Approximation through multicommodity flow

Philip Klein*† Ajit Agrawal† R. Ravi† Satish Rao‡

Abstract

In this paper, we prove the first approximate max-flow min-cut theorem for general multicommodity flow. We use it to get approximation algorithms for minimum deletion of clauses of a 2-CNF≡ formula, via minimization, and other problems. We also present approximation algorithms for chordalization of a graph and for register sufficiency, based on undirected and directed node separators .

1 Introduction

Max-flow min-cut theorems

The amount of flow one can push through a network from a source to a sink clearly cannot exceed the capacity of a cut separating them, and the celebrated max-flow min-cut theorem of Ford and Fulkerson [3] and of Elias, Feinstein and Shannon [2] showed that the capacity upper bound is always tight. The value of the maximum flow is equal to the capacity of the minimum cut. This result yielded as a byproduct an algorithm for finding a minimum capacity cut.

Ever since, researchers have sought to generalize the max-flow min-cut theorem to apply to cases of *multicommodity* flow, in which each of several commodities has its own source and sink. In 1963, Hu showed [11] that such a theorem held for 2-commodity flow. In subsequent work, various special cases have been identified for which the max-flow min-cut theorem holds. As was demonstrated fairly early, however, no such theorem can hold in general for all multicommodity flow instances.

Leighton and Rao [16] recently defined the notion of an *approximate* max-flow min-cut theorem, by showing that the capacity upper bound is within a logarithmic factor of being tight for a special class of multicommodity flow instances, called *uniform* multicommodity flow. In a uniform flow problem for a graph G, a flow of value 1 is required between every two nodes of G. Their approximate max-flow min-cut theorem also yields an algorithm for finding an approximately *sparsest cut*, which in turn is the basis for a variety of approximation algorithms for NP-complete graph problems.

Our approximate max-flow min-cut theorem for arbitrary multicommodity flow

In this paper, we have gone a step further. We have proved an approximate max-flow min-cut theorem that holds for *all* multicommodity flow problems. Our theorem is the first for arbitrary multicommodity flow.

Theorem 1.1 *Consider a multicommodity flow problem where the sum of the demands is D and the sum of the capacities is C (and demands and capacities are integral). In order for there to exist a feasible flow, it is sufficient that every cut's capacity exceeds its demand by a factor of $O(\log C \log D)$.*

Moreover, our theorem yields an algorithm for finding a cut that is (approximately) minimum *relative* to the flow that must cross it. By appropriate choices of source-sink pairs, one can use this algorithm to find cuts that are (approximately) minimum subject to certain criteria. Thus we significantly enhance the generality of the cut-based approach to approximation algorithms.

New approximation algorithms: Deleting 2-CNF≡ clauses to achieve satisfiability, and applications

As a consequence, we obtain new approximation algorithms. Our first result is the following theorem:

Theorem 1.2 *Given a 2-CNF≡ formula with weighted clauses of the form $(P \equiv Q)$, one can in polynomial time find an approximately minimum-weight set of clauses whose deletion yields a satisfiable formula.*[1] *The performance ratio is $O(\log^3 n)$.*

One of our aims in this research is to develop a framework for approximate solution of graph problems of the form: delete a minimum number of nodes (or edges) in order to

*Research supported by an NSF grant CCR-9012357

†Brown University, Providence, RI 02912.

‡Aiken Computation Laboratory, Harvard University, Cambridge, MA. Research supported by ONR Grant # N0014-88-k-0243.

[1] A complementary result was obtained by Johnson [12] in 1974; he showed that the maximum size of a satisfiable subset of clauses could be approximated to within a small constant factor; indeed, for every formula, all but at most a constant fraction of the clauses could be satisfied simultaneously. In view of this latter result, it makes sense to focus on how many clauses must be deleted to achieve satisfiability.

obtain a graph with a desired structural property. Our algorithm for 2-CNF≡ clause deletion provides a good basis for such algorithms; one can sometimes state the structural property in the language of 2-CNF≡ in such a way that deleting clauses corresponds to deleting edges or nodes of the graph.[2]

The *edge-deletion graph bipartization problem*—deleting a minimum number of edges to get a bipartite graph—was studied in [28]. Using our variant of 2-CNF≡ in which each clause has the form $(P \equiv Q)$, it is easy to state that a given graph is bipartite: For each node v, we have a Boolean variable P_v. For each edge $\{v, w\}$, there is a clause $(P_v \equiv \neg P_w)$ of weight 1. The resulting formula is satisfiable if and only if the graph is bipartite; any satisfying truth assignment gives a bipartition (a two-coloring). Moreover, the minimum number of clauses that must be deleted to achieve satisfiability equals the number of edges that must be deleted to achieve two-colorability.

Using similar techniques, we can solve some other problems. The *via minimization problem* (discussed in [1]) is a problem arising in the design of a printed circuit board, in which there are two layers and one wants to minimize the number of holes made in the board in order to connect wires on different layers. The geometry of the problem is fixed; the goal is to choose an assignment of pieces of wire to layers.

Theorem 1.3 *One can in polynomial time approximately solve*

- *the edge-deletion graph bipartization problem,*
- *the via minimization problem.*

The performance guarantee is the same as that in Theorem 1.2.

New approximation algorithms: Approximately minimum chordalization

By adapting the method of [16], we can find approximately minimum node separators. This was also observed by Leighton [17]. We use this technique to approximately solve two NP-complete problems.

It has been a longstanding open problem to find an approximately minimum *chordalization* of a graph, i.e. to find a set of edges whose addition to the input graph yields a chordal graph. This problem has several applications, of which the most well-known was discovered by Rose [21], Gaussian elimination for sparse linear systems. When Gaussian elimination is used to solve a sparse symmetric positive-definite linear system, the order in which variables are eliminated has a significant bearing on the complexity of the elimination process, both in terms of time and space. The reason is that each successive variable elimination typically results in "fill-in", new non-zero entries in the matrix. These new non-zero entries must be stored, and they enter into subsequent calculations, so in order to minimize storage and calculation time, it is desirable to minimize the fill-in. Heuristics such as the "minimum-degree" heuristic are commonly used in an attempt to do just that. Other heuristics can be found in [14].

Rose reformulated the notion of fill-in as a graph property. He showed that if we interpret the matrix as a graph—where a non-zero entry A_{ij} corresponds to the existence of an edge $\{i, j\}$—then the filled-in matrix corresponds to a *chordal* graph. He thereby characterized the matrices for which no fill-in is required—they are the matrices whose graphs are chordal. He also demonstrated that for any given matrix, minimizing the fill-in corresponds to adding a minimum number of edges to a graph to make it chordal.

The issue of minimizing fill-in is especially important in the solution of linear programs. When an interior-point method is used, the principal cost of an iteration is due to solving a symmetric positive-definite system of linear equations. The particular system changes at each iteration, but the graph associated with the system remains the same. Since it is the graph that determines what fill-in is achievable and how, it is worth spending considerable time finding a good chordalization of the graph in order to save time and storage at each iteration of the linear programming algorithm.[3]

Unfortunately, as Rose, Tarjan, and Lueker conjectured [22] and Yannakakis proved [29], finding a minimum chordalization is NP-complete. We are thus led to considering whether minimum chordalization can be approximated by a polynomial-time algorithm. Until now, no such approximation algorithms were known that provably achieved better-than-trivial performance.

Our new algorithm

In this paper, we report the discovery of such an approximation algorithm, based on a method in [16], on a modification of this method due to Leighton, Makedon, and Tragoudas [17], and on a separator theorem of Gilbert, Rose, and Edenbrandt [9]. The algorithm performs particularly well on graphs with small degree. Fortunately, linear systems arising in practice (e.g. from the finite-element method) often have graphs with small degree, because of constraints in the physical system being modelled.

Theorem 1.4 *For an input graph with n nodes and maximum degree k, the algorithm outputs a chordal graph such*

[2] Cf. Papadimitriou and Yannakakis's idea [20] of expressing NP-complete problems with quantified sentences in order to study their approximability.

[3] Note also that the cost of a floating-point operation, which depends on the precision required, typically exceeds the cost of a step in a chordalization algorithm.

that the number of edges is within a factor of $O(\sqrt{k}\log^4 n)$ of optimal.

We can also prove a performance bound when the degree is large, but the bound is weaker and somewhat complicated, and we postpone it until Section 6.

It follows from Theorem 1.4 and the work of Rose that one can find an order of elimination that approximately minimizes the storage required for symmetric positive-definite matrices where each row and column has a small number of non-zero entries.

Our algorithm is similar to the generalized nested dissection of Lipton, Rose, and Tarjan [18]. Their algorithm works for any class of graphs with $O(f(n))$ separators, and relies on a recursive decomposition based on such separators. The performance of the algorithm is determined by the function $f(n)$. Since we do not assume the input graph has any special structure, we cannot rely on the existence of small separators. Gilbert [6] has shown that for every graph with degree at most k, there exists a nested dissection order that yields fill at most $O(k\log n)$ times optimum. We build on his important result. Gilbert's argument, however, is inherently non-constructive in that his choice of separators depends crucially on the optimally filled graph. Our analysis shows that it is sufficient to choose nearly optimal node separators in order to achieve near-optimal fill.

It turns out that our algorithm also approximately minimizes the total time required for the elimination process. Here we use the customary measure of time for this process, the number of multiplications.

Theorem 1.5 *The algorithm outputs an elimination ordering such that the number of multiplications is within a factor of $O(k\log^6 n)$ of optimal.*

Solving linear systems in parallel

Our algorithm is particularly well-suited to use in parallel solution of linear systems. In a typical parallel implementation of Gaussian elimination, each iteration consists of selecting some set of variables and eliminating them from the system. In order to eliminate all the variables in the set simultaneously, they must not interact—no two variables can occur in the same equation. The *height* of an elimination ordering is the number of iterations necessary; thus minimizing height corresponds to minimizing the parallel time required by this method. Moreover, the choice of which variables to eliminate in each step determines fill-in; it is desirable to keep the fill-in small, both to keep the storage small and to keep the number of processors small. Gilbert [7] has conjectured that there is an ordering that simultaneously minimizes height and approximately minimizes (to within a constant factor) the number of edges in the filled-in graph. Our analysis represents progress towards proving this conjecture.

Theorem 1.6 *For degree k graphs, there exists an ordering that simultaneously minimizes height to within a logarithmic factor and minimizes number of edges to within a factor of $O(\sqrt{k}\log^2 n)$.*

Theorem 1.6 follows from the analysis we use to prove Theorem 1.4 and from the analysis of Gilbert and Hafsteinsson [8]. They propose essentially the same algorithm as us to find elimination orderings that simultaneously minimize three important measures: tree width, front size, and elimination tree height. As part of their analysis, they show that the elimination tree height is within a logarithmic factor of the size of a minimum balanced node separator. Their algorithm differs from ours only in that it finds small node separators by first finding small edge separators. Consequently, their algorithm's performance guarantee for height depends linearly on the maximum degree of the input graph. As they observe, use of an approximation algorithm for node separators would yield a guarantee that was independent of degree. Since we use such an approximation algorithm, we achieve such a guarantee.

Theorem 1.7 *The algorithm outputs an elimination ordering whose height is within a factor of $O(\log^2 n)$ of optimal.*

Register sufficiency

Given a computation dag, how many registers are needed to compute it? This NP-complete problem arises in compiler optimization, in the task of register allocation.

Theorem 1.8 *There is a polynomial time algorithm that finds an order for computing an n-node dag such that the number of registers required is within an $O(\log^2 n)$ factor of the optimal.*

2 An approximate max-flow min-cut theorem for general undirected multicommodity flow

2.1 Preliminaries

In this section we prove an approximate max-flow min-cut theorem that forms the basis for the 2-CNF≡ clause deletion algorithm of Section 5 and the applications: bipartization and via minimization.

Consider a multicommodity flow problem, where we have a network G with edge-capacities $\text{CAP}(vw)$, and commodities $\{(s_i, t_i, d(i)) : i = 1, \ldots, k\}$, where s_i and t_i are, respectively, the source and sink of commodity i, and $d(i)$ is the demand of commodity i. We assume all demands and capacities are integral. A *concurrent flow* [26] of *capacity utilization* u is a multicommodity flow satisfying the

demands and subject to the edge-capacities $u \cdot \text{CAP}(vw)$. The concurrent flow problem is to find a flow f with minimum capacity utilization. The concurrent flow problem relates to feasibility of ordinary multicommodity flow: a multicommodity flow is feasible if and only if the minimum capacity utilization is at most 1.

It is useful to think of each pair s_i, t_i of commodity endpoints as a *demand edge* $s_i t_i$ in the graph G, with a weight $\text{DEM}(s_i t_i)$ equal to the demand associated with the commodity. Thus G has two kinds of edges, demand edges and ordinary capacity edges.

Each subset V of the node set of G defines a *cut*, namely the set $\Gamma(V)$ of edges with exactly one endpoint in V. For a subset E of the edge set of G, let $\text{CAP}(E)$ denote the sum of capacities of capacity edges in E, and let $\text{DEM}(E)$ denote the sum of weights of demand edges in E.

The analogue of a minimum cut is a cut $\Gamma(A)$ that minimizes the ratio of capacity to demand. That is, define the *minimum cut ratio* S as follows:

$$S = \min_A \frac{\text{CAP}(\Gamma(A))}{\text{DEM}(\Gamma(A))} \tag{1}$$

It is easy to see that the value of the capacity utilization is at least $1/S$. In other words, if the minimum cut ratio is less than one, then the multicommodity flow problem is infeasible.

Let ℓ be any length function assigning lengths to the capacity edges of the graph G, and let $dist_\ell(v, w)$ be the resulting shortest path distance between v and w using capacity edges. The linear programming dual to the problem of minimizing the capacity utilization u is maximizing the sum

$$\sum_{\text{demand edge } st} dist_\ell(s,t)\text{DEM}(st) \tag{2}$$

subject to the constraint

$$\sum_{\text{capacity edge } vw} \text{CAP}(vw)\ell(vw) = 1 \tag{3}$$

In particular, the value of (2) is always at most the value of the capacity utilization u, and, when ℓ maximizes (2) and f minimizes u, then the sum (2) equals u. That is, at optimality, we have

$$\sum_i dist_\ell(s_i, t_i)d(i) = u \geq 1/S \tag{4}$$

Our main result in this section is to show that the leftmost expression is not much more than the rightmost expression.

Theorem 2.1 *There is a polynomial-time algorithm to find a particular node subset A such that*

$$\sum_i dist_\ell(s_i, t_i)d(i) = O(\log C \log D)\frac{\text{DEM}(\Gamma(A))}{\text{CAP}(\Gamma(A))}$$

where C is the sum of all capacities and D is the sum of all demands.

Corollary 2.1
(Approximate Max-flow Min-cut Theorem)
We have

$$\frac{S}{O(\log C \log D)} \leq 1/u \leq S$$

In particular, for a multicommodity flow to be feasible, it is necessary that the min cut ratio S be at least 1, and it is sufficient that it be at least $O(\log C \log D)$.

Corollary 2.2
(Approximation Algorithm for Min Cut)
The cut $\Gamma(A)$ found by the algorithm of Theorem 2.1 approximately minimizes the ratio

$$\frac{\text{CAP}(\Gamma(A))}{\text{DEM}(\Gamma(A))}$$

to within a factor of $O(\log C \log D)$.

The following theorem is useful in applications where either the capacities or demands are exponentially large.

Theorem 2.2 *If either C or D is polynomial in n, the number of nodes of G, then the performance guarantee can be improved to be $O(\log^2 n)$.*

2.2 The proof of Theorem 2.1

Our proof of the theorem follows the lines of Leighton and Rao's proof for their result concerning the uniform demand case (the case where there is a demand of 1 between every pair of nodes).

First solve the dual of the concurrent flow problem, obtaining an optimal length function ℓ satisfying the constraint (3). Next, assume (for now) that we know the value of S. We now give an algorithm to assign a path $P(s_i, t_i)$ to each commodity i such that

$$\sum_i \text{length}(P(s_i, t_i)) \cdot \text{DEM}(s_i t_i) = O((1/S)\log C \log D) \tag{5}$$

To initialize, let D_0 equal the sum D of all demands, and let $t = 0$.

Each stage t consists of the following steps. Decompose the graph into node-disjoint trees so that each tree has height $O((1/SD_t)\log C)$. (This step is described in more detail later.) For each commodity i whose endpoints lie in a common tree, assign to the commodity the path in that tree, and discard the commodity. Let D_{t+1} be the sum of the demands of remaining commodities, i.e. those whose endpoints are in different trees. Finally, increment t, and loop.

In each stage t, the length of every path assigned is $O((1/SD_t)\log C)$, and the sum of demands of commodities

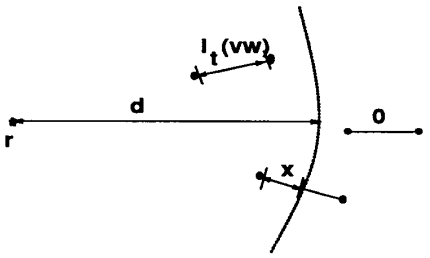

Figure 1: If one of the endpoints of edge vw is within distance $d-\ell_t(vw)$ of r, then all $\ell_t(vw)$ of the edge's tokens are within distance d. If neither endpoint is within distance d, then none of the edge's tokens is within distance d. If one endpoint is at a distance $d-x$ of r, and $x \leq \ell_t(vw)$, then x of the edge's tokens are within distance d.

that were assigned paths is $D_t - D_{t+1}$, so the total contribution of stage t to the left-hand side of (5) is $O((1/S)\log C)$. It remains to describe a procedure for decomposing G into trees of small height such that $D_{t+1} \leq D_t/2$, for then the number of stages is at most $\log D$, and the total contribution of all stages to the left-hand side of (5) is $O((1/S)\log C \log D)$.

Decomposition into trees: It is convenient to discretize or tokenize the distance. Each token of distance will correspond to distance $1/C$ in the original units. That is, define

$$\ell_t(vw) := \lceil C\ell(vw) \rceil \tag{6}$$

for each edge vw. Notice that since $\sum \ell(vw)\text{CAP}(vw) \leq 1$ and that rounding up adds only one extra token for each edge, we have that the total token weight is

$$\sum \ell_t(vw)\text{CAP}(vw) \leq 2C \tag{7}$$

Measured in tokens, we must decompose the graph into trees of token height $O((1/SD_t)C\log C)$.

Next, repeat the following step, forming a tree T (as described below), and then removing the nodes of the tree from the graph, as well as all incident capacity edges and demand edges, until the graph no longer contains a source or sink.

Forming a tree T: Start at any node r, and compute the shortest-path tree consisting of all nodes reachable from r. If every node in this tree has distance at most $\lceil (9C\ln 2C)/SD_t \rceil + 1$, then let T be this tree. Otherwise, we must select a distance $\hat{d} \leq \lceil (9C\ln 2C)/SD_t \rceil + 1$ at which to truncate the tree.

There is a natural notion of how much of an edge's tokens lies within a given distance d of the root r, as illustrated in Figure 1. Denote this value by amount(vw, d). We determine the total *weight* of the subgraph within distance d of r, defined as follows.

$$\text{weight}(d) = \sum_{\text{edge } vw} \text{amount}(vw, d) \cdot \text{CAP}(vw)$$

We use binary search on the range of distances $d = 1, 2, 3 \ldots \lceil (9C\ln 2C)/SD_t \rceil + 1$, to find a $\hat{d}$ such that

$$\text{weight}(\hat{d}+1) \leq (1 + \frac{SD_t}{8C})\text{weight}(\hat{d}) \tag{8}$$

The binary search to find such a $\hat{d}$ proceeds as follows. First, we set d_l to 1 and d_r to $\lceil (9C\ln 2C)/SD_t \rceil + 1$. Notice, that weight$(d_l)$ is at least $(1 + SD_t/8C)^{d_l - 1}$ and hence at least 1 since every edge has capacity one. Also notice that weight(d_r) is strictly less than $(1 + SD_t/8C)^{d_r - 1}$ since

$$(1 + SD_t/8C)^{d_r - 1} \geq (1 + SD_t/8C)^{(9C\ln 2C)/SD_t} \geq 2C,$$

and $2C$ is the *total* weight in the whole graph by equation 7. Thus, there must be a level between d_l and d_r where the weight does not expand by $(1 + SD_t/8C)$. Now, we consider the level $d' = \lfloor (d_l + d_r)/2 \rfloor$. If weight$(d')$ is at least $(1 + SD_t/8C)^{d'-1}$ then set $d_l = d'$, otherwise set $d_r = d'$. Notice that a level that does not expand still exists between d_l and d_r and the number of levels between d_l and d_r has been halved. Thus, after $O(\log C)$ iterations $d_l - d_r$ becomes a constant so $\hat{d}$ can easily be found. (We use this binary search in the algorithm so that the running time is polynomial in $\log C$ and n, i.e., the size of the input graph.)

Once $\hat{d}$ has been chosen, let T be the portion of the shortest-path tree spanning nodes of distance $\leq \hat{d}$ from the root, and define weight(T) = weight$(\hat{d})$. Let $\mathcal{E}_T$ denote the set of edges currently incident to T. Note that

$$\begin{aligned} \text{CAP}(\mathcal{E}_T) &\leq \text{weight}(\hat{d}+1) - \text{weight}(\hat{d}) \\ &\leq \frac{SD_t}{8C}\text{weight}(T) \end{aligned} \tag{9}$$

Finally we delete the nodes of T and all incident edges, and repeat, constructing another tree.

What the tree decomposition achieves: Now suppose that all sources and sinks have been deleted. We've assigned short paths to every commodity whose source and sink lay in the same tree. Let D_{t+1} denote the sum of demands of remaining commodities. We prove that $D_{t+1} \leq D_t/2$. Let D_T denote the sum of demands of commodities i with source s_i in T and sink t_i outside T. The trees partition D_{t+1}. Let $\Gamma(T)$ denote the boundary of T, i.e. the set of edges with exactly one endpoint in T. We have the following

$$SD_{t+1} = \sum_T SD_T \leq \sum_T \text{CAP}(\Gamma(T))$$

where the inequality follows from the definition of S. Since each edge belongs to the boundary of at most two trees and belongs to at least one $\mathcal{E}_T$, we have

$$\sum_T \text{CAP}(\Gamma(T)) \leq 2 \sum_T \text{CAP}(\mathcal{E}_T).$$

Using (9), we infer

$$2 \sum_T \text{CAP}(\mathcal{E}_T) < \frac{2SD_t}{8C} \sum_T \text{weight}(T)$$

In constructing the tree decomposition, once we count an edge towards the weight of the tree, we delete it. Hence $\sum_T \text{weight}(T)$ is no more than the total weight in the whole graph, which was shown to be $2C$ in (7). Putting these inequalities together, we obtain

$$SD_{t+1} \leq SD_t/2.$$

2.3 Turning the proof into an algorithm

In theorem 2.1, we stated that the algorithm would find a set A such that $\sum_i dist(s_i, t_i) d(i) = O(\log C \log D) \frac{\text{DEM}(\Gamma(A))}{\text{CAP}(\Gamma(A))}$. In fact, we only proved that $\sum_i dist_\ell(s_i, t_i) \text{DEM}(s_i t_i) = O(\log C \log D)/S$. Moreover, we assumed that the value of S was exactly known in advance.

We remedy this by making two observations. First, note that the proof still holds when S is replaced by the "observed value of S," which we denote by S_{obs}. The value of S_{obs} is defined to be the minimum value of $\frac{\text{CAP}(\Gamma(T))}{\text{DEM}(\Gamma(T))}$ over all trees T actually appearing in some decomposition during the algorithm. Thus if we knew S_{obs} before running the algorithm, we could substitute it for S, and proceed as before. We would end up showing that $\sum_i dist_\ell(s_i, t_i) \text{DEM}(s_i t_i) = O(\log C \log D)/S_{\text{obs}}$. By letting A be the nodes spanned by the tree that determined the value of S_{obs}, we obtain the bound in Theorem 2.1.

Of course, we can't expect to know the value of S_{obs} before running the algorithm. However, if the value substituted for S in (8) is no more than $(3/2)S_{\text{obs}}$, then we are still guaranteed that the remaining demand diminishes by a factor of 3/4 in each stage. We use this observation as follows. Start with some estimate of S, derived from looking at an arbitrary cut in the graph. Run the algorithm, and compare the resulting value of S_{obs} with the estimate. If the estimate is no bigger than $(3/2)S_{\text{obs}}$, then we are done, otherwise, reduce the estimate by a factor of 3/2, and repeat.

3 Balanced separators

In this section, we discuss the application of concurrent-flow based methods to finding balanced separators. In Subsection 3.1, we show how to apply our Theorem 2.1. These results are used in Section 5. In Section 4, we present the application of the method of [16] to finding node separators in undirected graphs and in directed acyclic graphs. These results are used in Sections 6 and 7.

3.1 Cutting all the demand

The cuts that are found by the algorithm of the previous section tend to achieve a low cut ratio. However, such a cut may only separate a small fraction of the *total* demand in the flow problem. In this section, we present some results about finding a small cut that cuts a large fraction of the total demand.

In particular, we consider a cut that cuts half the demand pairs.

Consider a multicommodity flow problem specified by $G = (V, E)$. We denote the capacity edges of G by E_C and denote the demand edges by E_D. Now we define E_{opt} to be a minimum capacity edge set such that if $\{H_1, ..., H_l\}$ are the connected components of $G_{\text{opt}} = (V, E_C - E_{\text{opt}})$ then

$$\sum_{s_i \in H_j, t_i \in H_k, j \neq k} d(i) \geq D/2.$$

That is, the edge set E_{opt} is the smallest cut that separates half the demand in G.

We can prove the following result about approximating such a cut.

Theorem 3.1 *There is a polynomial time algorithm, A, that will find a set of edges E_A such that if $\{H_1, ..., H_{l'}\}$ are the connected components of $G' = (V, E_C - E_A)$ then*

$$\sum_{s_i \in H_j, t_i \in H_k, i \neq k} d(i) \geq D/3,$$

and

$$\text{CAP}(E_A) \leq O(\log C \log D) \text{CAP}(E_{\text{opt}}).$$

Proof Sketch: We simply find an approximate sparsest cut using the algorithm of the previous section, remove the edges in the cut from the graph and repeat until $D/3$ demand sources and sinks are separated. We can show that the union of the cuts have small cost by using the fact that they have small sparsest cut cost. □

Thus we have an algorithm that finds a small cut that separates many of the demands. In fact, using the above theorem, we can find a small cut that separates all demands. First, we say an edge set, E_A, *completely cuts* the demand in a flow problem if there is no commodity whose source and sink are in the same connected component of $G' = (V, E_C - E_A)$. Now we state the following lemma.

Lemma 3.1 *There is a polynomial time algorithm, A, that given a flow problem on G will find a set of edges E_A that completely cuts the demand in G such that*

$$\text{CAP}(E_A) = O(\log C \log^2 D)\text{CAP}(E_{\text{opt}}),$$

where E_{opt} is the smallest capacity edge set that completely cuts G.

4 Separators in Graphs

As a consequence of the approximate max-flow min-cut theorem of Leighton-Rao, one can find weighted edge- and node-separators in a graph. We list their results below.

Lemma 4.1 ([16, 17]) *For a graph (directed or undirected), there exists a polynomial algorithm to find a $\frac{1}{3}$-balanced edge-separator (node-separator) with cost within $O(\log n)$ factor of the optimal $\frac{1}{2}$-balanced edge-separator (node-separator).*

We observe that one can find directed node separators in a DAG. That is, we find a partition, (L, X, R), of the nodes of G such that there are no edges of G between L and R, and all edges between X and R are directed from X to R. The *node cut ratio* of a directed node separator, (L, X, R), is defined to be $|X| / \min(|L \cup X|, |R \cup X|)$. The *sparsest node cut ratio* of a graph is the smallest node cut ratio of any directed node separator in the graph.

Lemma 4.2 *Given a DAG G, we can find a directed node separator of G whose node cut ratio is within a factor of $O(\log n)$ of the sparsest node cut ratio.*

Proof: We construct an auxiliary directed graph G' from G with the property that we can recover a node-separator of G from an edge-separator of G'. We augment the DAG G to G' as follows. For each node v in G, we add two nodes v_i and v_o in G' with a directed edge from v_i to v_o of infinite cost and a reverse edge from v_o to v_i of unit cost. We call such reverse edges the *pseudo-edges*. For each original edge (u, v) in G, directed edges (u_o, v_i) and (v_i, u_o) of infinite cost are added.

The algorithm in [16] finds a directed edge-separator in G' where the ratio of the capacity of the separator edges divided by the number of nodes on the smaller side is within a factor of $O(\log n)$ of the minimum value. We call this minimum value the *sparsest edge cut ratio*.

Any optimal or near-optimal edge separator for G' will only contain pseudo-edges, hence we can recover a node separator in G from these pseudo-edges. Since any node separator with a node cut ratio of n_r in G can be mapped to an edge separator in G' with an edge cut ratio of at most n_r, the sparsest edge cut ratio in G' is no more than the sparsest node cut ratio in G. Moreover, for any edge separator in G' with an edge cut ratio of e_r, the node separator recovered from it has a node cut ratio of no more than $2e_r$ in G. Hence from an approximate edge separator algorithm for G' with a performance guarantee of $c \log n$, one can find an approximate directed node separator in G with a performance guarantee of $2c \log n$. □

We use the above lemmas for the approximation algorithms presented hereafter.

5 Approximately minimizing unsatisfied clauses in a 2-CNF≡ formula

Given a 2-CNF≡ formula F with weighted clauses of the form $(p \equiv q)$, finding a minimum weighted set of clauses whose deletion yields a satisfiable formula is known to be NP-complete [4]. We use a well-known construction to reformulate this problem as a problem of deleting edges in a graph. We then show how to approximate this edge-deletion problem by defining a flow function in this graph and using the approximate max-flow min-cut theorem for general concurrent flow in Section 4.

Defn: We define a weighted 2-CNF≡ formula to be a conjunction of weighted clauses of the type $(p \equiv q)$ where p and q are literals, and $\equiv$ refers to logical equivalence.

5.1 Modeling the problem

Given a weighted 2-CNF≡ formula F, we define the corresponding boolean graph $G(F)$ as follows. Let the node-set $V(G)$ be the set of all the literals that occur in F. For every clause $(p \equiv q)$ in F, include the *undirected* edge (p, q) in $G(F)$. This edge is assigned a weight equal to the weight of the corresponding clause $(p \equiv q)$ in F. We shall henceforth use the terms "literal" and "vertex" interchangeably.

We can easily prove the following lemma [13].

Lemma 5.1 *A weighted 2-CNF≡ formula F is satisfiable iff no connected component of the graph G(F) constructed as above contains both a literal and its negation.*

By virtue of Lemma 5.1, we have the following corollary.

Corollary 5.1 *Given a weighted 2-CNF≡ formula F, finding a minimum weight set of clauses whose deletion yields a satisfiable formula is equivalent to finding a minimum weight set of edges in the corresponding boolean graph G(F) such that its deletion from G(F) leaves the graph with no connected component containing both a literal and its negation.*

5.2 The algorithm

Define a general concurrent flow problem on the boolean graph $G(F)$ by assigning one unit of a commodity to flow

from each literal x_i to $\neg x_i$ occuring in the same connected component. From Lemma 3.1, we can find a set of edges in $G(F)$ whose cost is at most within a factor of $O(\log^3 n)$ of the minimum cost separator that completely cuts all the demands in $G(F)$. From Corollary 5.1, such a separator corresponds to the minimum weight set of clauses that we are looking for and we have the following theorem.

Theorem 5.1 *There exists an algorithm that finds an approximately minimum-weight set of clauses to delete from a given weighted 2-CNF≡ formula to leave it satisfiable. The performance ratio is* $O(\log^3 n)$.

6 Minimum chordalization

In this section we give an approximation algorithm for minimum chordal graph completion of an input graph. For bounded degree graphs, the algorithm guarantees a polylog factor approximation to the number of edges in the optimal chordal graph. For unbounded degree graphs, the bound is weaker. This is the first known polynomial-time algorithm with a non-trivial performance guarantee.

As Rose has shown, and as discussed in the introduction, chordalization arises in solving a symmetric positive-definite system of linear equations. The order of elimination affects the storage, sequential time, and parallel time required to solve the system. Our algorithm outputs an ordering that approximately minimizes all these quantities simultaneously over all possible elimination orderings.

6.1 The Algorithm: Near-Optimal Generalized Nested Dissection

Given a graph $G(V, E)$, we define an elimination ordering α of its vertices. The graph G is then augmented to be the elimination graph G^* for the elimination ordering α [22]. G^* is the output chordal graph produced by the algorithm. The elimination graph is obtained as follows. At step i, the graph G_{i-1} is augmented to G_i such that all the higher numbered neighbors in G_{i-1} of the node numbered i (in the ordering α) form a clique. G_0 is set to the original graph G. The elimination graph corresponds to $G_{|V|}$.

Elimination Ordering Algorithm: Given a graph $G(V, E)$ with n nodes to be numbered in the range $[a, b]$, where $b = a + n - 1$, we proceed as follows. If $n = 1$, we number the node. Else, we find a balanced node separator X for G using Lemma 4.1. Let $|X| = s$. We number the vertices in the separator from $b - |X| + 1$ to b in any order. Let there be k connected subgraphs $A_1, \ldots, A_k$ of sizes $n_1, \ldots, n_k$ left on removing X from G. We recursively number the graph A_i in the range $[a+\sum_{j=1}^{i-1} n_j, a-1+\sum_{j=1}^{i} n_j]$ for each $i \in [1, k]$.

Separator Tree Representation: The elimination ordering of G can be related to the separator tree of the graph. The vertices in the separator X form the root of the tree with subtrees formed recursively for each of the pieces $A_1, \ldots, A_k$. The elimination numbering of the vertices is consistent with a postorder traversal of the tree nodes, with vertices in a tree node numbered in any order.

Defn: We shall refer to a node in the separator tree T as a *separator*, and to a node in the original graph G as a *vertex*. The vertices forming a separator are said to *belong* to the separator. Each vertex belongs to exactly one separator and we refer to this node as the *separator node* of the vertex. The *depth* of a separator is the distance of the separator node from the root of the tree. The depth of a vertex is the same as the depth of the separator it belongs to. The *subtree of a separator* is the subtree rooted at the separator in the separator tree. We say that a vertex u is an ancestor of v if u's separator node is either the same or is an ancestor of v's separator node in the separator tree.

We now state without proof a few lemmas that will be useful in subsequent arguments. All references to elimination orders or separator trees in Section 6 refer to the ones presented above.

Fact 6.1 *Every node induced subgraph of a chordal graph is also chordal.*

Theorem 6.1 ([9]) *Every chordal graph has a clique separator, and hence has a node separator of size at most* $O(\sqrt{E})$, *where* E *is the number of edges in the graph.*

Lemma 6.1 *For any vertex* v, *an edge* (u, v) *is in* G^* *only if for some edge* $(z, v) \in G$, v *is an ancestor of* z, *and* u *belongs to a separator node on the path from* v*'s separator node to* z*'s separator node. Note that this includes the case in which* u *and* v *belong to the same separator.*

Lemma 6.2 *The height of the separator tree is* $O(\log n)$.

6.2 Bounds on the number of edges

In this section we analyze the total number of edges in the resulting graph G^* for the cases of bounded and unbounded degree graphs.

Defn: We define the *weight of a separator* (w) to be the number of vertices belonging to the separator, and the *weight of a tree* (W) to be the sum of the weights of each of the separators in the tree. Edges with both their endpoints in vertices at the same depth and different depths will respectively be called *flat* edges and *jump* edges. The *cost of a vertex* (c) is the number of edges from the vertex to vertices at greater depth. Thus the total number of jump edges in G^* is the sum of the costs of the vertices. The *cost of a separator* is the sum of the costs of the vertices belonging to it.

Theorem 6.2 *The total number of flat edges in* G^* *at a given depth is* $O\left(E_{opt}\log^2 n\right)$. *Hence the total number of flat edges in* G^* *is* $O\left(E_{opt}\log^3 n\right)$.

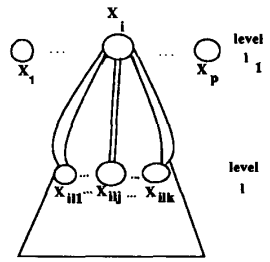

Figure 2: Counting the jump-edges in G^*- Bounded degree case.

Proof Sketch: By Lemma 6.1, the number of flat edges at a given depth are no more than those required to turn each separator into a clique. Let the subgraph induced by the subtree rooted at a separator X in the optimal chordal graph have $E_{opt,X}$ edges. By Fact 6.1, Theorem 6.1, and Lemma 4.1, $|X|$ is $\sqrt{E_{opt,X}}O(\log n)$. Hence the number of flat edges within X is $O(E_{opt,X}\log^2 n)$. Summing over node disjoint chordal graphs induced by the subtrees of each of the separators at the given depth, the claim follows. □

Theorem 6.3 *The total number of edges in G^* is $O\left(\sqrt{k}E_{opt}\log^4 n\right)$, where k is the maximum degree of the graph.*

Proof: Theorem 6.2 shows that the total number of flat edges is within the bound. Hence it suffices to show that the total number of jump edges is $O\left(\sqrt{k}E_{opt}\log^4 n\right)$.

For a vertex u, consider its neighbors in G at greater depth. Consider the tree formed by the union of the paths from u's separator to separators containing such neighbors. Call it the *associated tree* of u. Since G has bounded degree k, this tree can have at most k separator leaves, and hence at most k separators at any depth. By Lemma 6.1, the cost of the vertex u is the weight of u's *associated subtree*.

Let us estimate the sum of the costs of all vertices at a given level ℓ_1 in the tree (see Figure 2). Suppose that level consists of separators $X_1, \ldots, X_p$. For $i = 1, \ldots, p$, consider the highest-cost vertex of X_i, and let T_i be the associated subtree for this vertex. For each level ℓ, let $W_\ell(T_i)$ be the weight of T_i due to vertices at level ℓ. Then the sum of the costs of vertices at level ℓ_1 is no more than the sum, over all levels ℓ greater than ℓ_1, of the value

$$\sum_{i=1}^{p} |X_i| \cdot W_\ell(T_i). \tag{10}$$

The weight of T_i at level ℓ is the sum of the sizes of at most k separators $X_{i,\ell,1}, \ldots, X_{i,\ell,k}$. Substituting into (10), we get

$$\sum_{i=1}^{p}\sum_{j=1}^{k} |X_i| \cdot |X_{i,\ell,j}| \leq \sqrt{\sum_{i=1}^{p} k|X_i|^2}\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{k} |X_{i,\ell,j}|^2} \tag{11}$$

where the inequality follows from the Cauchy-Schwartz inequality.

Since the X_i's are all disjoint, it follows from the proof of Theorem 6.2 that $\sqrt{\sum_i k|X_i|^2} = O(\sqrt{kE_{opt}}\log n)$. Similarly, $\sqrt{\sum_{i=1}^{p}\sum_{j=1}^{k} |X_{i,\ell,j}|^2} = O(\sqrt{E_{opt}}\log n)$. Thus the right-hand side of (11) is $O\left(\sqrt{k}E_{opt}\log^2 n\right)$. Summing over all levels $\ell 1$ and ℓ, we conclude that there are $O\left(\sqrt{k}E_{opt}\log^4 n\right)$ jump-edges. □

For unbounded degree graphs, we state the following result without proof.

Theorem 6.4 *For an unbounded degree graph G with n vertices and m edges, the total number of edges in G^* is $O\left(min(\sqrt{E_{opt}}n\log^2 n, E_{opt}^{\frac{3}{4}}\sqrt{m}\log^{3.5} n)\right)$.*

It should be noted that the above value is no more than $O(m^{\frac{1}{4}}\log^{3.5} n)$ factor of the optimal.

6.3 Bounds on the number of multiplications

For solving a symmetric sparse system of linear equations without pivoting, we view the matrix as the adjacency matrix of a graph G. We define the elimination numbering of the nodes in G as in Section 6.1. We show that for a matrix with the number of non-zero entries in any row (or column) not exceeding k, the number of multiplications performed using this elimination order is within a $O\left(k\log^6 n\right)$ multiplicative factor of that required by the optimal elimination ordering.

Before we prove the main result of this section, we give some lemmas that will be used later. A system of linear equations in n variables given by $Ax = b$ is considered dense if the matrix A has $\Theta(n^2)$ non-zero entries.

Fact 6.2 *The number of multiplications required to solve a dense system of equations in m variables is $\Omega\left(m^3\right)$.*

The following lemma is a consequence of fact 6.2.

Lemma 6.3 *For any chordal graph G^*, if m is the size of its clique separator, then $\Omega\left(m^3\right)$ is a lower bound on the number of multiplications required for any elimination ordering.*

Let M_{opt} be the optimal multiplication count for any elimination ordering of G. Then we have the following results.

Theorem 6.5 *Let a given level in the separator tree have p separators with weights $w_1 \ldots w_p$. Then $\Omega\left(\sum_{i=1}^{p} \frac{w_i^3}{\log^3 n}\right)$ is a lower bound on M_{opt}.*

Proof Sketch: The vertices in the subtrees of different separators at a level are disjoint. Hence the subgraphs induced by them in the optimal chordal graph are node disjoint. Each induced subgraph is also chordal and hence Lemma 6.3 provides a lower bound on the optimal multiplication count for each. By summing over all the subgraphs, we obtain the theorem. □

For any vertex with elimination number i, the vertex along with all its neighbors with number higher than i form a clique in G^*. We shall refer to this clique as the *associated clique* for the vertex, and denote it by C_v. The number of multiplications (M_v) required to eliminate a variable v, is the total number of edges in the clique C_v. Let M be the number of multiplications required for the elimination ordering defined by the algorithm. M is given by $\sum_v \sum_{e \in C_v} 1$, which is the same as $\sum_e \sum_{v:C_v \ni e} 1$. We can hence sum over the multiplication contribution for each edge. The *multiplication contribution* (henceforth referred to simply as *contribution*) for an edge (v, u) is the total number of vertices with depth no less than that of u or v, containing (v, u) in its associated clique.

Theorem 6.6 *The elimination ordering defined above yields a multiplication count of $O\left(kM_{opt}\log^6 n\right)$.*

Proof Sketch: We count the contributions from edges that go between any two levels of the tree. To count this contribution, we count the contribution due to all vertices at a depth no less than those of the two levels. Using techniques similar to those for proving Theorem 6.3 we can show that this contribution is no more than k times the sum of the cubes of the separator sizes at the three levels under consideration, which using Theorem 6.5 is $O\left(3kM_{opt}\log^3 n\right)$. The claim then follows. □

6.4 Bounds on the height of the elimination ordering

Minimizing the height of the elimination ordering minimizes the time required to solve the system of equations in parallel. Let h_{min} be the minimum elimination height for any elimination ordering of a given set of linear equations. We show that the elimination ordering proposed also minimizes height to within a polylog factor of h_{min}.

Fact 6.3 *To solve a dense system of equations in m variables, h_{min} is $\Omega(m)$.*

Lemma 6.4 *For any chordal graph G^*, if m is the size of its clique separator, then h_{min} for G^* is $\Omega(m)$.*

Theorem 6.7 *If w_{max} is the maximum weight of a separator in the separator tree of G, then h_{min} for G is $\Omega\left(\frac{w_{max}}{\log n}\right)$.*

Proof: Let X be the separator in the separator tree with the maximum weight, and let V_i be the set of vertices in the subtree of the separator X. If G^*_{opt} corresponds to the optimal chordal graph with minimum height over all elimination orders, then let the graph induced by V_i in G^*_{opt} be G_i. G_i has a clique separator of size $\Omega\left(\frac{w_{max}}{\log n}\right)$ by Theorem 6.1, and this is a lower bound on h_{min} by Lemma 6.4. □

Theorem 6.8 *The elimination ordering defined in Section 6.3 yields a height of $O\left(h_{min}\log^2 n\right)$.*

Proof: Consider all the separators at each level. One variable from each of the separators can be eliminated simultaneously as there are no direct edges between the variables. Hence the height for eliminating all the variables at a level is no more than w_{max}. Since the number of levels is $O(\log n)$, the claim follows. □

It should be mentioned that a parallel prefix operation may be required at each elimination step to update the coefficients in the matrix as a result of eliminating multiple variables simultaneously.

7 Approximating Register Sufficiency

In this section, we present a polylogarithmic approximation to the *register sufficiency* problem. Given a DAG G (where $|G| = n$) with its vertices numbered by a topological ordering τ, the *register cost at step i* is defined as the number of nodes in the set $\{1, \ldots, i\}$ that are tails of edges that go from the set $\{1, \ldots, i-1\}$ to the set $\{i, \ldots, n\}$. The *maximum register cost* of this ordering, denoted by MRC_τ, is the maximum of the register costs over all steps i. We shall refer to the minimum value of the maximum register cost achievable by any topological ordering for G as the *optimum register cost* M for G. The register sufficiency problem is to find an ordering τ such that $MRC_\tau = M$. This problem is shown to be NP-complete in [23]. We give a polynomial time algorithm that finds a topological ordering of G with its maximum register cost within a polylogarithmic factor of M.

We now describe the algorithm for the register sufficiency problem.

We find an approximate sparsest directed node separator (L, X, R) in G as outlined in Lemma 4.2. If $|X| / \min(|L|, |R|)$ is greater than $1/\log n$ then we output an arbitrary topological ordering for G. Otherwise, we partition G into $L \cup X$ and R. We recursively order each of these subgraphs, and output the topological ordering of

G consisting of the recursively produced order of $L \cup X$ followed by the recursively produced order of R.

We now argue that this algorithm produces the desired result. First, it is easy to see that the following fact holds.

Fact 7.1 *The ordering τ defined by the above algorithm is a topological ordering.*

Now we proceed by showing that τ is a *good* topological ordering.

Theorem 7.1 *Given an n-node DAG G, one can in polynomial time find a topological ordering τ of G such that MRC_τ is within a factor of $O(\log^2 n)$ of the optimum register cost M for G.*

Proof: Consider the first cut, (L, X, R), that is used in the algorithm above to produce τ. Notice that if the algorithm recurses the schedule τ needs to use only as many registers as the size of X plus the maximum of the number of registers that τ needs for evaluating $L \cup X$ or to evaluate R.

Thus we can use the following recurrence to estimate the performance bound when the algorithm recurses.

$$S(n) \leq |X| + \max(S(|L \cup X|), S(|R|)) \tag{12}$$

If the algorithm does not recurse, we use the trivial bound of $S(n)$ being at most n.

We proceed by bounding the size of $|X|$ in terms of M. To do this we consider an optimal ordering, τ_{opt} of G. We form a partition, (A, B), of the nodes of G where A consists of the first $n/2$ nodes in the ordering τ_{opt}. Now consider the set of nodes in A with a neighbor in B: these form a directed node separator of G. Clearly, a register must be used for each of these. Thus, there are no more than M such nodes. Thus, there exists a node separator of G with sparsest cut cost of at most $M/|A| = M/(n/2)$. Now recall that (L, X, R) has sparsest cut cost of $O(\log n)$ times optimal. Thus,

$$\frac{|X|}{\min(|L \cup X|, |R \cup X|)} \leq c \log n \frac{M}{n/2}. \tag{13}$$

When the algorithm does not recurse, $\frac{\min(|L \cup X|, |R \cup X|)}{|X|}$ is at most $\log n + 1$. Thus from (13), n, and hence $S(n)$, is at most $2cM \log n(\log n + 1)$, which is $O(M \log^2 n)$.

Now we consider the case where the algorithm does recurse. With the above bound on $|X|$, we can rewrite recurrence (12) as

$$\begin{aligned} S(n) \quad \leq \quad & 2cM \log n \frac{\min(|L \cup X|, |R \cup X|)}{n} \\ & + S(\max(|L \cup X|, |R|). \end{aligned}$$

When the algorithm recurses, we have $|X| \leq \frac{\min(|R|, |L|)}{\log n}$, so we can rewrite the above inequality as

$$\begin{aligned} S(n) \quad \leq \quad & 2c'M \log n \frac{\min(|L \cup X|, |R|)}{n} \\ & + S(\max(|L \cup X|, |R|). \end{aligned}$$

where c' is no more than $c(1 + \frac{1}{\log n})$.

We further simplify the above equation to be

$$S(n) \leq 2rc'M \log n + S((1 - r)n), \tag{14}$$

where

$$r = \frac{\min(|L \cup X|, |R|)}{n} = (1 - \frac{\max(|L \cup X|, |R|)}{n}).$$

Finally, we note that at most M registers are needed to evaluate any subgraph of G. Inductively assuming that $S(n')$ is $c''M \log^2 n'$, we can infer from (14) that $S(n) \leq c''M \log^2 n$ for an appropriate constant c''. □

8 Final remarks

We have presented an approximate min-max theorem for general multicommodity flow. Recently, we have been able to generalize this theorem to apply to hypergraph networks; using this theorem, we can handle CNF≡ clauses with an arbitrary number of literals per clause.

We have not addressed running times of algorithms described in this paper, but we note that the algorithm of [15] can be used to quickly find approximate solutions to the concurrent flow problems.

Our approximate min-max theorem, while more general than that in [16], does not guarantee as good an approximation. In contrast to [16], we have no example demonstrating that our bound is existentially tight. It is therefore an open problem to improve our bound or show it cannot be improved.

Acknowledgements

We gratefully acknowledge helpful conversations with John Gilbert, Tom Leighton, John Reif, and David Shmoys.

References

[1] H. Choi, K. Nakajima and C.S. Rim, "Graph bipartization and via-minimizatiuon", *SIAM J. of Discrete Maths Vol. 2, No. 1* (1989), pp. 38-47.

[2] P. Elias, A. Feinstein and C.E. Shannon, "A note on the maximum flow through a network", *IRS Trans. Information Theory IT 2* (1956), pp. 117-119.

[3] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey (1962).

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, W. H. Freeman, San Francisco (1979).

[5] George, J. A., "Nested Dissection of a regular finite element mesh", *SIAM Journal on Numerical Analysis 10* (1983), pp. 345-367.

[6] J. R. Gilbert, "Some Nested Dissection Order is Nearly Optimal", *Information Processing Letters* 26 (1987/88), pp. 325-328.

[7] J. R. Gilbert, personal communication (1989).

[8] J. R. Gilbert and H. Hafsteinsson, "Approximating treewidth, minimum front size, and minimum elimination tree height", manuscript, 1989

[9] J. R. Gilbert, D. J. Rose, and A. Edenbrandt, "A separator theorem for chordal graphs", *SIAM J. Alg. Disc. Meth. 5* (1984), pp. 306-313.

[10] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York (1980).

[11] T. C. Hu, "Multicommodity network flows", *Operations Research 11*, (1963), pp. 344-360.

[12] D. S. Johnson, "Approximation algorithms for combinatorial problems", *Journal of Computer and System Sciences 9* (1974), pp. 256-278.

[13] N. D. Jones, Y. E. Lien and W. T. Lasser, "New problems complete for nondeterministic log space", *Math. Systems Theory, 10* (1976), pp. 1-17.

[14] U. Kjærulff, "Triangulation of graphs - Algorithms giving small total state space", *R 90-09, Institute for Electronic Systems, Department of Mathematics and Computer Science, University of Aalborg* (1990).

[15] P. Klein, C. Stein, and É. Tardos, "Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities", *Proceedings, 22nd ACM Symposium on Theory of Computing* (1990), pp. 310-321.

[16] F. T. Leighton and S. Rao, "An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms", *Proceedings, 29th Symposium on Foundations of Computer Science* (1988), pp. 422-431.

[17] F. T. Leighton, F. Makedon, and S. Tragoudas, personal communication, 1990

[18] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized nested dissection", *SIAM Journal on Numerical Analysis 16* (1979), pp. 346-358.

[19] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem, *SIAM Journal on Computing 9* (1980), pp. 615-627.

[20] C. H. Papadimitriou and M. Yannakakis, "Optimization, approximation, and complexity classes", *Proceedings, 20th ACM Symposium on Theory of Computing* (1988), pp. 229-234.

[21] D. J. Rose, "Triangulated graphs and the elimination process", *Journal of Math. Anal. Appl. 32* (1970), p. 597-609.

[22] D. J. Rose, R. E. Tarjan, and G. S. Lueker, "Algorithmic aspects of vertex elimination on graphs", *SIAM J. Comp. 5* (1976), pp. 266-283.

[23] R. Sethi, "Complete register allocation problems", *SIAM J. Comp. 4* (1975), pp. 226-248.

[24] P.D. Seymour, "Matroids and multicommodity flows", *European Journal of Combinatorics 2* (1981), pp. 257-290.

[25] P.D. Seymour, "On odd cuts and planar multicommodity flows", *Proc. London Math. Soc. 42* (1981), pp. 178-192.

[26] F. Shahrokhi and D. Matula, "The maximum concurrent flow problem," *Journal of the ACM* **37**:2 (1990), pp. 318-334

[27] R. Schreiber, "A new implementation of sparse Gaussian elimination", *ACM Trans. on Mathematical Software 8:3* (1982), pp. 256-276.

[28] M. Yannakakis, "Edge-Deletion problems", *SIAM J. Computing 10*, (1981), pp. 297-309.

[29] M. Yannakakis, "Computing the minimum fill-in is NP-complete", *SIAM J. Algebraic and Discrete Methods 2* (1981), pp. 77-79.