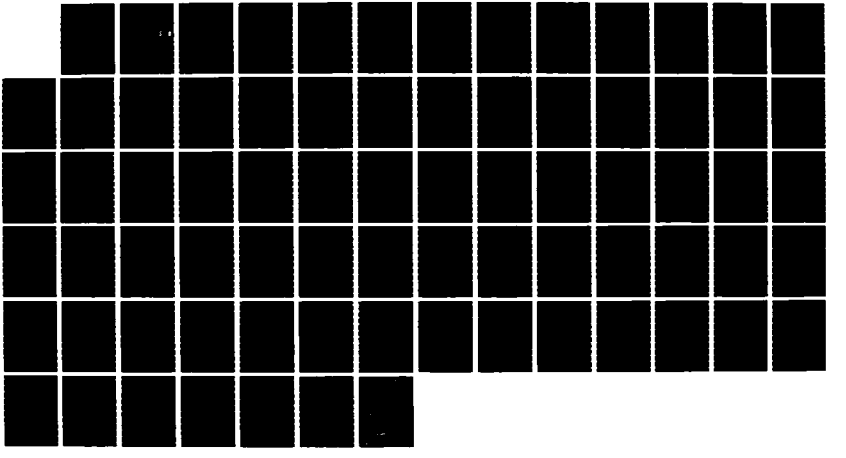
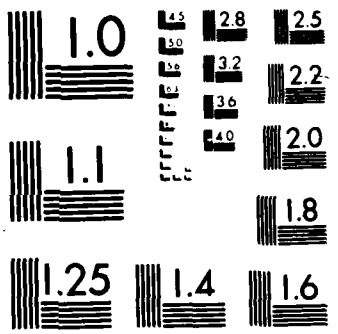
AD-A172 445 THE COMPLEXITY OF PARALLEL ALGORITHMS(U) STANFORD UNIV 1/1
CA DEPT OF COMPUTER SCIENCE R ANDERSON NOV 85
STAN-CS-86-1092 N00014-85-C-0731

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0

November 1985

Report No. STAN-CS-86-1092

(12)

The Complexity of Parallel Algorithms

by

Richard Anderson

Department of Computer Science

Stanford University
Stanford, CA 94305

N00014-85-C-0731

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Stanford University
Department of Computer Science

The Complexity of Parallel Algorithms

by

Richard Anderson

This research was supported in part by a National Science Foundation Graduate Fellowship, an IBM Faculty Development Award, and National Science Foundation grant DCR–8351757. Much of this work was done in collaboration with my advisor, Ernst Mayr. He was also supported in part by ONR contract N00014-85-C-0731.

Table of Contents

Abstract

This thesis addresses a number of theoretical issues in parallel computation. There are many open questions relating to what can be done with parallel computers and what are the most effective techniques to use to develop parallel algorithms. We examine various problems in hope of gaining insight to the general questions.

One topic that is investigated is the relationship between sequential and parallel algorithms. We introduce the concept of a P-complete algorithm to capture what it means for an algorithm to be inherently sequential. We show that a number of sequential greedy algorithms are P-complete, including the greedy algorithm for finding a path in a graph. However, a problem is not necessarily difficult if an algorithm to solve it is P-complete. In some cases, the natural sequential algorithm is P-complete but a different technique gives a fast parallel algorithm. This shows that it is necessary to use different techniques for parallel computation than are used for sequential computation.

We give fast parallel algorithms for a number of simple graph theory problems. The algorithms illustrate a number of different techniques that are useful for parallel algorithms. The most important results are that the maximal path problem can be solved in $\mathcal{RNC}$ and that a depth first search tree can be constructed in $O(n^{1/2+\epsilon})$ parallel time. This shows that substantial speed up is possible for both of these problems using parallelism.

The final topic that we address is parallel approximation of P-complete problems. P-complete problems probably cannot be solved by fast parallel algorithms. We give a number of results on approximating P-complete with parallel algorithms that are similar to results on approximating NP-complete problems with sequential algorithms. We give upper and lower bounds on the degree of approximation that is possible for some problems. We also investigate the role that numbers play in P-complete problems, showing that some P-complete problems remain difficult even if the numbers are small.

Chapter 1 Introduction

1.1. Parallel Computation

Parallel computation offers substantial opportunities for performing computations faster than they can be done with just a single processor. In some cases, problems can be decomposed into independent subproblems, with each subproblem being solved simultaneously. Ideally, this allows a speed up in the computation proportional to the number of processors employed. There are, however, many difficulties in parallel computation. Some of the difficulties are technological, relating to such issues as processor synchronization, resource contention, and wiring together processors. Other difficulties are more algorithmic in nature. These include the partitioning of problems into subproblems and the programming of parallel machines.

There is a substantial difference between parallel and sequential algorithms. Sequential algorithms can take advantage of many intermediate results. Processing can be done one step at a time, basing each decision on the previous decisions made. However, for a parallel algorithm to be efficient, the problem must be decomposed so that progress can be made on many subproblems at the same time. This often requires a very different approach than is used in sequential algorithms.

Two of the most important questions relating to parallel computation are: What are the problems that can be solved by fast parallel algorithms, and what general techniques work for parallel algorithms. Essentially the questions are "what" and "how." The classification problem is to identify the problems that can be effectively parallelized and also to identify the problems that require sequential processing. The other problem is to identify general algorithmic techniques. There are a number of general techniques that are commonly employed in sequential algorithms, such as *divide and conquer* and *dynamic programming*. It is important to develop a similar set of approaches for parallel algorithms. There are currently just a few techniques used in parallel algorithms. It is hoped that new techniques can be developed.

The goal of this thesis is to study particular problems to gain insight into these general questions. We take a theoretical approach by adopting an abstract model of parallel computation. The model of computation that we use is the P-RAM model. This is an idealization of a parallel machine. Most of the problems that we look at are very simple from the sequential point of view. However, these problems turn out to be a challenge to parallelize and illustrate many of the issues involved in parallel computation.

The study of parallel computation is very broad, and this thesis addresses issues related to only a portion of it. We are only concerned with what could be called "inherent parallelism." This refers to the parallelism possible in an ideal computer, where unit cost communication is available between all processors and memories, and there are no constraints imposed by physical layout. We neglect such issues as communication complexity and the layout of processors, although they are important, both in theory and in practice.

The next section covers some preliminary results that we base our work on. The discussion covers our model of parallel computation and also covers some of the methods that are available to show that problems are inherently sequential. Following the preliminary material is a discussion of some of the general issues in parallel computation. This thesis has three chapters of technical results. Chapter 2 is concerned with the relation between sequential and parallel algorithms. It discusses ways to show that certain algorithms are inherently sequential. Chapter 3 discusses some algorithms for certain path problems. The algorithms illustrate a number of important general techniques. Finally, Chapter 4 discusses ways to approximate problems that probably cannot be solved by fast parallel algorithms. The chapter shows that there is a high degree of similarity between parallel and sequential complexity theory.

1.2. Preliminaries

A substantial amount of work has been done in studying models of parallel computation in order to identify the appropriate theoretical basis for parallel computation. In this section we discuss some of the results that are directly relevant to our work. We do not attempt to give a complete survey of the various models of parallel computation. There are a number of papers that survey the work that has been done, including papers by Cook [C1], and Hoover and Ruzzo [HR].

1.2.1. The P-RAM Model

The standard model of synchronous parallel computation is the P-RAM (Parallel Random Access Machine). This model captures the intuitive idea of what a parallel machine is. The P-RAM model has been described by a number of authors [FW] [G2]. A P-RAM consists of a set of processors, $P_1, \ldots, P_n$ and a set of global memory cells, $M_1, \ldots, M_m$. Each processor is a RAM [AHU], with its own local memory. A processor can perform some standard arithmetic operations and can access its own memory with direct or indirect addressing. A processor can also communicate with the global memory by reading a value from or writing a value to any global memory cell. The global memory accesses are assumed to take unit time. This is one of the major idealizations of the P-RAM model. In a real parallel computer, one would expect that the access time is related to the number of processors. A P-RAM has a single program that all processors execute one step at a time. Each processor has a register which contains its processor number and instructions may depend on this number, so different processors may do different things on the same instruction. Some of the global memory cells are designated for the input to the problem and some of them are for the outputs. The time taken for an algorithm is the number of instructions that are executed. The space is the sum of the number of processors and the number of memory cells.

There are a number of variants of the P-RAM model that handle concurrent reads and concurrent writes to the global memory cells differently. The major variants are exclusive read, exclusive write (EREW), concurrent read, exclusive write (CREW) and concurrent

read, concurrent write (CRCW). For the latter model, there are additional variants on the nature of concurrent writes. There is a difference in the power of the various types of P-RAM. For example, it is trivial to compute the OR of n inputs with a CRCW P-RAM in constant time, while on a CREW P-RAM the problem requires $\Omega(\log n)$ time [CD]. There are also separation results for the various different types of CRCW P-RAMS [FRW]. However, the differences in the power of the models is not that great. It is not hard to show that a single instruction of an n processor, m memory CRCW P-RAM (any variant) can be simulated by an EREW P-RAM with nm processors and nm memories in $O(\log n + \log m)$ time. In this thesis, we use the CREW model. However, we are not that interested in the exact time or processor bounds of our algorithms, so our results carry over to the other variants of the P-RAM model.

1.2.2. Fast Parallel Algorithms

In this thesis, we deal with problems that can be solved by "fast" parallel algorithms that use a "reasonable" number of processors. The generally accepted definition of fast and reasonable is polylog ($O(\log^k n)$) parallel time and a polynomial number of processors [P]. This class is commonly referred to as $\mathcal{NC}$. The problems in $\mathcal{NC}$ are problems for which an exponential speedup is possible using parallelism; these problems can have their running times reduced from polynomial to polylog.

One of the reasons why $\mathcal{NC}$ is broadly accepted as the appropriate class to use in the study of parallelism is that it is a very robust complexity class. $\mathcal{NC}$ remains the same whether it is defined in terms of any variant of the P-RAM model, or in terms of some other models, such as uniform circuits [B][Ru]. More refined classes, such as problems that can be solved in $O(\log n)$ or $O(\log^2 n)$ parallel time depend upon the particular model of computation that is used. Even the weakest model of a P-RAM is an idealization of the type of machine that could actually be built. To convert the model to a more realistic model, such as a bounded degree network [Sc], a slow down of a factor of at least $\log n$ is needed. The theoretical models, however, are accurate models when factors of $\log n$ are ignored. The advantage of $\mathcal{NC}$ is that it allows us to ignore the factors of $\log n$ that separate the various models.

The class $\mathcal{RNC}$ is the probabilistic analogue of $\mathcal{NC}$. It denotes the set of problems that can be solved with a probabilistic P-RAM in polylog time with a polynomial number of processors. There are several ways that randomness can be introduced into the P-RAM model. For example, the processors can be given coins to flip or certain memory locations can be assigned random values at the start of the program's execution.

One of the drawbacks of the class $\mathcal{NC}$ is that many $\mathcal{NC}$ algorithms are reasonable only when the number of processors is very large. The basic problem is that $\log^k n$ is not a slowly growing function when n is small. The following table shows how large n must be so that $n \geq \log^k n$ for several values of k.

k	n
2	16
3	982
4	65,536
5	5,690,333
6	621,201,921

For example, if the constant factors are the same, an $O(\log^3 n)$ algorithm is not better than an $O(n)$ algorithm until n is about one thousand. The size of n where an $O(\log^3 n)$ algorithm is superior to an $O(n^{1/2})$ algorithm is astronomical. Showing that a problem is in $\mathcal{NC}$ is just the first step to getting a practical algorithm for the problem. The problems in $\mathcal{NC}$ can in principle be solved by fast parallel algorithms; they are not inherently sequential.

1.2.3. The Parallel Computation Thesis

Parallel time is closely related to sequential space. The parallel computation thesis is: For all reasonable models of computation, parallel time is polynomially related to sequential space [G2]. The equivalence of parallel time and sequential space has been proved for a number of specific models including alternating Turing machines [CKS], circuits [B] and P-RAMS [FW] [Wy]. Let PTIME($T(n)$) denote the class of problems that can be solved in $O(T(n))$ time on a CREW P-RAM and DSPACE($S(n)$) denote the class of problems that can be solved in $O(S(n))$ space on a multitape Turing machine. (For sequential space, only the work space is counted; the input is given on a separate read-only tape). It is shown in [FW] that PTIME($T(n)$) $\subseteq$ DSPACE($T^2(n)$) and DSPACE($S(n)$) $\subseteq$ PTIME($S(n)$) for $S(n) \geq \log n$.

The relationship between $\mathcal{NC}$ and Turing machine space is:

$$\text{DSPACE}(\log n) \subseteq \text{NSPACE}(\log n) \subseteq \mathcal{NC} \subseteq \bigcup_{k \geq 0} \text{DSPACE}(\log^k n).$$

To show that DSPACE($\log n$) $\subseteq \mathcal{NC}$, it is necessary to show that a Turing machine that uses $O(\log n)$ space can be simulated with a P-RAM with polynomial size. The size is polynomial since the number of possible states of the $O(\log n)$ space Turing machine is $O(n^k)$. A similar proof can be used to show that an $O(\log n)$ space non-deterministic Turing machine can be simulated by a P-RAM with polynomial size. The space efficient simulations of a polylog time P-RAM do not in general give polynomial time algorithms. However, $\mathcal{NC} \subseteq \mathcal{P}$, since a polynomial number of processors can be simulated by a single processor with a polynomial slowdown.

1.2.4. P-Completeness

One of the most difficult areas of complexity theory is lower bounds. Very few non-trivial lower bounds are known for general models of computation; this holds for parallel computation as well as for sequential computation. A different approach, which has proved far more successful is to show problems to be at least as difficult as other problems. The notion of *completeness* is a way to identify the most difficult problems in a particular class. A problem A is complete for a class C if it is in C and all problems in C are reducible to it by some appropriate form of reduction. If the problem A could be solved efficiently, then all problems in C could be solved efficiently by using the solution for A.

The parallel computation thesis allows us to apply results on space complexity to parallel computation. We show problems to be log-space complete for P (P-complete) to provide evidence that they are difficult to parallelize. A problem is log-space complete for P if it is in P and all problem in P are reducible to it by log-space reductions. A problem A is log-space reducible to a problem B if there exists a log-space Turing machine that converts instances of A into equivalent instances of B. Log-space reducibility is transitive, i.e., if A is reducible to B and B is reducible to C, then A is reducible to C. This means that if a P-complete problem could be solved in $O(\log^k n)$ space, then $P \subseteq \mathrm{DSPACE}(\log^k n)$.

If a problem is P-complete, then it is unlikely that there is a fast parallel algorithm for it. Log-space transformations can be done in $O(\log n)$ time on a P-RAM using a polynomial number of processors, so the P-complete problems are the most difficult problems in P to parallelize. If a P-complete problem is found to be in NC, then $P = NC$ and $P \subseteq \cup_{k>0}\mathrm{DSPACE}(\log^k n)$. If this were the case, then all problems in P could be solved very fast in parallel and could be solved sequentially using very little space. Both of these are considered to be very unlikely. There is of course, no known proof that $P \neq NC$, as there is no known proof that $P \neq NP$.

Many problems are known to be P-complete. An important P-completeness result is that the problem of computing the value of a circuit given its inputs is P-complete [Lad]. We discuss this problem in the next section. Other important P-complete problems are network flow [GSS], linear programming [DLR], and unification [DKM]. A list of currently known P-complete problems has been compiled by Hoover and Ruzzo [HR].

P-completeness is defined in terms of language recognition, so according to the definition, we are restricted to discussing problems that have a yes/no answer. However, it is often the case that we are interested in computing functions instead of just recognizing languages. For example, in the network flow problem, we wish to compute the value of the maximum flow in a network. One way to extend the definition of P-completeness to functions is to introduce a language associated with the function. For a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, we define the language

$$L_f = \{\langle x, k, a\rangle : \text{The } k\text{-th bit of } f(x) \text{ is } a\}.$$

By definition, the problem of computing f is P-complete if the problem of recognizing L_f is P-complete. The proof that network flow is P-complete [GSS] actually shows that the problem of computing the least significant bit of the maximum flow is P-complete.

1.2.5. The Circuit Value Problem

The fundamental P-complete problem is the circuit value problem. The circuit value problem is: Given a circuit with values for its inputs, compute the value of its output. The circuit value problem is clearly in $\mathcal{P}$, since it can be solved by evaluating the gates one at a time. Intuitively, a problem is P-complete if it is sufficiently powerful to simulate the computation of any polynomial time bounded Turing machine on a given input. The proof that the circuit value problem is P-complete is a generic reduction from an arbitrary problem in $\mathcal{P}$.

A problem can be shown to be P-complete by giving a log-space reduction from a known P-complete problem to it. The circuit value problem is by far the most frequently used problem for P-completeness proofs. The reason for this is that the circuit value problem seems to capture the complexity of P-completeness and the structure of the problem very often makes it convenient to use in reductions. Its role in P-completeness is similar to the role of satisfiability in NP-completeness. We now give a precise definition of the circuit value problem. A circuit is a string $\beta = \beta_1, \ldots, \beta_n$ where β_i is either an input, (0-INPUT or 1-INPUT), or a gate AND(j,k), OR(j,k), or NOT(j). The inputs for a gate are lower numbered gates, thus the gate β_i = AND(j,k) receives its inputs from the gates β_j and β_k with $j < i$ and $k < i$. The circuit value problem is to determine if a given string is in the language of all circuits that evaluate to *true*.

There are a number of important variants of the circuit value problem that are P-complete. The circuit value problem is P-complete for any collection of gates that form a complete basis, for example $\{NOT, OR\}$ or $\{NAND\}$. The circuit value problem is also P-complete for monotone circuits, i.e., if the logical gates are AND and OR [GJ]. A second version of the circuit value problem that is P-complete is the planar circuit value problem [G]. The problem is to evaluate a circuit that is laid out on the plane without wires crossing. The inputs are assumed to be along one edge of the circuit. The monotone planar circuit value problem, however, is apparently not P-complete, since it can be solved in $\mathcal{NC}$ [DC] [Ru].

There are a number of minor restrictions of the circuit value problem that are also P-complete. These are mentioned because they make a number of P-completeness proofs cleaner. The first restriction is that the gates are limited to having fanout at most two. It is not hard to simulate arbitrary fanout with a fanout two circuit by introducing extra gates. In all of the P-completeness proofs we give in this thesis we assume the logical gates are restricted to fanout two. We also assume that the inputs (0-INPUT and 1-INPUT), have fanout one. A second restriction is that the circuit can be assumed to be laid out in levels, with each gate connected only to gates on adjacent levels. The planar circuit value problem remains P-complete with this restriction even when the gates are restricted to NOT and OR (a one input OR is allowed). This variant is used in a proof in the next chapter.

P-completeness proofs are very similar to NP-completeness proofs. First, the problem must be shown to be in $\mathcal{P}$. In most cases of interest, this is obvious, linear programming being a notable exception. For a reduction from the circuit value problem, it is necessary to simulate a circuit. This entails having a way to represent the values *true* and *false*. It is also necessary to be able to combine values to simulate the gates. Often, the difficult part

of a P-completeness proof is fanning out values. To fan out a value, it must be replicated so that two other gates can receive the value. A technical detail in P-completeness proofs is to make sure that values are only propagated in the proper direction. If care is not taken, values may be propagated backwards, interfering with earlier gates. A final issue in P-completeness proofs is that the reduction must be a log-space reduction. The types of local transformations that are commonly done in NP-completeness proofs can be done in log-space. The proofs that a transformation can be done in log-space is generally omitted from a P-completeness proof.

1.3. Parallel and Sequential Algorithms

There are essentially two ways to design a parallel algorithm. One can either start with a sequential algorithm for the problem and attempt to adapt it to a parallel machine, or one can start from scratch and design a parallel algorithm. The first approach is appealing for a number of reasons. There has been a vast amount of work done on developing sequential algorithms, so it is hoped that some of it carries over to parallel computation. Some general techniques in sequential computation, such as *divide and conquer*, involve partitioning problems into independent subproblems. Some of these algorithms have natural parallel analogues. There are a few theoretical results that show that certain classes of computation can be converted to parallel algorithms. For example, it is known that programs that compute certain polynomials can be converted to fast parallel algorithms [VSBR]. There is also a practical interest in converting sequential algorithms to parallel algorithms. Over the years, many programs have been written for sequential computers. Many people want compilers that will compile the code for parallel machines, to avoid having to rewrite the code. In certain domains, such as numerical computation, this approach is likely to be at least partially successful. It is possible to identify a certain amount of parallelism in vector computations automatically.

The approach of directly converting sequential algorithms to parallel algorithms has its limitations. Some sequential algorithms process information in a way that seems to be inherently sequential. Each step may directly depend upon the previous step, so it is not possible to decompose the computation into independent sub-computations. In Chapter 2, we investigate the relationship between sequential and parallel algorithms. We introduce the notion of a P-complete algorithm. This gives us a way to identify inherently sequential algorithms. A P-complete algorithm cannot be converted to a fast parallel algorithm unless $P = NC$. We give a number of examples of simple algorithms that are P-complete.

Showing that an algorithm is inherently sequential does not show that the problem that the algorithm solves is necessarily difficult. There are a number of problems where the natural sequential algorithm is P-complete, but a different approach can be used to construct a fast parallel algorithm. In cases where an algorithm is P-complete, it is necessary to start from scratch to attempt to find a fast parallel algorithm. This shows that in some cases completely different approaches are needed for parallel algorithms than are used for sequential algorithms.

1.4. Techniques for Parallel Algorithms

The techniques that are used for parallel algorithms are quite limited. The technique that is most commonly used is referred to as *path doubling*. The essential idea in path doubling is that at each phase a processor doubles the amount of information that it has. For example, in a summation algorithm, each step doubles the number of values for which a processor has the sum. A second example is traversing a linked list. There is a processor associated with each item in the list, and the processor has a pointer to another item. Each step of the algorithm doubles the distance that is covered by each pointer. Path doubling also appears in more sophisticated guises. For example, it is used by Helmbold and Mayr in their algorithm to compute an optimal two processor schedule [HM2].

One of the promising developments in parallel algorithms is that new techniques are being discovered. One of the major new techniques is what we refer to as the *iterated improvement strategy*. Instead of seeking to solve the problem in one shot, an iterated improvement algorithm builds its solution in a number of phases. Often, each phase of an iterated improvement algorithm reduces the number of candidates for the solution by a constant fraction, so that there are only $O(\log n)$ phases. One of the first uses of this approach was by Karp and Wigderson in their maximal independent set algorithm [KW]. A maximal independent set in a graph is a maximal set of vertices with no edges between them. The algorithm maintains a set I which is eventually a maximal independent set. At each phase, the number of vertices that are not in I or adjacent to members of I is reduced significantly. A second important algorithm that uses iterated improvement is the Karp-Upfal-Wigderson *matching algorithm* [KUW]. This algorithm finds a perfect matching by identifying subsets of the edges that are contained in a perfect matching. Once edges are put into the solution set, they are not removed. This method is in sharp contrast to the sequential algorithms for matching which move edges into and out of the solution [PS].

The use of randomness has been gaining popularity in parallel algorithms. Quite often it is possible to generate certain objects with random choices, but it seems more difficult to do it deterministically. A typical situation is for random choices to be good with high probability, but to guarantee that the choices are good requires basing each choice on the other choices made. Randomness seems to reduce decisions from being global to being local. Probabilistic techniques are often used in conjunction with the iterated improvement strategy. In some cases, it is possible to get rid of the randomness by showing that a small sample space is sufficient, and then searching the sample space exhaustively [KW][Lu].

In Chapter 3 of this thesis, we look at parallel algorithms for some path problems and use some of these techniques. The path problems that we look at can be solved by simple sequential algorithms but are more difficult to solve with fast parallel algorithms. Much of our work on path problems was motivated by the problem of depth first search. A related problem is to compute a *maximal path*. A maximal path is a simple path that cannot be extended. Our major results of the chapter are that a maximal path can be found by an $\mathcal{RNC}$ algorithm, and that a depth first search tree can be constructed in time $O(n^{1/2+\epsilon})$ for an n vertex graph. We also give algorithms for some other path problems. The algorithms employ a number of the new techniques. None of them depend directly on path doubling.

1.5. Coping with P-Completeness

P-complete problems probably cannot be solved by fast parallel algorithms. However, it is still important to see what can be done with these problems using parallelism. One approach is to look for algorithms that are substantially faster than the known sequential algorithms. Algorithms which run in sublinear time, say $n^{1/2}$ can be of practical importance. This is in contrast to the analogous situation for sequential algorithms, where super polynomial algorithms are rarely practical. A second approach is to look for an approximate solution to the problem. In some cases, a solution that is close to the desired solution can be found by a fast parallel algorithm.

There has been a substantial amount of work done on the approximation of NP-complete problems. For some problems, there are polynomial time algorithms which find solutions that are close to the optimal solution. There are also results which give lower bounds on the degree of approximation that is possible assuming that $\mathcal{P} \neq \mathcal{NP}$. In Chapter 4 we study the parallel approximation of P-complete problems. Our results are similar to the results on approximating NP-complete problems. One problem that we look at is finding a subgraph of a graph that has all vertices with high degree. For a variant of the problem we establish bounds on the degree of approximation that is possible. The similarity between sequential and parallel approximation is particularly strong for number problems. Some NP-complete number problems are tractable if the numbers involved are small. This has motivated the distinction between strong and weak NP-completeness [GJ2]. We make the same distinction for P-complete problems and give an example of a strongly P-complete problem.

1.6. Notation and Conventions

Many of the problems that we look at in this thesis are simple graph theory problems. We use is fairly standard notation. We generally denote a graph by $G = (V, E)$ where V is the set of vertices and E is the set of edges. Where we neglect to state it explicitly, the number of vertices is n. We make frequent use of the notation that identifies a set of vertices with the induced subgraph on the vertices. For $V' \subseteq V$, the induced subgraph is the graph $G' = (V', (V' \times V') \cap E)$.

In this thesis, we describe algorithms at a moderately high level. In most cases our interest is to show that a problem can be solved in $\mathcal{NC}$, as opposed to giving the best possible parallel algorithm for the problem. We are not that worried about the exact power of the logarithm in an algorithm's running times or the degree of the polynomial for the number of processors used. In cases where we do claim explicit time and processor bounds, they are with respect to a CREW P-RAM implementation.

In our algorithms, we take advantage of many known parallel algorithms for graph problems. A very important algorithm that we use a number of times is the Karp-Upfal-Wigderson matching algorithm. The algorithm is an $\mathcal{RNC}$ algorithm that finds a maximum cardinality matching in a graph. The matching algorithm can be used to solve a number of other important problems. For example, it can be used to find a maximum flow in a unit capacity network. We also use quite a few subroutines to solve simple graph theory

problems. We use algorithms for such problems as finding connected components [V], finding articulation points [TV], and finding a shortest path between two vertices. We also rely on parallel algorithms for maintaining data structures and manipulating graphs. We do not go into the details of these operations. The parallel algorithms for these problems are not very complicated, especially when we are only concerned with getting $\mathcal{NC}$ algorithms, as opposed to getting the best algorithms possible.

We describe our algorithms in a PASCAL-like language. Many of the statements are English language descriptions. It would not be difficult to convert these to give a more detailed implementation of the algorithms. Some of our algorithms appear rather sequential, their parallelism arises from the parallel implementation of the individual statements. We do use a few explicit parallel control structures in our algorithm descriptions. We use a statement which has the form: **for each** x **do in parallel**. This has the natural meaning of running independent copies of the routine for each x and then combining the results when the routines are all done.

Chapter 2 P-Complete Algorithms

2.1. Introduction

One of the interesting and challenging aspects of parallel computation is that different techniques need to be used for parallel algorithms from those that are used for sequential algorithms. In sequential algorithms, processing is done one step at a time. This allows solutions to be constructed in many phases, with choices depending upon all of the earlier choices that were made. However, to get fast parallel algorithms, many choices need to be made simultaneously. The computations need to be localized, with only a small dependence between the various components of the computation. In this chapter, some simple sequential algorithms are examined. Strong evidence is presented that the techniques used in these algorithms are often inherently sequential, so there is little hope that they can be sped up substantially with parallelism.

Some sequential algorithms have fairly direct parallel counterparts. The parallel algorithm can be thought of as a parallel implementation of the sequential algorithm. A trivial example is matrix multiplication. The straightforward sequential algorithm computes the entries of the product one at a time, while the parallel algorithm computes the entries simultaneously. However, other sequential algorithms take advantage of being able to base decisions on accumulated information. For example, the sequential algorithms for matching start with an initial solution and improve the solution through a number of phases of augmentation. The known parallel algorithms [KUW] for matching take a completely different approach.

In this chapter we formalize what it means for an algorithm to be inherently sequential by relating computations to certain P-complete problems. The term "P-complete algorithm" is introduced to describe these algorithms. This provides strong evidence that some algorithms are inherently sequential. When we show that a certain approach to a problem probably cannot yield a fast parallel solution, this does not imply that the problem is difficult. There are a number of examples of problems where the natural sequential algorithm is P-complete, but a different approach can be used to construct a fast parallel algorithm.

The sequential algorithms that we examine are greedy algorithms. The greedy paradigm is a very important general technique used in many sequential algorithms. Examples of greedy algorithms include Kruskal's minimum spanning tree algorithm [K] and the well known algorithm for depth first search [T]. A greedy algorithm is one which builds its solution one step at a time. Items are added to the partial solution by picking the "best" choice by some generally simple criterion. Once an item is added to the solution, it will not be discarded, thus there is no backtracking. Greedy algorithms often seem very sequential in nature, since the choice of which item to add to the solution set frequently depends on many of the previous choices. In this chapter we show that the greedy algorithms for several simple problems are P-complete. We show that the greedy algorithms for finding a maximal path, for finding a set of disjoint paths, and for approximating a maximum cut are all P-complete.

2.2. Definition of P-completeness for Algorithms

The purpose of extending the notion of P-completeness to algorithms is to be able to capture the idea that an algorithm is (probably) inherently sequential. The definition of P-completeness for algorithms that we give must in some way capture what it means to implement a sequential algorithm as a parallel algorithm. In other words, the definition must establish some kind of correspondence between sequential and parallel algorithms. To justify the term "P-completeness," our definition should give as much evidence for the algorithm being inherently sequential as there is that a P-complete problem cannot be solved by a fast parallel algorithm. To do this, our definition should imply that if a P-complete algorithm could be implemented as an $\mathcal{NC}$ algorithm then $\mathcal{P} = \mathcal{NC}$. There are a number of different ways that P-completeness can be defined for algorithms. The basic idea in the definitions is that the problem of performing the same computation as is done by the sequential algorithm is a P-complete problem.

One way to define an algorithm to be P-complete is in terms of the full computation of a Turing machine. The computation of a polynomial time Turing machine on a particular input can be summarized by a string of polynomial length. For a Turing machine M, this string can be viewed as a function $f_M(x)$ of the input x. A possible definition of P-completeness of an algorithm A with Turing machine M is: A is P-complete if the problem of computing $f_M(x)$ is P-complete. The drawback to this approach is that it is heavily dependent on the actual Turing machine corresponding to an algorithm. It is rarely desirable to have to describe algorithms in terms of Turing machine implementations. The advantage of this approach is that it fully captures the computation of the algorithm.

The definition of a P-complete algorithm that we use is based on functions that solve *search* problems. A search problem Π consists of a set of instances D_Π and set of solutions $S[I]$ for each $I \in D_\Pi$ [GJ3 pp. 110]. The problem can be viewed as a relation $R = \{(x, y) \mid x \in D_\Pi, y \in S[x]\}$. An algorithm for a search problem is a function f such that $(x, f(x)) \in R$. A simple example of a search problem is the spanning tree problem. The solutions for a graph G form the set of all spanning trees of G. An algorithm that solves the spanning tree problem is one that finds some spanning tree. For a search problem, there are many possible algorithms that solve the problem. Our definition of a P-complete algorithm is:

Definition 2.1. *An algorithm A for a search problem is P-complete if the problem of computing the solution found by A is P-complete.*

A shortcoming of our definition of a P-complete algorithm is that it does not immediately relate to the internal computation of the algorithm. For some algorithms it is the method used to get the answer that appears sequential in nature. The way to handle this with our current definition is to redefine the result of the algorithm so that it includes a *trace* of the computation. This means that we include with the result a list of certain internal states of the computation. For example, if an algorithm computes the partial solutions $S_1, \ldots, S_n$ on its way to the solution S, then we could define the result as $S_1, \ldots, S_n, S$. This allows us to handle most cases of interest with our definition of P-completeness for algorithms without having to deal with the details of Turing machine implementation.

In order to prove that an algorithm is P-complete, it is necessary to have a fairly precise statement of the algorithm. When describing an algorithm, it is common practice to leave certain steps unspecified with statements like "pick an unmarked vertex." This is done when several choices are satisfactory and there is no need to encumber the description with superfluous detail. When implementing the algorithm, some arbitrary choices need to be made. Often a reasonable choice to make is to choose the lowest numbered element. When this choice is made, frequently the resulting solution is the lexicographically minimum solution. The natural lexicographic order on strings is $s_1 \cdots s_j <_{lex} t_1 \cdots t_k$ if $s_1 \cdots s_j = t_1 \cdots t_j$ and $j < k$ or $s_1 \cdots s_i = t_1 \cdots t_i$ and $s_{i+1} < t_{t+1}$. We shall use the word *lexmin* in place of the cumbersome phrase "lexicographically minimum." For graph problems with edges represented by adjacency lists, another reasonable approach for unspecified choices of edges is to take the first available edge from a list. The result of this approach is generally equivalent to choosing the lowest numbered adjacent vertex when the edges are ordered in the list by vertex numbers.

An example of a P-complete algorithm is the greedy algorithm for finding a maximal independent set. For a graph $G = (V, E)$ an independent set I is a subset of the vertices such that there are no edges between vertices in I. A maximal[1] independent set is an independent set that is not properly contained in any other independent set. The sequential algorithm constructs a maximal independent set by considering the items one at a time. If an item is not adjacent to the independent set when it is considered, then it is added to the independent set. The algorithm is:

```
MaximalIndependentSet(G)
  begin
    I ← ∅;
    for i ← 1 to |V| do
      if v_i ∉ N(I) then
        I ← I ∪ {v_i};
  end.
```

One approach to designing a fast parallel algorithm for finding a maximal independent set is to attempt to implement the sequential algorithm as a parallel algorithm. The algorithm would have to decide whether or not to include an element v_i in I without building I step by step. However, this approach is not likely to be successful. The solution that is found by this algorithm is the lexmin solution. The problem of computing the lexmin maximal independent set is P-complete, so the algorithm is a P-complete algorithm. This result is due to Cook, who showed that the complementary problem of computing the lexmin maximal clique is P-complete [C2]. Although this sequential algorithm apparently cannot be used to create a fast parallel algorithm, a different approach can be used to construct a fast parallel algorithm. Wigderson and Karp [KW] developed a probabilistic parallel algorithm for constructing a maximal independent set. Their algorithm can be converted into a deterministic algorithm, so the problem can be solved in $\mathcal{NC}$. A simpler maximal independent set algorithm has been found by Luby [Lu].

---

[1] In this thesis, we use *maximal* to denote something that cannot be extended, and we use *maximum* to indicate maximum cardinality.

2.3. Finding a Maximal Path

The first algorithm that we show to be P-complete is a simple algorithm for finding a path in a graph. The algorithm builds a path one vertex at a time by going from the current endpoint to its lowest numbered neighbor that is not already on the path. The algorithm runs until a vertex is encountered that has all of its neighbors on the path so that the path can not be extended. A path that cannot be extended is a *maximal path.* The greedy algorithm for finding a maximal path starting at a given vertex r is:

```
GreedyMaximalPath(G, r)
  begin
    P ← r; v ← r;
    while v has an unvisited neighbor do
      begin
        w ← lowest numbered unvisited neighbor of v;
        P ← Pw;
        v ← w;
      end
  end.
```

The greedy algorithm computes the lexmin maximal path. We show that computing the lexmin maximal path is a P-complete problem, so that this algorithm is P-complete. However, this does not mean that a maximal path cannot be found by a fast parallel algorithm. The next chapter gives an $\mathcal{RNC}$ algorithm for finding a maximal path. The greedy algorithm for finding a maximal path is closely related to the algorithm for finding a depth first search tree of a graph. The greedy maximal path algorithm finds the initial branch of the lexmin depth first search tree, so our results imply that the greedy algorithm for depth first search is P-complete. The original proof that computing the lexmin depth first search tree is P-complete is due to Reif [Re].

2.3.1. Directed Lexmin Maximal Path

We show that the problem of computing the maximal path found by the greedy algorithm is P-complete. We first show the result for directed graphs and then for undirected planar graphs. The proof for directed graphs is simpler and conveys the intuition of why the problem is difficult better than the proof for planar graphs. The second result is stronger since it applies to a very restrictive class of graphs. Finding the lexmin maximal path in an undirected graph is a special case of the problem for directed graphs, since an undirected edge can be viewed as a pair of directed edges.

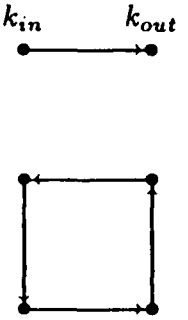
Theorem 2.1. *The problem of computing the lexmin maximal path is P-complete for directed graphs.*

Proof: The proof is a reduction from the monotone circuit value problem. Let $\beta = \beta_1, \ldots, \beta_n$ be an instance of the monotone circuit value problem. The circuit β is transformed in log-space to a graph with distinguished vertices r and v such that v will be on the lexmin path from r if and only if the circuit evaluates to *true.*

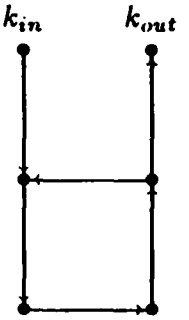
For each gate β_k there is a collection of vertices. A gate is simulated by the way that the lexmin path passes through the vertices corresponding to that gate. The gates are evaluated in order, with the path first passing through the vertices for β_1, then β_2, and so on. There are vertices k_{in} and k_{out} in the collection of vertices for β_k. The segment of the lexmin path between the vertices k_{in} and k_{out} visits certain vertices to test the values of the inputs to β_k and then visits other vertices to indicate the value of the output of the gate.

A key component of the simulation is a *switch* which is used to indicate the value of a wire. For each gate there is one switch for each output. The vertices of these switches are traversed during the simulation of the gate to indicate a *true* value, and they are bypassed to indicate a *false* value. If a switch for β_k is not visited when simulating β_k, it might be traversed when simulating a gate β_j which receives an input from β_k.

The gadgets for the gates are shown in the figures below. In the figures, the switches are the groups of four vertices drawn in a square. The gadget for gate β_k is connected to the gadget for β_{k+1} by an edge from k_{out} to $(k+1)_{in}$. If a gate β_k receives an input from gate β_i, then the gadget for β_k is connected to the output switch of β_i. The AND and OR gates are illustrated as 2-output gates β_k that receive inputs from β_i and β_j. The vertices of the graph that is constructed are labelled so that the labels of the vertices associated with β_i are less than those associated with β_k for $i < k$. In addition, within gate β_k, the labels are as indicated in the figures, where $k < k_r < k_{r+1}$.

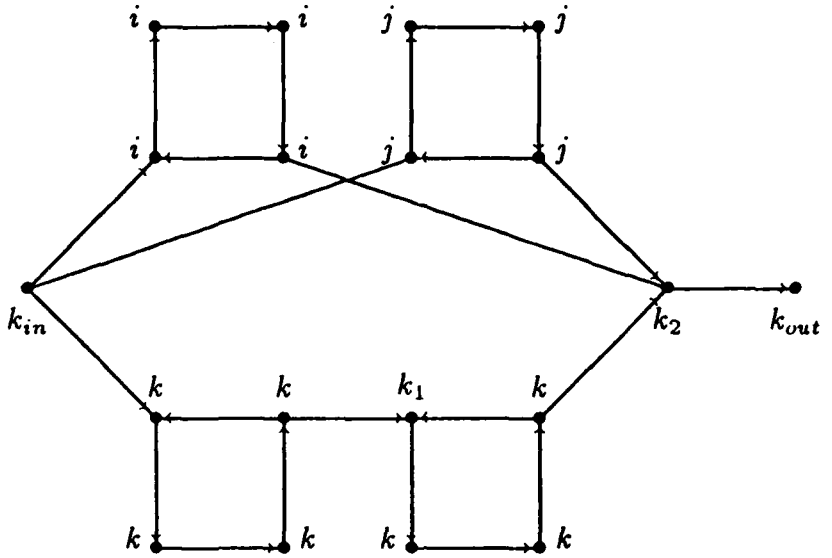


0-INPUT

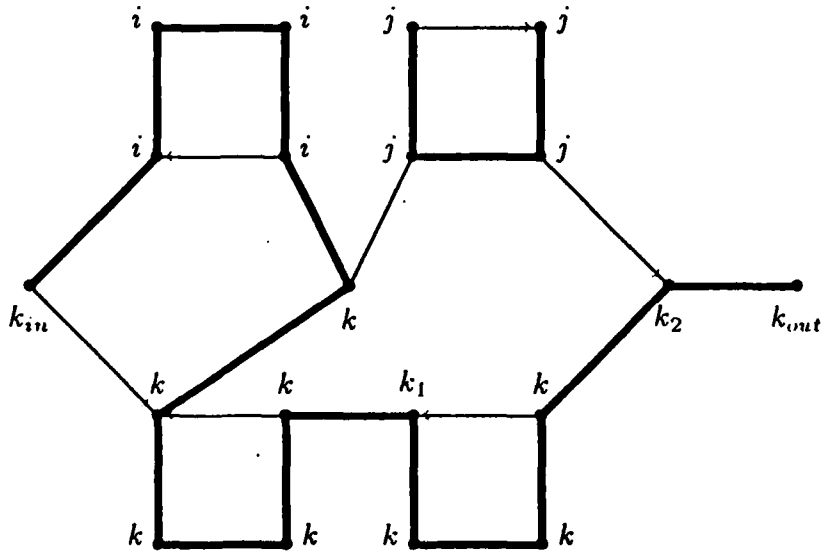


1-INPUT

The circuit is simulated by constructing the lexmin path starting at the vertex 1_{in}. From a vertex v the path goes to the lowest numbered neighbor of v that is not already on the path. For a 0-INPUT, the path goes directly k_{in} to k_{out}, and for a 1-INPUT the path traverses the switch on its route from k_{in} to k_{out}. For an AND gate if either of the input switches has not been traversed when the path gets to k_{in} (so the gate is receiving a *false* input), then the path goes through an input switch to k_{out}, bypassing the output switches. For an OR gate, if either of the switches has been visited then the path will go through the output switches. In the illustration of an OR gate, the highlighted path shows what happens when the gate receives a *false* input from β_i and a *true* input from β_j. If the lexmin path visits a vertex in the output switch of the final gate β_n, then the circuit evaluates to *true*, and if the path does not visit the output switch, the circuit evaluates to *false*. ∎



AND gate



OR gate

2.3.2. Planar Lexmin Path

The P-completeness result can be strengthened by showing that it applies to a very restricted class of graphs. We show that computing the lexmin path is P-complete even for undirected planar graphs with maximum degree three. Many graph problems appear to be much easier when they are restricted to planar graphs. For example, both depth first search [Sm] and network flow [JV] can be solved in $\mathcal{NC}$ for planar graphs. The P-completeness proof that we give is similar to the previous proof. We reduce from a variant of the circuit value problem and simulate the gates in the same manner as above. The circuit value problem that we use is for a layered planar circuit made up of NOT and OR gates. The gates are on levels, with the gate β_{k+1} immediately to the right of β_k unless β_k

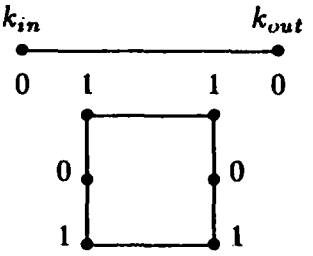
is the right most gate on a level, in which case β_{k+1} is the leftmost gate on the next level. The wires between gates run only between consecutive levels and the wires do not cross.

Theorem 2.2. *The problem of computing the lexmin maximal path in undirected planar graphs with maximum degree three is P-complete.*

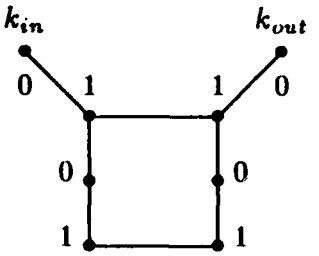
Proof: We prove this theorem by giving a log-space transformation from the circuit value problem for layered planar OR-NOT circuits. The planar circuit value problem remains P-complete with this restriction. The reduction of an arbitrary circuit to a planar circuit [G1] can be modified so that this type of planar circuit is genterated.

The construction is the same as used in Theorem 2.1 except that we use the gadgets below. The gadgets for the gates can be put in levels and the connections to the switches can be done without edge crossings. The only violation of planarity is the edge between k_{out} and $(k+1)_{in}$ when β_k and β_{k+1} are on separate levels. This problem can be solved by laying the graph out on a cylinder instead of the plane. The cylinder can then be projected onto the plane to achieve a planar layout. The vertex labels shown in the figures are just 0, 1, 2, 3, and 4. The vertex labels can be made unique by replacing each label k ($k \in \{0,\ldots,4\}$), by a label in $\{kn,\ldots,(k+1)n\}$.

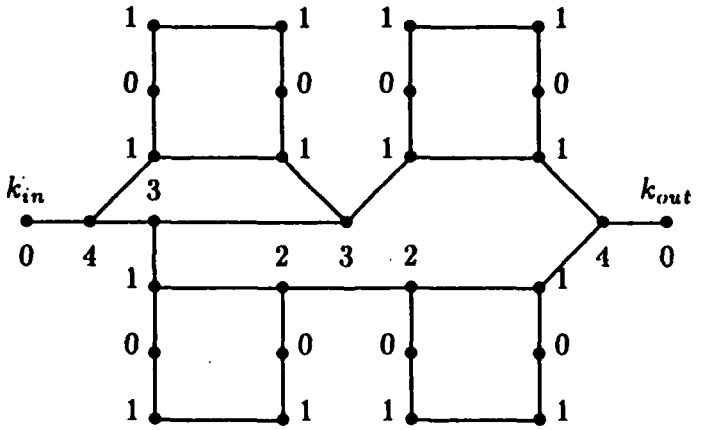
The circuit is again simulated by computing the lexmin path from the vertex 1_{in}. The path will visit the output switch of the final gadget if and only if the output of the circuit is *true*. ∎



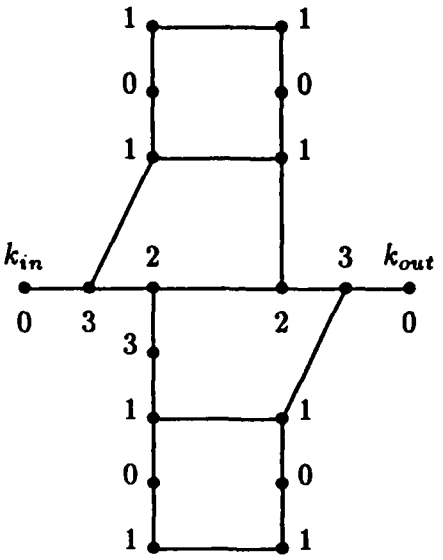
0-INPUT



1-INPUT



OR gate



NOT gate

Since the lexmin maximal path is the initial branch of the lexmin depth first search tree, we have the following corollary. This is an improvement of the result of Reif [Re].

Corollary 2.1. *Computing the lexmin depth first search tree is P-complete for planar graphs.*

2.4. Finding a Maximal Set of Disjoint Paths

A second path problem that can be solved by a simple greedy algorithm is to find a maximal set of disjoint paths. The problem is:

Given a graph $G = (V, E)$ and a subset U of V, find a maximal set of vertex disjoint paths joining the vertices of U.

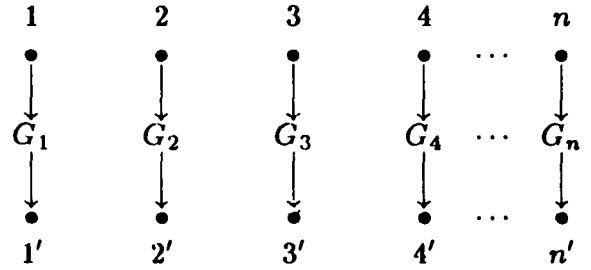
The set of paths is required to be maximal in the sense that no more paths joining vertices of U can be added to it. A greedy algorithm solves this problem by finding paths one at a time until no more paths can be found. In this section we examine the problem when it is restricted to a layered directed acyclic graph (dag). A layered graph has all of its vertices in levels with edges only between consecutive levels. The motivation for looking at this restricted case is that it occurs as a subroutine in a number of network flow and matching algorithms [HK].

We show that the greedy algorithm for finding a maximal set of vertex disjoint paths in a layered dag is P-complete. The greedy algorithm repeatedly finds paths from the first level to the last level and removes them. This process is repeated until the first level is separated from the last level. When a path is removed, some vertices might be separated from the last level, these vertices are also removed. When the algorithm constructs a path, it finds the lexmin path between the first and last level in the current graph. The complexity of the problem does not arise from finding lexmin paths, since they can be found easily in a dag. The complexity arises from the dependence of a path on the previous choices of paths.

Theorem 2.3. *The greedy algorithm for computing a maximal set of vertex disjoint paths in a layered dag is P-complete.*

Proof: The proof is a reduction from the circuit value problem with several minor restrictions. First, the circuit is restricted to be made up of only inputs, NOT gates, and AND gates. The fanout of all gates is assumed to be exactly two, although one of the outputs of a gate need not be connected to anything. The gates are numbered topologically so that a gate gets its inputs from lower numbered gates. The outputs of gate β_i are denoted i_1 and i_2, with i_1 going to the *higher* numbered gate receiving an input from β_i and i_2 going to the *lower* numbered gate. If only one gate gets an input from β_i, then that gate receives i_2. It is finally assumed that the AND gates get their inputs from distinct gates, so $\beta_k = \text{AND}(i_1, i_2)$ is not allowed. The circuit value problem clearly remains P-complete with these restrictions.

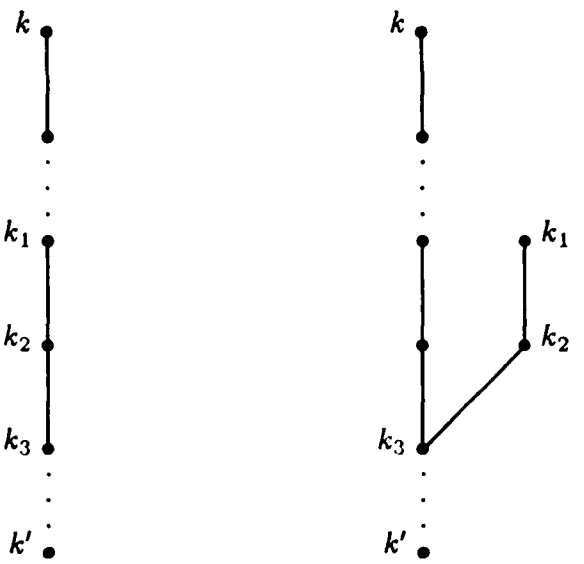
Let $\beta = \beta_1, \ldots, \beta_n$ be a circuit satisfying the above conditions. We construct a layered dag G such that the maximal set of disjoint paths found by the greedy algorithm with input G corresponds to the evaluation of the circuit β. The basic structure of G is:



G_k is the gadget which is associated with the gate β_k. The circuit is simulated by computing a sequence of paths $P_1, \ldots, P_n$. The path P_k is the lexmin path in the graph G after the paths $P_1, \ldots, P_{k-1}$ have been removed. The gate β_k is simulated by the path P_k. The path P_k goes from k to k' and lies entirely in G_k except when it tests the inputs to β_k. There are two distinguished vertices k_1 and k_2 in G_k that are visited by P_k if β_k is *true* and are not visited if β_k is *false*.

The gadgets for the inputs and the gates are illustrated in the figures below. The graph has $3n+2$ levels. The output vertices for G_k are numbered k_1 and k_2. The vertices k_1, k_2 and k_3 are on levels $3k-1$, $3k$, and $3k+1$ respectively. In the figures, all edges are directed downwards. For a 1-INPUT and a 0-INPUT, the path P_k goes directly from k to k'. The vertices k_1 and k_2 are visited for a 1-INPUT and are not visited for a 0-INPUT.

In the figure for the NOT gate, $\beta_k = \text{NOT}(i_p)$, $p \in \{1,2\}$, the vertex l is numbered higher than the vertex i_p. If the vertex i_p has not been visited when P_k is constructed, the path P_k goes through i_p to k_1 and k_2, otherwise the path goes through l and bypasses k_1 and k_2. Since i_p not being visited corresponds to β_k receiving a *false* value, the gadget simulates a NOT gate. In the AND gate $\beta_k = \text{AND}(i_p, j_q)$, $p, q \in \{1,2\}$ the path P_k goes through k_1 and k_2 if both i_p and j_q have been visited by earlier paths. If i_p or j_q has not been visited when P_k is constructed, then the path does not go through the vertices k_1 and k_2. Note that if both i_p and j_q have not been visited, the path P_k goes through both i_p and j_q.



1 - INPUT 0 - INPUT

In the NOT gate shown below it is essential that the vertex i_{p+1} is visited by some path before the path P_k is constructed. If this were not the case, then the path P_k would go out through i_{p+1} into the vertices of a different gadget, not returning to k_1 and k_2. This also applies to the vertices i_{p+1} and j_{q+1} in the AND gate. For any gate, the vertex k_3 is on the path P_k, so there is no problem for $p = 2$. If $p = 1$, then the gate β_k is the higher numbered gate to receive an input from β_i. Suppose β_j also gets an input from β_i. The path P_j contains the vertex i_2 if i_2 is not on P_i. Hence the vertex i_{p+1} is already on a path when P_k is constructed. This completes the proof that the circuit is successfully simulated by the set of disjoint paths found in G by the greedy algorithm. ∎

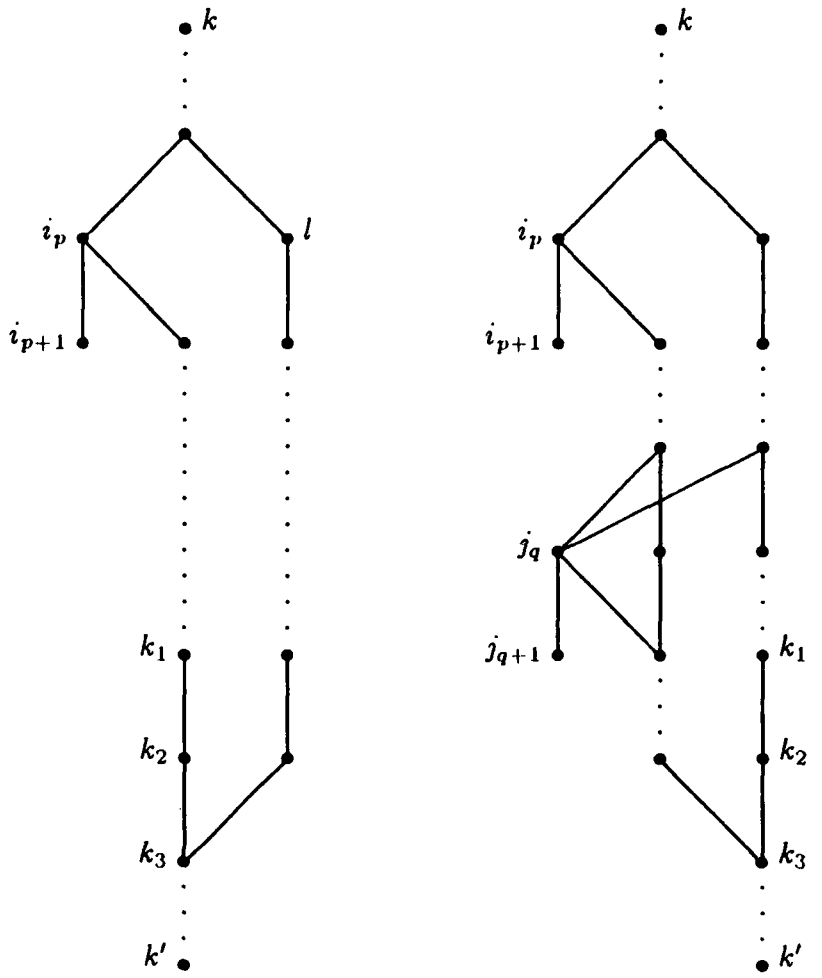
2.5. Neighborhood Heuristics for NP-Complete Problems

Greedy algorithms are often used to approximate NP-complete problems. The basic approach is to take some starting solution and attempt to improve it by local changes. The improvements are repeated until a local maximum is reached. Although these heuristics are difficult to analyze, they have been found effective in practice.

The greedy algorithms for many neighborhood search schemes appear to be inherently sequential. It seems to be difficult to perform more than a few steps of the search at a time. As an example of this we show that an approximation algorithm for the maxcut problem is P-complete. The maxcut problem is:

> Given a graph $G = (V, E)$, find a partition $\{V_1, V_2\}$ of V such that the number of edges between V_1 and V_2 is maximized.

The heuristic that is used is to move vertices between the two sets as long as moves increase the number of edges between the two sets. First some initial partition is chosen. Then a vertex is found that has more neighbors in its own set than in the other and it is moved to the other set. This step is repeated until each vertex has more neighbors in the opposite



NOT gate AND gate

set than in its own set. Since the number of edges between V_1 and V_2 is increased each move, there are at most $|E|$ phases.

An alternative way to view this scheme is as a coloring problem. The goal is to color the vertices with two colors such that each vertex has more neighbors of the opposite color than of its own color. Such a coloring is said to be *stable*. A stable coloring can be computed by switching the colors of vertices one at a time until the coloring is stable. A *valid swap* occurs when a vertex with more neighbors of its own color than of the opposite has its color changed. We show that given a graph with an initial assignment of colors, it is P-complete to compute a sequence of valid swaps that reaches stability. This shows that the greedy algorithm for approximating maxcut is P-complete. For this problem, we require that the output include the intermediate computations that are made. We do this so that all of the color swaps are valid. Our result does not imply that it is P-complete to compute a stable coloring. It is an open problem whether it is possible to compute a stable coloring with a fast parallel algorithm. An $\mathcal{NC}$ algorithm is known for computing a stable coloring of a graph with maximum degree three [KSS].

Theorem 2.4. *Given a graph $G = (V, E)$ and an initial assignment of colors to the*

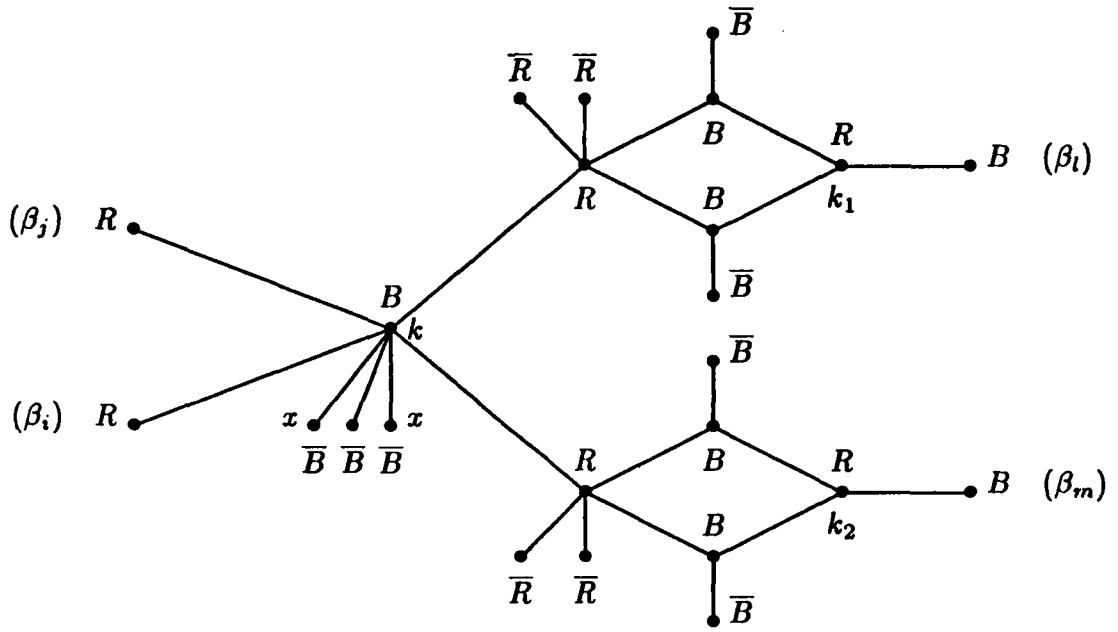
vertices, it is P-complete to compute a sequence of valid swaps that reaches a stable coloring.

Proof: The proof is a reduction from the monotone circuit value problem. Let $\beta = \beta_1, \ldots, \beta_n$ be a monotone circuit. A graph G is constructed such that a sequence of valid swaps that reaches stability corresponds to the evaluation of the circuit. There is a subgraph for each gate with an initial assignment of the colors R/B. The OR gate is shown below. The gates are connected together in a manner that corresponds to the connections of the circuit. The subgraph for β_k has three distinguished vertices, the vertex k is associated with the gate's inputs and the vertices k_1 and k_2 are associated with the gate's outputs. If β_k gives outputs to β_l and β_m $(l < m)$, then there are edges (k_1, l) and (k_2, m). In the illustration of the OR gate, the vertices labelled with $\overline{B}$ have color B and are connected to two vertices with color R (not shown in the figure). These vertices will never have more B neighbors than R neighbors, so they are colored B throughout the simulation. Similarly the vertices labelled $\overline{R}$ have color R and are connected to two vertices with color B. The AND gate has the same structure as the OR gate except that the two vertices labelled x are not present. A 0-INPUT β_k is a vertex k with color R connected to two vertices with color B and a 1-INPUT β_k is a vertex k with color B connected to two vertices of color R. If the input β_k goes to the gate β_l, then there is an edge (k, l).

The circuit is evaluated by finding a sequence of valid swaps that reaches a stable coloring. The graph G has the property that the coloring achieved by any maximal sequence of valid swaps is unique. The graph also has the property that any vertex can have its color changed at most one time. The initial coloring of the gate β_k is stable, except possibly for the vertex k. If the vertex k is recolored R then other vertices become unstable and eventually the vertices k_1 and k_2 are recolored B. If the vertex k is recolored R, then the gate β_k evaluates to *true*. When the vertex k is colored R, the swaps propagate so that the vertices k_1 and k_2 have their colors changed to B. For the OR gate β_k, if one of the vertices corresponding to its inputs is recolored, then the vertex k is recolored. Similarly, for the AND gate β_k, if both of its inputs are recolored, then the vertex k is recolored. A maximal sequence of swaps simulates the evaluation of the gates in roughly topological order. A gate is not set to *true* (i.e. the colors switched in the associated subgraph), until enough of its inputs are known to be *true* to make the gate *true*. The simulation proceeds until the subgraphs that correspond to all gates that evaluate to *true* have had their colors switched. ∎

2.6. Additional P-Complete Algorithms

Many other sequential algorithms can be shown to be P-complete. Here are a few additional P-complete algorithms. All of our P-completeness proofs for these algorithms are reductions from variants of the circuit value problem. The P-completeness proofs for the high degree subgraph problem and first fit bin packing are given in Chapter 4. The other proofs are left to the interested reader.



$\beta_k = \mathrm{OR}(\beta_i, \beta_j)$, Outputs to β_l and β_m

High Degree Subgraph Problem

PROBLEM: Given a graph $G = (V, E)$ and an integer k, construct the maximum induced subgraph that has all vertices of degree at least k.

SEQUENTIAL ALGORITHM: The sequential algorithm for this problem discards vertices of degree less than k one at a time until all remaining vertices have degree at least k.

Comment: The problem of determining if a graph has a nonempty induced subgraph with minimum degree k is also P-complete. Various methods of finding approximate solutions to the high degree subgraph problem are discussed in Chapter 4.

First Fit Bin Packing

PROBLEM: Given a finite set U of items with sizes $s(u) \in Z^+$ for each $u \in U$ and a bin capacity B, construct a first fit packing of the items into the bins.

SEQUENTIAL ALGORITHM: The sequential algorithm considers the items in the order $u_1, \ldots, u_n$ and places each item in the first bin with enough room left for the item.

Comment: This problem remains P-complete if the items are in decreasing order, but can be solved in $\mathcal{NC}$ if the items are in increasing order. The problem is discussed further in Chapter 4.

Alternating Breadth-first Search

PROBLEM: Given a graph $G = (V, E)$ with the edges partitioned into two sets M and U, and a distinguished vertex $r \in V$ construct an alternating breadth first search from r. An alternating breadth first search is a partition of the vertices into levels, with edges in U going from even levels to odd levels, and edges from M going from odd levels to even levels. A vertex v is on level i for i even (i odd) if it is not on any level less than i and there is a vertex w on level $i-1$ with $(v, w) \in M$ ($(v, w) \in U$). The vertex r is on level 0.

SEQUENTIAL ALGORITHM: The sequential algorithm assigns the vertices to levels, one level at a time.

Comment: This labelling procedure arises in some sequential matching algorithms when looking for *augmenting* paths [MV]. A variant of the algorithm which collapses *blossoms* when they are encountered is also P-complete.

Travelling Salesman 2-Opting
PROBLEM: Given a graph $G = (V, E)$ with edge weights, $w(e) \in Z^+$ for each $e \in E$ and an initial tour T_0, find a sequence of tours $T_0, \ldots, T_m$ such that T_i is the result of a *2-opt* [KL] of T_{i-1}, the cost of T_i is less than the cost of T_{i-1} and either T_m is a locally optimal tour or $m \geq |V|$. A 2-opt refers to a neighborhood transformation done on tours of the graph.
SEQUENTIAL ALGORITHM: Transformations are applied one at a time until a local optimum is reached.
Comment: It is necessary to put a bound on the number transformations, since examples are known where an exponential number of transformations may be made before a local optimum is reached [Lue].

2.7. Discussion

In this chapter, we have shown that a number greedy algorithms seem to be inherently sequential. However, there is one important greedy algorithm which has a parallel implementation, namely Kruskal's minimum spanning tree algorithm. The algorithm constructs a minimum spanning tree by considering the edges in order of their weights. It maintains as its solution set a collection of trees. Any edge which joins separate trees is added to the solution. This algorithm can be converted to a parallel algorithm by considering each edge independently. An edge is in the minimum spanning tree if and only if it joins distinct connected components of the edges that come before it. If the edges are ordered by their edge numbers, then this algorithm finds the lexmin spanning tree.

The reason that the greedy algorithm for the minimum spanning tree problem can be implemented as a fast parallel algorithm is that it is easy to know if an edge is in the solution set without knowing exactly what the solution set is when the edge is considered. The minimum spanning tree problem is a special case of the maximum independent set problem for weighted matroids. As long as the matroid has a rank function which can be computed by a fast parallel algorithm, then the associated greedy algorithm can be parallelized.

There are a number of interesting open problems concerning greedy algorithms. One of the most important is the status of the greedy algorithm for computing a maximal matching. This algorithm computes the lexmin maximal matching. Although this problem bears a close resemblance to the lexmin maximal independent set problem, it is not known to be P-complete. A P-completeness proof of lexmin maximal matching would be significant since it would imply that weighted matching is P-complete.

In this chapter we introduced the notion of a P-complete algorithm. This notion provides a means to identify techniques that probably will not work for a particular problem, and to direct the search for algorithms in more promising directions. The specific results of this chapter show that for quite a few problems, a greedy approach is not likely to

yield a fast parallel algorithm. However, since many of these problems can be solved by parallel algorithms using other methods, this shows that different approaches are needed for parallel computation from those that are used for sequential computation.

Chapter 3 Path Problems

3.1. Introduction

In this chapter we present parallel algorithms for a number of simple combinatorial problems. The problems that are examined all deal with finding certain paths in graphs. The most important results of this chapter are that it is possible to find a maximal path with an $\mathcal{RNC}$ algorithm and that a depth first search tree can be constructed in $O(n^{1/2+\epsilon})$ time. Some of the results in this chapter are complementary to the results of the previous chapter. The greedy algorithms for several problems studied in this chapter are P-complete, but these problems can be solved by fast parallel algorithms when different approaches are taken.

The first algorithm that we give is a simple probabilistic algorithm for finding a long path in a dense graph. Given a graph with all vertices of degree at least m, a path of length $m - o(m)$ is constructed with an $\mathcal{RNC}$ algorithm. The second problem we look at is finding a maximal set of disjoint paths. The problem is given a graph $G = (V, E)$ and a subset U of the vertices, find a maximal set of vertex disjoint paths with their endpoints in the set U. We show that this problem can be solved in $\mathcal{NC}$ for graphs with bounded degree. We then show that a *maximum* set of disjoint paths can be found in $\mathcal{RNC}$ using the Karp-Upfal-Wigderson matching algorithm. The major result of this chapter is that the maximal path problem can be solved in $\mathcal{RNC}$. The maximal path problem is:

> Given a graph $G = (V, E)$ and a vertex r, find a simple path starting from r that cannot be extended without encountering a vertex that is already on the path.

We also show that the restricted case of the maximal path problem for bounded degree graphs can be solved in $\mathcal{NC}$. Our final result is that a depth first search tree of an n vertex graph can be constructed in parallel time $O(n^{1/2+\epsilon})$.

There are a number of reasons to look for parallel algorithms for problems such as these. A major reason is to gain an understanding of the types of problems that are in $\mathcal{NC}$ and $\mathcal{RNC}$. Although these problems are fairly simple in nature, it is by no means obvious that they can be solved by fast parallel algorithms. These problems are not closely related to other problems known to be in $\mathcal{NC}$ or $\mathcal{RNC}$, so the positive results increase the variety of problems that can be solved by fast parallel algorithms. A second reason to look at particular problems is to identify techniques to use in parallel algorithms. There are relatively few general techniques used in parallel algorithms, so it is hoped that by looking at new problems, additional techniques can be discovered and added to the repertoire.

Some of the problems discussed in this chapter are important problems in their own right. In particular, depth first search is one of the major open problems in parallel computation. The algorithm in this chapter is the first sublinear algorithm for depth first search. Much of the work in this chapter was motivated by the depth first search problem. The initial reason for studying the maximal path problem is its close relationship to depth first search.

3.2. Finding a Long Path in a Graph

The first problem that we look at is the problem of finding a long path in a dense graph. Let $G = (V, E)$ be an n-vertex graph with all vertices of degree at least m. The graph clearly has a path of length at least m. This problem can be solved sequentially by the greedy algorithm discussed in the previous chapter, since any maximal path in G has length at least m. However, this problem is a little trickier to solve with a parallel algorithm. One plausible approach is to generate a random walk in the graph and take as the path the segment of the walk up until the walk's first intersection with itself. This does not work since if the graph is a complete graph, then the expected length of the path constructed by this method is just $O(\sqrt{n})$.

The algorithm that we give for this problem is a probabilistic algorithm. The basic idea is to construct a subgraph in which a long path can be found easily. The subgraph is constructed by making random choices of the edges. The randomization is done in a way where the choices of edges are not fully independent, so that the resulting graph has a long path with high probability. The algorithm finds a path of length at least $\frac{m}{c \log n}$ where c is a small constant. The algorithm runs in expected time $O(\log n)$ using $O(n^2)$ processors. The algorithm can be run several times to construct a path of length $m - o(m)$ in $\mathcal{RNC}$. As long as the path has length less than $(1 - \frac{1}{\log n})m$, the graph formed by deleting the path has all vertices of degree at least $\frac{m}{\log n}$, so a path can be found of length $\frac{m}{c \log^2 n}$. Thus at most $c \log^2 n$ paths need to be found to construct a path of length $m - o(m)$.

The first step in the algorithm is to randomly label the vertices of the graph. Each vertex, independently and uniformly picks a label from the set $\{0, \ldots, d-1\}$ where $d = \left\lfloor \frac{m}{c \log n} \right\rfloor$. For a random labelling, it is very likely that each vertex has at least one neighbor with label i for every $i \in \{0, \ldots, d-1\}$. We denote the set of labels of neighbors of v by $L(v)$. A labelling with $L(v_i) = \{0, \ldots, d-1\}$ for all $v_i \in V$ is referred to as a *good* labelling.

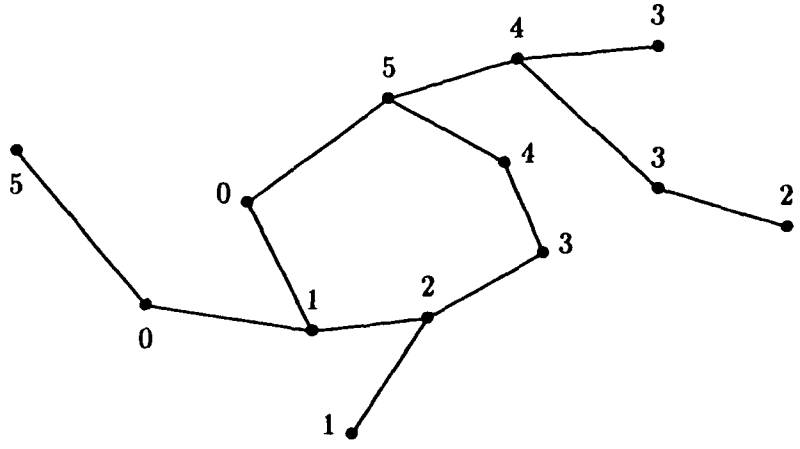
Lemma 3.1. *For a random labelling, $L(v_i) = \{0, \ldots, d-1\}$ for all $v_i \in V$ with probability at least $1 - \frac{1}{n}$.*

Proof: Let $c = 3 \ln 2$. The lemma is proved by bounding the probability that there is some vertex which is not adjacent to vertices with all of the labels in $\{0, \ldots, d-1\}$.

$$
\begin{aligned}
P\{\exists v_i \mid L(v_i) \neq \{0, \ldots, d-1\}\} &\leq \sum_{v_i \in V} P\{L(v_i) \neq \{0, \ldots, d-1\}\} \\
&\leq \sum_{v_i \in V} \sum_{0 \leq j \leq d-1} P\{j \notin L(v_i)\} \\
&\leq \sum_{v_i \in V} \sum_{0 \leq j \leq d-1} (1 - \frac{1}{d})^m \\
&\leq n^2 (1 - \frac{1}{d})^{d \cdot c \log n} \\
&\leq n^2 e^{-c \log n} = \frac{1}{n}
\end{aligned}
$$

∎

The labelling step is repeated until a good labelling is found. Once a good labelling is found, each vertex picks one of its neighbors $a(v)$ with a label one greater than its label, thus if vertex v has label k, it picks one of its neighbors with label $(k+1) \bmod d$. An auxiliary graph is constructed with vertices v and an edge from each vertex v to the associated vertex $a(v)$. A typical example of the auxiliary graph is illustrated below. Since this graph has $|V|$ edges, it must have a cycle. On the cycle, the labels increase by exactly one going from a vertex to its neighbor, so the graph has a cycle of length at least d. This cycle can be found by path doubling in $O(\log n)$ time.



Auxiliary Graph

3.3. Finding a Maximal Set of Disjoint Paths

Another path problem is to find a maximal set of disjoint paths. The problem is:
Given a graph $G = (V, E)$ and a subset U of the vertices, find a maximal set $P = \{P_1, \ldots, P_k\}$ of vertex disjoint paths that join vertices of U.

This means that no more paths can be added to P that have their endpoints in U. We require that the paths are non-trivial (i.e., they contain at least two vertices), and that vertices of U only appear in the paths of P as endpoints.

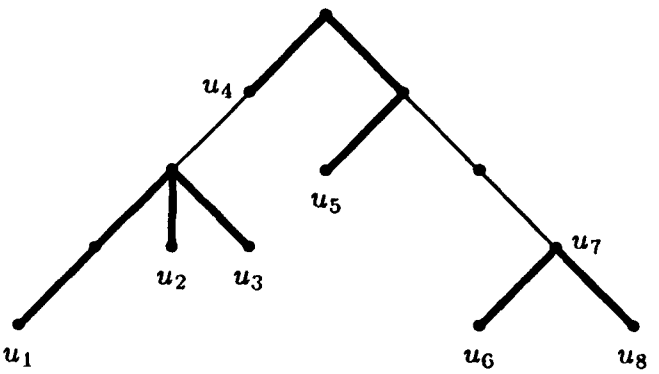
The maximal set of disjoint paths problem is a generalization of maximal matching. If $U = V$, then the problem is to find a maximal matching in the graph. The maximal set of disjoint paths problem is an important subroutine for a number of sequential and parallel algorithms. The directed variant of the problem is a key step in the Hopcroft-Karp bipartite matching algorithm [HK]. We use an algorithm for finding a maximal set of disjoint paths in our depth first search algorithm. We present two algorithms for finding disjoint paths. The first algorithm is an $\mathcal{NC}$ algorithm for graphs with bounded degree. The second algorithm is actually for a more difficult problem; for finding a *maximum* set of disjoint paths instead of just finding a maximal set of disjoint paths. The algorithm reduces the problem to matching, so it can be solved in $\mathcal{RNC}$ using the Karp-Upfal-Wigderson matching algorithm. It is straightforward to generalize our second algorithm

to directed graphs. However, the bounded degree algorithm applies only to undirected graphs.

When the maximum degree of the graph is bounded by d, a maximal set of disjoint paths can be found in $O(d \log^3 n)$ time. Thus if d is $O(\log^k n)$, the problem can be solved in $\mathcal{NC}$. The algorithm uses the *iterated improvement strategy* mentioned in Chapter 1. Each phase of the algorithm finds a number of disjoint paths and deletes them from the graph. The number of vertices in U is reduced by a factor of about $1 - \frac{2}{d}$ each phase.

A phase begins by finding a spanning tree of the graph. Then a maximal set of disjoint paths is found in the spanning tree. The paths are then deleted and another phase is run. If the graph becomes disconnected, the separate components are considered independently.

A maximal set of disjoint paths in the tree is constructed by following paths from the vertices of U up towards the root. Whenever two or more paths intersect, two of the paths are joined. For example, in the figure below, paths would be found between the pairs of vertices (u_1, u_3), (u_4, u_5), and (u_7, u_8). The vertices u_2, and u_6 would be left for the next phase.



It is not difficult to find the paths quickly in parallel. One method is to assign values to the edges of the tree: 1 indicates it is a path edge, and 0 indicates it is not a path edge. If a vertex is not in U, then the edge coming out of it has value 1 if exactly one of the edges coming into it has value 1 and the value is 0 otherwise. For a vertex in U, the value of its outgoing edge is 1 if all of the edges coming into it have value 0 and the value is 0 otherwise. The values of all the edges can be computed by treating the tree as a type of circuit and using the standard technique for evaluating circuits with fanout one.

Whenever some paths are joined at a vertex, some other paths might be cut off. The tree can be partitioned into connected components that consist of the edges that are assigned the value 1. There is at most one component that contains exactly one vertex of U. For the other components, the worst case is if a vertex is in U and all of its incoming edges represent paths. Since the maximum degree is assumed to be d, this vertex accounts for two vertices in U being joined, and $d - 1$ being left unjoined. Thus, the number of vertices in U that are joined is at least $\frac{2}{d+1}(|U| - 1)$. The number of phases of the algorithm is bounded by $\frac{\log |U|}{\log(1+\frac{2}{d})} = O(d \log |U|)$. Each phase takes $O(\log^2 n)$ time.

3.4. Finding a Maximum Set of Disjoint Paths

We now turn our attention to finding a maximum set of disjoint paths instead of just a maximal set of disjoint paths. The maximum set of disjoint paths problem (MDP) is:

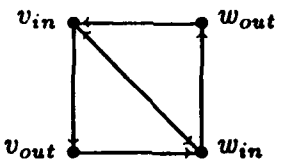
Given a graph $G = (V, E)$ and a set of vertices U, find a maximum cardinality set of nontrivial vertex disjoint paths that have their endpoints in U.

This problem is a much more difficult problem than finding a maximal set of disjoint paths. If $U = V$, then the problem is to find a maximum matching. Finding disjoint paths is the central step in our maximal path algorithm. We show that MDP is in $\mathcal{RNC}$ by reducing it to matching.

If the problem were to find a maximum set of disjoint paths from a set U_1 to a set U_2, it could be expressed as a flow problem with unit capacities and then it could be reduced to bipartite matching [ET]. However, since matching is a special case of MDP, the reduction is a little more difficult. Instead of reducing MDP to a flow problem, we reduce it to a *bidirectional* flow problem [PS ex. 8.6] [Law] and then reduce the bidirectional flow problem to a matching problem. A bidirected graph is a set of vertices, a set of directed edges, and a set of bidirected edges. A directed edge $a \rightarrow b$ can carry a unit of flow from a to b. A bidirected edge $a \leftrightarrow b$ can either give a unit of flow to both a and b, or give no flow to either. A bidirected edge can be thought of as a special source that must give the same amount of flow to both of its neighbors. The flow problem is to determine how much flow can be delivered to a sink vertex t.

Lemma 3.2. *MDP can be reduced in log-space ($O(\log n)$ parallel time) to a unit capacity bidirectional flow problem.*

Proof: We transform MDP into a bidirectional flow problem, where the flow corresponds to a set of disjoint paths. Each vertex v is replaced by a pair of vertices v_{in} and v_{out}, with a directed edge $v_{in} \rightarrow v_{out}$ between them. For an edge (v, w) in the graph there are directed edges $v_{out} \rightarrow w_{in}$ and $w_{out} \rightarrow v_{in}$ and a bidirected edge $v_{in} \leftrightarrow w_{in}$ as shown:



There is a sink t, and for each vertex u_{out} corresponding to $u \in U$, there is an edge $u_{out} \rightarrow t$. The problem is to find a maximum flow to t. The bidirectional edges serve as the sources of the flow.

There is a direct correspondence between a 0–1 flow of $2k$ in the flow graph and a set of k disjoint paths in the original graph. Suppose the bidirected graph has a flow of $2k$. The flow is introduced on k bidirected edges. The flow introduced at $v_{in} \leftrightarrow v'_{in}$ follows paths $v_{in}v_{out}v_{1,in} \cdots u_{j,in}u_{j,out}t$ and $v'_{in}v'_{out}v'_{1,in} \cdots u_{i,in}u_{i,out}t$ to the sink. This corresponds to the path $u_j \cdots v_1 v v' v'_1 \cdots u_i$ in the graph. Similarly, suppose we have a set of k disjoint paths in the graph with their endpoints in U. For each path we pick an edge (v, v') and introduce flow on the bidirected edge $v_{in} \leftrightarrow v'_{in}$. The flow then follows the two segments of the path to the sink. ∎

Lemma 3.3. *Unit capacity bidirectional flow can be reduced in log-space to matching.*

Proof: We use a reduction that is similar to the standard reduction of unit capacity flow to bipartite matching [CSV] [Wag]. The handling of bidirected edges causes the reduction to be to general matching instead of bipartite matching. A graph is constructed that has a perfect matching if and only if the bidirected graph has a flow of size $2k$. The maximum flow is found by constructing graphs for each possible value of k. The flow in the network can be reconstructed from a perfect matching.

The table below gives the graph to test for a flow of $2k$ in the bidirected graph as follows. The outdegree of a vertex v is denoted by $out(v)$.

Bidirected Graph	Matching Graph
Sink t	Vertices $t_1, \ldots, t_{2k}$
Vertex i	Vertices $i_1, \ldots, i_{out(i)}$
Edge $i \rightarrow j$	Vertex $\overrightarrow{ij}$
	Edges $(\overrightarrow{ij}, i_1), \ldots, (\overrightarrow{ij}, i_{out(i)})$
	$(\overrightarrow{ij}, j_1), \ldots, (\overrightarrow{ij}, j_{out(j)})$
Edge $i \rightarrow t$	Vertex $\overrightarrow{it}$
	Edges $(\overrightarrow{it}, i_1), \ldots, (\overrightarrow{it}, i_{out(i)})$
	$(\overrightarrow{it}, t_1), \ldots, (\overrightarrow{it}, t_{2k})$
Bidirected Edge $i \leftrightarrow j$	Vertices $\overleftrightarrow{ij}_1$, $\overleftrightarrow{ij}_2$
	Edges $(\overleftrightarrow{ij}_1, \overleftrightarrow{ij}_2)$
	$(\overleftrightarrow{ij}_1, i_1), \ldots, (\overleftrightarrow{ij}_1, i_{out(i)})$
	$(\overleftrightarrow{ij}_2, j_1), \ldots, (\overleftrightarrow{ij}_2, j_{out(j)})$

There is a direct correspondence between a flow of $2k$ and a perfect matching. In a perfect matching, if $\overrightarrow{ij}$ is matched with j_p, there is flow from $i \rightarrow j$, and if $\overrightarrow{ij}$ is matched with i_p there is no flow on $i \rightarrow j$. For a bidirected edge $i \leftrightarrow j$, if $\overleftrightarrow{ij}_1$ is matched with i_p and $\overleftrightarrow{ij}_2$ is matched with j_q, then $i \leftrightarrow j$ delivers flow to both i and j, if $\overleftrightarrow{ij}_1$ is matched with $\overleftrightarrow{ij}_2$, then $i \leftrightarrow j$ does not deliver any flow.

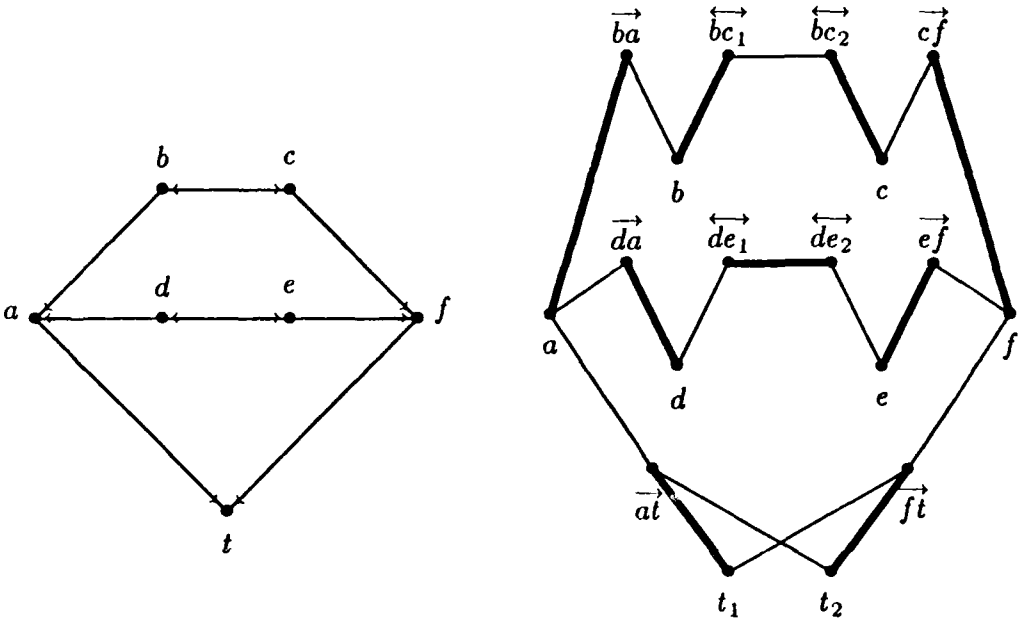
We now prove using the above correspondence that the graph has a perfect matching if and only if there is a flow of $2k$.

Suppose the graph has a perfect matching. The vertices i_p are matched for $1 \leq p \leq out(i)$. Suppose a of the vertices i_p are matched to vertices of the form $\overrightarrow{vi}$ and the other $out(i) - a$ vertices i_p are matched to vertices $\overrightarrow{iv}$. There is a flow of a going into vertex i. Since there are $out(i)$ vertices of the form $\overrightarrow{iv}$, a of the vertices $\overrightarrow{iv}$ are matched to vertices v_q, so there is a flow of a out of i. Hence, flow is conserved at the vertices, so it is a valid flow. Since the vertices $t_1, \ldots, t_{2k}$ are all matched with vertices of the form $\overrightarrow{vt}$, the flow is of size $2k$.

For the other direction of the proof, assume there is a flow of size $2k$. Suppose the flow into i is a. Then a vertices $\overrightarrow{vi}$ and $out(i) - a$ vertices $\overrightarrow{iv}$ can be matched with vertices

i_p corresponding to i. So all of the vertices associated with the vertices in the original graph can be matched. The vertices $\overleftrightarrow{ij_1}$ and $\overleftrightarrow{ij_2}$ can be matched together if flow is not introduced on $i \leftrightarrow j$. Since the flow is of size $2k$, all the vertices $t_1, \ldots, t_{2k}$ are matched. ∎

An example of the reduction from bidirectional flow to matching is shown in the following diagram. The matching illustrated with bold edges corresponds to a bidirectional flow introduced on the edge $b \leftrightarrow c$ that flows to t along the paths $c \rightarrow f \rightarrow t$ and $b \rightarrow a \rightarrow t$.



Bidirected Graph Matching Graph

Combining the two previous lemmas and using the Karp-Upfal-Wigderson probabilistic matching algorithm [KUW], we have the following theorem:

Theorem 3.1. *The maximum set of disjoint paths problem can be solved by an $\mathcal{RNC}$ algorithm.* ∎

3.5. The Maximal Path Problem

In this section we present two algorithms for the maximal path problem. The maximal path problem is:

> Given a graph $G = (V, E)$ and a vertex $r \in V$, find a simple path P starting at r such that P cannot be extended without encountering a vertex that is already on the path.

Our results are that the restricted case of the maximal path problem for bounded degree graphs can be solved in $\mathcal{NC}$ and the general case can be solved in $\mathcal{RNC}$.

The maximal path problem can be solved sequentially by the simple greedy algorithm discussed in Chapter 2. We showed that the greedy algorithm for constructing a maximal

path is P-complete, so it probably cannot be sped up substantially with parallelism. Here we show that by taking a different approach, the problem can be solved by a fast parallel algorithm.

It is a significant result that the maximal path problem can be solved by a fast parallel algorithm. The construction of a maximal path appears to be a very sequential process, since to add a vertex to the path we need to know that the vertex is not already on the path. The initial motivation for looking at the maximal path problem is its relationship to depth first search. Any branch of a depth first search tree a maximal path. The maximal path problem captures some of the difficulties involved with a parallel depth first search. In the next section we describe an $O(n^{1/2+\epsilon})$ parallel algorithm for depth first search. The algorithm works along the same lines as the maximal path algorithm and uses a number of tools developed for it. However, our depth first search algorithm does not depend directly upon the maximal path algorithm.

Both of our maximal path algorithms use a divide and conquer strategy. A path is found which allows the problem to be reduced to finding a maximal path in a graph of less than half the original size. This path is referred to as a *splitting path*. If the graph has a vertex of low degree, then a splitting path can be found relatively easily. However, the general case is substantially more complicated. To find a splitting path in a general graph, we use the probabilistic matching algorithm of Karp-Upfal-Wigderson. We first describe the algorithm to find a maximal path in a graph with all vertices of degree less than d, and then describe the algorithm for general graphs.

3.5.1. Maximal Path Algorithm for Bounded Degree Graphs

Let $G = (V, E)$ be a graph with all vertices having degree at most d. We give a deterministic algorithm to find a maximal path in time $O(d \log^k n)$. This gives an $\mathcal{NC}$ algorithm for families of graphs with a degree bound of $O(\log^j n)$.

The basic step of the algorithm is to find a path that reduces the problem to finding a maximal path in a smaller graph.

Definition 3.1. *A path P starting from r is called a splitting path if $V - P$ has at least two connected components.*

Lemma 3.4. *If splitting paths can be found in time $O(T(n))$, then a maximal path can be found in time $O(T(n) \log n)$.*

Proof: Suppose $P = ru_1 \cdots u_k$ is a splitting path. Let u_j be the last vertex on P such that u_j is adjacent to at least two components of $V - ru_1 \cdots u_j$. Let C be the smallest of the components of $V - ru_1 \cdots u_j$ adjacent to u_j, and let v be a vertex in C that is adjacent to u_j. If P' is a maximal path from v in C, then $ru_1 \cdots u_j P'$ is a maximal path from r in V, so the problem is reduced to finding a maximal path in C. Since $|C| < \frac{n}{2}$, it takes at most $\log n$ iterations of finding a splitting path and reducing the problem to find a maximal path. Hence the maximal path problem can be solved in $O(T(n) \log n)$ time. ∎

We now describe how to find a splitting path. We first take care of the case where the graph is not biconnected. If r has degree one, then we can follow a path from r until we reach a vertex of degree at least three, so we may as well assume that r has degree at least two. If the graph is not biconnected, there is an articulation point v that is in the same biconnected component as r. Any shortest path from r to v is a splitting path.

The more interesting case for finding a splitting path is if the graph is biconnected. For the remainder of the section we assume that G is biconnected. The basic idea in finding a splitting path is to pick a vertex v different from r and find a path that cannot be extended to contain any more neighbors of v. This either gives us a splitting path or a way to construct a maximal path directly. Let v be a vertex different from r. We construct a path, one segment at a time, with each segment going to another neighbor of v without passing through v. We stop when we cannot add another neighbor of v to the path. This is done by the following simple algorithm.

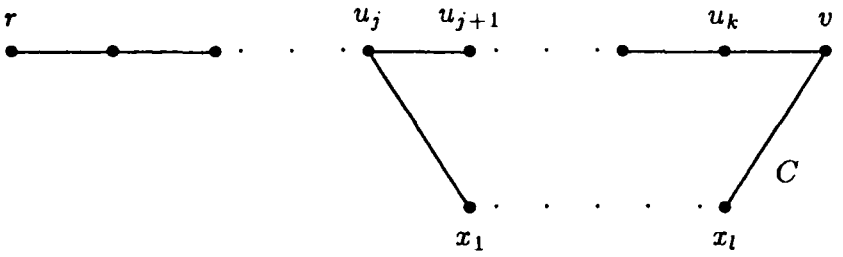
```
VisitNeighbors(r, v)
  begin
    Let v_1, ..., v_m be the neighbors of v;
    P ← r;
    w ← r;
    V' ← V − v;
    for i ← 1 to m do
      if there is a path wP'v_i with wP'v_i ⊆ V' then
        begin
          P ← PP'v_i;
          w ← v_i;
          V' ← V' − wP';
        end
    return P;
  end.
```

After we call *VisitNeighbors*, we add v to P. There are three cases that can occur. First, if P contains all of v's neighbors, then Pv is a maximal path and we are done. Otherwise if $V - Pv$ has more than one connected component, then we have a splitting path. The last case is that all the unvisited vertices are in a single component and some neighbor of v is not on P. This case is handled by the following lemma.

Lemma 3.5. *Let $P = ru_1 \cdots u_k v$ be a path such that $V - P$ has a single connected component C. If v is adjacent to C but u_k is not adjacent to C, then a maximal path can be found by an $\mathcal{NC}$ algorithm.*

Proof: Let u_j be the last vertex other than v on P that is adjacent to C. There must be such a vertex u_j since we are assuming the graph is biconnected. Let $x_1 \cdots x_l$ be a path in C with u_j adjacent to x_1 and v adjacent to x_l. The path $r \cdots u_j x_1 \cdots x_l v u_k \cdots u_{j+1}$ is a maximal path. The path is shown in the figure below. ∎

The individual steps such as finding a path between two vertices, testing for articulation points, and finding connected components can all be done in $O(\log^2(n))$ time on $O(n^2)$ processors. Since the problem size is reduced by at least half every time a path is



found that splits the graph, no more than $\log n$ stages are needed. If all vertices have degree at most $D(n)$, no more than $D(n)$ paths are found by *VisitNeighbors*. The algorithm therefore runs in $O(D(n)log^3(n))$ time on $O(n^2)$ processors. The restriction that all of the vertices obey a global degree bound is not necessary for this algorithm to be a fast parallel algorithm. It is only necessary that at each stage a vertex of low degree can be found. For example, if the graph is planar, it is always possible to find a vertex of degree at most five, so this algorithm runs in $O(\log^3(n))$ for planar graphs.

3.5.2. Maximal Path Algorithm for General Graphs

We now describe an algorithm for the maximal path problem in general graphs. This algorithm also relies on finding a splitting path and reducing the problem to a problem of less than half the size. The problem of finding a splitting path is much more involved than in the previous algorithm. The discussion of finding a splitting path is divided into two parts. A set Q of vertex disjoint paths is said to *separate* the graph if $V - Q$ has at least two connected components. We first show how we can construct a splitting path from a set of paths that separates the graph. The construction is similar to the one for finding a splitting path in a sparse graph discussed above. We then describe how the separating paths are found. This is the most complicated part of the algorithm and relies on some very powerful machinery The algorithm uses as a subroutine the algorithm for finding a maximum set of disjoint paths described in Section 3.4. The resulting algorithm is probabilistic, but its only use of randomness is in the matching subroutine in the algorithm for finding disjoint paths.

We now describe how a splitting path is found. Once the splitting path is found, the algorithm proceeds as the one discussed above. We also assume that the graph is biconnected. A splitting path can be found in a graph that is not biconnected as is done in the bounded degree case.

We show how to construct a splitting path from a small set of vertex disjoint paths that separates the graph. Suppose $Q = \{Q_1, \ldots, Q_k\}$ is a set of vertex disjoint paths. A subroutine that builds a single path using the paths in Q is used in the construction of a splitting path. The routine *ExtendPath*(r, Q, V) constructs a path starting from r that cannot be extended to include any more vertices of Q. In other words, if the constructed path is $P = ru_1 \cdots u_k$, then no vertex lying on any path in Q is contained in any connected component of $V - P$ adjacent to u_k. Such a path is said to be *maximal with respect to* Q. *ExtendPath* is essentially a sequential greedy algorithm. It builds the path by adding segments of the paths in Q to the current path for as long as possible. The only use of parallelism is in the low-level routines which find shortest paths and maintain connected components. The routine is:

```
ExtendPath(r, Q, V)
  begin
    P ← ∅; s ← r;
    while there is a path in V − P from s to a vertex in Q do
      begin
        Let su_1 ··· u_k be the shortest path in V − P from s to Q;
        Suppose u_k ∈ Q_i, and Q_i = v_1 q' u_k q'' v_2 with |v_1 q'| ≤ |q'' v_2|;
        P ← P s u_1 ··· u_k q'';
        Q_i ← v_1 q';
        s ← v_2;
      end
    return Ps;
  end.
```

Each iteration of the while loop halves the length of some path in Q, so if there are initially c paths, there can be at most $c \log n$ iterations. Each of the steps within the while loop (such as finding shortest paths) can be done in $O(\log^2 n)$ time, so the total time is $O(c \log^3 n)$.

To show how to construct a splitting path from a set of paths that separates the graph, we begin with the special case where we have a single path which contains all the neighbors of a vertex.

Lemma 3.6. *Let Q_1 be a path and v a vertex different from r that has all of its neighbors on Q_1. There is an $\mathcal{NC}$ algorithm that finds either a splitting path or a maximal path.*

Proof: If v lies on Q_1, let $Q = \{Q', Q''\}$ be the two segments formed by removing v from Q_1, otherwise let $Q = \{Q_1\}$. Construct a path $P = ru_1 \cdots u_k$ that is maximal with respect to Q in $V - v$ by calling $ExtendPath(r, Q, V - v)$. There are four cases to consider.

1) Suppose u_k is not adjacent to v. If there was a path from u_k to v that contained no vertices of P other than u_k then P could be extended to contain an additional neighbor of v without including v. Hence the component of $V - P$ that contains v is not adjacent to u_k, so P is either maximal, or $V - P$ has more than one component.

Assume that u_k is a neighbor of v and let $P' = Pv$.

2) If all neighbors of v are on P, then P' is a maximal path.
3) If $V - P'$ has more than one component then P' is a splitting path.
4) The final case is when $V - P'$ has a single connected component C and v is adjacent to a vertex in C. This is precisely the case that is covered in Lemma 3.5, so a maximal path can be constructed. ∎

Theorem 3.2. *Let $Q = \{Q_1, \ldots, Q_k\}$ be a set of vertex disjoint paths where $k \leq c$ for some fixed constant c. If $V - Q$ has more than one connected component then a splitting path or a maximal path can be found by an $\mathcal{NC}$ algorithm.*

Proof: Use *ExtendPath* to construct a path P that is maximal with respect to Q. Suppose that P is not a splitting path, so that $V - P$ has a single connected component C. If $Q \not\subseteq P$, then C is not adjacent to the last vertex of P, so P must be a maximal path. Assume that $Q \subseteq P$ and let x_1 and x_2 be vertices in different components of $V - Q$. If both x_1 and x_2 had neighbors in C, then there would be a path from x_1 to x_2 with all of its interior

vertices in C, but this would mean that there was a vertex of Q in C. Hence, either x_1 or x_2 has all its neighbors on P. The previous lemma then shows that a splitting path or a maximal path can be constructed. ∎

We now show how to construct a small set of paths that separates the graph. The paths that we construct are referred to as *isolated*.

Definition 3.2. *A set of vertex disjoint paths $Q = \{Q_1, \ldots, Q_k\}$ is isolated if every path between endpoints of different paths has at least one interior vertex in Q.*

A set of at least two isolated paths can be transformed into a set of paths that separates the graph. Let $Q = \{Q_1, \ldots, Q_k\}$ be a set of isolated paths, and let x_1 be an endpoint of the path Q_1 and x_2 be an endpoint of Q_2. The paths $Q' = \{Q_1 - x_1, Q_2 - x_2, Q_3, \ldots, Q_k\}$ separate the graph with x_1 and x_2 in different components of $V - Q'$.

A set of isolated paths can be constructed from a set of paths by repeatedly combining paths. A maximum set of disjoint paths is found between the endpoints of the paths. If a path P is found between endpoints of Q_i and Q_j, $i \neq j$, then the paths Q_i, P and Q_j are joined to form a single path. Phases of joining paths are repeated until no more paths can be joined. The routine *JoinPaths* constructs a set of isolated paths. The input to *JoinPaths* is a set of vertex disjoint paths $Q = \{Q_1, \ldots, Q_k\}$ and a set of vertices T not on the paths. *JoinPaths* constructs a set of isolated paths $Q' = \{Q'_1, \ldots, Q'_j\}$. Every path in Q is a segment of some path in Q'.

```
JoinPaths(Q, T)
  begin
    while the paths in Q are not isolated do
      begin
        Construct an auxiliary graph G' with vertices x_i for each Q_i ∈ Q, and vertices v_i for
          each v_i ∈ T. The edge (v_i, v_j) is in G' if (v_i, v_j) is an edge in the original graph;
          (x_i, v_j) is in G' if there is an edge from an endpoint of Q_i to v_j, and (x_i, x_j) is in
          G' if there is an edge between endpoints of Q_i and Q_j, i ≠ j;
        Find a maximum set of disjoint paths P = {P_1, ..., P_j} in G' with their endpoints in
          {x_1, ..., x_k};
        for each P_l ∈ P do in parallel
          begin
            Suppose P_l = x_i P'_l x_j with i < j;
            Q_i ← Q_i P'_l Q_j;        (joined appropriately)
            Q_j ← ∅;
            T ← T − P'_l;
          end
      end
  end.
```

Earlier in this chapter we showed how a maximum set of disjoint paths can be computed in $\mathcal{RNC}$ by using matching. The manipulation of paths and the construction of the auxiliary graph can be done easily in $\mathcal{NC}$. To show that *JoinPaths* is an $\mathcal{RNC}$ algorithm, we show that the number of phases (i.e. iterations of the outer loop), is $O(\log n)$.

Lemma 3.7. *The process of joining the paths $\{Q_1, \ldots, Q_m\}$ requires $O(\log m)$ phases.*

Proof: Suppose k joins are performed at phase j. Any subsequent join must involve at least one unjoined endpoint of a path joined at phase j. Thus there are at most $2k$ subsequent joins. The number of paths that are joined at a phase is no greater than the number joined the previous phase. Therefore, there are at most $\frac{k}{2}$ joins at phase $j+4$. Because there are at most $\frac{m}{2}$ joins during the first phase, there are $O(\log m)$ phases. ∎

In the maximal path algorithm, it is important to insure that the joining process does not result in a single path. The following lemma gives a simple case where the joining process does not form a single path.

Lemma 3.8. *If at most $\frac{m}{3} - 1$ joins are performed in the first phase of the joining process, then there are at least two paths left when the joining process is done.*

Proof: The total number of joins performed is at most $\frac{m}{3} - 1 + 2(\frac{m}{3} - 1) = m - 3$. The number of joins required to combine the paths into a single path is $m - 1$. ∎

The set of paths joined in a single phase depends upon the particular set of disjoint paths found by the probabilistic matching algorithm. The number of paths joined in a phase is however a fixed number, independent of the probabilistic choices made. We denote the number of paths joined in the first phase of *JoinPaths* by $J(Q_1, \ldots, Q_k)$. The *JoinPaths* procedure does not actually require finding a *maximum* set of disjoint paths to construct isolated paths, it is sufficient to find a *maximal* set of disjoint paths. However, for technical reasons explained later, our maximal path algorithm relies on the fact that *JoinPaths* does find a maximum set of disjoint paths. A second reason for using a maximum set of disjoint paths is that it is not known how to construct in parallel a maximal set of disjoint paths without constructing a maximum set of disjoint paths.

A major portion of the maximal path algorithm is to construct a small set of isolated paths. We construct a set of isolated paths $Q = \{Q_1, \ldots, Q_k\}$ where $2 \leq k \leq 5$. (There is nothing special about the the bound of five, it is just necessary to have $k < c$, for some constant c.) An initial set of isolated paths is constructed by calling $JoinPaths(V, \emptyset)$ (treating each vertex as a path of length one). If we are very lucky and a single path is constructed, then we can find a path starting at r that contains at least half the vertices and reduce the problem without bothering with a splitting path. If between two and five paths are constructed, a splitting path can be found directly, otherwise the number of paths must be reduced. The basic idea is to discard some of the paths and use the vertices of the discarded paths as well as the vertices not on any of the paths to join the remaining paths. This process is repeated until between two and five isolated paths remain. It is easy to reduce the number of paths significantly with a phase, since any number of paths may be discarded. It is necessary to make sure that at least two paths remain when the paths are joined.

The algorithm for reducing the number of paths works in phases, where each phase reduces the number of paths on hand by at least a factor of $\frac{1}{4}$. A phase starts with a set of isolated paths $Q = \{Q_1, \ldots, Q_m\}$. To reduce the number of paths in Q, we choose a suitable k and replace Q by $\{Q_1, \ldots, Q_k\}$ and join the paths in the new Q. The first value of k that we try is $k = \frac{3m}{4}$. If $J(Q_1, \ldots, Q_{\frac{3m}{4}}) < \frac{m}{4} - 1$, then Lemma 3.8

guarantees that the paths do not collapse to a single path. Suppose $J(Q_1,\ldots,Q_{\frac{3m}{4}}) \geq \frac{m}{4}$. Since $J(Q_1,\ldots,Q_m) = 0$, there is a $k > \frac{3m}{4}$ such that $J(Q_1,\ldots,Q_k) \geq \frac{m}{4} - 1$ and $J(Q_1,\ldots,Q_{k+1}) < \frac{m}{4} - 1$. This value can be found by checking all values of k in parallel. This value of k still might not be satisfactory, since joining $Q_1,\ldots,Q_k$ could cause the paths to collapse to a single path, while joining $Q_1,\ldots,Q_{k+1}$ might not give enough reduction. If this is the case, we find a segment P of Q_{k+1} such that $\frac{m}{4} - 2 \leq J(Q_1,\ldots,Q_k,P) \leq \frac{m}{4} - 1$. If we remove a vertex from Q_{k+1}, we change the number of joins in the first phase of *JoinPaths* by at most two. This is because we change the auxiliary graph used in *JoinPaths* by two vertices. When finding a maximum set of disjoint paths in a graph that has been altered in this way, the number of paths found changes by at most two. This "continuity" property is why we chose to use a maximum set of disjoint paths instead of settling for maximal. We can find this segment P by testing each of the initial segments of Q_{k+1} in parallel. When we join paths, we guarantee that the number of paths is reduced by at least $\frac{m}{4} - 2$ since the first phase performs at least that many joins.

ReducePaths$(Q = \{Q_1,\ldots,Q_m\}, V')$
begin
if $J(\{Q_1,\ldots,Q_{\frac{3m}{4}}\}) < \frac{m}{4} - 1$ **then**
JoinPaths$(\{Q_1,\ldots,Q_{\frac{3m}{4}}\}, V' \cup \{Q_{\frac{3m}{4}+1},\ldots,Q_m\})$;
else
begin
Find a $k > \frac{3m}{4}$ such that $J(Q_1,\ldots,Q_k) \geq \frac{m}{4} - 1$ and $J(Q_1,\ldots,Q_{k+1}) < \frac{m}{4} - 1$;
Suppose $Q_{k+1} = p_1 \cdots p_j$;
Find an initial segment $P = p_1 \cdots p_{j'}$ of Q_{k+1} such that
$\frac{m}{4} - 2 \leq J(Q_1,\ldots,Q_k,P) \leq \frac{m}{4} - 1$;
JoinPaths$(\{Q_1,\ldots,Q_k,P\}, V' \cup \{Q_{k+2},\ldots,Q_m\} \cup \{p_{j'+1} \cdots p_j\})$;
end
end.

This completes our description of the pieces of our algorithm for finding a maximal path. We now put them together and give the full algorithm.

MaximalPath(V, r)
begin
$P \leftarrow$ *FindSplittingPath*(V, r);
if P is a maximal path **then**
return P;
Let $ru_1 \cdots u_k$ be a subpath of P such that $V - ru_1 \cdots u_k$ has at least two connected components adjacent to u_k. Suppose C is the smallest of the components adjacent to u_k and $v \in C$ is adjacent to u_k;
$P' \leftarrow$ *MaximalPath*(C, v);
return $ru_1 \cdots u_k P'$;
end.

```
FindSplittingPath(V, r)
  begin
    if V is not biconnected then
      begin
        Let v be an articulation point in the same biconnected component as r;
        Let P be a shortest path from r to v;
        return P;
      end
    else
      begin
        comment First we find a set of paths that separates the graph;
        Q = {Q_1, ..., Q_m} ← JoinPaths(V, ∅);
        if m = 1 then
          reduce the problem directly;
        V' ← ∅;
        while m > 5 do
          ReducePaths(Q, V');
        Let x_1 be an endpoint of Q_1 and x_2 be an endpoint of Q_2, x_1, x_2 ≠ r;
        Q' ← {Q_1 − x_1, Q_2 − x_2, ..., Q_m};
        comment The paths Q' separate the graph, we now find the splitting path;
        P ← ExtendPath(r, Q', V);
        if P is a splitting path or a maximal path then
          return P;
        without loss of generality, assume x_1 has all of its neighbors on P;
        if x_1 ∈ P then
          Q'' ← {P_1, P_2} where P_1 and P_2 are the segments of P − x_1;
        else
          Q'' ← {P};
        P ← ExtendPath(r, Q'', V − x_1);
        if P is a splitting path then
          return P;
        else if Px_1 is a splitting path or a maximal path then
          return Px_1;
        else
          begin
            Construct a maximal path P' using Lemma 3.5;
            return P';
          end
      end
  end.
```

Each of the calls to *MaximalPath* reduces the problem by half, so it is called at most $\log n$ times. The most time consuming steps in *FindSplittingPath* are the calls to *ReducePaths*. *ReducePaths* runs in polylog time and is called $O(\log n)$ times so the entire procedure runs in polylog time. Hence, we have the following theorem:

Theorem 3.3. *The maximal path problem is in RNC.* ∎

3.6. Depth First Search

In this section a parallel algorithm for depth first search is described. The depth first search problem is:

> Given a graph $G = (V, E)$ and a vertex $r \in V$, find a spanning tree T of G that could be constructed by some depth first search of G with root r.

Our algorithm runs in $O(\sqrt{n}\log^k n)$ and uses a polynomial number of processors. The original motivation for looking at the maximal path problem is its relation to depth first search. Any branch from the root to a leaf in a depth first search tree is a maximal path. The maximal path algorithm does not apply directly to give a fast parallel algorithm for depth first search, since $O(n)$ maximal paths might be necessary to construct a depth first search tree. However, the general techniques and tools developed for the maximal path problem are used for depth first search.

The depth first search algorithm uses the same strategy as was used for finding a maximal path. A partial solution is found that allows the problem to be reduced to smaller problems which are solved recursively. The algorithm finds a set Q of disjoint paths such that the size of the largest connected component of $V - Q$ is less than $\frac{1}{2}n$. These paths are referred to as a *separating set* of paths. Note that this usage of the term *separate* differs from our usage in the description of the maximal path algorithm. An initial segment of a depth first search tree containing Q is constructed. Depth first search trees are then found for the remaining components. Since the problem is halved at each level, the depth of recursion is at most $\log n$. The procedures for finding separating paths and constructing an initial segment of the tree both take $O(\sqrt{n}\log^k n)$ time. We first describe the construction of the initial segment and then discuss the more complicated step of finding the separating paths.

The routine *InitialSegment* is given a set of disjoint paths Q and constructs a subtree T' of some depth first search tree T with all the vertices of Q contained in T'. In the depth first search algorithm, there will be $O(\sqrt{n}\log n)$ paths in Q when *InitialSegment* is called. *InitialSegment* is essentially a sequential algorithm; however it uses the parallel routine *ExtendPath* described above. *InitialSegment* maintains the connected components of the vertices not in the subtree T'. A component is said to be *active* if it contains a path from the set Q.

```
InitialSegment(Q, r)
  begin
    T' ← r;
    while there is an active component of V − T' do
      begin
        Let v be the lowest vertex on T' adjacent to an active component C of V − T';
        P ← ExtendPath(v, Q, C);
        Add P to T';
        Recompute the connected components of V − T';
      end
  end.
```

Each phase of the routine *ExtendPath* reduces the length of some path in Q by a factor of at least one half. If there are initially m paths in the set Q, then *InitialSegment* will take $O(m \log^k n)$ time.

Lemma 3.9. *The tree T' constructed by InitialSegment can be extended to a depth first search tree.*

Proof: It suffices to show that there are no paths between separate branches of T' that have all their interior vertices in $V - T'$. This condition holds throughout the execution of *InitialSegment* since the extensions are made at the lowest vertex adjacent to some component. ∎

We now show how to find a separating set of paths. We construct a set Q of disjoint paths where Q contains $O(\sqrt{n} \log n)$ paths and the largest component of $V - Q$ has size at most $\frac{n}{2}$. The procedure runs in $O(\sqrt{n} \log^k n)$ time. The routine *Separate* constructs a set of paths and then attempts to reduce the number of paths while keeping the connected components of the vertices not on the paths small. The paths are reduced by joining them together or by removing vertices from them. The routine maintains several sets of paths. The set Q contains paths that are in the separator. Once a path is put into Q it is committed to the separating set and is not be removed. The set S stores the paths that are currently being worked on. The set R is used for temporary storage; paths are put into R to set them aside for the next phase. The vertices not on any of the paths in Q, R, or S are in a set T. The size of the largest component of T is at most $\frac{n}{2}$. The algorithm runs until R and S are empty.

In *Separate* an iteration of the outer while loop is referred to as a phase and an iteration of the inner while loop is a subphase. Each phase halves the number of paths in S. A subphase reduces the length of each path in S by one. Subphases are repeated until all of the vertices in paths of S have been moved to R or T. The paths of R are then moved back to S for the next phase. Phases are repeated until R and S are empty. In the routine below, $Join(S, T)$ is a procedure that performs the first phase of $JoinPaths(S, T)$. $Join(S, T)$ finds a maximum set of vertex disjoint paths in T between the endpoints of the paths in S. The paths in S are then joined using the disjoint paths that were found.

```
Separate
  begin
    Q ← ∅; R ← ∅; T ← ∅;
    S ← V;
    while S ≠ ∅ do
      begin
        while S ≠ ∅ do
          begin
            Join(S, T);
            Move the joined paths from S to R;
            Move one endpoint of each remaining path in S to T;
            if there is a component in T of size ≥ n/2 then
              Fix the component size by moving a vertex from T to Q;
          end
        S ← R; R ← ∅;
```

Move paths of length $\geq \sqrt{n}$ from S to Q;
end
end.

The goal of the routine *Separate* is to remove all the paths from S and R while making sure that the largest connected component of T remains of size at most $\frac{n}{2}$. *Separate* accomplishes this by alternately joining paths of S and by moving some vertices from paths in S to T. The paths are removed in two different ways. When paths are joined, any paths of length at least $\sqrt{n}$ are put into Q. The other way paths can be removed is if all of their vertices are put into the set T.

A subphase begins by joining as many paths as possible using the vertices of T. The paths of S that are joined are then put into R and set aside until the next phase. The next step is to move one endpoint of each of the paths remaining in S to T. The moving of endpoints accomplishes a dual purpose; it makes new vertices of S endpoints, allowing additional joins in the next subphase, and reduces the lengths of the paths in S. Before the endpoints are removed, each connected component is adjacent to the endpoints of at most one path. This means that when the endpoints are moved from S to T, the only way components of T are merged is if they are adjacent to the endpoints of the same path in S. At most one component of size $\frac{n}{2}$ can be formed each subphase. If a large component is formed, then removing the endpoint that caused it to merge reduces the components to size less than $\frac{n}{2}$. When the vertex is removed, it is placed into Q as a path of length one.

The following lemmas establish that *Separate* constructs the desired separator in $O(\sqrt{n}\log n)$ time.

Lemma 3.10. *The largest connected component of T has size at most $\frac{n}{2}$.*

Proof: The only time vertices are added to T is when endpoints of paths in S are moved to T. When this is done, if a component of size greater than $\frac{n}{2}$ is formed, a vertex can be moved from T to Q to reduce the components to size at most $\frac{n}{2}$. ∎

Lemma 3.11. *There are at most $\sqrt{n}$ subphases per phase.*

Proof: At the start of a phase, the paths in S have length at most $\sqrt{n}$. Each subphase reduces the lengths of all the paths remaining in S by one. ∎

Lemma 3.12. *The number of phases is at most $\log n$.*

Proof: Each path at the start of a phase is the join of two paths from the previous phase, so the number of paths in S is halved by each phase. ∎

Lemma 3.13. *When Separate is finished, there are $O(\sqrt{n}\log n)$ paths in Q.*

Proof: At most $\sqrt{n}$ paths of length $\sqrt{n}$ can be placed in Q. Each subphase places at most one singleton in Q. Since there are at most $\sqrt{n}\log n$ subphases altogether, there are $O(\sqrt{n}\log n)$ paths in Q. ∎

A subphase takes polylog time since the time it takes is dominated by the time it takes to find a maximum set of disjoint paths. Since *Separate* has at most $\sqrt{n}\log n$ subphases, *Separate* runs in $O(\sqrt{n}\log^c n)$ time. The resulting algorithm is probabilistic since it uses

the algorithm of Section 3.4 to find a maximum set of disjoint paths. The depth first search algorithm does not actually require a maximum set of disjoint paths, it would suffice to find a maximal set of disjoint paths. However, as was mentioned above, it is not currently known how to construct a maximal set of disjoint paths without finding a maximum set of disjoint paths. For graphs with bounded degree, the deterministic algorithm for finding a maximal set of disjoint paths can be used, so for that restricted case the depth first search algorithm is deterministic with approximately the same time bound.

3.7. Discussion

In this chapter we have presented parallel algorithms for several path problems. The major results of the chapter were that a maximal path can be found by an $\mathcal{RNC}$ algorithm and that a depth first search tree can be constructed in parallel time $O(n^{1/2+\epsilon})$. The algorithms of this chapter illustrate a number of techniques for parallel algorithms. None of the algorithms depends on path doubling, although path doubling is present in a number of implementation details. The algorithm for finding a long path used a novel probabilistic approach. Probabilistic methods seem important for quite a few parallel algorithms. The algorithms for finding a maximal set of disjoint paths and the maximal path algorithm both make use of the *iterated improvement strategy*. The maximal set of disjoint paths algorithm built up its solution by adding paths to the solution and the maximal path algorithm reduces the size of a separator by joining paths. The algorithm for finding a maximum set of disjoint paths reduced the problem to matching.

There are a number of open problems related to these path problems. By far the most important open problem is whether depth first search can be solved by an $\mathcal{RNC}$ or $\mathcal{NC}$ algorithm. The major obstacle to speeding up our algorithm to an $\mathcal{RNC}$ algorithm is that it appears difficult to reduce the lengths of the paths substantially when joining paths while still making sure that the connected components of the vertices not on the paths remain small. Although these difficulties are technical in nature, it might be necessary to take a different approach to get a substantially faster depth first search algorithm. A second interesting open problem is whether the maximal path problem can solved in $\mathcal{NC}$. One way this could be done is to find a deterministic algorithm for matching. It would also be interesting to find a simpler algorithm than ours for the maximal path problem, even if it still relied on randomness. Our results on the maximal path problem and depth first search do not carry over to directed graphs; the directed variants of the problems are open. A final open problem is whether a maximal set of disjoint paths can be found by a fast parallel algorithm without using matching.

Chapter 4 Approximating P-Complete Problems

4.1. Introduction

In this chapter we investigate various ways of finding approximate solutions to P-complete problems. The type of approximation that we are interested in is fast parallel algorithms that give solutions that are close to the desired solution. We present results for parallel approximation of P-complete problems that are very similar to the results on sequential approximation of NP-complete problems.

There are a number of reasons to look at approximating P-complete problems. The main reason is since P-complete problems probably are not amenable to fast parallel solution, it is important and interesting to see what can be done on these problems using parallelism. A second reason to look at approximation is to develop the theory of parallel approximation in analogue to the sequential theory. The problems which arise when looking at approximate solutions are often problems that are close to the "boundary" of what can and cannot be done efficiently with parallelism, thus they are important for the general study of parallelism.

In this chapter we look at a number of P-complete problems and examine to what extent they can be approximated by fast parallel algorithms. The first problem is the high degree subgraph problem. This problem is to find a vertex induced subgraph of a graph that has all vertices of degree at least as big as some given k. We show tight bounds on the degree of approximation achievable for a variant of this problem by a fast parallel algorithm assuming that $P \neq NC$. The next topic that we look at is approximating number problems and show that it is very similar to the situation for sequential computation. Some P-complete number problems can be solved by fast parallel algorithms when the numbers are small. We give two examples of problems that can be approximated very well by solving restricted cases of the problem and then relating the solution to the original problem. We define *strong* P-completeness analogously to strong NP-completeness so that we can identify number problems that remain difficult even if the numbers involved are small. We show that the problem of computing a first fit decreasing bin packing is strongly P-complete. We also show how a related packing scheme which performs as well as first fit decreasing can be computed in NC.

4.2. The High Degree Subgraph Problem

The high degree subgraph problem is:

Given a graph $G = (V, E)$ and an integer k, find the maximum induced subgraph of the graph that has all vertices of degree at least k.

It is interesting that this problem is P-complete since it is so simple. Most known P-complete graph theory problems have some other device, such as weights or an ordering, that make them difficult. However, this problem could be called a "purely combinatorial

problem." The high degree subgraph problem can be approximated in several different ways. In the next section we derive bounds on the degree of approximation that is possible for a variant of the problem. We also discuss other approaches that construct subgraphs with all vertices of high degree.

The high degree subgraph problem can be solved by a simple sequential algorithm. The algorithm discards vertices of degree less than k one at a time until all vertices have degree at least k or the graph is empty. The correctness of this algorithm follows from two easy lemmas. The first lemma establishes that there is a unique maximum induced subgraph of G with minimum degree at least k. We denote this subgraph by $HDS_k(G)$.

Lemma 4.1. *Let S and T be maximum induced subgraphs of G that have minimum degree at least k, then $S = T$.*

Proof: The induced subgraph on $S \cup T$ has minimum degree at least k. Since S and T are maximum, $|S| = |T| = |S \cup T|$, so $S = T$. ∎

Lemma 4.2. *The sequential algorithm outlined above finds $HDS_k(G)$.*

Proof: Let S be the induced subgraph found by the algorithm. Since the vertices of S have degree at least k, $S \subseteq HDS_k(G)$. Suppose that $S \neq HDS_k(G)$. Let v be the first vertex of $HDS_k(G)$ discarded by the algorithm and let T be the graph just before v is discarded. Since v has degree less than k in T and $HDS_k(G) \subseteq T$, v must have degree less than k in $HDS_k(G)$, a contradiction. Hence $S = HDS_k(G)$. ∎

It is possible that $HDS_k(G)$ is empty. The following lemma due to Erdős [E] establishes an important case where $HDS_k(G)$ is nonempty.

Lemma 4.3. *If a graph has n vertices and m edges then it has an induced subgraph with minimum degree at least $\lceil \frac{m}{n} \rceil$.*

Proof: The proof is by induction on the number of vertices in the graph. The result holds for graphs consisting of a single vertex. Suppose the result holds for all graphs with fewer than n vertices. Let G be a graph with n vertices and m edges. If all vertices of G have degree at least $\lceil \frac{m}{n} \rceil$, then the graph itself is an induced subgraph with minimum degree at least $\lceil \frac{m}{n} \rceil$. Otherwise we can delete a vertex of minimum degree along with its incident edges, leaving a graph with $n - 1$ vertices and $m - k > m - \lceil \frac{m}{n} \rceil$ edges. By the induction hypothesis the remaining graph has an induced subgraph with minimum degree

$$\left\lceil \frac{m-k}{n-1} \right\rceil \geq \left\lceil \frac{m - \lceil \frac{m}{n} \rceil}{n-1} \right\rceil \geq \left\lceil \frac{m - \frac{m}{n}}{n-1} \right\rceil = \left\lceil \frac{m}{n} \right\rceil .$$

∎

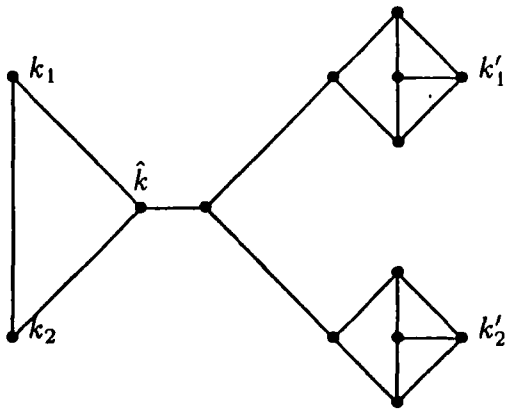
The high degree subgraph problem can be reformulated as a decision problem HDS, by asking if a specific vertex v is in $HDS_k(G)$. We show that HDS is P-complete by giving a reduction from the monotone circuit value problem. We also give a stronger result by showing that it is P-complete to determine if $HDS_k(G)$ is nonempty.

Theorem 4.1. *HDS is P-complete.*

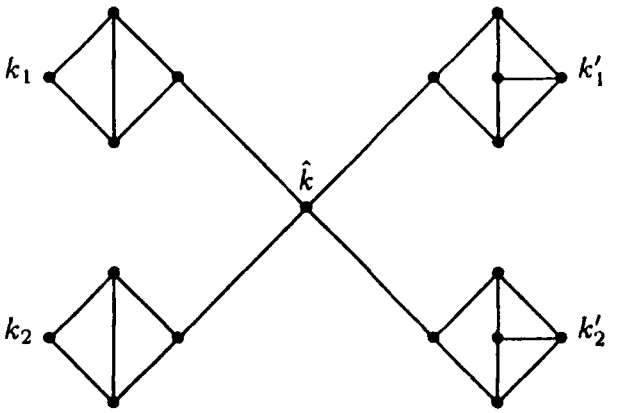
Proof: The proof is a reduction from the monotone circuit value problem. The AND and OR gates are assumed to be connected to one or two other gates, and the inputs are assumed to be connected to just one gate. Let $\beta = \beta_1, \ldots, \beta_n$ be a monotone circuit. The inputs and outputs of each gate are assumed to be numbered, with the connections given explicitly. For example, if β_k receives an input from β_j, we might be given that output 1 of β_j goes to input 2 of β_k. We construct a graph G with a distinguished vertex v such that $v \in IIDS_3(G)$ if and only if the circuit evaluates to *true*.

The gate β_k is simulated by a collection of vertices. There is a vertex $\hat{k}$ in the subgraph associated with β_k that is in $IIDS_3(G)$ if and only if β_k evaluates to *true*. The subgraphs for an AND gate and an OR gate are shown below. The vertices k_1 and k_2 are associated with the inputs of β_k and k'_1 and k'_2 are associated with the outputs of β_k. The subgraph for a 0-INPUT β_k is a singleton vertex k'_1 and the subgraph for a 1-INPUT β_k is a clique of four vertices with one of them k'_1. The subgraphs are connected together in a manner that corresponds to the gates of the circuit. If output p of β_j goes to input q of β_k, then there is an edge from j'_p to k_q.

The circuit is simulated by computing $IIDS_3(G)$. This can be done by discarding vertices of degree one and two until all vertices have degree at least three. A gate β_k receives a *false* input at k_p if the edge going into k_p is removed. All the vertices of the subgraph associated with an AND gate are removed if it receives a single *false* input, and all of the vertices of an OR gate are removed if it receives two *false* inputs. The vertex associated with a 0-INPUT is always removed, and the vertices associated with a 1-INPUT are never removed. Note that values do not propagate backwards. For example, if an edge going into k'_p from a different subgraph is removed, the vertex k'_p is not removed because of that. ∎



AND gate



OR gate

Our proof that HDS is P-complete is just for $k = 3$. However, it is not difficult to modify the proof to show that the problem is P-complete for any $k \geq 3$. For $k = 2$, it is possible to find $IIDS_k(G)$ with an $\mathcal{NC}$ algorithm. The algorithm for computing $IIDS_2(G)$ has $\log n$ phases, where each phase removes all *chains*. A chain is a path of vertices that starts with a vertex of degree one and contains no vertex of degree greater than two. The

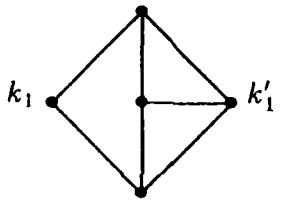
chains can easily be identified by path doubling techniques. When the chains are deleted, more vertices of degree one might be created, however each new vertex of degree one requires the removal of at least two chains, so the number of chains decreases by at least half each phase.

The P-completeness result can be made stronger in the sense that the problem of determining if $IIDS_k(G)$ is nonempty is P-complete. In the next section, we discuss the problem of determining the largest k such that $IIDS_k(G)$ is nonempty. The stronger P-completeness result shows that this problem probably cannot be solved exactly by an $\mathcal{NC}$ algorithm.

Theorem 4.2. *The problem of determining if $IIDS_k(G)$ is nonempty is P-complete.*

Proof: The proof is a reduction from the monotone circuit value problem. The construction of the previous theorem is modified so that given a monotone circuit $\beta_1, \ldots, \beta_n$, a graph G is constructed such that $IIDS_3(G)$ is nonempty if and only if the output of the circuit is *true*.

The subgraphs and connections for the AND gates, OR gates, and 0-INPUTS are the same as in the previous construction. The subgraph for the 1-INPUT β_k is:



If the gate β_j receives an input from β_k, there is and edge from k_1' to j_p. There is a binary tree that has as its leaves the vertices k_1 of the 1-INPUTS β_k. The output vertex of the subgraph for the final gate is the root of this tree.

The computation of $IIDS_3(G)$ simulates the circuit in the same manner as the previous theorem. If the final output is removed, then all of the vertices of the tree are removed and then all of the 1-INPUTS are removed. This causes all of the vertices to be removed, so $IIDS_3(G)$ is empty. If the final output is not removed, then the vertices of the tree and the 1-INPUTS are not removed, so a subgraph with minimum degree three is left. ∎

4.3. Approximations to the High Degree Subgraph Problem

If a problem is known to be P-complete, there is little hope of finding an $\mathcal{NC}$ algorithm to solve it. As is often done with NP-complete problems, we can lower our sights and attempt to find an approximate solution. The high degree subgraph problem is well suited for approximation since it can be rephrased as an optimization problem. The optimization problem is to ask what is the largest k such that $IIDS_k(G)$ is nonempty. This value is denoted $IIDS(G)$. It follows from Theorem 4.2 that it is P-complete to compute $IIDS(G)$. An approximate solution to this problem is to find a k such that $IIDS(G) \geq k \geq \frac{1}{c} IIDS(G)$ for some fixed $c > 1$. We say that this is an approximation within a factor of c. We show that this high degree subgraph problem can be approximated to a factor of c for any $c > 2$

in $\mathcal{NC}$, but cannot be approximated in $\mathcal{NC}$ for $c < 2$, unless $\mathcal{P} = \mathcal{NC}$. This result is analogous to a number of results on approximating NP-complete problems where lower bounds on the degree of approximation are known assuming that $\mathcal{P} \neq \mathcal{NP}$. For example it is known that graph coloring cannot be approximated to a factor of less than two [GJ1] and precedence constrained scheduling cannot be approximated to a factor of less than $\frac{4}{3}$ [LR].

Theorem 4.3. *For any constant $c > 2$, the optimization problem can be solved by an $\mathcal{NC}$ algorithm to a factor of c.*

Proof: Let $\epsilon > 0$. The following routine $Test(V, k)$ returns an answer which is either "the graph has no subgraph with minimum degree k", or "the graph has a subgraph with minimum degree at least $\frac{1-\epsilon}{2}k$".

```
Test(V, k);
  begin
    while V ≠ ∅ do
      begin
        U := {v ∈ V | deg(v) < k};
        if |U| < ε|V| then
          return "IIDS(G) ≥ (1/2 − ε)k";
        V := V − U;
      end
    return "IIDS(G) < k";
  end.
```

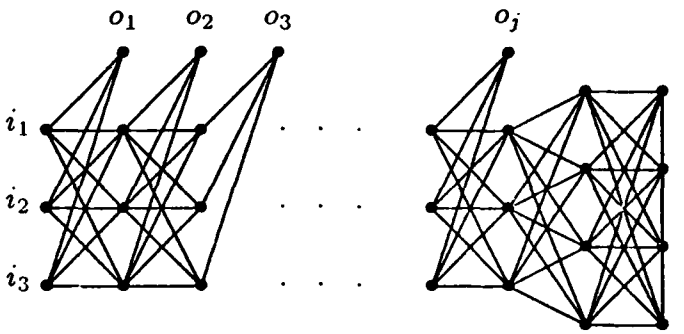
Each iteration of the while loop discards all vertices with degree less than k. Since a constant fraction of the vertices is discarded in each iteration, there are $O(\log n)$ iterations. The algorithm can be implemented so that a single iteration takes $O(\log n)$ time, so it is an $\mathcal{NC}$ algorithm. If $Test(V, k)$ terminates with an empty set of vertices, the graph does not have a subgraph with minimum degree k. Suppose that $Test(V, k)$ terminates with n' vertices, claiming that $IIDS(G) > (1/2 - \epsilon)k$. The number of edges when $Test(V, k)$ terminates is at least $(1/2 - \epsilon)kn'$, so by Lemma 4.3, G has a subgraph with minimum degree at least $\frac{1-\epsilon}{2}k$. The procedure $Test(V, k)$ is applied for each value of k between 1 and n. A value k is found where the graph has a subgraph of degree at least $\frac{1-\epsilon}{2}k$ but no subgraph of degree $k + 1$. This gives an approximation to within a factor of $\frac{2}{1-\epsilon}$. ∎

The next theorem shows that the previous result is essentially the best possible assuming that $\mathcal{P} \neq \mathcal{NC}$. We show that a circuit can be simulated by a graph which has $\overline{IIDS}(G) \leq 2k$ if the output of the circuit is *true* and $\overline{IIDS}(G) \geq k + 1$ if it is *false*. If the problem could be approximated by a factor of less than two then the following construction could be used to solve the monotone circuit value problem.

Theorem 4.4. *If $\mathcal{P} \neq \mathcal{NC}$, then it is not possible to approximate $\overline{IIDS}(G)$ in $\mathcal{NC}$ by a factor less than two.*

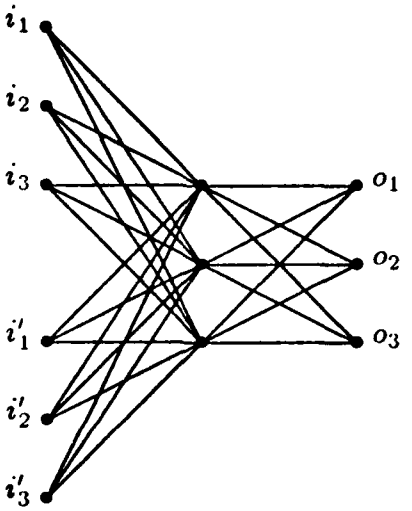
Proof: This theorem is proved by giving a log-space transformation of a monotone circuit to a graph G which has $IIDS(G) \leq 2k$ if the output of the circuit is *true*, and $\overline{IIDS}(G) = k + 1$ if it is *false*. The figures are for $k = 3$, the generalization to other values of k is

straightforward. For each AND and OR gate there is a subgraph. These subgraphs are connected together in a manner that corresponds to connections of the circuit. There are also subgraphs which are called *expanders*. An expander is shown below. The expanders are used to fanout values and are also used in the AND gate. The vertices $i_1, \ldots, i_k$ are the expander's inputs and $o_1, \ldots, o_j$ are the expander's outputs. An expander consists of a number of levels of k vertices each. Adjacent levels are connected by complete bipartite graphs. Each level is connected to an output of the expander. The expander is terminated with two layers that have $k+1$ vertices. The vertices of the last level are joined to form a $k+1$ clique.

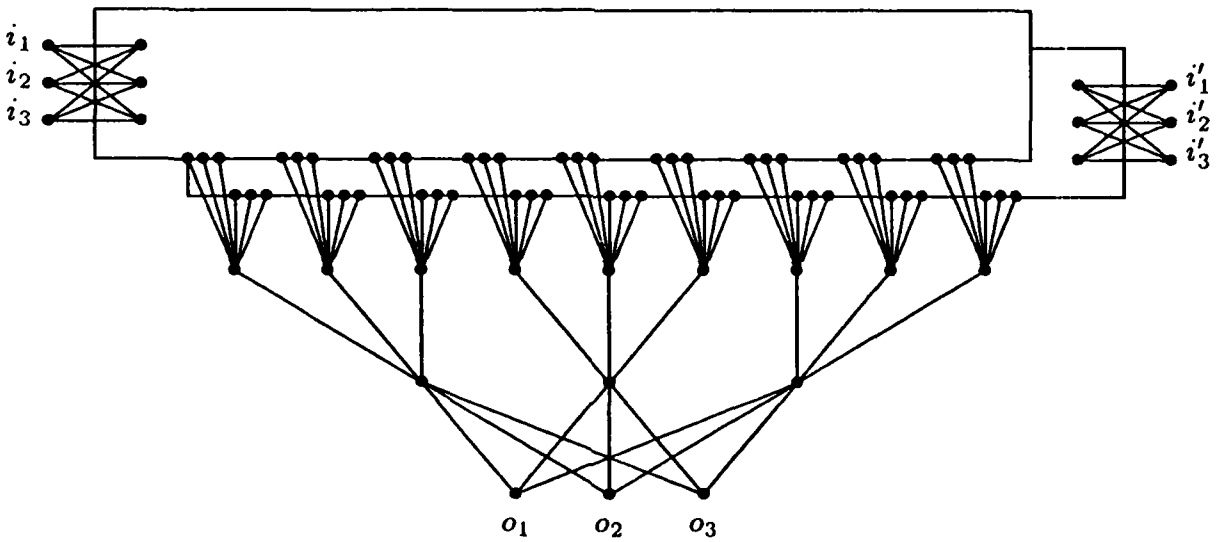


The OR and AND gates are illustrated in the figures below. The gates have two sets of input vertices $i_1, \ldots, i_k$ and $i'_1, \ldots, i'_k$ and a set of output vertices $o_1, \ldots, o_k$. The output vertices of a gate are connected to an expander which is connected to the input vertices of the appropriate gates. The expanders fanout values and insure that information is propagated correctly. In the AND gate, the long rectangle is a k^3-expander. The output of the final gate is connected to an expander which goes to all of the input vertices of gates that correspond to connections from 1-INPUTS in the circuit. The input vertices of gates that receive 1-INPUTS have degree $2k$ and the input vertices of gates that receive 0-INPUTS have degree k.

The circuit is simulated by computing $IIDS_{k+2}(G)$. This "evaluates" the gates in topological order. A value is represented by a set of k vertices. A group of vertices all with degree at least $2k$ indicate *true* and a group of vertices of degree at most $k+1$ indicate *false*. In computing $IIDS_{k+2}(G)$, the vertices indicating *false* values are removed. If the input vertices $i_1, \ldots, i_k$ of an expander have degree k, then all of the vertices of the expander are removed, whereas if they have degree $2k$, they are left. This is the manner in which values are propagated. When an AND gate is simulated, all of its vertices are removed if at least one of its inputs is *false*, and all the vertices of an OR gate are removed if both of its inputs are *false*. If the output of the final gate is *false*, then the expander that goes back to the 1-INPUTS is removed. This causes all of the remaining vertices to be removed, so $\overline{IIDS}(G) \leq k+1$. If the final output is *true*, then a graph with minimum degree $2k$ is left, so $\overline{IIDS}(G) = 2k$. $\overline{IIDS}(G) \geq k+1$ since an expander has a subgraph with minimum degree $k+1$. ∎



OR gate



AND gate using a k^3-expander

4.4. Finding a High Degree Subgraph

A second approach to approximating the high degree subgraph problem is to attempt to find a subgraph with high degree without insisting that it is the maximum subgraph with that degree. We discuss two algorithms for this type of approximation. One algorithm discards vertices of low degree until all vertices have a certain degree. This algorithm constructs a supergraph of $HDS_k(G)$. The algorithm exhibits an interesting relationship between the time it takes and how good the approximation to $HDS_k(G)$ is. The second approach is to relate a high degree subgraph to a *maximum density* subgraph. The problem of constructing a maximum density subgraph can be reduced to a unit capacity network flow problem, so it can be solved in $\mathcal{RNC}$.

An approximation to the problem of computing $HDS_k(G)$ is to find a supergraph of $HDS_k(G)$ that has all vertices with degree at least $\frac{k}{c}$. The closer c is to one, the better the approximation. The result of the previous section shows that this is not possible with an $\mathcal{NC}$ algorithm for c a constant less than two, unless $\mathcal{P} = \mathcal{NC}$. In this section we give a family of algorithms for this type of approximation. The algorithm A approximates to a factor $c_A(n)$ and runs in time $T_A(n)$. The degree of approximation improves as the run time increases.

The sequential algorithm for computing $HDS_k(G)$ discards vertices of degree less than k until all vertices have degree at least k. The reason that this algorithm appears inherently sequential is that it is difficult to predict which vertices eventually have degree less than k. When vertices are discarded, some other vertices have their degrees reduced to less than k, and then they are also discarded. It is possible that all vertices that initially have degree at least k are removed. One way to control the number of vertices that are discarded is to throw out vertices of degree much less than k. If vertices of degree $\frac{k}{4}$ are removed, then the number of vertices that initially have degree at least k that get removed is bounded. This is formalized in the following lemma:

Lemma 4.4. *If $G = (V, E)$ if an n vertex graph with p vertices of degree less than k, then G contains a subgraph with minimum degree $\frac{k}{4}$ that has at least $n - \frac{3}{2}p$ vertices.*

Proof: Suppose vertices of degree less than $\frac{k}{4}$ are removed until all vertices have degree at least $\frac{k}{4}$. Let S_k be the set of vertices that initially have degree at least k that are removed. Before a vertex of S_k is removed, it must have had $\frac{3}{4}k$ edges removed that go to it. Each vertex that is removed can have at most $\frac{k}{4}$ edges which go to members of S_k that are removed after it is. Putting these two facts together we have $\frac{3k}{4}|S_k| \leq \frac{k}{4}p + \frac{k}{4}|S_k|$, so $|S_k| \leq \frac{p}{2}$. Hence at most $\frac{3}{2}p$ vertices are removed. ∎

The lemma provides a way to find a subgraph with minimum degree $\frac{k}{4}$ in $O(n^{1/2}\log n)$ time assuming that the graph has a subgraph with minimum degree k. If $HDS_k(G)$ is empty, then the algorithm might terminate with an empty set of vertices. The algorithm is:

FindSubgraph1(V, k)
begin
  while at least $n^{1/2}$ vertices have degree less than k **do**
    remove vertices of degree less than k;
  while there is a vertex of degree less than $\frac{k}{4}$ **do**
    remove vertices of degree less than $\frac{k}{4}$;
end.

Each iteration of a loop takes $O(\log n)$ time. The first loop cannot be executed more than $n^{1/2}$ times since it removes at least $n^{1/2}$ vertices each iteration. Lemma 4.4 insures that the second loop does not remove more than $\frac{3}{2}n^{1/2}$ vertices, so it also has $O(n^{1/2})$ iterations.

This algorithm can be generalized to one that uses more than two phases of discarding vertices. In *FindSubgraph2*, the approximation factor and the runtime depend upon the function $f(n)$. The algorithm *FindSubgraph1* corresponds to *FindSubgraph2* with $f(n) =$

$n^{1/2}$. The algorithm *FindSubgraph2* maintains two counts, *bound* and *threshold*. A phase consists of discarding vertices with degree less than *bound*. As long as there are at least *threshold* vertices of degree less than *bound*, then vertices with degree less than *bound* are discarded. When the number of vertices of degree less than *bound* is smaller than *threshold*, the values of *bound* and *threshold* are changed. Phases are run until all vertices have degree at least *bound* or the graph becomes empty.

```
FindSubgraph2(V, k)
  begin
    threshold := n/f(n);
    bound := k;
    while V ≠ ∅ and there is a vertex of degree less than bound do
      begin
        S := {v | deg(v) < bound};
        if |S| ≥ threshold then
          V := V − S;
        else
          begin
            bound := bound/4;
            threshold := threshold/f(n);
          end
      end
  end.
```

Theorem 4.5. *If $HDS_k(G)$ is nonempty, the algorithm finds a subgraph with minimum degree at least $\frac{k}{4^{\lceil \log_{f(n)} n \rceil - 1}}$ in $O(f(n) \log n \log_{f(n)} n)$ time.*

Proof: A phase is the group of iterations for which *bound* and *threshold* have fixed values. There are $\log_{f(n)} n$ phases, since the value of *threshold* is reduced by a factor of $f(n)$ at the end of each phase and *threshold* is initially $\frac{n}{f(n)}$. When a phase ends, less than *threshold* vertices have degree less than *bound*, so the minimum degree of the subgraph that is found is the value of *bound* when *threshold* = 1, which is $\frac{k}{4^{\lceil \log_{f(n)} n \rceil - 1}}$. When a phase begins, there are less than $f(n) \cdot$ *threshold* vertices of degree less than $4 \cdot$ *bound*. It follows from Lemma 4.4 that at most $\frac{3}{2} f(n) \cdot$ *threshold* vertices are removed by removing vertices of degree less than *bound*. Since at least *threshold* vertices are removed at each iteration, there are at most $\frac{3}{2} f(n)$ iterations per phase. Hence there are at most $\frac{3}{2} f(n) \log_{f(n)} n$ iterations altogether. An iteration takes $O(\log n)$ time, so the algorithm takes $O(f(n) \log n \log_{f(n)} n)$ time. ∎

If different values are used for $f(n)$ an interesting time/performance trade off is exhibited. When $f(n) = \log n$ the algorithm is in $\mathcal{NC}$ and a subgraph with minimum degree $O(dn^{-\epsilon})$, for any $\epsilon > 0$ is found. Time and performance figures are given in the table below. The time is given as the total number of iterations, neglecting the factor of $\frac{3}{2}$.

	Iterations	Approximation Factor
$f(n)$	$f(n)\log_{f(n)} n$	$4^{\log_{f(n)} n - 1}$
$n^{1/k}$	$kn^{1/k}$	4^{k-1}
4^k	$\frac{4^k \log n}{2k}$	$\frac{1}{4}n^{1/k}$
$\log^k n$	$\frac{\log^{k+1} n}{k \log\log n}$	$\frac{1}{4}n^{2/k \log\log n}$
$2^{\sqrt{\log n}}$	$2^{\sqrt{\log n}}\sqrt{\log n}$	$\frac{1}{4}4^{\sqrt{\log n}}$

Time/performance relationship

4.5. Finding a Maximum Density Subgraph

An alternate approach in constructing a subgraph with high minimum degree is to look for a *maximum density* subgraph. The results that we get by this approach are stronger than the previous results for certain cases, although the resulting algorithms are probabilistic. The *density* of a graph is the ratio of the number of edges to the number of vertices, so a maximum density subgraph is an induced subgraph for which this ratio is as large as possible. We denote a maximum density subgraph of the graph G by $MD(G)$ and the maximum density by $\overline{MD}(G)$. Our first lemma shows that a maximum density subgraph has high degree.

Lemma 4.5. *A maximum density subgraph of G has all vertices of degree at least $\overline{MD}(G)$.*

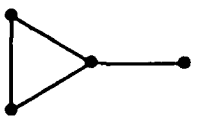
Proof: If the maximum density subgraph had a vertex of degree less than $\overline{MD}(G)$, then the density would be increased by deleting that vertex. ∎

The maximum density subgraph is related to the optimization problem discussed in the previous section. Computing the value of the maximum density subgraph gives an approximation of $\overline{HDS}(G)$ that is guaranteed to be within a factor of two.

Lemma 4.6. $2\overline{MD}(G) \geq \overline{HDS}(G) \geq \overline{MD}(G)$.

Proof: Let H be a subgraph of G with minimum degree $\overline{HDS}(G)$. Since H has at least $\frac{1}{2}|H|\,\overline{HDS}(G)$ edges, its density is at least $\frac{1}{2}\overline{HDS}(G)$. The density of H is no greater than the maximum density, so $2\overline{MD}(G) \geq \overline{HDS}(G)$. Lemma 4.5 implies that $\overline{HDS} \geq \overline{MD}$. ∎

The minimum degree of a maximum density subgraph can differ from $\overline{HDS}(G)$ by a factor of two. An example of such a graph is:



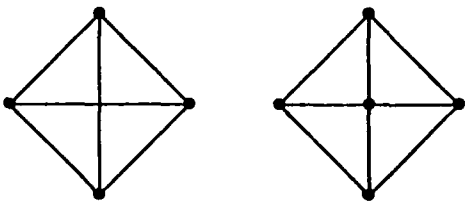
The value of $\overline{HDS}(G)$ is two, while the minimum degree of a maximum density subgraph is just one.

The maximum density subgraph can be constructed by an $\mathcal{RNC}$ algorithm. The algorithm for constructing a maximum density subgraph relies on a reduction due to Goldberg [Go], which reduces finding a maximum density subgraph to a unit capacity network flow problem. The flow problem is solved using the techniques of [KUW].

Theorem 4.6. *The problem of finding a maximum density subgraph can be reduced in log-space to the problem of finding a maximum flow in a unit capacity network.* ∎

Corollary 4.1. *A maximum density subgraph can be found in $\mathcal{RNC}$.* ∎

There is a slight difference in the types of approximations achieved by our two algorithms. The first approximation constructs a graph which is a supergraph of $HDS_k(G)$. The maximum density subgraph is an approximation to $HDS_d(G)$, where $d = \overline{HDS}(G)$. The maximum density subgraph is not necessarily a supergraph of $HDS_d(G)$. In the following example, $HDS_d(G)$ is the entire graph, while the maximum density subgraph is just the component on the right.



4.6. Number Problems

One class of problems where approximation is particularly important is number problems. A number problem is one in which arbitrary sized integers can be part of the problem. For example, the numbers could be weights of edges or coefficients of linear constraints. It is quite common for number problems to have natural approximations. For problems that have an objective function which is being opimized, an approximate solution is one that is close to the optimum. Number problems can be approximated in other ways as well. For example, an approximate solution to a packing problem could be one where the constraints are violated by a small amount. Some number problems can be solved efficiently if the numbers are small. Problems of this type can often be approximated by modifying the problem by truncating the values, solving the modified problem exactly, and then relating the solution of the modified problem to an approximate solution of the original problem. In this section we give a couple of examples of problems where this technique is used in parallel approximation algorithms.

The first problem that we look at is network flow. This problem illustrates how the difficulty in a problem can be caused by the presence of large numbers. The problem of determining a maximum flow in a network is P-complete [GSS]. The P-completeness proof uses large integers as capacities to encode the circuit value problem. If the capacities are bounded by a polynomial in the number of edges, then the problem can be solved in $\mathcal{RNC}$ [KUW]. We show that for every $\epsilon > 0$, a flow that differs by a factor of at most $1 + \epsilon$ from the maximum flow can be found in $\mathcal{RNC}$.

Theorem 4.7. *For any $\epsilon > 0$, maximum flow can be approximated to a factor of $1+\epsilon$ by an $\mathcal{RNC}$ algorithm.*

Proof: Let $G = (V, E)$, $|V| = n$, be an instance of network flow with edge capacities c_i for each $e_i \in E$. Suppose that the maximum flow has size f. We give an $\mathcal{RNC}$ algorithm that finds a flow of size $\hat{f}$ where $\left|f - \hat{f}\right| \leq \epsilon f$. The basic idea in the approximation algorithm is to truncate the values of the edge capacities so that they can be represented by numbers with $O(\log n + \log \frac{1}{\epsilon})$ bits. The modified flow problem can then be solved using the Karp-Upfal-Wigderson matching algorithm.

First we need bounds on the maximum flow f. Let c be the maximum capacity such that there is a path from s to t using only edges of capacity at least c. The network must have a cut that is made up of edges with capacity at most c, so $f \leq n^2 c$. Since there is a path from s to t with edges having capacity at least c, $c \leq f$. Since the flow is at most $n^2 c$, there is a maximum flow where no edge carries flow of more than $n^2 c$, so any edge with capacity greater than $n^2 c$ can be replaced by an edge with capacity exactly $n^2 c$.

Let $j = \max(0, \lfloor \log \frac{\epsilon c}{n^2} \rfloor)$. We now create a new network with capacities $\hat{c}_i = 2^j \lfloor c_i 2^{-j} \rfloor$. This is setting the j least significant bits of each capacity to 0. Since the largest capacity is at most cn^2, the number of significant bits in the modified capacities is at most $\lceil \log cn^2 \rceil - j \leq \lceil \log cn^2 \rceil - \lfloor \log \frac{\epsilon c}{n^2} \rfloor \leq 1 + 4 \log n + \log \frac{1}{\epsilon}$. The problem can then be solved in $O(\log^k n \log \frac{1}{\epsilon})$ time to get an approximate flow $\hat{f}$. The most that the approximate flow $\hat{f}$ could differ from the original flow f is $f - \hat{f} \leq \sum_i c_i - \hat{c}_i \leq n^2 \frac{\epsilon c}{n^2} = c\epsilon \leq \epsilon f$. Thus the algorithm achieves the desired approximation. ∎

A second number problem that can be approximated is list scheduling. This is a simple scheduling problem that involves scheduling jobs on two processors. The problem is:

> Given a list of jobs $j_1, \ldots, j_n$, each with an execution time $t(j_i) \in Z^+$, construct a two processor schedule such that j_i is started no later than j_{i+1} and there is no idle time until after all jobs have been started.

A list schedule can be computed by considering the jobs in order and assigning a job to the first processor that becomes available. The problem of computing a list schedule is P-complete [HM1]. The P-completeness proof requires the use of large numbers for the job times. The computation of a circuit is encoded by the times that the jobs are scheduled, with certain bits of the times corresponding to the values of the gates. However, the problem can be solved by a fast parallel algorithm if the numbers are small. Helmbold and Mayr [HM1] have shown that if the job times are bounded by $L(n)$, then a schedule can be computed in $O(\log L(n) \log n)$ time using $O(n^2)$ processors.

For list scheduling, we define an approximate solution to be a schedule that has the same first come, first served property as a list schedule, but we allow idle time between the jobs. The smaller the total idle time, the better the approximation. Using an $\mathcal{NC}$ algorithm to compute a list schedule for problems with small job times, we can construct an $\mathcal{NC}$ algorithm to approximate list scheduling with the idle time an arbitrarily small fraction of the schedule length.

Theorem 4.8. *For all $\epsilon > 0$, list scheduling can be approximated by an $\mathcal{NC}$ algorithm such that the proportion of the idle time is less than ϵ.*

Proof: The algorithm for approximating a list schedule rounds the job times up so that they are multiples of 2^k with $O(\log n)$ significant bits and then solves the modified problem exactly. Let $j_1, \ldots, j_n$ be a list of jobs with job times $t(j_i)$, and let $\epsilon > 0$. Let t be the largest of the job times. If $t \leq \frac{n}{\epsilon}$, the problem can be solved exactly in $\mathcal{NC}$, so suppose $t > \frac{n}{\epsilon}$. Suppose $2^k \leq \frac{t\epsilon}{n} < 2^{k+1}$. We create a modified problem with job times $\hat{t}(j_i) = 2^k \lceil t(j_i) 2^{-k} \rceil$. The modified problem can be solved exactly, and then the jobs of the original problem can be scheduled at the times of the modified problem. The idle time after job j_i is run is at most $\hat{t}(j_i) - t(j_i) < 2^k$. The total idle time is at most $n2^k \leq n\frac{t\epsilon}{n} = t\epsilon$. Since the length of the schedule is at least t, the proportion of idle time is at most ϵ. ∎

The approximation algorithms for network flow and list scheduling both depend on being able to solve the problems efficiently when the numbers are small. However, some problems remain difficult when the numbers are small. In the theory of NP-completeness, a distinction is made between *weakly* and *strongly* NP-complete problems in order to classify number problems. If an NP-complete problem remains NP-complete when restricted to instances of the problem involving only small numbers, it is NP-complete in the strong sense, while if it can be solved in polynomial time when the numbers are small, the problem is NP-complete in the weak sense. We can make the same distinction for P-complete problems.

Definition 4.1. *A P-complete problem is strongly P-complete if there exists a polynomial p such that the problem remains P-complete when restricted to instances I with the largest number bounded by $p(|I|)$.*

A P-complete problem that is not a number problem, such as the circuit value problem is P-complete in the strong sense, so the distinction is only interesting for number problems. An important example of a number problem that is P-complete in the strong sense is linear programming. Linear programming is a number problem since the coefficients of the equations can be arbitrary integers. Cook has shown (see [HR]), that the problem of determining if a set of linear inequalities has a solution is P-complete. This problem remains P-complete when the coefficients are restricted to $+1$ and -1, so linear programming is P-complete in the strong sense. In light of this result, a fast parallel approximation for linear programming is unlikely. In the next section we present another strongly P-complete problem.

4.7. First Fit Bin Packing

In this section and the next, we examine the problem of computing a first fit decreasing (FFD) bin packing. We show that the problem is strongly P-complete, and that the problem can be approximated in a reasonable sense. The bin packing problem is:

> Given a list of items $U = u_1, \ldots, u_n$ with sizes $s(u_i) < 1$ for $u_i \in U$, find an assignment of the items to unit capacity bins such that the number of bins used

is as small as possible. The sum of the sizes of the items assigned to a bin must be at most one.

For ease of exposition, we refer to $s(u_i)$ just as u_i. Bin packing is a well known NP-complete problem [GJ3]. Sequential approximation schemes to bin packing have been developed and extensively analyzed [GJ3]. An important approximation algorithm for bin packing is the first fit algorithm. First fit considers the items one at a time, and places each item in the first bin with enough room. If the list is sorted so that the items are non-increasing, then the algorithm is first fit decreasing (FFD), and if the list is sorted so that the items are non-decreasing, the algorithm is first fit increasing (FFI). An FFI packing is within $\frac{17}{11}$ of optimal and an FFD packing is within $\frac{11}{9}$ of optimal.

The result of this section is that it is P-complete in the strong sense to compute an FFD packing. The problem of computing an FFD packing is a number problem, since the items may have arbitrary sizes. However, our result shows that this problem is difficult even if the item sizes are "small." Many P-complete number problems, such as network flow and list scheduling are only P-complete in the weak sense. This is one of the first results that shows a number problem to be P-complete in the strong sense. The result shows that the source of the difficulty in computing an FFD packing is from the arrangement of items in the bins as opposed to being from the numbers involved in the problem.

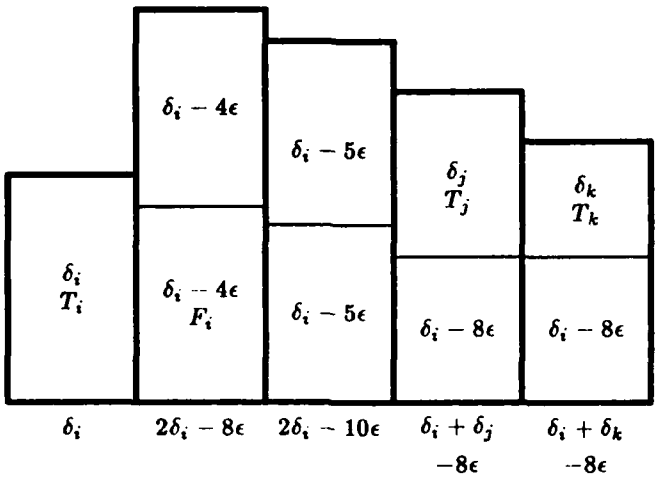
The strong P-completeness result suggests that a scaling approach is not likely to lead to a good approximation to an FFD packing. However, in the next section we show that a reasonable approximation to an FFD packing can be computed by an $\mathcal{NC}$ algorithm. We show that if the item sizes are bounded below by a constant, then the FFD packing can be computed in $O(\log n)$ time. This gives us an $\mathcal{NC}$ algorithm to compute a packing that obeys the same performance bound as an FFD packing.

Theorem 4.9. *The problem of computing a first fit decreasing packing is strongly P-complete.*

Proof: The proof is a reduction from the monotone circuit value problem. The reduction has two stages. The first stage reduces the monotone circuit value problem to computing an FFD packing into bins of variable size (the sizes of the bins are specified as part of the instance). The second stage reduces computing an FFD packing into bins of variable size to computing an FFD packing into unit capacity bins.

Let $\beta = \beta_1, \ldots, \beta_n$ be a monotone circuit. We transform β into a non-increasing list of items and a list of bins. There is a distinguished item $\hat{u}$ and a distinguished bin $\hat{b}$. The item $\hat{u}$ is placed in $\hat{b}$ by a first fit packing if and only if the output of the circuit is *true*.

For each gate there is a list of items and a list of bins. The items and bins are ordered by gate number, so if $i < j$, the items and bins for gate β_i come before the items and bins for gate β_j. Among the items for gate β_i are two pairs of items, T_i, T_i and F_i, F_i which indicate the values of the inputs to the gate. Exactly two of these items are placed by the first fit packing in the bins for β_i, the other two are packed in bins for lower numbered gates. The two that are placed in the bins for β_i give the value for the inputs to the gate. If the output of the gate β_i is connected to the gates β_j and β_k, the bins for β_i get either T_j, T_k or F_j, F_k, depending upon the value of the gate. If T_j and T_k are placed in the bins for β_i, then the gates β_j and β_k receive *false* values for β_i.



Packing for AND gate β_i with one *true* input and one *false* input

Suppose there are n gates in the circuit. Let $\delta_i = 1 - \frac{i}{n+1}$ and $\epsilon = \frac{1}{9(n+1)}$. The items for gate β_i have sizes:

$$\delta_i, \quad \delta_i, \quad \delta_i - 4\epsilon, \quad \delta_i - 4\epsilon, \quad \delta_i - 4\epsilon, \quad \delta_i - 5\epsilon, \quad \delta_i - 5\epsilon, \quad \delta_i - 8\epsilon, \quad \delta_i - 8\epsilon.$$

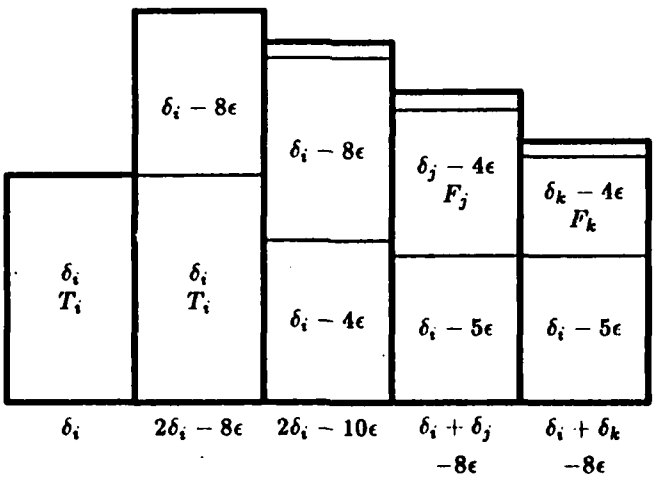
The first four items are T_i, T_i, F_i, and F_i respectively. The list of items is non-increasing. The bins for an AND gate β_i with outputs to β_j and β_k have sizes:

$$\delta_i, \quad 2\delta_i - 8\epsilon, \quad 2\delta_i - 10\epsilon, \quad \delta_i + \delta_j - 8\epsilon, \quad \delta_i + \delta_k - 8\epsilon.$$

The bins for an OR gate β_i with outputs to β_j and β_k have sizes:

$$2\delta_i - 8\epsilon, \quad \delta_i, \quad 2\delta_i - 10\epsilon, \quad \delta_i + \delta_j - 8\epsilon, \quad \delta_i + \delta_k - 8\epsilon.$$

The first two bins evaluate the value of the gate and the last three bins propagate the value. Packings of the AND gate for the inputs TT and TF are illustrated in the figure below. The OR gate is similar. For gates β_i that have a constant input, either a T_i or a F_i is deleted to give the gate the right input.



Packing for AND gate β_i with two *true* inputs

To finish the proof, we give a reduction of FFD packing with variable bin sizes to an FFD packing with unit capacity bins. Let $u_1, \ldots, u_q$ be a list of items and $b_1, \ldots, b_r$ be a list of bins. We construct a list of decreasing items $v_1, \ldots, v_{2r}$ which when packed into bins of size C leave space b_i in the i-th bin. The packing of $u_1, \ldots, u_q$ into $b_1, \ldots, b_r$ is transformed into packing $v_1, \ldots, v_{2r}, u_1, \ldots, u_q$ into bins of size C. The sizes can then be normalized to give a packing into unit capacity bins.

Let b be the largest of the b_i's and $C = (2r + 1)b$. Without loss of generality we assume that all the b_i's have size greater than 0. The sizes of the items are:

$$v_i = \begin{cases} C - ib - b_i, & \text{if } i \leq r; \\ C - ib, & \text{if } i > r. \end{cases}$$

The items v_i and v_{2r+1-i} are put into the i-th bin, leaving b_i empty space. The list $v_1, \ldots, v_{2r}$ is non-increasing.

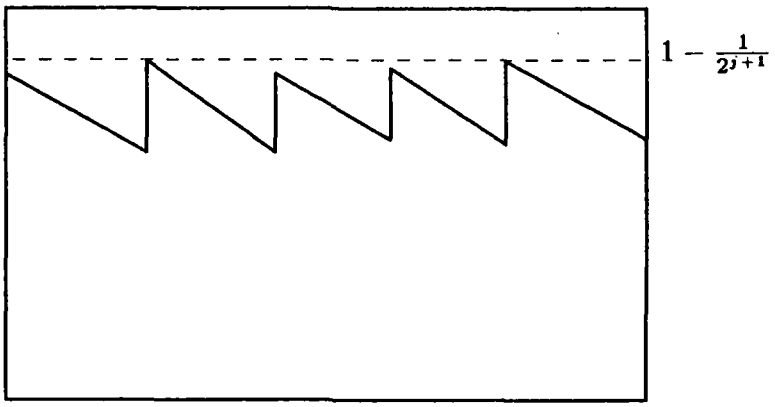
The largest number involved in the reduction is polynomial in the size of the circuit, so the proof shows that it is strongly P-complete to compute an FFD packing. ∎

4.8. Approximating an FFD packing

Although it is probably not possible to compute a first fit decreasing packing in $\mathcal{NC}$, we can approximate a first fit decreasing packing in a reasonable sense. We show that if the sizes of all of the items are at least $\frac{1}{k}$, then an FFD packing can be computed in $\mathcal{NC}$. Thus we can find a packing that agrees with FFD on all of the big items. As a corollary to our result, we show how to construct a packing in $\mathcal{NC}$ that obeys the same performance bound as FFD.

Our algorithm for computing an FFD packing of items of size at least $\frac{1}{k}$ is relatively simple. The basic idea is to decompose the problem into a series of packing problems that have a simple structure. We show that the number of subproblems is independent of the number of items in the list, although it does depend upon k.

The FFD algorithm is broken into phases, with the j-th phase packing all of the items in the interval $(\frac{1}{2^j}, \frac{1}{2^{j+1}})$. The bins which are filled to at least $1 - \frac{1}{2^{j+1}}$ cannot be of use in the j-th phase, so they are ignored for the phase. The bins which are filled to less than $1 - \frac{1}{2^{j+1}}$ consist of groups of consecutive bins in which the amount of remaining space is increasing, as is illustrated in the figure below. We prove that the number of groups is bounded by a constant depending on j but not on the number of items. The packing into groups is done sequentially, first by packing into the first group, then into the second group, and so on. The routine *Pack* computes the packing of items of size $(\frac{1}{2^j}, \frac{1}{2^{j+1}})$ into a list of increasing sized bins in $O(\log n)$ time. The details of the routine *Pack* are described below. The algorithm *FFD* runs in $O(\log n)$ time since the number of phases and the number of calls to *Pack* is constant.

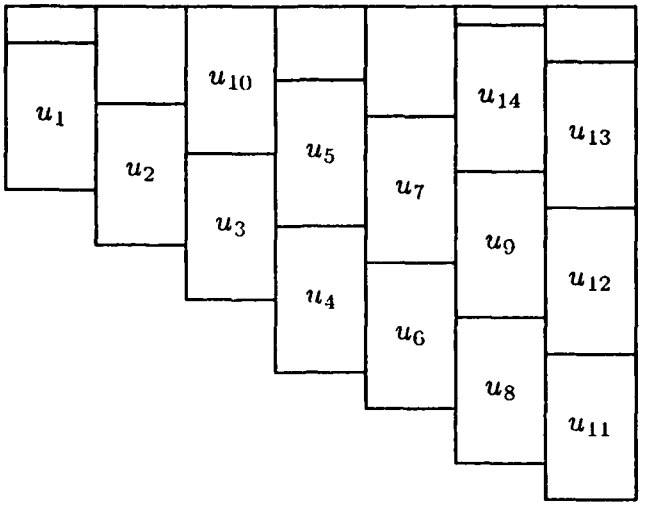


Bins filled to at most $1 - \frac{1}{2^{j+1}}$

```
FFD(U = u_1, ..., u_n);
  begin
    for j ← 1 to k − 1 do
      begin
        U_j ← {u_i ∈ U | 1/2^{j+1} < u_i ≤ 1/2^j};
        Let B' be the sublist of bins filled to less than 1 − 1/2^{j+1};
        Divide B' into groups of consecutive bins such that the empty space in each group is
          increasing.
        for each group G do
          Pack(U_j, G);
      end
  end.
```

The major subroutine of our bin packing algorithm is to pack a decreasing list of items $U = u_1, \ldots, u_n$ with $u_i \in [\frac{1}{2^j}, \frac{1}{2^{j+1}})$ into an increasing list of bins. The reason that this packing problem is easy to solve is that the resulting packing has a simple structure. The number of items placed in the bins increases with bin number. The routine consists of a number of phases. The i-th phase packs consecutive bins that can fit i items each. The items are taken from the start of the list and packed i per bin until a bin is encountered that fits $i+1$ items. Some of the bins that received i items can accommodate one additional item from further down the list. These items are added and then the next phase is run using the remaining items and the remaining bins. In the example below, the items $u_4, \ldots, u_9$ are placed in consecutive bins by the second phase, and the item u_{14} fits in the bin containing u_8 and u_9. When packing items in the range $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$, there are fewer than 2^{j+1} phases since no more than $2^{j+1} - 1$ items can be placed in a bin. A phase can easily be implemented to run in $O(\log n)$ time.

```
Pack(U = u_1, ..., u_n, B = b_1, ..., b_m)
  begin
    for i := 1 to k do
      begin
        Find s such that u_{(r−1)i+1} + ··· + u_{ri} ≤ b_r < u_{(r−1)i+1} + ··· + u_{ri+1} for r ≤ s and
          u_{si+1} + ··· + u_{(s+1)i+1} ≤ b_{s+1}.
        Place the items u_1, ..., u_{si} into the bins b_1, ..., b_s.
```



An example of a packing performed by *Pack*

for each item u_j, $j > si + 1$
Find the first bin b_r, $r \leq s$ that has enough space remaining for u_j.
Add the additional items to the bins $b_1, \ldots, b_s$ by placing items in the first possible bin, pushing overflow items towards the higher numbered bins.
Remove the placed items from U and the filled bins from B and renumber the lists.
end
end.

The following lemma establishes that *Pack* computes a first fit packing.

Lemma 4.7. *Let $U = u_1, \ldots, u_n$ be a decreasing list of items with $u_i \in [\frac{1}{2^j}, \frac{1}{2^{j-1}})$ and $B = b_1, \ldots, b_m$ be an increasing list of bins with $b_m \leq 1$. The routine Pack computes a first fit decreasing packing of U into B.*

Proof: The potential place for the algorithm to be incorrect is in the placement of the s groups of i items each in the first s bins of B. Suppose an item u' that is placed in a bin b_r, $r \leq s$ actually fit in a bin $b_{r'}$, $r' < r$. So $u_{r'i+1} + \cdots + u_{(r'+1)i} + u' \leq b_{r'}$. But, $u_{ri+1} + \cdots + u_{r(i+1)} + u_{r(i+1)+1} \leq u_{r'i+1} + \cdots + u_{(r'+1)i} + u' \leq b_{r'} \leq b_r$. Hence, $r > s$. ∎

In order to prove that our algorithm for constructing an FFD packing of items of size at least $\frac{1}{k}$ is a fast parallel algorithm, we must show that the number of groups of bins encountered in the algorithm is bounded by a constant. We prove a slightly stronger result than we need by showing that the constant is in fact only polynomial in k. Thus our algorithm remains an $\mathcal{NC}$ algorithm even if k is some slowly growing function in n, such as $\log n$.

We must now show that the number of groups of consecutive bins that the algorithm considers when packing items of size greater than $\frac{1}{k}$ is bounded by a constant C_k. Our proof shows that this constant is $O(k^4)$. We prove the theorem by keeping track of the number of intervals of bins of increasing size at the start of each phase. We can bound the increase in the number of intervals by considering in some detail the way items are packed into the bins.

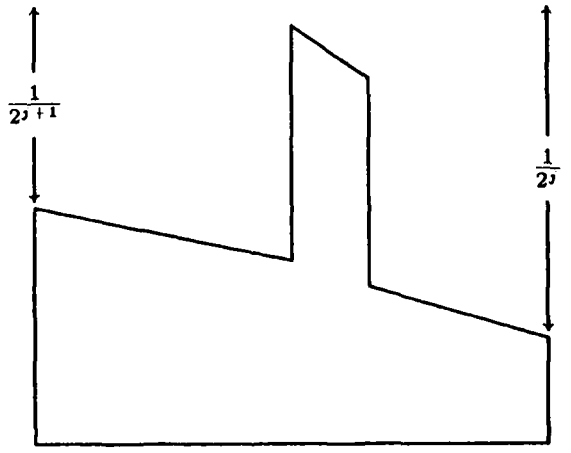
Let $U = u_1, \ldots, u_n$ be the list of items and $B = b_1, \ldots, b_m$ be the list of bins. We denote the amount of space left in bin b_i at a given time by $s(b_i)$. This value depends upon the phase of the packing. The projection of a packing onto a subset B' of the bins is the set of items U' that are placed into the bins B'. It is a folk theorem of bin packing that U' is packed into B' by a first fit packing in exactly the same manner as the items U' are packed in the full packing. A $\frac{1}{2^j}$-projection is the projection of the packing onto the set of bins b_i such that $s(b_i) > \frac{1}{2^j}$. In the j-th phase, when we pack items in the interval $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$, we only consider the $\frac{1}{2^{j+1}}$-projection of the packing up to that point.

The major portion of this proof is accounting for groups of bins with increasing space. Basically, an *interval* is a set of consecutive bins with remaining space increasing. However, for bookkeeping reasons, we break up the intervals at powers of two, so all bins of an interval have space in $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$ for some j. We also relax the condition that the bins are consecutive; we allow some bins with less space to be between the bins of the interval. Finally, we insist that our intervals are maximal, so that bins cannot be added to an interval without violating one of the defining properties.

Definition 4.2. *A $\frac{1}{2^j}$-interval is a set of bins $b_{i_1}, \ldots, b_{i_r}$ such that:*

1. $\frac{1}{2^{j+1}} < s(b_{i_1}) \leq s(b_{i_2}) \leq \cdots \leq s(b_{i_r}) \leq \frac{1}{2^j}$.
2. *Let $\hat{b}_1, \ldots, \hat{b}_m$ be the bins of the $\frac{1}{2^{j+1}}$-projection of the packing. The image of $b_{i_1}, \ldots, b_{i_r}$ is a set of consecutive bins $\hat{b}_s, \ldots, \hat{b}_{s+r-1}$.*
3. $s = 1$ *or* $s(\hat{b}_{s-1}) > s(\hat{b}_s)$.
4. $s + r - 1 = m$ *or* $s(\hat{b}_{s+r}) > \frac{1}{2^j}$ *or* $s(\hat{b}_{s+r}) < s(\hat{b}_{s+r-1})$.

An important detail in our definition of an interval is that we allow intermediate bins to have less space than the bins of the interval. The figure below shows a situation that might occur. The $\frac{1}{2^j}$ interval is separated by an interval of bins with much less space. The reason that we allow this in our definition, is that we do not want to subdivide the bins too early, or else we generate too many intervals.



Each $\frac{1}{2^j}$-interval is assigned a weight of 2^{-2j}. We show that the sum of the weights of the intervals is $O(4^j)$ at the start of phase j. This allows us to bound the number of groups of consecutive bins that we consider in the algorithm.

We begin by examining the packing of items in the range $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$ into a set of consecutive bins with increasing space in the range $[\frac{1}{2^l}, \frac{1}{2^{l+1}})$. Here, we assume that the bins are fully packed, meaning that after the items are placed, none of the bins have more than $\frac{1}{2^j}$ space. This is the type of packing that is performed by the routine *Pack* described above.

The items that are packed can be divided into two types of items, the *forward* items and the *fill-in* items. The routine *Pack* places items r per bin from the front of the list until a bin is encountered that can hold at least $r+1$ items, then the routine looks ahead in the list of items and finds additional items that can fit with the items placed r per bin. The items placed in groups from the front of the list are the *forward* items, and the items placed by looking ahead in the list are the *fill-in* items.

If we just look at the forward items, then the bins are divided into runs of bins with increasing space. The set of bins that receive the same number of items forms a run. The number of runs formed is at most $\frac{2^{j+1}}{2^l} - \frac{2^j}{2^{l+1}} = \frac{3}{2}\frac{2^j}{2^l}$. The amount of space left in a bin is at most $\frac{1}{2^j}$, so the weight of each run is less than $\sum_{i \geq j} 2^{-2j} = \frac{4}{3}2^{-2j}$. The total weight is less than $\frac{3}{2}\frac{2^j}{2^l}\frac{4}{3}2^{-2j} = \frac{2}{2^l 2^j} \leq 2^{-2l}$ if $j > l$. Thus the weight is not increased by the forward packing.

We now consider packing items of size $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$ into an increasing list of bins of size $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$. This covers the fill-in items.

Lemma 4.8. *The number of $\frac{1}{2^i}$-intervals generated by a packing of items of size $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$ into an interval of size $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$ is at most $\frac{2^i}{2^j}$.*

Proof: Let $\hat{b}_1, \ldots, \hat{b}_m$ be the $\frac{1}{2^{i+1}}$-projection of the packing. A *reversal* occurs when an item u_{i+1} is placed in a lower numbered bin than the previous item u_i. Each $\frac{1}{2^i}$-interval (except possibly the first one formed) begins with a reversal. Let u_{i+1} be the first item of a $\frac{1}{2^i}$-interval, and suppose it was placed in the bin $\hat{b}_s$. There is at least $\frac{1}{2^{i+1}}$ space left in $\hat{b}_s$ after u_{i+1} is placed. Since u_i did not fit in $\hat{b}_s$, $u_{i+1} + \frac{1}{2^{i+1}} < u_i$. Thus, each $\frac{1}{2^{i+1}}$-interval other than the first one accounts for a gap of more than $\frac{1}{2^{i+1}}$ in the item sizes. Since the difference between the first and the last item is at most $\frac{1}{2^{j+1}}$, the number of $\frac{1}{2^i}$-intervals is at most $\frac{2^i}{2^j}$. ∎

The weight of the resulting packing is bounded by $\sum_{i \geq j} \frac{2^i}{2^j} 2^{-2i} = 2 \cdot 2^{-2j}$. The weight is at most double the weight of the original interval.

The j-th phase of the algorithm packs the items in the range $[\frac{1}{2^j}, \frac{1}{2^{j+1}})$, so for the j-th phase, we can neglect all of the bins with space at most $\frac{1}{2^{j+1}}$. Suppose the weight at the start of phase j is w_j. Since the $\frac{1}{2^j}$-intervals have weight 2^{-2j}, there are at most $2^{2j}w_j$ intervals to consider. However, since these intervals might overlap, some of them may have to be split. If a $\frac{1}{2^i}$-interval encloses a $\frac{1}{2^l}$-interval, $(i < l \leq j)$, then we split the outer interval. The number of intervals that are added is bounded by the original number of intervals. At most $2 \cdot 2^{2j}w_j$ intervals need to be considered in packing the j-th phase.

The packing of the *forward* items in the $\frac{1}{2^l}$-intervals $(l < j)$ does not increase the total weight, except when a bin is only partially filled. At most one bin is partially filled by a phase, so this adds at most a constant to the weight. (The constant is in fact at

most $\frac{5}{4}$.) In computing the weight, the forward packing can be considered before the splitting of intervals, since the forward packing cost does not depend on the bins being consecutive. The splitting of intervals can double the number of $\frac{1}{2^j}$-intervals, so this can double the weight. The weight is again doubled by packing into the $\frac{1}{2^j}$-intervals. Hence $w_{j+1} \leq 4w_j + \frac{5}{4}$. Since $w_0 = 1$, we have $w_j \leq \frac{5}{3} \cdot 4^j$.

When the items packed all have size at least $\frac{1}{k}$, there are at most $\log k$ phases. The weight is then $O(k^2)$ at the end of the algorithm, so the number of groups of bins considered in the final phase is $O(k^4)$. Thus, we have the following theorem:

Theorem 4.10. *The algorithm FFD computes a first fit decreasing packing of items of size greater than $\frac{1}{k}$ in $O(\log n)$ time.* ∎

We can use our algorithm to construct a packing that obeys the same performance bound as FFD. The algorithm for doing this combines a first fit decreasing packing and a first fit increasing packing. The algorithm first packs all items that are in the interval $[1, \frac{1}{6})$ using a first fit decreasing packing, and then packs the remaining items using a first fit increasing packing. Let $L_D(I)$ be the length of the first fit decreasing packing and $OPT(I)$ be the optimal packing for a list I of items. First we show that this packing is relatively close to optimal.

Theorem 4.10. *The length of the composite packing $L_C(I)$ satisfies*

$$L_C(I) \leq \max(L_D(I), \frac{6}{5}OPT(I) + 1) \leq \frac{11}{9}OPT(I) + 4$$

Proof: Let L be the length of the FFD packing of the items with size greater than $\frac{1}{6}$. Clearly $L \leq L_D(I)$, so if all the items are placed in the first L bins, then $L_C(I) \leq L_D(I)$. If more than L bins are used, then all bins except for possibly the last one is filled to at least $\frac{5}{6}$, so $L_C(I) \leq \frac{6}{5}OPT(L) + 1$. ∎

The first fit increasing part of the packing can be done fairly easily with a fast parallel algorithm. The following lemma sketches how an FFI packing is computed.

Lemma4.9. *A first fit increasing packing into variable sized bins can be computed in NC.*

Proof: The property that an FFI packing has that makes it easy to compute with a fast parallel algorithm is that the order of the items in the bins is the same as the initial order of the items in the list. The key part of computing the FFI packing is to identify the first item placed in each bin. One way this can be done is to compute for each bin b_j and each item u_i, the first item available for bin b_{j+1} assuming that u_i is the first item available for bin b_j. The first item for each bin can then be computed using path doubling. This algorithm can be implemented to run in $O(\log n)$ time using $O(n^2)$ processors. ∎

The composite packing is computed by partitioning the items into the items of size at least $\frac{1}{6}$ and the items of size less than $\frac{1}{6}$. The first group is packed using the FFD algorithm, and the second group is packed using FFI. A similar parallel algorithm for computing a packing that obeys the same performance bound as FFD has been independently discovered by Warmuth [War].

4.9. Discussion

The results on approximating P-complete problems are similar to approximation results for NP-complete problems. Our results in this chapter illustrate a number of points of similarity. Our result on approximating the high degree subgraph problem shows that it is possible to get tight bounds on the degree of approximation that is feasible by an $\mathcal{NC}$ algorithm assuming that $\mathcal{P} \neq \mathcal{NC}$. This result parallels a number of results on approximating NP-complete problems with parallel algorithms. Some P-complete number problems can be solved in $\mathcal{NC}$ when the numbers involved are small. Efficient parallel approximation schemes can often be found for problems of this type. Other number problems remain difficult when they are restricted to instances that involve small numbers. The notion of strong P-completeness captures this, being analogous to strong NP-completeness. The problem of computing an FFD bin packing is strongly P-complete. This problem can still be approximated in a reasonable sense, since an FFD packing can be computed in $\mathcal{NC}$ if the sizes of the items are bounded below by a constant.

References

[AHU] Aho, A. V., Hopcroft, J. E., Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[B] Borodin, A., "On relating time and space to size and depth," *SIAM Journal of Computing*, **6**, 1977, pp. 733–744.

[CKS] Chandra, A. K., Kozen, D. C., Stockmeyer, L. J., "Alternation," *JACM*, **28**, 1, January 1981, pp. 114–133.

[CSV] Chandra, A. K., Stockmeyer, L. J., Vishkin, U., "Constant depth reducibility," IBM Research Report RC9548, August 1982.

[C1] Cook, S. A., "Towards a complexity theory of synchronous parallel computation," *L'Enseignement Mathematique XXVII*, 1981, pp. 99–124.

[C2] Cook, S. A., "The classification of problems which have fast parallel algorithms," Technical Report No. 164/83, Department of Computer Science, University of Toronto 1983.

[CD] Cook, S., Dwork, C. "Bounds on the time for parallel RAM's to compute simple functions," *Proc. 14th STOC*, 1982, pp. 231–233.

[DLR] Dobkin, D., Lipton, R. J., Reiss, S., "Linear programming is log-space hard for P," *Information Processing Letters*, 8 1979, pp. 96–97.

[DKM] Dwork, C., Kanellakis, P. C., Mitchell, J. C., "On the sequential nature of unification," Manuscript, 1983.

[DC] Dymond, P. W., Cook, S. A., "Hardware complexity and parallel computation," *Proc. 21st FOCS*, 1980, pp. 360–372.

[ET] Even, S., Tarjan, R. E., "Network flow and testing graph connectivity," *SIAM Journal of Computing*, 4, no. 4 1975, pp. 507–518.

[E] Erdős, P., "On the structure of linear graphs," *Israel Journal of Mathematics*, **1** 1963, pp. 156–160.

[FRW] Fich, F. E., Ragde, P. L., Wigderson, A., "Relations between concurrent-write models of parallel computation," *Proc. 3rd PODC*, 1984, pp. 179–189.

[FW] Fortune, S., Wyllie, J., "Parallelism in random access machines," *Proc. 10th ACM STOC*, 1978, pp. 114–118.

[GJ1] Garey, M. R., Johnson, D. S., "The complexity of near-optimal graph coloring," *JACM*, **23**, 1, 1976, pp. 43–49.

[GJ2] Garey, M. R., Johnson, D. S., "Strong NP-completeness results: motivations, examples, and implications" *JACM*, **25**, 3, 1978, pp. 230–237.

[GJ3] Garey, M., Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, H. Freeman, San Francisco, 1978.

[Go] Goldberg, A. V., "Finding a maximum density subgraph," Technical Report UCB/CSD 84/171, Computer Science Division, University of California, Berkeley, May 1984.

[G1] Goldschlager, L. M., "The monotone and planar circuit value problems are log space complete for P," *SIGACT News*, **9**, 2 1977, pp. 25–29.

[G2] Goldschlager, L. M., "A unified approach to models of synchronous parallel machines," *Proc. 10th STOC*, 1978, pp. 89–94.

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

[HM1] Helmbold, D., Mayr, E., "Fast scheduling algorithms on parallel computers," Technical Report No. STAN-CS-84-1025, Computer Science Department, Stanford University, November 1984.

[HM2] Helmbold, D., Mayr, E., "Two processor scheduling is in NC," Manuscript, 1985.

[HR] Hoover, H. J., Ruzzo, W. L., "A compendium of problems complete for P," Manuscript, October 1984.

[HK] Hopcroft, J. E., Karp, R. M., "A $n^{5/2}$ algorithm for maximum matching in bipartite graphs," *SIAM Journal of Computing*, **2**, 1973, pp. 225–231.

[JV] Johnson, J. B., Venkatesan, S. M., "Parallel algorithms for minimum cuts and maximum flows in planar networks," *Proc. 23rd FOCS*, 1982, pp. 244–254.

[KSS] Karloff, H., Shmoys, D., Soroker, D., "Efficient parallel algorithms for graph coloring and partitioning problems," Manuscript, 1984.

[KUW] Karp, R. M., Upfal, E., Wigderson, A., "Constructing a maximum matching is in random NC," *Proc. 17th STOC*, 1985, pp. 22–31.

[KW] Karp, R. M., Wigderson, A., "A fast parallel algorithm for the maximal independent set problem," *Proc. 16th STOC*, 1984, pp. 266–272.

[KL] Kernigan, B. W., Lin, S., "A heuristic algorithm for the travelling salesman problem," *Operations Research*, **21**, 1973, pp. 498–516.

[K] Kruskal, J. B., "On the shortest spanning subtree of a graph and the travelling salesman problem," *Proceedings fo the American Mathematical Society*, **7**, 1956, pp. 48–50.

[Lad] Ladner, R. E., "The circuit value problem is log space complete for P," *SIGACT News*, **7**, 1 1975, pp. 583–590.

[Law] Lawler, E. L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[LR] Lenstra, J. K., Rinooy Kan, A. H. G., "Complexity of scheduling under precedence constraints," *Operations Research*, **26**, 1978, pp. 22–35.

[Lu] Luby, M., "A simple parallel algorithm for the maximal independent set problem," *Proc. 17th STOC*, 1985, pp. 1–10.

[Lue] Lueker, G. S., Private communication.

[MV] Micali, S., Vazirani, V. V., "An $O(\sqrt{|V|}\,|E|)$ algorithm for finding a maximum matching in general graphs," *Proc. 21st FOCS*, 1980, pp. 17–27.

[PS] Papadimitriou, C. H., Steiglitz, K. *Combinatorial Optimization*, Prentice-Hall, Englewood Cliffs, N.J.

[P] Pippenger, N., "On simultaneous resource bounds," *Proc. 20th FOCS*, 1979, pp. 307–311.

[Re] Reif, J., "Depth first search is inherently sequential," Aiken Computation Lab Technical Report TR-27-83, November 1983.

[Ru] Ruzzo, W. L., "On uniform circuit complexity," *Journal of Computer and System Sciences*, **22**, 1981, pp. 365–383.

[Sc] Schwartz, J. T., "Ultracomputers," *ACM TOPLAS*, **2**, 4, 1980, pp. 484–521.

[Sm] Smith, J. R, "Parallel algorithms for depth-first searches: I. Planar graphs," To appear, *SIAM Journal of Computing*.

[T] Tarjan, R. E., "Depth-first search and linear graph algorithms," *SIAM Journal of Computing*, **1**, 1972, pp. 146–160.

[TV] Tarjan, R. E., Vishkin, U., "Finding biconnected components and computing tree functions in logarithmic parallel time," *Proc. 25th FOCS*, 1984, pp. 12–20.

[VSBR] Valiant, L. G., Skyum, S., Berkowitz, S., Rackoff, C., "Fast parallel computation of polynomials using few processors," Aiken Computation Lab Technical Report TR-17-82, May 1982.

[V] Vishkin, U., "An optimal parallel connectivity algorithm," RC 9149, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., 1981.

[Wag] Wagner, H. M., "On A class of capacitated transportation problems," *Management Science*, 5, 1959, pp. 304–318 (cited by [PS]).

[War] Warmuth, M., "Parallel approximation algorithms for one-dimensional bin packing," Manuscript, 1984.

[Wy] Wyllie, J. C., *The Complexity of Parallel Computations*, Phd. Thesis, Department of Computer Science, Cornell University, 1979.

END

11-86

DTIC