APPROXIMATING A PARALLEL TASK SCHEDULE USING MINIMUM K-CUT

Joel Nelson
Computer Science Department
University of Minnesota, Morris
nels0354@mrs.umn.edu

Mark Logan (Advisor)
Math Department
University of Minnesota, Morris
loganm@mrs.umn.edu

Dian Lopez (Advisor)
Computer Science Department
University of Minnesota, Morris
lopezdr@mrs.umn.edu

Abstract

Presented are two new approximation algorithms for solving two different NP-hard graph problems. The first is the minimum *k*-cut problem. The new algorithm runs within the same time complexity as the current best-known algorithm and may have equal or better performance given any input. The second new algorithm uses the *k*-cut algorithm to approximate a solution to the parallel task scheduling problem. Both of these problems have important applications in the real world. Also discussed are the algorithms that lead up to the new ones, along with their respective applications.

1. Introduction

There are many problems in graph theory that are NP-hard. And many of them have very efficient approximation algorithms with very good performance. Conversely, we can use the graph algorithms that solve problems to optimality in a polynomial amount of time to help us generate these approximation algorithms.

The max flow – min cut problem allows us to perform the Gomory-Hu algorithm which helps us in an approximation algorithm for the *k*-cut problem. The purpose of all of this is to generate an approximation algorithm for the parallel task scheduling problem with communication and latency constraints imposed on it. This will be accomplished by minimizing the amount of inter-processor communication required to execute a job composed of tasks represented by nodes in a rooted acyclic directed graph with exactly one leaf node. We show that a *k*-cut algorithm can provide us with a partitioning of tasks that will allow us to minimize the time that a processor spends uploading and downloading from the network instead of executing a task. Cumulatively, this paper shows us that many graph theoretic problems are somehow linked to one another, no matter how different they may seem to each other.

2. Background

2.1 Graph Theory

A *graph* is a theoretic structure composed of *nodes* or *vertices* that are connected with *arcs* or *edges*. Each arc or edge is connected at each side to exactly one node or vertex in the graph. Both the nodes and edges in graphs may or may not have varying values, sizes, or capacities associated with them. Because graphs are such versatile theoretic tools they can be used to represent many real world structures from communication networks to airline schedules. A *directed graph*, G = (V,A), has arcs that have a head and a tail, meaning that the arc points in a certain direction. An *undirected* graph, G = (V,E), graph has edges that simply connect two nodes with no direction or order implied in the edge. In this paper, if it is not specified whether the graph is directed or undirected, then an undirected graph is what is being used.

There exists a *path* between nodes i and j if and only if node i is *connected* to node j through a series of edges from node i to node j.

A graph has k *components* only if there exists k disjoint subsets of nodes in G such that there exists no edges that connect two nodes in different subsets and every pair of nodes in each subset are connected to each other.

A *rooted graph* is a graph where every node has a directed edge pointing at it except one. In other words, every node, except one, has at least one parent.

An *acyclic graph* is a graph where every node i does not have a path back to itself.
A *leaf node* is a node in a graph that has no directed edges pointing away from it. In other words, it has no children.

In a directed graph, a node j is a *decendant* of node i if and only if there exists a directed path from node i to j.

2.2 Approximation Algorithms

Many problems in graph theory are NP-hard. This means that the optimal solutions can only be determined in an exponential amount of time, unless P = NP. We consider these problems to be incomputable. While it would be great to have the optimal solutions to these problems, we will not be able to generate them within our lifetimes for most reasonable sized input. However, there exist feasible solutions, that aren't optimal, whose solution values are near the optimal value. This means that these solutions are almost as good as the optimal ones. We call them approximate solutions. And fortunately there are many approximation algorithms that can be determined in a polynomial amount of time, which is usually considered to be a reasonable amount of computation time. If an approximation algorithm is to be used widely, it should have a proven bound on its performance. This means that one would prove that his or her approximation algorithm will always generate solutions within a certain factor over the optimal value. So basically, an approximation algorithm sacrifices a small level of accuracy in determining a good solution for a large saving in the amount of computation time.

Approximation algorithms are evaluated on two characteristics. The first is how *efficient* the algorithm is, this is measured in Big-O computational complexity. The other is the quality of the outputted results, or the algorithm's *performance*. This is measured by proving an upper bound on the value of the solution that the algorithm produces. The bound is usually reported as a factor over the optimal value such that every solution produced by the algorithm will always have a value within that factor.

3. Maximum Flow – Minimum Cut

3.1 Maximum Flow

The maximum flow problem consists of a graph G = (V,E) where the V is the set of nodes and E is the set of edges connecting the nodes. We also impose a constraint that each edge has a specific capacity associated with it. To visualize this we can think of this graph setup as a model to a water pipe system. Each edge is a pipe with a certain capacity for water flow per time unit and each node is a specific point in space connecting two edges, like a pipe joint. With this graph model, the problem is to find how much water we can pump through the network of pipes, (ie- the graph), starting at one node in the graph (the source node) and ending at another node (the sink node). And more specifically, we need to find

out how much water flow per time unit we need to send through each pipe. Note that we must abide by the following constraints in order to have a feasible flow of water going through our network:

b_{ij} = the capacity of edge i to j in E
x_{ij} = the flow assignment for each edge i to j in E

$$\sum_i x_{ij} - \sum_k x_{jk} = \begin{cases} -v & \text{if } j = s \\ 0 & \text{if } j \neq s,t \\ v & \text{if } j = t \end{cases}$$

$0 \leq xij \leq bij$ for all i,j in V

This just says is that the sum of all the water flowing into each joint from all the pipes flowing into it is equal to the sum of all the water flowing out of the joint through all the pipes flowing out of it. Except the source and the sink, they send and receive *v* amount of water per time unit, respectively. In this case, *v* is the value of this feasible flow. The maximum feasible flow value, *v*, is the optimal solution to the problem.

3.2 Minimum Cut

The minimum cut problem consists of a graph G = (V,E) where V is the set of nodes and E is the set of edges connecting the nodes. We also impose a constraint that each edge has a specific weight associated with it. Let there be a source node and a sink node. With this graph model, the problem is to find a subset of the edges in E such that removing them from G will cause the source node to be disconnected from the sink node in G and the sum of the edge values in the subset is minimized.

3.3 Similarity between Max Flow & Min Cut

Theorem 1: *The maximal flow value between two nodes, in a graph G, is equal to the minimum cut between the same two nodes in G.*

Ford and Fulkerson proved this in 1956 [3]. They also defined an algorithm to find both the max flow and min cut at the same time in $O(|V|^3)$ time. A high level explanation of the algorithm follows: randomly search for paths from the source to sink nodes in order to find feasible flows between the two nodes. Each time it finds one it modifies the graph to reflect that the potential for flow through that path has been explored. Once the graph has been modified enough so that there exists no more feasible flows from source to sink, the algorithm halts.

3.4 Applications

Both max flow and min cut have many applications. Max flow can be used to solve questions dealing with sewer systems, traffic networks, building evacuations, communications network traffic, and any time one needs to pump items through a network such that there are multiple means by which the items can get from the source to the sink. There are also many applications to other graph theoretic problems, such as the Gomory-Hu algorithm, discussed below. The min cut problem can be used any time something needs to be disconnected from something else when there exists multiple things holding them together. It has fewer real world applications than max cut, by itself, but when used in other graph theoretic problems, such as minimum *k*-cut, which is discussed later, there are numerous applications.

4. Gomory-Hu Algorithm

Theorem 2: *Let G = (V,E). There exists a set of |V| - 1 cuts in G that contains the min cut between every pair of nodes in G.*

Let |V| = n. Notice that if one wanted to easily get all minimum cuts between any two nodes in G, the simple operation of computing all min cuts between the $(n^2 – n) / 2$ possible unordered pairs of nodes in V would complete the task. But Gomory and Hu proved the theorem that says one needs only n – 1 of them in order to minimally separate any pair of nodes in G. More importantly, that they can be obtained through n – 1 max flow computations, instead of the previously implied $(n^2 – n) / 2$ max flow computations. The algorithm to determine the set of the n – 1 max flows referred to in theorem 2 is very advanced and requires a lot of wording and example space, therefore it won't be defined here. It is quite clever and interested readers should refer to [6].

This theorem and algorithm can be applied in many other places in graph theory, specifically it is used in the current best known approximation algorithm for the minimum *k*-cut problem, discussed below. Also, the solution for minimum *k*-cut for fixed k=2 follows trivially from the Gomory-Hu algorithm; just choose the smallest cut.

5. Minimum *k*-Cut

5.1 Statement of Problem

The minimum *k*-cut problem consists of a graph G = (V,E) where V is the set of nodes and E is the set of edges connecting the nodes. Each edge in E also has a weight associated with it. With this graph model, the problem is to find a subset of edges in E such that removing them from G will cause G to have k components and the sum of the edge values in the subset is minimized.

5.2 Applications

The minimum *k*-cut problem was initially formulated to help generate the cutting planes used to solve the traveling salesman problem [2], so it is certainly of use in that respect. There are many other applications as well. One of its most common uses is in VLSI design. Another use is with parallel computing and network partitioning. This is where *k*-cutting helps with the placement of data into distributed memory in a multiprocessor computer where it is desirable to balance computation load with a minimized amount of communication. It is used as a determinant of network reliability, where the value of the optimal solution is directly proportional to the reliability of the network. It can also be used in most cluster type setups where the edge weights designate the similarity between clusters [1], [2]. However, in this paper it will be used to help solve the parallel task scheduling, discussed below.

5.3 Current Best Approximation Algorithm

Since the optimal solution to this problem can only be determined in exponential time, (it is NP-hard [5]), we try to find the next best thing to an optimal solution, a good approximate solution. Fortunately, an approximate solution can be derived for any graph such that the value of the solution will always be within 2 – 2/k times the actual optimal solution. If we let G=(V,E), Saran and Vazirani define this algorithm [11]:

1. Compute the n – 1 min cuts in G using the Gomory-Hu algorithm.
2. Sort the cuts by ascending cut weight and call them $g_1,\ldots,g_{n-1}$.
3. Find the minimum i such that removing all the edges in $g_1,\ldots,g_i$ will cause G to have at least k components.

This algorithm runs in almost the same amount of computation time as the Gomory-Hu algorithm. Using the current best known max flow algorithm [4], it runs in $O(mn^2 + n^{3+\varepsilon})$ where |E| = m and ε is any fixed positive real number.

5.4 Proposed Approximation Algorithm

4.4.1 The 2-cut problem

Let G=(V,E) where |V| = n and |E| = m. The 2-cut problem is simply the *k*-cut problem for fixed k=2. As stated above, the Gomory-Hu algorithm can be used to solve this problem, and since 1961 it was the best known way to compute the optimal 2-cut of a graph. Recall that it would take n – 1 max flow computations to obtain the solution. Then in 1992, Nagamochi and Ibaraki introduced an algorithm to solve it in $O(mn)$ time [10].

5.4.2 The k-2-cut Algorithm

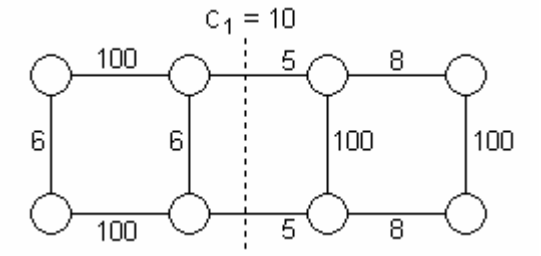
Let G = (V,E) where |V| = n and |E| = m. Introduced is a new *k*-cut approximation algorithm utilizing the 2-cut algorithm:

1. Let i := 1
2. Compute the 2-cut for each component in G, call them $c_1,\ldots,c_i$
3. Find c_j in $c_1,\ldots,c_i$ with the least cut value and let E := E \ c_j
4. Let i := i + 1
5. If i < k Then goto step 3 Else return G

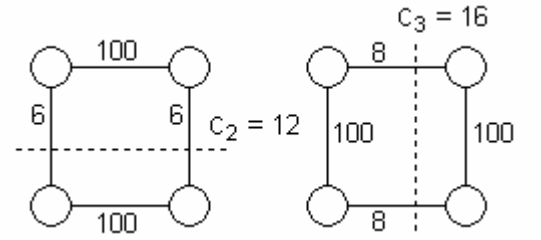
Note: in step 3, the "\" operator references set difference.

5.4.3 Sample Run

Iteration i = 1



Iteration i = 2



Iteration i = 3

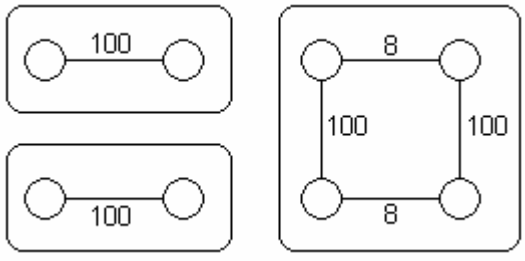


Figure 1. Sample run of the k-2-cut algorithm with k = 3.

Total Cut Weight: $c_1 + c_2 = 22$
Note: c_2 was picked over c_3 because $12 < 16$.

5.4.4 Analysis

5.4.4.1 Complexity

Implemented naively, the algorithm appears to be $O(k^2mn)$. This is because step 2 will occur k – 1 times and each time it is executed, it calls for $O(k)$ number of 2-cuts, (ie- it computes the 2-cut for every component in G, and on average there will be $O(k)$ components throughout execution). But notice that in each iteration we are only introducing 2 new components that haven't had their 2-cut calculated, the rest of the components have already been done in previous iterations. A quick computation lets us know that there will be exactly 2k – 3 calls to the 2-cut algorithm throughout execution, which is equal to $O(k)$. So the total complexity is $O(kmn)$, or k calls to the 2-cut algorithm, hence the name k-2-cut. And since $O(kmn) \cong O(mn^2 + n^{3+\varepsilon})$, the two algorithms run in about the same number of computations.

5.4.4.2 Performance

There hasn't been any testing done on this algorithm, that's why it is still in the proposal phase. But in theory, it should perform as well or better than the algorithm defined by Saran and Vazirani, (the SaVa algorithm), discussed above. This is because the k-2-cut algorithm considers every cut that the SaVi algorithm considers – either as a single cut or as a composite of cuts – but it also considers other cuts with possibly lower cut values. And if there are better cuts to use, the k-2-cut algorithm will use them instead. But keep in mind that this is being presented as a conjecture without proof. However, figure 2 illustrates the cuts that the SaVa algorithm would make on the same graph as the k-2-cut algorithm made in 4.4.3, and it does so with a higher total cut value of $g_1 + g_2 = 26$.

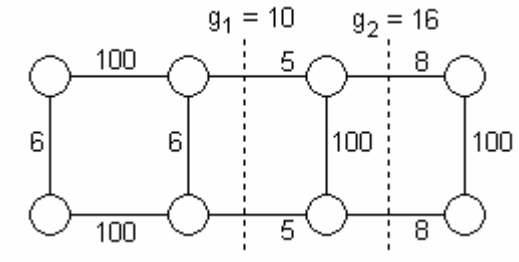


Figure 2. Sample run of the SaVa algorithm with k = 3.

5.5 Modified k-2-cut Algorithm

In the parallel task scheduling problem, discussed below, it may be advantageous to *k*-cut the graph so that the sum of the values of the nodes in each component are as equal as possible. With a slight modification to the k-2-cut algorithm we can give solutions that favor a more balanced set of components, but not guarantee it. It should be noted, however, that the modified algorithm will generate cuts with higher cut value. Also, it will run in the same time complexity as the original.
Let G = (V,E) where |V| = n and |E| = m. Introduced is the modified k-2-cut approximation algorithm:

1. Let i := 1
2. Remove the edges in E that are contained in the 2-cut for the component in G with the maximum sum of node values
3. Let i := i + 1
4. If i < k Then goto step 3 Else return G

6. Parallel Task Scheduling Problem

6.1 High Level Problem Definition

Consider a job composed of multiple tasks, each of which has a varied interdependency on other tasks in the job. Meaning that one task may not be able to start executing until it receives some data from one or more other tasks. Also consider an unlimited network of identical processing entities. The problem is to find out how to allocate tasks to different processors in the network in order to minimize the total computation time. Additional constraints include communication time that it takes to upload and download data to and from the network when communication between processors is required and the latency time that it takes for the data to travel across the network. Groundwork for this problem was established in [7] and [8]. It should be noted at this point that the words node and task are interchangeable, as are graph and job.

6.2 Precise Problem Definition

As many other real world problems, this can also be formulated as a graph problem. Valid input comes in the form of a rooted acyclic directed graph with exactly one leaf node such that there exists a path between the root node and every other node in the graph, call it G = (V,A). Cij is the amount of time it takes to upload and download data for the transmission from task i to task j and L is the latency constant for the network. The optimal solution must constrain to the following:

1) If there exists an edge from task i to task j, task i must finish computation before task j can begin.

2) If there exists an edge from task i to task j and they are scheduled to be executed on different processors then the processor that task i is scheduled on must spend Cij amount of time uploading data to the network starting some time after task i has finished executing. The processor that task j is on must spend Cij amount of time downloading data from he network starting some time that is L amount of time after the corresponding upload started and it must finish before task j can start executing.
3) Two tasks cannot execute simultaneously on the same processor.
4) Processors can't communicate more than one task transmission at one time.
5) Processors can't communicate and execute tasks at the same time.
6) The end time of the execution of the task represented by the leaf node in G is minimized.

This problem is NP-hard, [7], therefore we are interested in finding approximate solutions in a polynomial amount of time.

6.3 Approximation Algorithms

The general goal of an approximation algorithm when solving a problem like this is to somehow summarize large amounts of data about the graph, (ie- the many different combinations of possibilities), so that a small and simple decision can be made. Each algorithm usually has one or two simple theories of how to consider only the possible solutions that are the best. These can be found by looking at the characteristics that many optimal solutions are likely to have for general graphs. We can refer to this as a theory of optimization.

6.3.1 Current Approximation Algorithm

The theory of optimization for this algorithm is that, for each node in the graph, there are a few decisions that the processor currently executing a task can make. For each child that the task has, it can either send it off to another processor or keep it and process the task itself. To make an intelligent decision here, we define the summed execution of each task, i, as the sum of the execution times of every decedent of task i. So the decision of whether to send a child to another processor for processing is made according to the summed execution time. It keeps the task with the smallest summed execution time for itself and sends off all other children to other processors.

This algorithm can be implemented in $O(|V|^2)$ time. Unfortunately there exists no known proof on the performance bound. However, extensive testing suggests that the solution will always be within 10% of the optimal solution within the range of feasible solutions between the optimal and worst [9].

6.3.2 Proposed Approximation Algorithm

The theory of optimization for this algorithm is that communicating is a waste of time. Though we must incur some penalty of communication time in order to realize the benefits of concurrent computation, doing too much communication has the potential for causing a solution value to degrade from optimal. We now notice that if there aren't very many arcs between two sets of nodes that are scheduled to be processed on different processors, there won't be a lot of communication time incurred. So in order to minimize the amount of communication between processors, we can partition the graph into disjoint subsets of nodes such that the sum of the arcs between these subsets is minimized. Unfortunately, though, the optimal solution to the question that we just asked about the disjoint subsets is equal to the *k*-cut problem, which is NP-hard, therefore it can't be used in a polynomial time algorithm. However, the high performance polynomial time *k*-cut approximation algorithms discussed above can be used. So we define the algorithm:

1. Let numClusters := 1
2. Compute the minimum k-cut on G for k = numClusters
3. Determine the level* of each node in G
4. Let p_i := i for $0 \leq i < k$
5. For each level in G: lvl
    a. For each node on level lvl
        i. Let sNode := the node on level lvl that has the maximum sum of arc values going to nodes that aren't in the same cluster
        ii. Let i := the cluster that sNode exists in
        iii. If no other node from cluster i has been scheduled, then
            1. Let arcSum := 0
            2. For each r, such that $0 \leq r < k$ and p_j != r for all $0 \leq j < k$
                a. Let tempArcSum := the sum of the arcs going from cluster r to cluster i
                b. If tempArcSum > arcSum
                    i. Let arcSum := tempArcSum
                    ii. Let p_i := r
        iv. Add sNode to the solution under processor p_i
        v. If all nodes from cluster i have been scheduled Then Let pi := -1
6. If this solution is the best so far then save it
7. Let numClusters := numClusters + 1
8. If numClusters ≤ |V| Then Goto step 2
9. Return the best solution that was found

*Note: The level of a node is equal to the length of the longest path from the root node to that node if all the arc values were equal to 1.

Basically, all this algorithm says is that the i^{th} cluster of tasks will be placed on the i^{th} processor unless there is a cluster that has already been completely scheduled before it, (ie- that processor is not being used any more), then the i^{th} cluster will be put on that processor instead. This saves on wasted communication time between the two clusters. The

algorithm also needs to try building solutions for all possible number of clusters in G, that is to say for each k such that $1 \leq k \leq |V|$, a new feasible solution is built and compared to the rest. This is because we have no way of knowing how many clusters are going to be best for the inputted graph G.

The algorithm runs in $O(n^5)$ or $O(mn^3)$ if $|V| = n$ and $|A| = m$, depending on which *k*-cut algorithm is used. This is because we need to call it n times from step 2, and the rest of the algorithm, which also executes n times, runs in $O(k^2)$ which is less than both of the *k*-cut algorithms.

There is no proof of a performance bound given here. But the algorithm is likely to run the best for inputted jobs that possess cluster structure. That is to say that it will have a higher probability of generating a near optimal solution if the value of the solution given by the minimum *k*-cuts divided by the sum of the arcs in the graph is as near to zero as possible. This ratio being low is important because it means that we were able to split up the tasks in the job so that we will have to do very little communicating in comparison to how much we would have to if the tasks in the same clusters were put on different processors. For this reason, it would be reasonable to restrict the input graphs for this algorithm to those possessing cluster structure. This is because given a perfectly uniform graph, (ie- a graph with zero cluster structure), the algorithm isn't likely to exercise its main theory of optimization very effectively and will then probably not yield solutions near optimal. This could be implemented by calculating the ratio between the average *k*-cut on G and the sum of the edges in G and only allowing input graphs that fall within a certain ratio.

6.4 Applications

This problem can be applied any time a job exists that can be separated out by tasks. All that needs to be done is for the data dependencies to be represented as arcs in the graph and the tasks can be scheduled out onto a network of processing entities using the schedule generated by the approximation algorithm. This could result in the overall running time of the job to finish twice, 10 times, or even 100 times faster than it would if it were to be executed sequentially on a single processor. There are many problems that exist that could benefit from faster overall computation time. An example of this would be weather forecasting, where a lot of data must be calculated within a relatively short amount of time before the predicted event actually occurs.

7. Conclusion

The k-2-cut algorithm used for computing the minimum *k*-cut problem looks very promising. It is very simple, easy to implement, efficient, and has high performance. Future work on this algorithm will certainly be to prove a performance bound. The new approximation algorithm for the parallel task scheduling problem also seems to have a valid theory of optimization – given that the inputted graph has cluster structure. It would also be very desirable to prove a performance bound for this algorithm as well. All of the

algorithms discussed do a good job of solving the problems they were designed to handle. They can also model many real world problems. Finally, we observe that the max flow problem seems to be very different from the parallel task scheduling problem, but they could very well be a lot closer related than we might think. This is probably true of a lot more problems than we even realize.

8. References

[1] R. Battiti and A. Bertossi, "Greedy, Prohibition, and Reactive Heuristics for Graph Partitioning," IEEE Transactions on Computers, Vol. 48, No. 4, April 1999.

[2] M. Burlet and O. Goldschmidt. “A new and improved algorithm for the 3-cut problem.,” Operations Research Letters, 21:225-227, 1997.

[3] Ford, L. R., and D. R. Fulkerson, “Maximal flow through a network,” Canad. J. Math., 8 (3) 1956, 399-404.

[4] A. V. Goldberg and R. E. Tarjan, “A new approach to the maximum flow problem,” Proc. Of the 18th Annual ACM Symp. On Theory of Computing, 1987, pp. 136-146.

[5] O. Goldschmidt and D. S. Hochbaum, “Polynomial algorithm for the *k*-cut problem,” Proc. 29th Annual Symp. On the Foundations of Computer Science, 1988, pp. 444-451.

[6] R. Gomory and T. C. Hu, “Multi-terminal network flows,” J. SIAM, 9 (1961), pp. 551-570.

[7] Hollermann, L., Hsu, T.S., Lopez, D. and Vertanen, K., "Scheduling Problems in a Practical Allocation Model," Journal of Combinatorial Optimization, Vol. 1 (2), pp.129-149, 1997.

[8] Hsu, T.s., Lee, J., Lopez, D.R., and Royce, W., "Task Allocation on a Network of Processors," IEEE Transactions on Computers, Vol. 49, No. 12, pp. 1339-1353, December 2000.

[9] Lopez D., Nelson J., O'Brien D., and Wilfart R., "Algorithm Design and Testing for Parallel/Distributed Processing Models," Proceedings of the Midwest Instruction and Computing Symposium (MICS), Cedar Falls, Iowa, April 2002.

[10] Nagamochi, H., Nishimura, K., and Ibaraki, T., “Computing all small cuts in undirected networks,” Ding-Zhu Du, Xiang-Sun Zhang (Eds.), Lectures Notes in Computer Science, Vol. 834, Springer, Berlin, Algorithms and Computation, ISAAC’94, 1994, pp. 190-198.

[11] Saran, H. and Vazirani, V. V., “Finding *k*-Cuts within Twice the Optimal,” SIAM J. Computing, Vol. 24, No. 1, pp. 101-108, February 1995.

9. Acknowledgements

The author would like to thank Dian Lopez and Mark Logan for all of their help in developing this paper.