly different RAM that has the ability to flip an n-sided coin in constant time on inputs of length n. A randomized machine model is said to compute a function f if it outputs the correct value of f with probability $> 1/2$.

3.5 The Advantages of Randomized Algorithms

The first and most obvious advantage of randomized algorithms is that we no longer have to assume anything about the distribution of the inputs in order to prove that they run quickly. A second advantage is that they need not be any less accurate than deterministic algorithms. This is

because there is always a non-zero probability that the hardware itself might fail, and if we want to, we can always ensure that the probability of our algorithm returning a wrong answer is less than this value [2]! Third, randomized algorithms are usually simpler and easier to understand than deterministic algorithms for the same problem with similar running times. And fourth, there are many problems for which the best-known randomized algorithm is faster than the best-known deterministic algorithm (including DFS, as we will see). In fact, there are at least two problems (primality testing[1] and exact matching [42]) that are in the class RNC (defined below) but are not known to be in P.

4 Parallel Randomized Algorithms

This section is structurally identical to Section 2, in which we discussed deterministic parallel algorithms.

4.1 The Basic Idea

Because parallelism and randomization are completely different approaches, we can combine them, thereby obtaining the advantages of both. Reif was apparently the first to propose this idea and demonstrate its effectiveness on a variety of algebraic and graph-theoretic problems [51].

The definition of optimality is similar to that of deterministic parallel algorithms: a parallel randomized algorithm with processor bound $P(n)$ and time bound $T(n)$ is optimal if $P(n)T(n) = \tilde{O}(S)$, where S is the running time of the best known sequential algorithm for the same problem.

4.2 Models of Computation

If we can randomize models of sequential computation, we can easily randomize models of parallel computation too. In the PRAM model, for instance, we simply replace each RAM with a randomized RAM (described above). Similarly, a randomized Boolean circuit is just a Boolean circuit in which each node can also do coin flips. As before, a parallel randomized machine model is said to compute a function f if it outputs the correct value of f with probability $> 1/2$.

[1] Miller has shown that primality testing is in P assuming that the extended Riemann Hypothesis holds [41]; the best known deterministic algorithm that does not make this assumption takes time $n^{O(\log \log n)}$ [1].

4.3 The Class *RNC*

We define the class *RNC* in much the same way as the class *NC*. Let RNC^k be the set of all functions computable by a uniform family of randomized Boolean circuits $< \alpha_n >$ with size $n^{O(1)}$ and depth $O(\log^{O(1)} n)$, and let $RNC = \bigcup_k RNC^k$. The intuition is also similar: *RNC* is the set of all problems that have fast parallel randomized algorithms. As with *NC*, *RNC* remains the same across a wide range of machine models.

With these definitions in hand, we can now turn our attention to the main subject of this paper. In the next section, we present a history of parallel DFS algorithms and show where our three main papers fit in this history. After that, we will discuss these three papers in chronological order, introducing additional topics as necessary.

5 History of Parallel DFS Algorithms

As we stated in the introduction, the history of parallel DFS algorithms is interesting because for a long time, many researchers believed that no such algorithms existed. Wyllie was the first to suggest that it might be possible to parallelize DFS [58]. After he made this conjecture, several researchers examined the problem and concluded that there was no way to do so [15, 49, 50]. Starting in 1983, however, breakthroughs in this area began to occur. We list those breakthroughs below in chronological order, from 1983 to the present.

- Ghosh and Bhattacharjee gave an $O(\log^2 n)$ algorithm for DFS in directed acyclic graphs, thereby becoming the first to give a sub-linear time algorithm for any special case of DFS [23]. Their algorithm has an error which is corrected by Zhang [59].

- Reif proved that the lexicographic version of DFS (defined below) is *P*-complete, thereby providing strong evidence for the claim that general DFS is inherently sequential [50].

- Smith showed that DFS for planar undirected graphs is in *NC*; his algorithm takes time $O(\log^3 n)$ [52].

- Anderson gave the first sub-linear time algorithm for DFS in general undirected graphs. His algorithm is randomized and takes time $\tilde{O}(\sqrt{n}\log^c n)$ for some small constant c [7].

- Anderson improved the running time of his previous algorithm somewhat, to $\tilde{O}(2^{2\sqrt{\log n}})$ [6].

- Aggarwal and Anderson gave the first *RNC* algorithm for DFS in general undirected graphs. The running time of their algorithm is $O(T_{mm}(n)\log^3 n)$, where $T_{mm}(n)$ is the time needed to find a minimum weight perfect matching on an n vertex graph with maximum edge weight n [3]. Currently $T_{mm} = \tilde{O}(\log^2 n)$ [42], for a bound of $\tilde{O}(\log^5 n)$.

- Kao gave the first *NC* algorithm for DFS in planar directed graphs [35].

- Goldberg et. al. gave the first sub-linear time *deterministic* algorithm for undirected graphs; their algorithm takes time $O(\sqrt{n}\log^5 n)$ [25].

- Aggarwal et. al. gave the first *RNC* algorithm for DFS in general directed graphs. The running time of their algorithm is $O(\log^5 n(T_{mm}(n) + \log^2 n))$, where T_{mm} is as defined above; letting $T_{mm} = \tilde{O}(\log^2 n)$, this simplifies to $\tilde{O}(\log^7 n)$. They also built on the results in Goldberg et. al.'s paper to give the first sub-linear deterministic DFS algorithm for general directed graphs, which takes time $O(\sqrt{n}\log^{11} n)$ [4].

- Hagerup showed how to perform DFS in planar undirected graphs in time $O(\log n)$. Other researchers had previously improved the time (and processor) bounds of Smith's algorithm somewhat; see Hagerup's paper for details [28].

In the remainder of this paper, we discuss three of the above results in depth: Reif's proof that lexicographic DFS is (in all likelihood) difficult to parallelize, Smith's *NC* algorithm for planar graphs, and Aggarwal and Anderson's *RNC* algorithm for general undirected graphs. We begin with Reif's result, but first we touch on two topics that are crucial to understanding it—namely, *P*-completeness and the parallel computation thesis.

6 *P*-Completeness and the Parallel Computation Thesis

Let us consider non-trivial ways to formally reduce problems in the class P to one another. Obviously, we cannot use polynomial-time reduction, as we do for the class NP; we use a *space* reduction instead.

Let L and L' be languages over a finite alphabet Σ. We say that L' is *log-space reducible* to L (written $L \leq_{\log} L'$) if there exists a function f such that f is computable on a log-space Turing machine and, for each $w \in \Sigma^*$, $w \in L'$ if and only if $f(w) \in L$. L is *P-complete* if $L \in P$ and, for each $L' \in P$, $L \leq_{\log} L'$. As with polynomial-time reducibility, we know that log-space reducibility

is transitive [30]. Therefore, to show that a problem $L \in P$ is P-complete, it suffices to show that some P-complete problem L' is log-space reducible to L.

Many problems are known to be P-complete. Cook gave the first P-completeness proof (for path systems), although he did not state it as such [12]. Jones and Lasser showed that seven important problems (such as the emptiness problem for CFG's) were P-complete, thereby doing for P-completeness what Karp did for NP-completeness [33]. Goldschlager et. al. showed that the maximum flow problem is P-complete [26].

We have not yet said anything about the significance of P-complete problems. On a superficial level, all we can conclude about these problems is that, if we could solve one of them in log-space, we could solve every problem in P in log-space. It turns out, however, that the P-complete problems are precisely the problems in P that are (in all likelihood) not in NC, i.e., the ones that are hard to parallelize. Stated another way, if any P-complete problem is in NC, then $P = NC$. This result follows from an important conjecture of theoretical computer science called the *parallel computation thesis*, which states that time-bounded parallel machines and space-bounded sequential machines are polynomially related; i.e., for any function $T(n)$,

$$\bigcup_k \text{PARALLEL-TIME}(T^k(n)) = \bigcup_k \text{SPACE}(T^k(n)).$$

If we let $T(n) = c \log n$ in this equation (for some constant c), the result follows easily.

Pratt and Stockmeyer were the first to prove that this thesis held for their model of parallel computation, although they did not state it as such [46]. Chandra and Stockmeyer were the first to state it explicitly [10]. Goldschlager provided strong evidence for its validity [27]. Pratt and Stockmeyer were not the only ones to prove that the thesis held for their model; see, for example, Goldschlager [27] and Fortune and Wyllie [19].

There are many obvious similarities between the P-complete problems and the NP-complete problems, which we will not bother to list here. One important *difference* between the two classes is that NP-complete problems have received far more attention than their counterparts in the class P. We believe, however, that the increasing importance of parallel algorithms will rectify this imbalance in the near future.

7 Reif's P-Completeness Proof

Reif studied a special case of DFS known as *lexicographic DFS*, and proved that it is P-complete. In lexicographic DFS, for each vertex v in G, the order in which we explore v's outgoing edges is

fixed. (If we use adjacency lists to represent G, [54], we say that G's adjacency lists are fixed.) We present the proof below.

As with all problems, we must convert DFS into a decision problem in order to analyze its complexity. Let (U)DFS-ORDER be the following problem: Given a directed (undirected) graph G, a starting vertex s, and two vertices u and v, and assuming that the order in which we visit each vertex's outgoing edges is fixed, do we visit u before v in a depth-first search of G starting from s?

Let us also define the *circuit value problem* [50]. A *Boolean circuit* B (not to be confused with the Boolean circuits of Section 2.2.2) is a sequence $(B_0, B_1, \ldots, B_n)$ such that each B_i is either *true*, *false*, or an expression $op(B_{i_1}, B_{i_2})$, where $i_1, i_2 < i$ and op is a binary Boolean operation. We recursively define $value(true) = true$, $value(false) = false$, and $value(op(B_{i_1}, B_{i_2})) = op(value(B_{i_1}), value(B_{i_2}))$. Finally, we define $value(B)$ to be $value(B_n)$. The circuit value problem is as follows: Given a Boolean circuit B, is $value(B) = true$?

Ladner proved that the circuit value problem is P-complete [38]. We also have the following proposition [50]:

Proposition 1: The circuit value problem remains P-complete under the restrictions that $B_0 = true$ and the only Boolean operation in B is NOR, i.e., all non-trivial B_i's have the form $B_i = \neg(B_{i_1} \vee B_{i_2})$ where $i_1, i_2 < i$.

Proof: This follows easily from the fact that { NOR } is a functionally complete set. □

Reif proved that DFS-ORDER is P-complete by reducing this special case of the circuit value problem to it. Then he reduced DFS-ORDER to UDFS-ORDER to show that UDFS-ORDER is P-complete. We present these two reductions below.

Theorem 1: DFS-ORDER is P-complete.

Proof: For each Boolean operation $B_i = \neg(B_{i_1} \vee B_{i_2})$ in B, where B_i occurs in subsequent operations $B_{j_1}, B_{j_2}, \ldots, B_{j_k}$, we construct the "digraph gadget" G_i in Figure 1. For each vertex v in G_i, the numbers on the edges leaving v indicate the order in which they appear in v's adjacency list. The G_i's are linked together; for example, the node Exit$(i-1)$ is a part of gadget G_{i-1} and the node In(j_k, i) is a part of gadget G_{j_k}.

Suppose that we are performing a depth-first search of G_i starting at Exit$(i-1)$, and we have not previously visited either node In(i, i_1) or node In(i, i_2). Then the DFS tree edges for G_i will

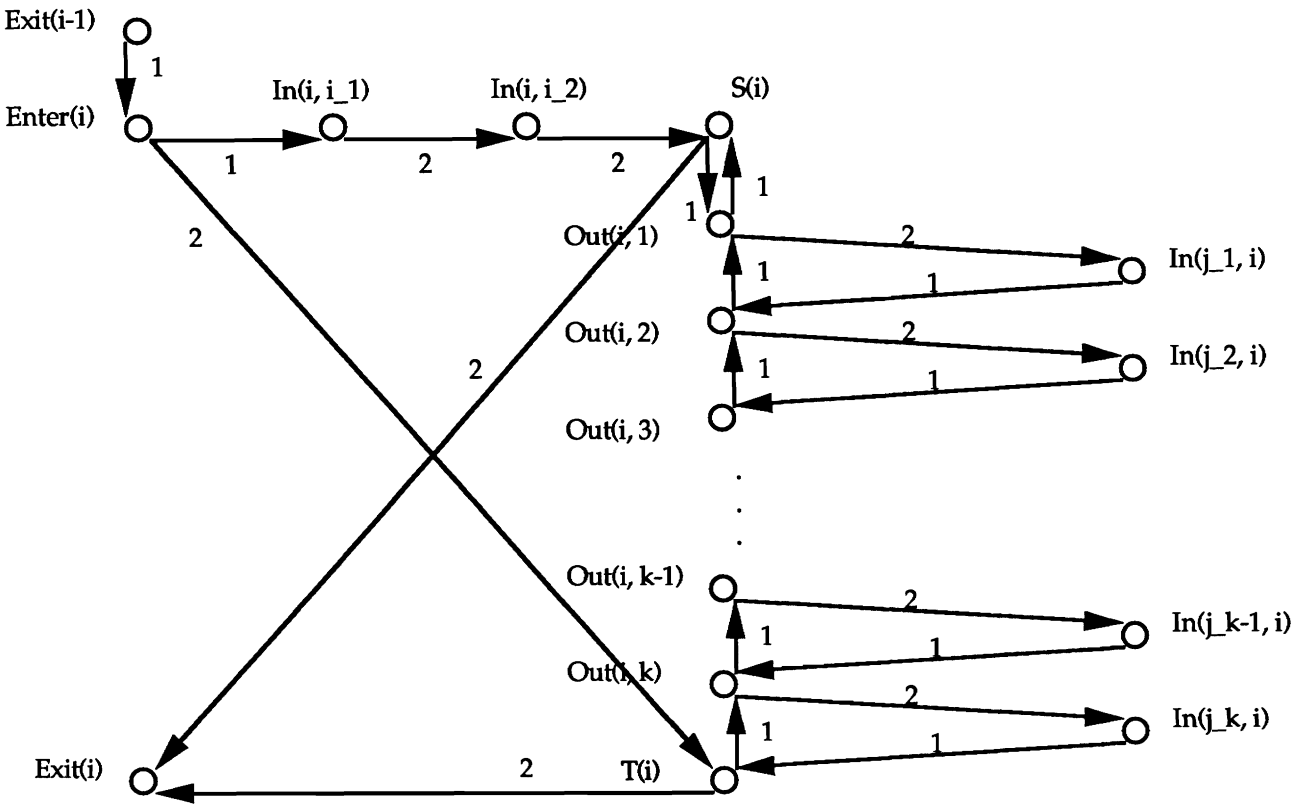

Figure 1: The digraph gadget G_i for Boolean operation $B_i = \neg(B_{i_1} \vee B_{i_2})$ in B, where B_i occurs in subsequent operations $B_{j_1}, B_{j_2}, \ldots, B_{j_k}$.

look as they do in Figure 2. Conversely, suppose that we have previously visited either In(i, i_1) or In(i, i_2) or both; then the DFS tree edges for G_i will look as they do in Figure 3.

Let G be the graph consisting of the union of all the G_i's, and let Exit(0) be the root of G. We have the following lemma.

Lemma 1: In a depth-first search of G, we visit $S(n)$ before $T(n)$ iff *value*$(B) =$ *true*.

Proof: We prove by induction that, for all i, $1 \leq i \leq n$, *value*$(B_i) =$ *true* iff the DFS tree edges in G_i are as shown in Figure 2, and *value*$(B_i) =$ *false* iff they are as shown in Figure 3.

$\Rightarrow$: The base case is trivial. Fix an i and assume that $B_i = \neg(B_{i_1} \vee B_{i_2})$ (where $i_1, i_2 < i$) and the condition holds for all $i' < i$. If *value*$(B_i) =$ *true*, then *value*$(B_{i_1}) =$ *value*$(B_{i_2}) =$ *false*. By the inductive hypothesis, the DFS trees for G_{i_1} and G_{i_2} are as shown in Figure 3, which means that we have not already visited either In(i, i_1) or In(i, i_2), so the DFS tree for G_i must be as shown in Figure 2. Conversely, suppose that *value*$(B_i) =$ *false*. Then either *value*$(B_{i_1}) =$ *true* or *value*(B_{i_2}) $=$ *true*, so by the inductive hypothesis, the DFS tree for either G_{i_1} or G_{i_2} is as in Figure 2, which means that we have already visited either In(i, i_1) or In(i, i_2), so the DFS tree for G_i must be as shown in Figure 3.

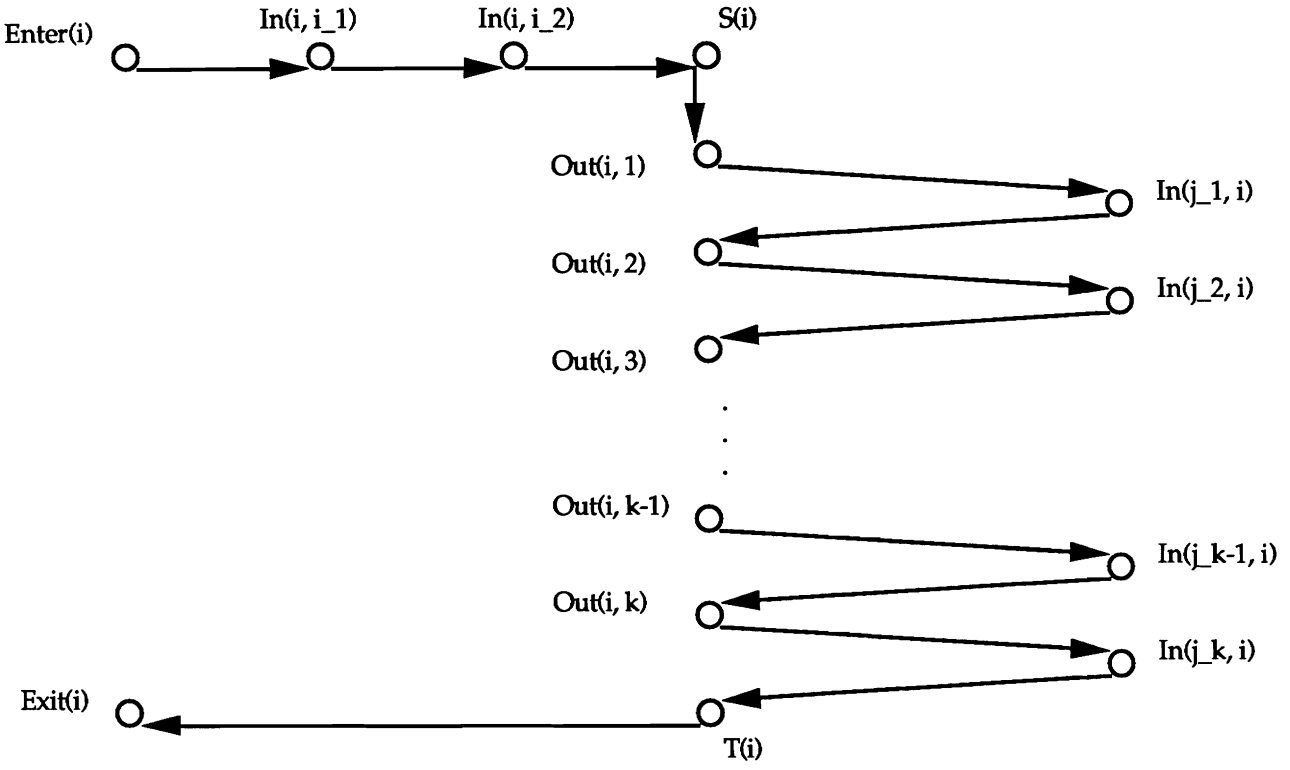

Figure 2: The DFS tree edges for G_i, assuming we have not previously visited either $\mathrm{In}(i, i_1)$ or $\mathrm{In}(i, i_2)$.

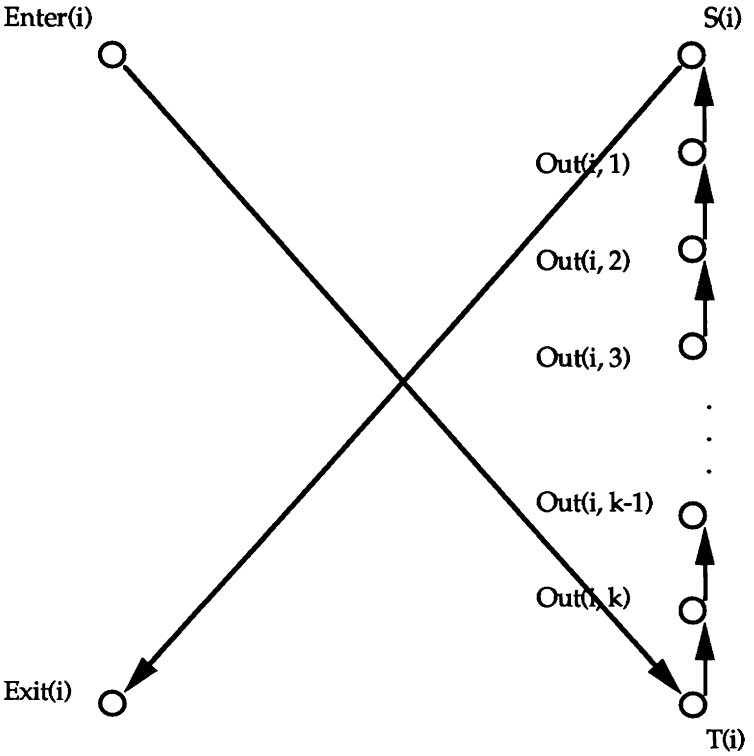

Figure 3: The DFS tree edges for G_i, assuming we have already visited either $\mathrm{In}(i, i_1)$ or $\mathrm{In}(i, i_2)$ or both.

$\Leftarrow$: The argument is similar.

The lemma follows immediately when we let $i = n$. □

Given B, a deterministic Turing machine can easily construct G in log-space. This fact, together with Lemma 1 and Proposition 1, implies that DFS-ORDER is P-complete. □

Theorem 2: UDFS-ORDER is P-complete.

Proof: Given the directed graph $G = (V, E)$ defined above, we create an undirected graph $G' = (V, E')$ by replacing every edge (u, v) or $(v, u) \in E$ with an undirected edge $\{u, v\}$ such that the following property holds: For each vertex $v \in V$, let $E_{v_{\text{out}}}$ and $E_{v_{\text{in}}}$ be the edges in G leaving v and entering v, respectively. In G', after visiting v we first visit the edges in $E_{v_{\text{out}}}$, in exactly the same order in which we visited them in G; then we visit the edges in $E_{v_{\text{in}}}$ in any order whatsoever. The root of G' is also Exit(0). It is easy to see that G and G' have the same DFS trees. Therefore Lemma 1 holds for G' as well, and the theorem follows. □

7.1 Implications

Reif's proof helped to support the belief that DFS is inherently sequential. There are several problems other than DFS, however, whose lexicographic versions are P-complete but whose general versions are in either NC or RNC, including the maximal independent set problem [37] and the maximal path problem [7]. Thus it seems that the complexity of a problem's lexicographic version does not tell us much about the complexity of its general version.

8 Smith's NC Algorithm for Planar Graphs

Smith showed that DFS of undirected planar graphs is in NC. This was the first fast parallel algorithm for any significant special case of DFS. We outline his algorithm below, but first we present some definitions and preliminary results [52].

8.1 Definitions and Essential Theorems

Proposition 2: Let G be a connected graph with biconnected components G_i and a given vertex x. For each G_i, let x_i be the articulation point of G_i connected to x by the shortest possible path. If T_i is a DFS tree of G_i rooted at x_i, then the union of the T_i's is a DFS tree for G rooted at x. □

Let G be a graph and suppose that P is a simple path in G such that $G - P$ is a disjoint union of components $\{G_i\}$ and one end of P is called its *root*. Let e be an edge in G. We say that e *touches* P if precisely one vertex in e is in P. e in *inessential* for G_i if e touches P but there exists another edge e' in G_i that touches P at a point farther away from the root than the point at which e touches P; otherwise e is *essential* for G_i.

Given these definitions, *the reduction of G by P* is a graph G' that results from deleting all of the inessential edges (for the components of $G - P$) from G (and not deleting their end vertices). G' consists of the union of P, the essential edges of G, and the components $\{G_i\}$ of the graph that results from deleting all of the vertices of P from G.

Proposition 3: Let e_i be an essential edge for G_i connecting G_i to P and let T_i be a DFS tree of G_i rooted at the end vertex of e_i that is in G_i. Then the union of P, the $\{e_i\}$ and the $\{T_i\}$ is a DFS tree of G.

Proof: Let T denote this union. We prove that T satisfies Tarjan's condition for a DFS tree[2]: if we say that $v_1 < v_2$ whenever v_2 is an ancestor of v_1 in T, then T is a DFS tree iff for all edges $(u, v) \in E$, either $u < v$ or $v < u$ [54]. Consider an edge e in G. If e connects two vertices in some tree T_i, then they will be comparable, because the T_i's are DFS trees by assumption. e cannot connect two distinct trees T_i and T_j because, again by assumption, they are contained in distinct components of $G - P$. Finally, consider the case where $e = \{v_{T_i}, v_P\}$ connects a tree T_i to P. e is either inessential or essential for G_i. If it is inessential, its end vertices are comparable because every vertex in T_i is a descendant of v_P. If it is essential, then its end vertices are comparable because e and e_i both touch P at v_P, e_i's other vertex is trivially a descendant of v_P, and v_{T_i} is a descendant of e_i's other vertex. □

Let G be a connected graph. Define G_s, the *solid subgraph*, to consist of all the edges in G contained in some cycle; define G_t, the *treelike subgraph*, to consist of all other edges in G. We can decompose G into G_s and G_t in (parallel) time $O(\log^2 n)$ [11].

The following theorem is the most significant result in the paper.

Theorem 3: Let G be a planar, biconnected graph with n vertices. Then there exists a parallel algorithm for finding a simple cycle in G with the property that its interior and exterior subgraphs have $\leq 2n/3$ vertices. This algorithm runs in time $O(\log^3 n)$ (and requires $O(n^4)$ processors).

[2] This condition is equivalent to the condition we gave in Section 1.1.

Proof: The proof is complicated and is not suitable for presentation in a paper of this nature. □

8.2 Description of the Algorithm

Here is a high-level description of the algorithm [52].

Input: An undirected planar graph G and an entry vertex r.

Output: A DFS tree T for G rooted at r.

Method:

1. Embed G in the plane. (See [32] for a description of what this operation involves.)
2. Find the biconnected components of G.
3. Find G_s and G_t and add G_t to T. The remainder of the algorithm is performed on G_s.
4. For each biconnected component C of G_s, do in parallel:
   (a) For C_s, find a partitioning cycle, as described above.
   (b) Find a path connecting the entry vertex to the partitioning cycle:
      i. Find a spanning tree of C_s using the method in [11].
      ii. Direct the spanning tree, with the entry vertex as its root, using the Euler tour method [56].
      iii. Delete all branches of the spanning tree that are directed away from the partitioning cycle.
      iv. Select a branch of the directed spanning tree that enters the partitioning cycle and propagate a marker backwards to the root. This is the path that connects the root to the partitioning cycle.
   (c) Delete an edge of the partitioning cycle (found in 4.a.) that is adjacent to the point where the path found in 4.b. touches it. Call the resulting path P and add it to T.
   (d) Find the set $\{C_i\}$ of connected components of $C - P$.
   (e) For each of the C_i's, find the set of all edges that touch P and determine which of them are essential.
   (f) Select any one of the essential edges. Let v_i be the endpoint of this edge that is not in P.

(g) Apply steps 3 and 4 to each of the C_i's using the v_i's as the entry vertices.

The correctness of this algorithm follows from Proposition 3.

8.3 Analysis

Let us analyze this algorithm step by step. Step 1 can be done in time $O(\log^2 n)$ [32]. Step 2 can also be done in time $O(\log^2 n)$ [56]. Step 4.a. takes time $O(\log^2 n)$, as mentioned above. Step 4.b. also takes total time $O(\log^2 n)$. Step 4.c. takes unit time; step 4.d. takes time $O(\log^2 n)$ [11]; and finally, step 4.e. takes time $O(\log n)$. Therefore one phase of this algorithm takes time $O(\log^2 n)$. Since each of the components we find in step 4.d. has $\leq 2n/3$ vertices, we will only have to repeat steps 3 and 4 $O(\log n)$ times, for a total time bound of $O(\log^3 n)$.

Before moving on to third and final paper, we discuss one more supplementary topic—the problem of *matching*. Matching is easy to think about and has many important applications, as we will see.

9 Matching

Matching is a problem that many researchers know well but does not appear in most introductory algorithms textbooks. Matching algorithms show up as subroutines in many other important algorithms, including a DFS algorithm that we will describe below.

In this section, we define the most common matching problems, list some important applications of matching, present a history of matching algorithms, and provide a list of open problems.

9.1 Definitions

A *matching* in a finite undirected graph $G = (V, E)$ is a set $M \subseteq E$ such that no vertex in V is incident with more than one edge in M. A matching on a bipartite graph is called a *bipartite matching*. Some people say "matching" when they really mean "bipartite matching", thereby causing needless confusion. A matching of maximum cardinality is called a *maximum matching*; a matching that includes every vertex in G (when it exists) is called a *perfect matching*. If every edge has a numerical weight, the weight of a set of edges is the sum of the weights of each edge. A *minimum weight perfect matching* is a perfect matching of smallest possible weight; a *maximum weight perfect matching* is a perfect matching of largest possible weight. Each of these kinds of matchings defines a particular matching problem, of course.

9.2 Applications of Matching

There are many applications of matching, both general and bipartite. matching. Applications of bipartite matching seem to outnumber applications of general matching, however, which explains why so many researchers have focused on speeding up bipartite matching.

Applications of bipartite matching include: the assignment problem, finding chain decompositions in posets, finding coset representatives in groups, finding systems of distinct representatives, finding block-triangular decompositions of sparse matrices [31], and the maximal path problem [7].

Applications of general matching include scheduling the tasks of multiprocessor computers and scheduling transmissions on packet radio networks [44], and DFS in both directed and undirected graphs [3, 4].

9.3 Relationships Among Matching Problems

We should note that the various matching problems listed above are not completely independent of one another. For example, perfect matching trivially reduces to maximum matching, since every perfect matching is maximum; we need only find a maximum matching and then check whether it has cardinality $n/2$. Also, maximum matching reduces to minimum-weight (or maximum-weight) perfect matching on an edge-weighted graph in which every edge has weight 0 or 1: Given the original graph G, we assign each of its edges a weight of 0, then turn it into a complete graph G' by adding extra edges of weight 1. Then a maximum matching on G is clearly a minimum-weight perfect matching on G', and vice-versa [36].

9.4 History of Matching Algorithms

Here are the most significant breakthroughs in matching algorithms (both general and bipartite), in chronological order:

- Edmonds showed that general matching was in P, thereby solving an important open problem [16]. He did not formally analyze the running time of his algorithm, but a straightforward implementation takes time $O(n^4)$ [43].
- Hopcroft and Karp gave an $O(n^{1/2}m)$ algorithm for maximum matching in bipartite graphs [31].
- Gabow improved the running time of Edmonds's algorithm to $O(n^3)$ [20].

- Kameda and Munro gave an $O(nm)$ algorithm for general matching [34].
- Even and Kariv gave an $O(n^{5/2})$ algorithm for general matching [17].
- Micali and Vazirani gave an $O(n^{1/2}m)$ algorithm for general matching [40]. Peterson and Loui correct a few minor errors in the original paper and give a clear exposition of the algorithm [44]. This is still the best known deterministic algorithm for general matching.
- Borodin et. al. showed that the *size* of a maximum matching in a bipartite graph is in *RNC*. Running time: $\tilde{O}(\log^2 n)$ [9].
- Karp et. al. showed that constructing a perfect matching in a general graph is in *RNC*. Running time: $\tilde{O}(\log^3 n)$ [36].
- Galil and Pan improved the processor bound of Karp et. al.'s algorithm by a factor of n^4 without increasing the running time [22].
- Mulmuley et. al. gave a randomized algorithm for *maximum* matching in general graphs that takes time $\tilde{O}(\log^2 n)$ [42].
- Gabow and Tarjan gave a deterministic algorithm for bipartite matching whose worst-case running time is $O(n \log^2 n)$ [21].
- Goldberg, Plotkin and Vaidya gave the first *sub-linear* deterministic algorithm for maximum matchings in bipartite graphs. Running time: $O(n^{2/3} \log^3 n)$ [25].
- Goldberg, Plotkin, Shmoys, and Tardos gave a deterministic algorithm for bipartite matching that runs in time $O(\sqrt{m} \log^k m)$ for some constant k. This algorithm is faster than the previous one for graphs of low to moderate density [24].

9.5 Open Problems

As the above history indicates, the best known randomized algorithms for bipartite and general matching are much faster than their deterministic counterparts. Therefore the only significant open problems that remain concern deterministic matching. First, is bipartite matching in *NC*? And second, is there a deterministic algorithm for general matching that runs in quadratic time (or better)?

10 Aggarwal and Anderson's *RNC* Algorithm

In this section, we describe the first *RNC* algorithm for DFS in general undirected graphs, due to Aggarwal and Anderson [3]. Their algorithm calls general matching algorithms as subroutines, and these subroutines are the costliest parts of the algorithm in terms of both time and processors. This algorithm is in *RNC* because general matching is in *RNC*; if general matching were in *NC*, this algorithm would be too.

10.1 Definitions

A *path* in a graph $G = (V, E)$ is an ordered set of distinct vertices $p = p_1, p_2, \ldots, p_k$ with edges $(p_i, p_{i+1}) \in E$ for $1 \leq i \leq k$. A set Q of vertex disjoint paths is a *separator* if the largest connected component of $V - Q$ has size at most $n/2$. An *initial segment* of G is a rooted subtree T' that can be extended to some DFS tree T.

10.2 Overview of the Algorithm

Here is a high-level description of the algorithm [3]. Given G and r, we construct an initial segment T' rooted at r such that the largest connected component of $V - T'$ has size at most $n/2$. Let the connected components of $V - T'$ be $C_1, C_2, \ldots, C_p$. For each C_i, there is a unique vertex $x_i \in T'$ of greatest depth that is adjacent to some vertex y_i of C_i. For each C_i (in parallel), we find these two vertices, recursively construct a DFS tree for C_i rooted at y_i, and then connect this tree to T' with an edge from x_i to y_i. The size of the recursive problem is at most half of the size of the original problem, so the running time of the algorithm is $O(\log n)$ times the time needed to construct T'.

To construct an initial segment, we first construct a separator Q such that the number of paths in Q is less than a fixed constant (hence Q is a "small separator"), then we construct the initial segment from Q. Constructing Q is the most complicated and time-consuming part of the algorithm. We discuss each of these two tasks in order below.

10.3 Constructing a Separator

We wish to construct a set of vertex disjoint paths $Q = \{q_1, q_2, \ldots, q_k\}$ where k is less than a fixed constant and the largest component of $V - Q$ has size at most $n/2$. To do this, we make use of a procedure *Reduce*(Q) which reduces the number of paths in Q by a constant fraction while maintaining the separator property.

Q is initially just V, and the separator property trivially holds. Each call to *Reduce* eliminates $1/12$ of the paths in Q, so we only have to call it $O(\log n)$ times to ensure that Q contains no more than 11 paths. Then we can use Q to construct an initial segment (see Section 10.6).

10.4 Reducing the Number of Paths

We now describe the operation of *Reduce* [3]. The basic idea is to find a set of vertex disjoint paths between pairs of paths in Q, then merge each of these pairs. We divide the paths in Q into two sets, L and S (for *L*ong and *S*hort paths, respectively). Suppose we find a set of vertex disjoint paths P, and consider a path $p \in P$ that connects a path $l \in L$ to a path $s \in S$. If the endpoints of p are x and y, then $l = l'xl''$ and $s = s'ys''$. Now we would like to use p to merge l and s. If $|s'| \geq |s''|$, then we replace l by $l'xs'$ and replace s by s''; otherwise, we replace l by $l'xs''$ and replace s by s'. In either case, we discard l''. We have now decreased the length of s by at least half, and we may be able to merge s completely, thereby eliminating it from S. Clearly, we merge these (l, s) pairs in parallel. Our goal is to proceed in this way until we have reduced the number of paths in Q by a constant fraction. Note that we also want P to satisfy two obvious constraints: first, P should be as large as possible; and second, the lengths of the segments we discard (l'' in our example) should be as large as possible.

Unfortunately, the process we have described will not necessarily work the way we would like it to. There are two things that could go wrong:

1. Discarding a segment (i.e., l'') might cause two of the connected components of $V - Q$ to merge, thereby violating the separator property.

2. The set of vertex disjoint paths we find might not be large enough to substantially decrease the lengths of the paths in S.

Let us assume for the moment that neither of these cases arises. If initially $|Q| = K$, we begin by putting $K/4$ paths in L and $3K/4$ paths in S; then we find a set of vertex disjoint paths $P = \{p_1, p_2, \ldots, p_\alpha\}$. We take constraint 1 to mean that the number of paths found must not be less than $K/12$, so we assume that $|P| > K/12$, i.e., we can reduce the lengths of at least $K/12$ paths in S by at least half. Since there are $3K/4$ paths in S initially, we will only have to find and merge paths $9 \log n$ times at the most before S becomes empty. We repeat these steps until $|Q| \leq 11K/12$.

If either of these cases arises, it turns out that, as long as P satisfies the constraints listed above, we can still reduce the number of paths in Q by a constant fraction. First, we consider case 1. Let $\hat{L}$ be the paths in L not joined to paths in P, $\hat{S}$ be the paths in S not joined to paths in P, and L^* be the set of path segments in L that are in danger of being discarded. Finally, let $T = V - Q - P$ be the set of vertices not on any path. We have the following lemma:

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: Since Q is a separator for G, the largest connected component of T has size at most $n/2$. There cannot be a path from any vertex in L^* to any vertex in $S \cup \hat{S}$ that uses vertices of T; if there were, we could have found a different set P with the same cardinality and strictly less cost. The induced subgraph on $T \cup L^* \cup (S - \hat{S})$ must contain at least 2 connected components. Either the components containing vertices of L^* or the ones containing vertices of $S - \hat{S}$ must have total size at most $n/2$ since L^* and $S - \hat{S}$ are in different components. We assumed that the largest connected component of $T \cup L^*$ has size at least $n/2$, so the size of the components containing $S - \hat{S}$ in $T \cup L^* \cup (S - \hat{S})$ must be at most $n/2$. Thus the largest connected component of $T \cup (S - \hat{S})$ has size at most $n/2$. □

Thus if case 1 arises, we can simply add the paths in $S - \hat{S}$ to T instead. This will reduce the total number of paths in Q by a constant fraction [3]. If case 2 arises, we can show by similar reasoning that we can discard either the paths in $L - \hat{L}$ or the paths in $S - \hat{S}$ without violating the separator property; this will also reduce the number of paths in Q by a constant fraction.

10.5 Finding Sets of Disjoint Paths

Given L and S, we need to find a set of vertex disjoint paths $P = \{p_1, p_2, \ldots, p_\alpha\}$ such that α is as large as possible and, when we use P to merge L and S, the lengths of the paths we discard are as small as possible. We do this by reducing the problem to two general matching problems (this approach, as opposed to more direct approaches, has a lower processor bound) [3].

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 0 or 1 only.

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.

Proofs: The proofs are fairly difficult and are ignored. □

Given these lemmas, we can find a min-cost maximum matching in two steps: first we apply the reduction of Lemma 3 to find the *size* of the matching we want, and then we apply the reduction of Lemma 4 to find the matching itself. We can do both reductions in $O(\log n)$ time using n^2 processors.

10.6 Constructing an Initial Segment

Once we have constructed the small separator Q, we must use it to construct an initial segment T' rooted at r such that the largest connected component of $V - T'$ has size at most $n/2$. This part of the algorithm is straightforward.

Initially, T' is just r. We repeatedly pick a path $q \in Q$ and extend T' to contain at least half of q. Since the number of paths in Q is a constant (at most 11), we can perform this operation sequentially. Given q, we find the lowest vertex in T' from which there is a path to q. Suppose that we find such a vertex x, and the path from x leads to a vertex $y \in q$, i.e., $q = q'yq''$. If q' is at least as long as q'', we add the path pyq' to T' and replace q by q'', otherwise we add the path pyq'' to T' and replace q by q'. Either way, we reduce the length of q by at least half, so we will repeat this process at most $11 \log n$ times.

We need to show that this algorithm constructs an initial segment with the desired properties. To show that we can extend T' to a DFS tree, it suffices to show that there are no paths between separate branches of T' whose interior vertices are all in $V - T'$, but this condition holds throughout the execution of the algorithm because we extend T' from the lowest possible vertex. Also, the largest connected component of $V - T'$ has size at most $n/2$, because T' consists entirely of paths in Q and Q is a separator.

10.7 Pseudo-Code and Run-Time Analysis

We can describe the algorithm more formally with the following pseudo-code [3]:

procedure *DFS*(G, r)
  $T' \leftarrow$ *Initial-Segment*(G, r);

```
    for each connected component C of G − T' do
        Recursively compute a DFS tree for C;
        Add this tree to T';
    end { for }
end

procedure Initial-Segment(G, r)
    Q ← V;
    while |Q| > 11 do
        Reduce(Q);
    end { while }
    Build the initial segment from Q;
end

procedure Reduce(Q)
    K ← |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, ..., p_α} between L and S;
        if α < K/12 then
            if lcc(T ∪ (S − Ŝ)) < n/2 then
                Q ← L ∪ Ŝ ∪ P;
            else
                Q ← S ∪ L̂ ∪ P;
            return
        else if lcc(T ∪ L*) > n/2 then
            Q ← L ∪ Ŝ ∪ P;
            return
        else
            Extend the paths of L̂. Suppose p joins l and s, x and y are the endpoints of p
            and l = l'xl'', x = s'ys''. If |s'| ≥ |s''| then l ← l'ps' and s ← s'';
            otherwise, l ← l'ps'' and s ← s'. In both cases, l'' is discarded.
    end { while }
end
```

Let us analyze the running time of this algorithm. The running time of *DFS* is $O(\log n)$ times the running time of *Initial-Segment*, which is $O(\log n)$ times the running time of *Reduce*, which is $O(\log n)$ times the time needed to find mincost disjoint paths. Thus the running time is $O(\log^3 n)$ times the time needed to find mincost disjoint paths. We reduced this last step to two successive matching problems. The best known algorithm for matching takes time $\tilde{O}(\log^2 n)$ [42], so the entire algorithm takes time $\tilde{O}(\log^5 n)$.

11 Conclusion

We have discussed three important papers in the history of parallel DFS algorithms. The result of the first one seemed to imply that DFS was inherently sequential. The second one established that an important special case of DFS, namely DFS in planar undirected graphs, is in *NC*. The third one showed that DFS in general undirected graphs is in *RNC*. We have also discussed several other peripheral topics, including *P*-completeness, the parallel computation thesis, and matching.

The two most important open problems for DFS are these: Is DFS for either directed or undirected graphs in *NC*? Also, since the complexity of Aggarwal and Anderson's algorithm depends so strongly on the complexity of general matching, we would like to know: Is general matching in *NC*?

Research on parallel DFS algorithms (and on parallel matching algorithms) has progressed rapidly in the last few years. It would not be unrealistic to expect an answer to one or more of these problems sometime in the near future.

References

[1] L. Adleman, C. Pomerance, and R. S. Rumley. On distinguishing prime numbers from composite numbers. *Annals of Math*, 117:173–206, January 1983.

[2] Leonard Adleman and Kenneth Manders. Reducibility, randomness, and intractibility. In *Proc. 9th ACM Symposium on Theory of Computing*, pages 151–163, 1977.

[3] Alok Aggarwal and Richard J. Anderson. A random *NC* algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[4] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM Journal on Computing*, 19(2):397–409, April 1990.

[5] A. U. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[6] R. J. Anderson. A parallel algorithm for depth-first search. 1986. *Extended Abstract, Mathematical Science Research Institute* (1986).

[7] R. J. Anderson. A parallel algorithm for the maximal path problem. *Combinatorica*, 7(4):315–326, 1987.

[8] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6:733–744, 1977.

[9] Allan Borodin, Joachim von zur Gathen, and John Hopcroft. Fast parallel matrix and GCD computations. *Information and Control*, 52:241–256, 1982.

[10] A. K. Chandra and L. J. Stockmeyer. Alternation. In *Proc. 17th IEEE Annual Symposium on the Foundations of Computer Science*, pages 98–108, October 1976.

[11] F. Y. Chin, J. Lam, and I. Chen. Optimal parallel algorithms for connected-component problems. In *Proc. 1981 Conference on Parallel Processing*, pages 170–175, 1981.

[12] Stephen A. Cook. An observation on time-storage trade off. *Journal of Computer and System Sciences*, 9:308–316, 1974.

[13] Stephen A. Cook. An overview of computational complexity. *Communications of the ACM*, 26(6):401–408, June 1983.

[14] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[15] D. Eckstein and D. Alton. Parallel graph processing using depth first search. In *Proc. of the Conference on Theoretical Computer Science at the University of Waterloo*, pages 21–29, 1977.

[16] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[17] S. Even and O. Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *Proc. 16th IEEE Annual Symposium on the Foundations of Computer Science*, pages 100–112, 1975.

[18] Shimon Even and R. Endre Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, December 1975.

[19] Steven Fortune and James Wyllie. Parallelism in random-access machines. In *Proc. 10th ACM Symposium on Theory of Computation*, pages 114–118, 1978.

[20] Harold N. Gabow. An efficient implementation of edmonds's algorithm for maximum matching on graphs. *Journal of the ACM*, 23(2):221–234, April 1976.

[21] Harold N. Gabow and Robert E. Tarjan. Almost-optimum speedups of algorithms for bipartite matching and related problems. In *Proc. 20th ACM Symposium on Theory of Computing*, pages 514–527, 1988.

[22] Z. Galil and V. Pan. Improved processor bounds for combinatorial problems in *RNC. Combinatorica*, 8(2):189–200, 1988.

[23] Ratan K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24:134–150, 1984.

[24] Andrew V. Goldberg, Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Interior-point methods in parallel computation. In *Proc. 30th IEEE Annual Symposium on the Foundations of Computer Science*, pages 350–355, 1989.

[25] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Proc. 29th IEEE Annual Symposium on the Foundations of Computer Science*, pages 174–185, 1988.

[26] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is log-space complete for *P. Theoretical Computer Science*, 21:105–111, 1982.

[27] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proc. 10th Annual ACM Symposium on the Theory of Computing*, pages 89–94, 1978.

[28] Torben Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, August 1990.

[29] J. E. Hopcroft and R. M. Karp. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, October 1974.

[30] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[31] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973.

[32] Joseph Ja' Ja' and Janos Simon. Parallel algorithms in graph theory: planarity testing. *SIAM Journal on Computing*, 11(2):314–328, May 1982.

[33] Neil D. Jones and William T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3:105–117, 1977.

[34] T. Kameda and I. Munro. An $O(|V| \cdot |E|)$ algorithm for maximum matching of graphs. *Computing*, 12:91–98, 1974.

[35] Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in *DNC*. In *Proc. 3rd Aegean Workshop on Computing*, Corfu, Greece, 1988. Springer-Verlag Lecture Notes in Computer Science 319, pp. 53–63.

[36] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random *NC*. *Combinatorica*, 6(1):35–48, 1986.

[37] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32:762–773, 1985.

[38] R. E. Ladner. The circuit value problem is log space complete for *P*. *SIGACT News*, 7(1):18–20, 1975.

[39] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, April 1979.

[40] S. Micali and V. Vazirani. An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st IEEE Annual Symposium on the Foundations of Computer Science*, pages 17–27, 1980.

[41] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.

[42] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[43] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[44] Paul A. Peterson and Michael C. Loui. The general maximum matching algorithm of Micali and Vazirani. *Algorithmica*, 3:511–533, 1988.

[45] Nicholas Pippenger. On simultaneous resource bounds (preliminary version). In *Proc. 20th IEEE Symposium on Foundations of Computer Science*, pages 307–311, 1979.

[46] Vaughan R. Pratt and Larry J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12:198–221, 1976.

[47] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–36, Academic Press, New York, 1976.

[48] Sanguthevar Rajasekaran and John H. Reif. Randomized parallel computation. In *Proc. International Conference on Fundamentals of Computation Theory*, Kazan, June 1987. Springer-Verlag Lecture Notes in Computer Science 278, pp. 364–376.

[49] Eshrat Reghbati and D. G. Corneil. Parallel computations in graph theory. *SIAM Journal on Computing*, 7(2):230–237, May 1978.

[50] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[51] John H. Reif. On synchronous parallel computations with independent probabilistic choice. *SIAM Journal on Computing*, 13(1):46–56, 1984.

[52] Justin R. Smith. Parallel algorithms for depth-first searches I. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.

[53] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977.

[54] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[55] Robert Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, March 1974.

[56] Robert E. Tarjan and Uzi Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proc. 25th IEEE Annual Symposium on Foundations of Computer Science*, pages 12–20, 1984.

[57] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal of Computing*, 14:348–355, 1985.

[58] J. C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell University, Ithaca, NY, 1979. Also available as Technical Report 79-387.

[59] Y. Zhang. *Parallel algorithms for problems involving directed graphs.* PhD thesis, Drexel University, Philadelphia, PA, 1986.