Approximating Concurrent Flow with Uniform Demands and Capacities: An Implementation

Philip N. Klein and Sarah Kang

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-58
September 1991

Approximating concurrent flow with uniform demands and capacities: an implementation

Philip N. Klein
Sarah Kang

Abstract

We have implemented the concurrent flow approximation algorithm described in "Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities," by Klein, Stein, and Tardos, which appeared in STOC 90. The implemented algorithm works for unit demands and unit capacities, and makes use of the idea of randomized path selection, introduced in that paper. We make use of a simplification of the path selection technique proposed by A. Goldberg. The implemented algorithm differs from the published algorithm in another respect as well; an effort is made to keep small the number of flow paths used to realize the concurrent flow. This paper describes our experience with the implementation thus far, and our plans for further experimentation.

1 Definition of the problem

An instance consists of a network G with edge- or node-capacities and a set of *commodity* specifiers, each consisting of a source node s_i, a sink node t_i, and a demand d_i, a positive real. A multicommodity flow consists of a flow of value d_i from s_i to t_i for every commodity i. The multicommodity flow is *feasible* if the sum of the flows through each edge/node does not exceed the edge or node's capacity. An instance is feasible if there exists a feasible multicommodity flow.

The *concurrent flow problem* is, given an instance, to output the minimum positive real λ such that by multiplying all capacities by λ, one obtains a feasible instance. As proof of feasibility, the feasible multicommodity flow in the multiplied network should also be output. The problem was proposed by Matula [12].

The special case handled by our implemented algorithm requires that all demands d_i and all capacities are one. While more general algorithms are available, this special case arises in several applications of considerable interest, as we describe in the next section.

2 Applications

Two principle application areas for the algorithm are finding graph separators and VLSI routing. Applications in turn for graph separators are numerous; we focus in this paper on two we have investigated, chordalization of a graph (which arises in solving sparse linear systems) and layout of a graph (which arises in VLSI and PC board design, as well as in user interfaces).

2.1 Finding graph separators

A crucial ingredient in many divide-and-conquer graph algorithms is a method for choosing edges or nodes whose removal breaks a graph into pieces. The set of removed edges or nodes is called an *edge-separator* or a *node-separator*. The divide-and-conquer approach to a problem is recursive; one removes a separator from the input graph, recursively solves the problem on the resulting pieces, and puts the subsolutions together to obtain a solution for the input graph. The "cost" of putting the subsolutions together typically depends on the size of the separator, where "cost" could be in terms of the computational effort required or the quality of the resulting solution. Thus it is desirable that the separator be small. At the same time, the depth of recursion also typically affects computational effort and the quality of the solution. Thus it is also desirable that after removal of the separator, the resulting pieces of the graph are small.

A priori, these two goals are in conflict. The greater the size of the separator allowed, the smaller can be the pieces. A compromise must be reached. Two kinds of compromises suggest themselves. One is to require that, say, the pieces be no larger than a prespecified threshold, and to try to find a smallest separator satisfying the requirement. The result is called a *balanced* separator, because typically one requires that no piece be larger than, say, one-half the size of the input graph; one can achieve this by splitting the grpah into two pieces of roughly equal sizes.

The second compromise is to try to minimize the product of the size of the separator with some measure of the sizes of the resulting pieces. Intuitively, this approach allows one to consider the full range of possible balance thresholds and choose the one that yields the "best" compromise. One measure of the sizes of the resulting pieces is the number of pairs of nodes in different pieces. This quantity tends to be big when the pieces are small. Let us assume there are exactly two pieces (if there are more components, group them into two pieces to get the best balance). In this case, the number of pairs of nodes in different pieces is just the product of the sizes of the two pieces. Let us refer to this measure as the *sparsity measure* of the separator.

To use this quantity in evaluating a proposed separator, we try to minimize the ratio of separator size to the sparsity measure. A separator that minimizes the ratio of size to sparsity measure is called a *sparsest separator*. The notion was introduced by Matula and Shahrokhi [13] in connection with the concurrent flow problem.

The two compromises are not really all that different. Given an algorithm for finding a minimum blanced separator, one can find a sparsest separator, by trying different thresholds. Conversely, Leighton and Rao have shown [9] that iteratively removing the edges or nodes of a sparsest separator, one can obtain an approximately minimum approximately balanced separator. Indeed, as Rao has pointed out [14] the proof shows that for many divide-and-conquer purposes, a sparsest separator is approximately as good as a balanced separator. In the applications we describe, we have used the former.

2.2 Leighton and Rao's algorithm and its uses

In an important paper that has catalyzed much recent work on algorithms for the concurrent flow problem, Leighton and Rao [9] gave an algorithm to approximate the sparsest cut to within a factor of $O(\log n)$. The computational bottleneck of their algorithm was solving the linear programming dual of concurrent flow. Their work was foreshadowed by, e.g., that of Matula and Shahrokhi [13].

Leighton and Rao also showed that the algorithm could be used in approximation algorithms for a variety of optimization problems in graph theory. Other researchers [3, 8, 11, 6, 15] have added still more problems approximable by these techniques.

The first step of Leighton and Rao's algorithm is solving the dual of a concurrent flow problem. The network is the input graph, where the capacity of an edge or node is taken to be the *cost* of having it in the separator. In the applications we describe, all edges/nodes have equal cost, so the network capacities are all the same (without loss of generality, they are all 1). The demands are also all the same; there is a demand of 1 between every pair of nodes. In a computationally easier variant proposed by Leighton, each node is the source node s_i for some small number of commodities (e.g. 3) where the sink nodes t_i are randomly chosen nodes.

The dual of concurrent flow involves assigning lengths to edges/nodes in order to maximize a certain sum of shortest paths. Once the dual has been solved (or approximately solved), the lengths are processed in order to come up with a separator. This involves a discretized shortest-path computation, which yields numerous candidate separators; each is evaluated, and the best is

chosen to be output.

Ajit Agrawal has implemented a variant of this second part of the algorithm. This implementation will not be described in this paper. However, the work described in Subsections 2.3 and 2.4 depend on this implementation.

2.3 Graph chordalization: an application of node separators

Sparse linear systems are those systems $Ax = b$ where most of A' entries are zero. Such systems often arise. One can take advantage of sparsity by storing and manipulating only the nonzero elements. However, in applying Gaussian elimination to symmetric sparse linear systems, one finds that the order in which variables are eliminated has an impact on the time and storage required by the compuation, in addition to several other computational resources. The reason is that, as the elimination progresses, nonzero elements are introduced, depending on which equations are used to eliminate variables. Rose [16] modelled this phenomenon using graph theory; he showed that, when solving $Ax = b$, there is an ordering such that no new nonzero elements are introduced if and only if the graph $G(A)$ of the matrix A—the graph whose incidence matrix has zeroes exactly where A does—has a property called *chordality*. Moreover, the problem of finding a good ordering is related to finding a graph G^* containing $G(A)$ as an edge-subgraph, Various predictors of the computational difficulty of solving the system are translated into graph measures of G^*. For example, the storage required is proportional to the number of edges of G^*.

In [6] is proposed an approximation algorithm for chordalization. We show that the same algorithm simultaneously minimizes (approximately) a number of measures: storage, time, and parallel time. (That other measure are approximately minimized was shown by [4]. The algorithm is essentially what has been known as generalized nested dissection [10], but using Leighton and Rao's algorithm to find node separators. One finds a sparsest node separator, orders its nodes last, and recursively finds an ordering for each of the pieces.[1] Ajit Agrawal has implemented this algorithm, using our implementation of concurrent flow and his implementation of Leighton and Rao's algorithm. That implementation and our experience with it will not be discussed here, except to say that the implemented algorithm seems to perform well—the output quality is competitive with that obtained using the most highly-tuned heuristic code (due to Joseph Liu)—despite the fact that our implemented algorithm has not yet been tuned or refined.

2.4 Graph layout: an application of edge-separators

Another problem to which a separator algorithm is applicable is that of laying out a graph in a rectangle. This involves finding positions for each of the nodes. Our intended application is to placement of circuit components for VLSI or PC board design, but the algorithm is also useful for finding a way to display a graph that makes it easy to see. There is a vast literature on both these topics. In this paper, we will content ourselves with showing some of the results of our experiments.

The algorithm is a variant of one proposed in [1]. We are given a graph and a rectangle in which to lay it out. Find an approximately sparsest edge-separator. Group the resulting pieces into two subgraphs. Divide the rectangle into two subrectangles and recursively lay out each subgraph in its subrectangle. Eventually, each node is assigned its own rectangle.

[1]The original algorithm proposed in [6] used approximately balanced separators, but this one is equivalent.

In dividing up the rectangle into subrectangles, we follow two rules. First, the sizes of the subrectangles are chosen to be proportional to the sizes of their subgraphs. It follows inductively that the rectangles belonging to individual nodes are all the same size. Second, the algorithm chooses whether to divide a rectangle vertically or horizontally depending on its current aspect ratio. This technique is intended to keep the aspect ratio from becoming too big.

We have applied our implemented algorithm to a number of the ISCAS benchmark circuits, used as benchmarks in test generation. We do not yet have comparisons between our outputs and those of other systems. However, the displayed graphs are visually appealing–at least in comparison to the random layouts—and we expect that subsequent routing will perform well with these placements.

One possible disadvantage of using sparsest separators instead of balanced separators become apparent while we observed the program running. If the separation is very imbalanced, one of the resulting subgraphs will be very small compared to the other, and so one of the resulting subrectangles will have a very high aspect ratio. Of course, subsequent recursion on the small subgraph will reduce the aspect ratio quickly, because of the second subrectangle rule described above. It is not clear, then, whether this disadvantage is reflected in the quality of the final layout. Eventually, we will compare the quality of these outputs to those obtained using approximately balanced separators.

3 The algorithm

3.1 The framework

The framework for the algorithm, was originated by Shahrokhi and Matula [17]. We start with some multicommodity flow in which, for each commodity i, the flow from source s_i to sink t_i consists of a collection of paths. The flows define lengths on the edges/nodes by an exponential formula: the length of an edge/node x is $e^{\alpha f(x)}$, where $f(x)$ is the sum of flows through that edge/node, and α is a parameter of the algorithm. The algorithm consists of a sequence of iterations; in each iteration, one chooses a flow path, say from s_i to t_i, computes a shortest path from s_i to t_i, and, depending on the length of the flow path compared to that of the shortest path, reroutes some amount of flow from the flow path to the shortest path (thus creating a new flow path). Eventually the flow assignment defines a nearly optimal solution to the concurrent flow problem, and the lengths define a nearly optimal solution to the linear programming dual.

The linear programming dual is as follows. Assign nonnegative lengths to the edges or nodes (depending on whether the edges or nodes have capacities in the primal problem). The length assignment must obey the constraint that the sum of all lenths is at most 1. The goal is to maximize the sum, over all commodities i, of the shortest-path distance from source s_i to sink t_i. This is equivalent to maximizing the ratio of the shortest path sum to the sum of lengths, where the length assignment is not subject to the sum-of-lengths constraint. This latter formulation is easier to work with because it obviates the necessity to rescale all lenths when some of them change.

The algorithm uses duality to determine when to terminate. Recall that the objective in the primal is to route the flow so as to minimize the maximum flow on an edge/node. The dual objective is to maximum the sum of shortest path lengths. When the primal objective value is not much more than the dual objective value, both are nearly optimal, and the algorithm may

terminate, outputting the flow assignment (the primal solution) and the length assignment (the dual solution).

The approach uses a potential function to show termination. The potential function is the sum of lengths, where each length is computed according to the exponential formula $e^{\alpha f(x)}$. The goal is reroute from those edges/nodes with high flow value $f(x)$ to those with smaller flow value. If done with care, this has the effect of reducing the sum of lengths. In particular, one only reroutes from a flow path to a shortest path if by doing so one significantly reduces the sum of lengths. It can be shown that if no such flow paths exists, then primal and dual are nearly optimal.

3.2 The implemented algorithm

Now we consider the specific algorithm implemented. The measure of quality of solution at any point in the algorithm is the ratio of $|f||\ell|_1$ to the sum of shortest paths, where $|f|$ is the maximum flow on any edge/node, and $|\ell|_1$ is the sum of lengths. This ratio is (barring floating-point roundoff error) always at least one; if it is close to 1, the primal and dual are close to being optimal.

Between $|f||\ell|_1$ and the shortest path sum is another quantity, the sum $\sum_x f(x)\ell(x)$ over all edges/nodes x, of the flow $f(x)$ on x times the length $\ell(x)$ assigned to x. We can express the above ratio in terms of two ratios

$$\frac{|f||\ell|_1}{\text{shortest path sum}} = \frac{|f||\ell|_1}{\sum_x f(x)\ell(x)} \frac{\sum_x f(x)\ell(x)}{\text{shortest path sum}} \tag{1}$$

Thus when each of the two ratios is not much more than 1, the solutions are nearly optimal.

3.2.1 The relaxed optimality conditions

In order to show the algorithm progresses towards near-optimality, we use two conditions, derived from the complementary slackness conditions of linear programming. The conditions are modified version of conditions in [7].

Condition I: $\frac{|f||\ell|_1}{\sum_x f(x)\ell(x)} \leq 1 + \epsilon$

Condition II: $\sum_P f(P)[\ell(P) - \text{short}(P)] \leq \epsilon|f||\ell|_1$

The first condition states that the first of the ratio in the right-hand side of (1) is nearly 1. The second condition needs some clarification. The sum is over all flow paths P. This includes the paths carrying flow from s_i to t_i for every commodity i. For a flow path P, $f(P)$ denotes the amount of flow carried b y P, $\ell(P)$ denotes the length of the path P with respect to the edge/node lengths $\ell(\cdot)$, and short(P) denotes the lenth of the shortest path with the same endpoints as P. Intuitively, condition II states that the lengths of flow paths are typically not much more than the lengths of corresponding shortest paths. It can be shown that if conditions I and II hold, then the ratio (1) is not much more than one.

3.2.2 Handling condition I

In order to achieve condition I, we periodically compute the ratio in I. If the ratio is not sufficiently close to 1, we double the parameter α appearing in the length formula $\ell(x) = e^{\alpha f(x)}$. This tends to concentrate more length on those edges/nodes x with particularly high flow $f(x)$, which tends

to reduce the ratio. Indeed, it is shown in [7] that if α is sufficiently high, then the ratio is sufficiently close to 1. We do not set α to this sufficiently high value all at once, for reasons that will be discussed later.

3.2.3 Handling condition II

Recall that the algorithm consists principally of rerouting flow from a flow path P to a shortest path with the same endpoints. The potential function by which we measure progress is the sum of lengths $|\ell|_1$. It is shown in [7] that by rerouting an amount σ of flow from a path P to a shortest path, we reduce $|\ell|_1$ by at least

$$.95\alpha\sigma[(1-\epsilon)\ell(P) - \text{short}(P)] \tag{2}$$

as long as α and σ satisfy the condition

$$1.05\alpha\sigma \leq \epsilon \tag{3}$$

Thus we make significant progress when $[(1-\epsilon)\ell(P) - \text{short}(P)]$ is large for the selected flow path P and α and σ satisfy (3).

To ensure that $[(1-\epsilon)\ell(P) - \text{short}(P)]$ tends to be large, we look to condition II. Condition II is a variant of condition II in [7] which was proposed by Goldberg. We can assume it is not satisfied, for otherwise (assuming I is satisfied) the primal and dual are nearly optimal and we can terminate. When II is not satisfied, for a flow path P randomly selected with probability proportional to $f(P)$, the expected value of $[(1-\epsilon)\ell(P) - \text{short}(P)]$ is at least $\epsilon|f||\ell|_1/D$, where D is the sum of the flows of all paths, i.e. the sum of all demands.

The random path selection technique, first proposed in [7], provides, at least in theory, a major improvement in performance over the deterministic method, which would involve looking at all commodities. One goal of this implementation effort was to evaluate the effectiveness of this method, or, rather, of Goldberg's variant.

3.2.4 Granularity of flow

Recall that rerouting was only guaranteed to achieve a good reduction in the length-sum $|\ell|_1$ when the condition $1.05\alpha\sigma \leq \epsilon$ holds. This condition is problematic in practice. In the algorithm of [7], one assigns α a value such that condition I is guaranteed to hold, and then assigns σ a value so that the condition holds. Taking this seriously would result in too small a σ for practical implementation.

Recall that σ is the amount rerouted in an iteration. There are two disadvantages to σ being small. First, the resulting reduction in $|\ell|_1$ would be too small, so the algorithm would take too long. Second, a small σ would mean that the flow for each commodity would be broken up into too many flow paths. This would especially be a problem if we explicitly decomposed each commodity's flow into distinct flow paths of value σ.

In the implementation, therefore, we made several design decisions intended to alleviate this potential problem. First and foremost, we did not explicitly decompose flows into flow paths of value σ. We maintain each commodity's flow as a collection of flow paths of different values. This means we must select a flow path with probability proportional to its flow value $f(P)$. To do this quickly, we maintain the flow paths in a heap. Using the heap, we can select from K flow paths

one path with the right probability in $O(\log K)$ time. In the same time, we can add a new flow path or decrease the flow path of an existing flow path.

Second, we only increase α and only decrease σ on an as-needed basis. Every so many iterations (proportional to the number of flow paths), we evaluate how well the algorithm is doing. First, we do a check to estimate whether condition II is satisfied (described further in a subsequent section). If the test succeeds, we evaluate the ratio in condition I and double α if that condition is not satisfied. If the test fails, we estimate the average amount by which the sum of lengths has been reduced recently. If the amount is not sufficiently long (compared to the amount predicted by (2), we deem it time for σ to be reduced.

Third, whenever possible we reroute more than σ units of flow. Each time a path is selected for rerouting, we find roughly the largest amount of flow we can reroute from that path to the shortest path such that we still achieve a sufficiently good reduction in the length sum. This tends to keep the flow from getting unnecessarily fragmented into many flow paths.

Fourth, we use a technique from Orlin's min-cost flow algorithm. We inductively maintain that each flow path has a flow value that is a multiple of σ. In particular, this ensures that no flow path has a flow value less than σ. To achieve this, first, when we reduce σ, we reduce it by an integral factor (two), and, second, when we reroute an amount of flow, we make sure that the amount is a multiple of σ, rather than rerouting the absolute most flow we could.

3.2.5 When to terminate

We can evaluate the left-hand side of (1) to determine whether the current solution is nearly optimal. However, this is a somewhat expensive computation; it involves an all-pairs shortest-paths computation. For this reason, we would ideally only carry out this check when we have good reason to believe that the answer is yes.

In the implementation, we use a random technique to estimate whether condition I is satisfied. Namely, whenever a path P is randomly selected, we evaluate the quantity $[(1-\epsilon)\ell(P) - \text{short}(P)]$. We add up the quantities over a series of iterations, and then use the average to estimate the value of the sum in Condition II.

So far, it seems this approach is only partially successful. The estimate is not always a good estimate, depending on the random selections made during the algorithm. The same seems to be true when we sample short(P) instead, thus avoiding subtraction, which accentuates the errors. Our preliminary judgement is that, while this technique provides a useful filter, which reduces somewhat the time required for repeatedly testing termination, it is not a good estimator. More work is needed to find a better filter.

4 Some numbers

At the end of this paper we give data concerning our algorithm running on some graphs. The data given are for running concurrent flow with capacities on the nodes. Thus these experiments indicate roughly the time required to find an approximately sparsest node-separator. For most graphs, there are three commodities per source node, each with a random sink. For newmatrix, each source node has 11 sinks. For outc17, there was a commodity for every pair of nodes.

The graphs beginning with "outc" are ISCAS '85 benchmark circuits [2]. Matrix.tmp and newmatrix.tmp are from sparse matrices used in testing graph chordalization; matrix.tmp is from the

Harwell-Boeing collection of sparse matrices, and newmatrix.tmp is a smaller matrix we devised.

Target approx is a number used as ϵ in the algorithm. The approximation error should be at worst 6 times ϵ, and is typically less.

Times are in seconds on a sparc 2 workstation.

The data indicate, first, that at least for moderate degrees of precision ($\epsilon \geq .025$), the algorithm runs fairly fast. In contrast, note that, for example, to set up outc1908 as a linear program would take $938 \cdot (938 \cdot 3) = 2639532$ node-capacity constraints and the same number again of conservation constraints, since the number of commodities is $(938 \cdot 3)$. This could be reduced by a factor of 3, by combining all commodities with the same source, but that still leaves 1759688 constraints.

Second, the random selection method receives mixed reviews. On the larger graphs (matrix.tmp and above), an average of 7.8% of all trials resulted in reroutings, excluding one particularly good case of 81%. This seems to indicate that the method is reasonably effective, but sometimes the fraction was as low as 3.5%. On a positive note, this indicates a potential for exploiting parallelism. More experiments are needed to show how the percentage varies over the course of the algorithm, and how much time is spent in unsuccessful trials (those that do not result in reroutings).

Third, although we do not include the data in the table below, we determined how much of the time was spent in simply evaluating the approximation error. In most cases the time on evaluating the error was less than 10% of the total time, but in one case in was as high as 18%. This indicates that more work needs to be put into getting a good estimate of the error, so as to avoid unnecessarily computing the error exactly.

Fourth, it is a bit difficult to estimate from these data how the algorithm's performance depends on ϵ. In several cases, the time required for one value of ϵ was not significantly more than that required for twice that error. (In one case, for example, fewer reroutings were needed to achieve an error of .03 than to achieve an error of .11) In the more well-behaved cases, it looks as though a factor-of-two decrease in ϵ resulted in an increase in time by a factor roughly between 2 and 3. This would lead one to conjecture that the dependence of the algorithm's running time on $1/\epsilon$ is somewhat less than quadratic.

graph	:	outc17		
no of nodes:	13			
no of edges:	14			
target approx:	0.1	0.05	0.025	0.0125
achieved approx:	0.272736	0.055321	0.055858	0.068185
no of iterations:	234	819	1287	1548
no of reroutings:	3	54	113	160
time required:	0.083330	0.199992	0.316654	0.933296

graph	:	newmatrix.tmp		
no of nodes:	21			
no of edges:	41			
target approx:	0.1	0.05	0.025	0.0125
achieved approx:	0.012758	0.086711	0.035015	0.045147
no of iterations:	1038	692	4770	1038
no of reroutings:	32	19	116	17
time required:	0.416650	0.283322	1.499940	1.233284

graph	:	matrix.tmp		
no of nodes:	136			
no of edges:	354			
target approx:	0.1	0.05	0.025	0.0125
achieved approx:	0.250779	0.082580	0.027356	0.019093
no of iterations:	3672	11322	14763	58517
no of reroutings:	253	451	903	4137
time required:	9.232964	27.048918	36.098557	130.261459

graph	:	outc499		
no of nodes:	275			
no of edges:	440			
target approx:	0.1	0.05	0.025	0.0125
achieved approx:	0.454109	0.109836	0.030208	0.020795
no of iterations:	3711	13867	14844	50140
no of reroutings:	487	688	525	4420
time required:	19.382558	65.980698	66.297348	596.326172

graph	:	outc1355	
no of nodes:	619		
no of edges:	1096		
target approx:	0.1	0.05	0.025
achieved approx:	0.681885	0.093607	0.071506
no of iterations:	2785	40370	125518
no of reroutings:	2264	5056	9216
time required:	67.213982	521.745789	1520.372559

graph	:	outc1908	
no of nodes:	938		
no of edges:	1523		
target approx:	0.1	0.05	0.025
achieved approx:	0.146848	0.103238	0.068035
no of iterations:	21105	50652	219278
no of reroutings:	2606	3541	18906
time required:	506.646393	994.193542	2391.5

References

[1] S. N. Bhatt and F. T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences 28* (2) (1984).

[2] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran," *Proc. IEEEE Int. Symposium on Circuits and Systems*, Special Session on ATPG and Fault Simulation, June 1985.

[3] Mark D. Hansen, "Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems," *Proceedings, 30th Symposium on Foundations of Computer Science* (1989), pp. 604-609.

[4] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, T. Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height.

[5] anuscript, 1990.

[6] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 726–727, 1990.

[7] P. Klein, C. Stein, and É. Tardos. Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 310–321, May 1990. Revised version (with S. Plotkin added as author) submitted to J. ACM, 1991.

[8] F. T. Leighton, F. Makedon, and S. Tragoudas. Approximation algorithms for VLSI partition problems. Manuscript, 1990.

[9] 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.

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

[11] F. Makedon and S. Tragoudas. Approximating the minimum net expansion: near optimal solutions to circuit partitioning problems. In *Proceedings of the 1990 Workshop on Graph Theoretic Concepts in Computer Science*, June 1990.

[12] Matula, D. W., "Concurrent flow and concurrent connectivity in graphs," in *Graph Theory and its Applications to Algorithms and Computer Science*, ed. Alavi, Y., et al., Wiley, New York (1985), pp. 543-559.

[13] D. W. Matula. and F. Shahrokhi The maximum concurrent flow problem and sparsest cuts. Technical Report, Southern Methodist University, March 1986.

[14] S. Rao. personal communication, 1990.

[15] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In *Proceedings of the 1989 ICALP Conference*, 1991.

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

[17] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37:318 – 334, 1990.