Multiple-Way Network Partitioning

Laura A. Sanchis*
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 181
March 1986

Abstract

We present an algorithm for partitioning the cells of a network into an arbitrary number of segments based on a recent 2-way network partitioning algorithm by B. Krishnamurthy [4]. By efficient use of data structures the complexity of the algorithm is shown to increase only linearly in the number of segments in the majority of cases. Through theoretical and experimental methods we show that the concept of "level gain" introduced in [4] becomes more useful as the number of segments increases.

Key Words: network partitioning, graph partitioning

*AT&T Bell Labs Scholar

1. Introduction

This paper deals with the problem of partitioning the cells of a network into two or more disjoint subsets in a way such as to minimize the number of nets that have cells in more than one of the segments of the partition.

This section describes the problem and some of its variants, and compares the network partitioning problem to the graph partitioning problem. The next section is a survey of recent attempts to deal with the problem. Most of these restrict the problem to graph partitioning or to partitioning a network into only two segments. The third section considers the adaptation of one of these algorithms to the more general multiple-way network partitioning problem. The fourth, fifth, and sixth sections describe the adapted algorithm with a complexity analysis. Section seven gives experimental results.

A *network* consists of a set of cells connected by a set of nets. Each net connects two or more cells. Thus a graph is a special case of a network in which each net is an edge connecting exactly 2 cells. A partition of the network is a partition of the cells of the network into disjoint segments. The *cutset* of a partition is the number of nets with cells in more than one of the segments of the partition. The network partitioning problem consists of finding a partition into s segments ($s \geq 2$) such that the size of the cutset is minimized, for a given s. Generally the segments are constrained to be of a certain size or within a range of sizes. (If this were not so then the trivial partition where all cells are in one of the segments would always be the optimum solution).

Variants to the problem which arise in practical situations include assigning costs to the nets, assigning sizes to the cells, or constraining certain cells to lie in a given segment. If costs are assigned to the nets, then the partition with a cutset of minimal cost is sought. If cells have different sizes, then these must be taken into account in calculating and constraining the size of each segment of the partition.

The network partitioning problem cannot be trivially reduced to the graph partitioning problem, as is shown in the following analysis. One can try to convert a network into a graph by making the cells of the network vertices of the graph and connecting two vertices by an edge if the two corresponding cells are connected by a net in the original network. Thus each net of k cells in the network would yield $k(k-1)/2$ edges in the graph. If, however, one attempts to partition this graph so that the smallest possible number of edges will be in the cutset, this will not necessarily yield a correspondingly good partition of the network. This is because two or more edges in the graph model corresponding to the same net in the network will be counted separately when calculating the cutset of the graph partition, while in the network partition they should be counted as only one connection. Thus the graph partitioning algorithm will disregard good partitions from the network's point of view in favor of other partitions whose cutsets contain less edges but possibly belonging to more nets. An example of this is shown in Figure 1. Figure 1a shows a network of 6 cells and 5 nets, and the best partition of this network into two segments, with a cutset of size 2. Figure 1b shows the graph transformation of this network, and the best partition for the graph; this partition gives a cutset of size 3 for the network, which is not optimal.

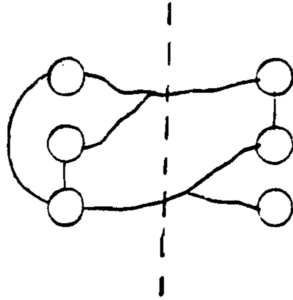


Figure 1a

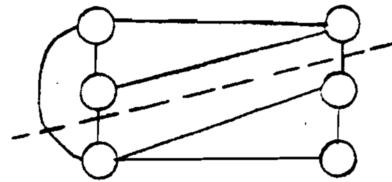


Figure 1b

Another approach would be to assign costs to the edges in the graph in such a way that the total cost of the edges deriving from a single net will add up to one. For instance, if a net is connected to three cells, then the three resulting edges in the graph model would each be assigned cost 1/3. This however also leads to problems, as the graph partitioning may now underestimate the cost of network partitions whose cutsets contain edges from each of many different nets. An illustration of this is shown in Figure 2. Figure 2a shows a network of 8 cells and 3 nets, and the best partition into two segments, with a cutset of size 1. Figure 1b shows the graph transformation, where each of the nets of size 4 is broken up into 6 edges, each with weight 1/6. We thus obtain a partition with a cutset size of 1 (Figure 1b), which includes edges from 2 different nets. This partition would thus be considered as good as the partition whose cutset consists of only 1 edge, even though from the network point of view it is worse.

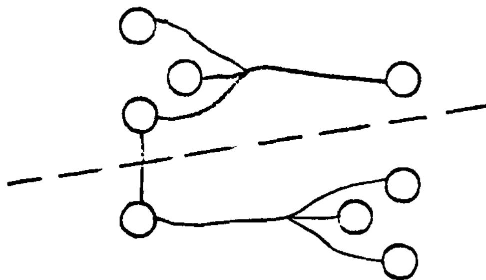


Figure 2a

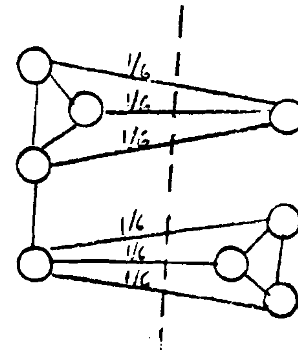


Figure 2b

2. Survey of Solutions

It is known that the graph and the network partitioning problem are NP-hard. (See [1]). Therefore attempts to solve this problem have concentrated on finding heuristics which will yield approximate solutions in polynomial time, or sometimes exact solutions in polynomial time with high probability for certain classes of graphs.

Two general approaches have been taken, namely the development of "constructive" algorithms and of "improvement" algorithms. Constructive algorithms attempt to find a partition that is near-optimal by doing some kind of analysis on the graph or network using graph theory or other applicable methods. Improvement algorithms take a given starting partition and try to optimize it locally by making small changes such as iteratively switching cells from one segment to another. In the following we will look at a few algorithms of the first type and then at a series of papers showing developments in the second approach, which is the one which will be pursued in this paper.

2.1. Constructive Methods

This section will briefly describe some algorithms for partitioning of graphs that have appeared in the literature and which use constructive methods. Except for the clustering procedures mentioned at the end, these algorithms are not easily adapted to networks.

An algorithm for graph partitioning which uses methods in linear algebra and linear programming to find an approximate solution was introduced by Earl R. Barnes in [6].

The problem dealt with is that of partitioning a graph into k segments of given sizes. Very briefly, the matrices A and P are considered, where A is the adjacency matrix of the graph and P represents a partition for the graph. P_{ij} is 1 if cells i and j are in the same segment, and 0 otherwise. Note A is a constant for the problem while P is the variable for which we want to find an optimum value. It is shown that finding a minimum partition is equivalent to minimizing $||A-P||$ where $||C||$ is the Frobenius norm of matrix C. By using the Hoffman-Wielandt inequality this norm is related to an expression involving the eigenvalues of A and P, which eventually reduces the approximation to solving a linear programming problem.

Bui,Chaudhuri, Leighton, and Sipser [7] describe a polynomial time graph partitioning algorithm which is based on network flow techniques. It is however restricted to bisections (i.e. the 2 segments of the desired partition are assumed to have the same sizes). It differs from other approximation algorithms in that instead of generally finding suboptimal solutions, it sometimes fails to find any solution at all; however, when it does find a solution, the solution is optimal. From theoretical and experimental data, is seems the algorithm works well (finds the optimal solution with high probability) for many natural classes of graphs, and in particular for graphs with small optimal bisections and small degree.

Other constructive algorithms are based on clustering techniques which attempt to put strongly connected cells in the same segment. Examples of these may be found in [9].

2.2. Improvement Methods

This section traces the development of improvement algorithms for graph and network partitioning.

Kernighan and Lin [2] described a heuristic procedure for graph partitioning which became the basis for most of the improvement type partitioning algorithms generally used. Their paper concentrated on the problem of producing a partition of a graph with $2n$ vertices (or cells) into 2 segments of n cells each.

The main idea is to start with a random (or not so random) partition and to improve it by iteratively choosing one cell from each of the segments and exchanging them. The cells to be switched are chosen to that a maximum decrease in cutset size may be obtained. Formulas are provided for computing and easily updating the gains or improvements in cutset size that would be obtained by switching any 2 given cells.

The algorithm consists of a series of passes; in each pass 2 cells are interchanged in turn until all 2n cells have been moved. At each iteration the cells to be moved are chosen from among the ones that have not yet been moved during the pass, and in order to produce the maximum possible improvement in cutset size. Note that at a given iteration, all of the possible switches may actually act to increase the cutset size; then the switch which would produce the minimum increase is chosen. At the end of the pass, since all cells have been exchanged, the cutset of the partition should be exactly the same as at the beginning. The n partitions produced during the pass are examined and the one with the smallest cutset is chosen as the starting partition for the next pass. Passes are performed until no improvement in cutset size can be obtained.

Note that for any given partition, there exist 2 sets of k cells, one set in each segment, which if interchanged will produce the optimum partition. It is not possible in polynomial time to determine what these sets are. Each pass in the algorithm just described provides an approximation to this optimum exchange.

The running time of each pass of this algorithm is shown in [2] to be $O(n^2 \log n)$. However, the total running time depends on the number of passes necessary before the process stabilizes. From experimentation, it was determined that the number of passes was almost always between 2 and 4; thus it is claimed the number of passes is independent of n.

This paper also briefly considers extending the method to partitioning into unequal-sized segments and to multiple-way partitions. The addition of "dummy" elements for unequal-sized segments is proposed. The adaptations to multiple-way partitions involve variants of reducing the original problem to several 2-way partition problems.

In a paper by Schweikert and Kernighan [5], an adaptation of the above algorithm for networks was presented. The authors discuss the difference between graph and network models and then show how the Kernighan-Lin algorithm can be easily adapted for nets by changing the manner of computing gains in cutset size resulting from the exchange of two cells.

Fiduccia and Mattheyses [3] gave a detailed analysis of the effects that moving a cell has on the neighboring cells, and from this they were able to devise efficient data structures which permit a linear running time per pass for the network adaptation of the Kernighan-Lin algorithm.

One modification suggested in [3] was to move one cell at a time instead of switching pairs. This allows for more flexibility in segment sizes and also allows implementation of the following idea.

At first glance it appears that it is necessary to search through $O(n^2)$ items to find the best pair of cells to swap each time. In [2] it was proposed to sort the pairs in order to do the search, thus requiring $O(n \log n)$ operations for choosing each pair to be moved. So the total sorting time during a pass was $O(n^2 \log n)$.

In [3] a method is proposed for keeping the candidates in each segment sorted at all times. Let $gain(C)$ be the number of nets by which the cutset would decrease if cell C were moved from one segment to the other. Let p be the largest degree of any cell in the network. Note that $gain(C)$ must be between $-p$ and p (inclusive) for all cells C. Then a bucket array of size $2p+1$ may be maintained indexed by possible gain values: $[-p:p]$. Each entry in this array consists of a pointer to a linked list of cells whose gains are equal to the index. A pointer is maintained to the highest gain value with a non-empty cell list. This structure allows for fast retrieval of the cell(s) with the biggest gain, as well as constant time transferring of a cell to the appropriate bucket when its gain changes.

Finally, the amount of time needed to maintain the gains of the cells during a single pass is analyzed and found to be also linear in the size of the network. This is based on the observation that for each net, the gains of the cells on the net are updated at most 4 times during the pass.

This paper also introduced the idea of preserving balance in the sizes of the segments. Note that since only one cell is moved at a time, the sizes of the two segments cannot be constrained to be constant during the pass. Instead, each segment's size is constrained to lie within a given interval. When choosing the next cell to be moved, the cell with the highest gain in each segment is examined. It will always be possible to move at least one of these cells while preserving balance. If both may be moved, the one with the highest gain is chosen.

Several versions on this type of algorithm have been used for VLSI applications (See [10]).

The latest development in this class of algorithms, which is also limited to 2-way partitioning, comes from a paper by Krishnamurthy [4], where a refinement of the method for choosing the best cell to be moved next is introduced. This version of the algorithm forms the basis for the adaptation to multiple-way partitioning presented in the rest of this paper. A brief introduction to it is given below. See [4] for more details.

Note that if a net contains more than two cells, and if the net is in the cutset of the current partition, then moving one of the cells on this net will not necessarily remove the

net from the cutset; however, it may make it possible to remove the net in a future iteration when another cell on that net is moved. To be concrete, suppose net N connects cells $C1$, $C2$, $C3$, $C4$, and $C5$; and that $C1$ and $C2$ are in segment 1 while $C3$, $C4$, and $C5$ are in segment 2. Moving any one of these cells will not remove net N from the cutset. However, moving cell $C1$ or $C2$ would be better than moving the other cells, since in the next iteration $C2$ or $C1$ might be moved to remove N from the cutset. Although this example is rather simplified, (because in general the other nets the cells are connected to will also play a role in the choice of cell to be moved), it shows how one is led to the concept of different level gains, which is introduced next.

Since the *level gain* concept will be adapted in the algorithm for multiple-way partitioning, we will here give a detailed definition of it, taken from [4].

If S is a set of cells, N is a net, we define

$$\alpha_S(N) = |\{C \mid C \varepsilon S \text{ and } C \varepsilon N\}|$$

$\alpha_S(N)$ is the number of cells on net N which are in set S.

A cell is labelled *free* if it has not yet been moved during the pass; otherwise it is labelled *locked*. Let A be a segment of the partition, A_F the set of free cells in A, and A_L the set of locked cells in A. The *binding number* of a net N with respect to the segment A is

$$\beta_A(N) = \begin{cases} \alpha_{A_F}(N) & \text{if } \alpha_{A_L}(N) = 0 \\ \infty & \text{if } \alpha_{A_F}(N) > 0 \end{cases}$$

The binding number of a net with respect to a segment of a partition indicates how tightly the net is bound to the segment. If there are locked cells on a net in the segment, then the net will be bound to the segment for the rest of the pass, since locked cells can no longer be moved during the pass.

The *ith level gain* of C, $\gamma_i(C)$, is defined as

$$\gamma_i(C) = |\{N \varepsilon N_C \mid \beta_A(N) = i \text{ and } \beta_B(N) > 0\}| - |\{N \varepsilon N_C \mid \beta_A(N) > 0 \text{ and } \beta_B(N) = i-1\}|$$

Note that the first level gain is the actual decrease in cutset size which would result from moving cell C; i.e. the first level gain corresponds to the regular gain concept used in the earlier algorithms.

The gain vector for a cell C is then defined as

$$\Gamma_l(C) = \langle \gamma_1(C), \ldots, \gamma_l(C) \rangle$$

where l is the number of levels used. These vectors are ordered lexicographically. At each iteration, the free cell with the largest such gain vector is the one chosen to be moved next.

In [4] it is claimed that the algorithm using l levels runs in time $O(lm)$, where m is the total number of connection points in the network, which is a measure of the size of the network. (We show in a later section that a factor involving the maximum cell degree also enters into the complexity).

Computing higher level gains enables the algorithm to better distinguish between cells whose first level gains are the same. The concept should be of even greater help in multiple-way partitioning, because the probability that more than one cell will have to be moved in order to remove a net from the cutset will tend to increase with increasing number of segments. This idea is made more precise in a later section where the choice of number of levels to use is discussed.

3. Multiple-way Partitioning Algorithm

In this section the adaptation of the algorithm in [4] to multiple-way partitioning is discussed.

3.1. Terminology

Following [4], we let a network consist of a set of c cells and n nets. For a given cell C, N_C will denote the set of nets incident on C, and n_C will denote the size of N_C. For a given net N, C_N will denote the set of cells on the net N, and c_N will denote the size of C_N. p is the maximal number of nets on any cell and q is the maximal number of cells on any net. m will denote the total number of connection points in the network:

$$m = \sum_{all\ C} n_C = \sum_{all\ N} c_N$$

m may be regarded as a measure of the size of the network.

An s-way partition of the network is described by the s-tuple $(A_1, A_2, \ldots, A_s)$ where the A_i are disjoint sets of cells whose union is the entire set of cells in the network. Each A_i is said to be a segment of the partition.

3.2. Choice of Strategy

In adapting the algorithm in [4] to multiple-way partitioning, the following strategies were considered.

1 Partitioning the initial network into segments A_1 and B_1, using the original algorithm for partitioning into two segments; here A_1 would be constrained to have the size of the desired first segment of the partition. Then in a similar manner partition B_1 into A_2 and B_2, where A_2's size is as desired. Continue this process until the s segments $A_1, \ldots, A_s$ have been obtained.

2 Start with an initial partition of s segments. For each i, i = 1 to s, perform a pass in which pairwise optimization is done by switching cells in A_i with cells in any of the other segments.

3 Start with an initial partition of s segments. At each iteration during a pass, consider all possible moves of each free cell from its home segment to any of the other segments and choose the best such move. Perform passes until no improvement in cutset size is obtained.

The first strategy is similar to one proposed in [2] for multiple-way partitioning of graphs. As mentioned there, a bad choice in the first partitioning will bias the second one, and so on, with the largest errors occurring for large s. Also the first partitioning will try to minimize the number of connections between A_1 and B_1, thus tending to maximize the connections inside B_1, making it harder to obtain a good partition of B_1; and similarly for the subsequent partitions.

The second strategy runs the risk of destroying in one pass the gains made in previous passes and thus having to perform a large number of passes before convergence is obtained.

The third strategy seems to offer the best hope of improving the partition in a homogeneous way, and is the strategy we will adopt in the following.

3.3. Balancing

We will adopt a balancing requirement for the sizes of the segments of the following form (analogous to [3]):

Let $r_1, r_2, ..., r_s$ be such that

$$0 \le r_i \le 1$$

for each i and

$$\sum_{i=1}^{i=s} r_i = 1$$

We want to have the size of A_i close to $r_i c$. (Recall that c is the total number of cells in the network). We choose a parameter $w > 0$ and allow the following range for the size of A_i:

$$r_i c - w \le |A_i| \le r_i c + w$$

That is, the size of A_i may be as much as "w off" from $r_i c$. A cell move from A_i to A_j is allowed if it preserves the above relationship for A_i and A_j. In the case of 2-way partitioning, it is always possible to move a cell in one direction or the other (or both), since if A_1 has more than $r_1 c$ elements, A_2 will have less than $r_2 c$ elements, and viceversa. This result generalizes to s-way partitioning as follows:

Proposition 1: Of the $s(s-1)$ possible cell move directions at least $\left\lceil \frac{s}{2}(\frac{s}{2}-1) \right\rceil$ directions will be compatible with the balance requirements at any one time.

Proof: Let M be the set of segments which may act as sources for a move; that is, those segments A_i whose sizes are strictly greater than $r_i c - w$. Let m be the size of M. Because the sizes of all of the segments must add up to c, and no segment has size greater than $r_i c + w$, we must have $m \geq \left\lceil \frac{s}{2} \right\rceil$.

Likewise, for each element σ in M, there are at least $\left\lceil \frac{s}{2} \right\rceil - 1 = \left\lceil \frac{s}{2} - 1 \right\rceil$ segments which may serve as targets for a move with source σ. This is because there are at least $\left\lceil \frac{s}{2} \right\rceil$ segments which may act as targets for a move, i.e. which have size less than $r_i c + w$, and at least $\left\lceil \frac{s}{2} - 1 \right\rceil$ of these are not equal to σ. Hence at any one time there are at least $\left\lceil \frac{s}{2} \right\rceil \left\lceil \frac{s}{2} - 1 \right\rceil \geq \left\lceil \frac{s}{2} (\frac{s}{2} - 1) \right\rceil$ possible directions for a cell move. □

The above shows that there are $\Theta(s^2)$ legal cell directions at all times. (With more work a slightly better bound of $\left\lceil \frac{s(s-1)}{4} \right\rceil$ can actually be obtained).

3.4. Computation of gains

In this section we will present the gain concept for a cell move adapted to multiple-way partitioning.

Instead of defining the function α as in [4], for convenience we will define φ and λ as follows:

$$\varphi_{A_i}(N) = |\{C \mid C \varepsilon A_i \text{ and } C \varepsilon C_N \text{ and } C \textit{ is free }\}|$$

$$\lambda_{A_i}(N) = |\{C \mid C \varepsilon A_i \text{ and } C \varepsilon C_N \text{ and } C \textit{ is locked }\}|$$

So $\varphi_{A_i}(N)$ is the number of free cells on the net N which are in the segment A_i, while λ_{A_i} is the number of locked cells on the net N which are in the segment A_i. For each segment A_i and each net N, we define β as in [4]:

$$\beta_{A_i}(N) = \begin{cases} \varphi_{A_i}(N) & \text{if } \lambda_{A_i}(N) = 0 \\ \infty & \text{if } \lambda_{A_i}(N) > 0 \end{cases}$$

For multiple-way partitioning we will also now need the function β', defined as follows:

$$\beta'_{A_i}(N) = \sum_{j \neq i} \beta_{A_j}(N)$$

That is, $\beta'_{A_i}(N)$ is the sum of all the binding numbers of net N with respect to all of the segments of the partition except segment A_i; it gives a measure of how tightly N is bound to the "side" of the partition "opposite" A_i.

Finally, we define the ith level gain associated with moving cell C from segment A_j to segment A_k.

$$\gamma_i^k(C) = |\{N \varepsilon N_C \mid \beta'_{A_k}(N) = i \text{ and } \beta_{A_k} > 0\}| - |\{N \varepsilon N_C \mid \beta'_{A_j}(N) = i-1 \text{ and } \beta_{A_j} > 0\}|$$

The first term in the above formula measures the ith level "goodness" of moving cell C from the side of the partition consisting of all segments except A_k, to A_k. The second term measures the ith level "badness" of moving C from A_j to the side of the partition consisting of all segments except A_j. Note that, as is the case for the level gains defined in [4], the first level gain is the actual decrease in cutset size that would result from making the move.

Now we will show that if β, φ, and λ values are maintained for each net N, that $\beta'_{A_k}(N)$ can be computed in constant time (independent of s, the number of segments in the partition).

In [4], a net is defined to be *locked* when it is connected to locked cells in both segments of the partition. Since locked cells can no longer be moved during a pass, once a net is locked it cannot be removed from the cutset during that pass. Note that a net N is connected to locked cells of a segment A_k if and only if $\beta_{A_k}(N) = \infty$. In s-way partitioning, a net is locked as soon as its binding values reach infinity for at least two of the segments of the partition.

For net N define

status(N) = *free* if none of its β values are ∞
status(N) = *loose* if exactly 1 of its β values is ∞
status(N) = *locked* if 2 or more of its β values are ∞

Note that the status of a net can be easily updated each time one of its binding values changes.

Proposition 2: $\beta'_{A_k}(N)$ can be computed in constant time from the α, λ, and β values for N, independent of s.

Proof: If status(N) is free, then there are no locked cells on N. Hence c_N is equal to the sum of all free cells of N on each of the segments. From this it follows that

$$\beta'_{A_k}(N) = \sum_{i \neq k} \beta_{A_i}(N) = \sum_{i \neq k} \varphi_{A_i}(N) = c_N - \varphi_{A_k}(N).$$

If status(N) is not free, then N has a locked cell on at least one of the segments, with a corresponding β value of ∞ on this segment. If $\beta_{A_k}(N) \neq \infty$, then $\beta_{A_i}(N) = \infty$ for some $i \neq k$, implying that

$$\beta'_{A_k}(N) = \sum_{i \neq k} \beta_{A_i}(N) = \infty.$$

The other case to be considered is that in which status(N) is not free and $\beta_{A_k}(N) = \infty$. There are two possibilities. If status(N) is loose, then there are no locked cells on any of the other segments. So $\beta_{A_i}(N) = \varphi_{A_i}(N)$ for $i \neq k$, and

$$c_N = \sum_{i \neq k} \varphi_{A_i}(N) + \varphi_{A_k}(N) + \lambda_{A_k}(N).$$

Hence

$$\beta'_{A_k}(N) = \sum_{i \neq k} \beta_{A_i}(N) = \sum_{i \neq k} \varphi_{A_i}(N) = c_N - \varphi_{A_k}(N) - \lambda_{A_k}(N)$$

If status(N) is locked, then there is at least one locked cell on some segment A_i other than A_k. For this segment $\beta_{A_i}(N) = \infty$, which implies that

$$\beta'_{A_k}(N) = \sum_{i \neq k} \beta_{A_i}(N) = \infty.$$

We have therefore shown that the following algorithm computes $\beta'_{A_k}(N)$:

if status(N) = free then
    $\beta'_{A_k}(N) = c_N - \varphi_{A_k}(N)$
else if $\beta_{A_k}(N) \neq \infty$ then {$\beta_{A_j}(N)$ must be ∞ for some $j \neq k$}
    $\beta'_{A_k}(N) = \infty$
else if status(N) = loose then {$\beta_{A_k}(N) = \infty$, all other β''s for N are not ∞}
    $\beta'_{A_k}(N) = c_N - \varphi_{A_k}(N) - \lambda_{A_k}(N)$
else {status(N) = locked}
    $\beta'_{A_k}(N) = \infty$
endif

□

3.5. Updating Gains

In this section we will investigate the effect a cell move has on the values of the gain vectors of free cells connected to the moved cell. Let l be the highest level gain computed for the cells. Note that since the cell which is being moved becomes locked, its gain values do not need to be maintained after the move.

Suppose cell C is being moved from segment A_j to segment A_k. Consider a single net N connected to C. The move implies that $\beta_{A_j}(N)$ will decrease by one, and $\beta_{A_k}(N)$ will become ∞. This means that $\beta'_{A_h}(N)$ becomes ∞ for all $h \neq k$, and $\beta'_{A_k}(N)$ is decreased by one.

Let D be any free cell distinct from C and connected to N. We will first consider the changes in the gains associated with D caused by the changes in $\beta'_{A_h}(N)$ and $\beta_{A_h}(N)$ for each $h \neq k$. Suppose D is not in A_h. If the old value of $\beta'_{A_h}(N)$, call it i, was less than or equal to l, and if the old value of $\beta_{A_h}(N)$ was greater than zero, then these terms were contributing to the ith level gain of cell D for moving to segment A_h. This contribution is no longer in effect since $\beta'_{A_h}(N)$ is now ∞. Hence 1 must be substracted from the ith level gain of cell D for moving to A_h. In a similar way it may be seen that if D belongs to A_h, and if the old value of $\beta'_{A_h}(N)$ was i-1, with $i \leq l$, and if the old value of $\beta_{A_h}(N)$ was greater than zero, that these terms were contributing (in a negative way) to the ith level gain of cell D for moving to each of the other segments; so the ith level gain of D must be incremented.

Similarly, if D is not in A_k, and if the old value of $\beta'_{A_k}(N)$, call it i, was $\leq l$, and if the old value of $\beta_{A_k}(N)$ was greater than 0, then the ith level gain of moving cell D to A_k must be decremented. Also in this case, however, if the new value of $\beta'_{A_k}(N)$ is greater than 0, its contribution must be added to the $\beta'_{A_k}(N)$ level gain of moving D to A_k. Similar updates may take place in the gains for moving cell D to each of the other segments, if D is in A_k and if $\beta'_{A_k}(N) \leq l$.

It is important to note that none of these updates need take place unless the old value of $\beta'_{A_h}(N)$ was $\leq l$ for $h \neq k$, or unless the old or new $\beta'_{A_k}(N)$ is $\leq l$. This fact will be used in the section dealing with complexity of the algorithm.

4. Data Structures

This section describes the main data structures used by the algorithm.

4.1. Cell Data Structure

The following information needs to be kept and updated for each cell.

1 A list of the nets incident on the cell.

2 The segment to which the cell currently belongs.

3 An indication of whether the cell is free or locked.

4 If the cell is free, the gain values associated with moving the cell to each of the other $s-1$ segments.

5 If the cell is free, $s-1$ pointers to the $s-1$ gain nodes included in the gain structures described in a later section. (These pointers make it easy to update the structures when the gain value of the cell changes).

4.2. Net Data Structure

A table may be maintained with entries for each net. Each entry should contain the following:

1 A list of the cells on the net.

2 The net's current status: free, loose, or locked.

3 φ, λ, and β values for the net corresponding to each of the s segments of the partition.

4.2.1. Segment Data Structure

The number of cells currently in each segment must be maintained.

4.3. Gain Structures

Recall that in [3] a data structure was introduced which allows fast retrieval of the cell with the biggest gain in each segment. This structure consists of a bucket array indexed by the possible gain values for a cell: $[-p:p]$. Each entry in the array consists of a pointer to a doubly-linked list of cells whose gains are equal to the index of the entry. We will call each of the entries in these lists a gain node for the cell; a gain node consists of a cell number, a forward pointer, and a backward pointer. A special maxgain pointer points to the highest index in the array whose list of gain nodes is not empty.

With the introduction of different level gains in [4], each cell has associated with it a gain vector instead of a single gain value. In [4], the following adaptation of the bucket array described above is proposed. If l is the number of levels used, the gain structure may consist of an l-dimensional array, each dimension being indexed from $-p$ to p; thus there is an entry in this array for each of the $(2p+1)^l$ possible gain vector values. Each entry would as before point to a list of cells having that gain vector value.

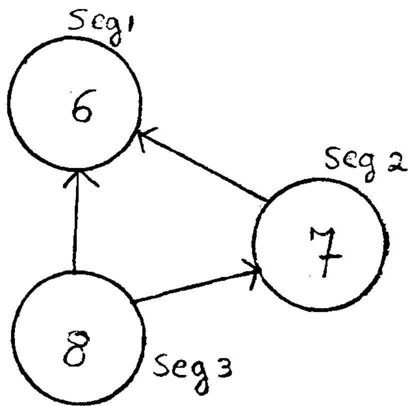
We will return to the issues involved in the implementation of this structure below, but first we will examine the complications introduced in increasing the number of segments.

For 2-way partitioning, one of these structures must be maintained for each of the 2 segments. In s-way partitioning, $s(s-1)$ structures must be maintained, corresponding to the $s(s-1)$ possible directions for a cell move.

At first glance it may seem that $\Omega(s^2)$ steps will now be required to locate the cell that may be legally moved (i.e. moved while preserving balance) with the biggest gain. Since each cell move causes the legal move directions to change (affecting potentially $2(s-1)$ of these directions) it is not sufficient to keep track of the biggest maxgain pointer, since this pointer may correspond at any one time to an illegal move direction. Hence $\Omega(s^2)$ comparisons may have to be made in order to determine the next cell move. What is needed in order to decrease this complexity is to keep a sorted list of the maxgain pointers corresponding to legal move directions.

This may be done by using a binary heap (see [11]) whose entries are maxgain pointers for the currently legal move directions. The pointer with the highest value will be found at the root of the tree. An array indexed by move directions must be maintained holding pointers to the elements in the heap.

Figure 3a shows a network of 21 cells partitioned into 3 segments of sizes 6, 8, and 7. The arrows between the segments show the currently legal move directions. Each of these directions corresponds to an entry in the heap, which is also shown. If a cell move is then made from segment 2 to segment 1, the legal cell move directions and the heap will change as shown in Figure 3b.



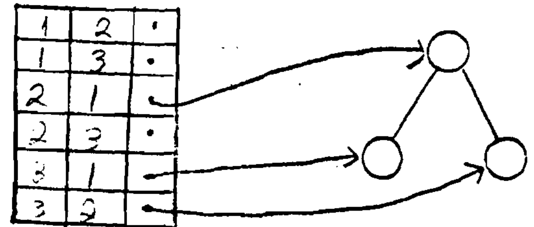
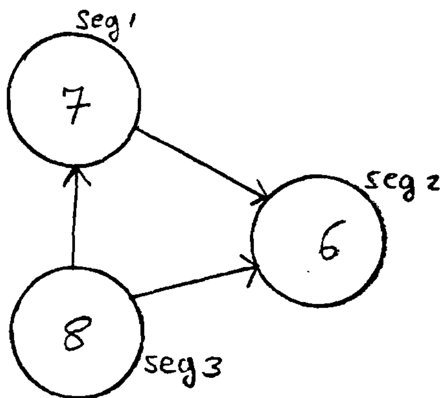


Figure 3a



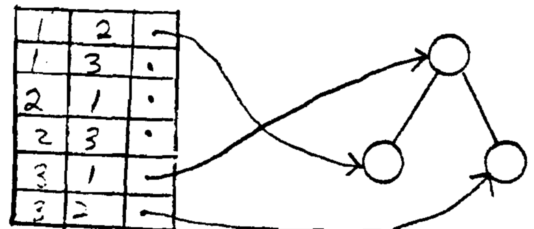


Figure 3b

We will now look at the operations necessary to maintain the heap, and in particular at what happens when a cell is moved from one segment to another. This movement will cause changes in the legal move directions which will in turn cause changes to the heap. Whenever a cell is moved from segment A_i to segment A_j, it is possible that segment A_j becomes full, so that no more cells may be added to it, and that segment A_i becomes too small to remove any more cells from it. This may result in at most $2(s-1)-1$ possible cell move directions which before were legal becoming illegal (namely those involving A_j as target or A_i as source). In the same way it may be seen that the move from A_i to A_j may result in at most $2(s-1)-1$ move directions becoming legal which before were illegal. So at each cell move $O(s)$ insertions and/or deletions will have to be made to the heap to keep it consistent with the legal move directions.

The number of elements in the heap will be $\Theta(s^2)$ (see Proposition 1), hence each insertion and deletion should take $O(\log s^2)=O(\log s)$ time. So the total complexity involved is $O(s\log s)$ for each cell move.

Deletions and insertions to the heap will also need to be made because of changes in cell gains resulting in corresponding changes to the maxgain pointers stored in the heap. As shown later in section 6, there will be $O(lms)$ of these gain updates, hence the complexity involved here in heap maintenance will be $O(lms\log s)$ during an entire pass.

Turning now to the implementation of each single gain structure, we note that the l-dimensional array proposed in [4] will if implemented naively add a factor of $2^l p^l$ to the complexity of the algorithm. The reasons for this are as follows.

Whenever a gain node with highest gain is removed from a gain structure, and it happens that there was only one gain node associated with this highest gain, then the maxgain pointer must be reset. In order to do this it is necessary to search down the bucket array looking for the next non-empty gain node list. The time spent doing this must be included in the complexity of the algorithm.

In [3], where the use of the bucket array was introduced for the single-level algorithm, it is pointed out that the total time spent searching down the bucket array is $O(p+R)$, where R is the sum of all the amounts by which the maxgain pointer is reset upwards during the pass. $R=O(g)$, where g is the total number of gain adjustments performed. Since $g=O(m)$, the total time spent searching down the bucket array is $O(m)$.

For the multilevel algorithm, assuming for the moment 2-way partitioning, this complexity is $O(2^l p^l+R)$. This is because the size of the bucket array is now $O(2^l p^l)$. Moreover in this case $R=O(g2^{l-1}p^{l-1})$, because incrementing a 1st level gain actually adds $((2p+1)^{l-1})$ to the gain vector in the lexicographic ordering. Hence, since $g=O(lm)$ for the multilevel algorithm, the total complexity is $O(2^l p^l+lm2^{l-1}p^{l-1})$. This is higher than the desired $O(lm)$ complexity for the entire algorithm.

However, by adding more pointers to the l-dimensional bucket array, the complexity of searching down the bucket array may be reduced to $O(ml(p+l))$. This is done as

follows.

For each level i, $0 \leq i \leq l$, define the *ith* level hyperplane induced by the constants $a_1,...,a_i$, where each a_j is between $-p$ and p, to be the set of entries of the array whose first i indices are $a_1,...,a_i$. For example, if $i=2$ then the bucket array is a matrix, the 0th level hyperplane is the whole array or matrix, the 1st level hyperplanes are the rows of the matrix, and the 2nd level hyperplanes are the entries of the matrix. For $0 \leq i < l$, the *ith* level hyperplane is composed of $(2p+1)$ $(i+1)st$ level hyperplanes, which will be referred to as its component hyperplanes.

For each *ith* order hyperplane, where $0 \leq i < l$, we define a minpointer and a maxpointer which will point respectively to the first and last of its component hyperplanes (in lexicographic order) which are not empty. Because there is 1 0th level hyperplane, $2p+1$ 1st level hyperplanes, $(2p+1)^2$ 2nd level hyperplanes, etc., this entails $O(2^{l-1}p^{l-1})$ pointers.

Each time a gain node is inserted or removed from the structure, some of these pointers may have to be adjusted. Suppose an entry of the array pointed to by an $(l-1)st$ maxgain pointer becomes empty. If the corresponding mingain pointer does not point to the same entry, then it will only be necessary to traverse at most $2p+1$ entries to reset the maxgain pointer and no lower order pointers need be adjusted. If on the other hand the two pointers were equal, then they are both set to null and the pointers belonging to the enclosing $(l-2)st$ hyperplane must be examined and possibly updated. Following this line of reasoning it is not difficult to see that a traversal of (at most) $2p+1$ entries or pointers need only take place at one of the levels, so the time required is $O(p+l)$. Hence the time required to delete a gain node is $O(p+l)$. The time required for insertion of a gain node is $O(l)$ since at most 2 pointers will have to be adjusted at each of l levels and no sequential search is necessary.

The only problem arises in the initialization of the $O(2^{l-1}p^{l-1})$ pointers. However, this initialization may be performed on line, as it were, as gain nodes are inserted into the structure. For a hyperplane whose entries are all empty, it is not necessary to maintain values for pointers of its component hyperplanes. At the beginning of the pass all entries of the array are empty and hence only two pointers need to be initialized, corresponding to the single 0th level hyperplane (which is actually the whole array). As gain nodes are inserted into the structure, more pointers are initialized. Each insertion will require at most the initialization of $2(l-1)$ pointers. So the time required to insert a gain node is still $O(l)$.

Returning to the case of varying number of segments, we can now prove the following:

***Proposition* 3**: Insertion of a gain node requires $O(\log s+l)$ time; deletion of a gain node requires $O(\log s+l+p)$ time.

Proof: The log s term comes from the possible update to the maxgain heap caused by a change in the maxgain pointer for the structure. The other terms were discussed in the

preceding discussion. □

Note that the above analysis indicates that for large values of $2^l p^l$, only a small fraction of the array is actually used. In our implementation we ran into memory allocation problems for this array, so we devised another data structure which requires less space. Although as shown below the time complexity is higher, in practical situations it may be a more expedient implementation in terms of memory requirements.

The more space efficient data structure involves l levels of one-dimensional bucket arrays, each bucket array consisting of $2p+1$ entries. At level 1 there will be one bucket array indexed from $-p$ to p. Each of the entries in this array will either be null or will point to a bucket array at level 2. The non-null entries in the bucket arrays at level 2 will point to bucket arrays at level 3, etc.. A bucket array at the last level will consist of pointers to gain nodes. Thus in order to find the gain node corresponding to a given gain vector, it is necessary to traverse l pointers.

Each bucket array will have a maxgain pointer and a mingain pointer indicating the highest and lowest indices whose entries are non-empty. Thus because of the lexicographic ordering of the gain vectors, in order to find the gain nodes corresponding to the maximum gain, it is only necessary to follow l maxgain pointers.

Bucket arrays are allocated and deallocated as needed. Since there are no more than c different gain nodes per structure, there will be no more than c bucket arrays at each level at any one time, for a total of at most cl bucket arrays. (In fact there will be less at the lower levels, since there is only one bucket array at level 1, at most $2p+1$ at level 2, etc.). So the space required is $O(clp)$.

Now accessing a gain node with highest gain will take $O(l)$ time. Removing a node may cause a downward search to look for the next nonempty bucket at at most 1 of the levels, so the complexity of removing a gain node will be $O(l+p)$. However inserting a gain node may take $O(pl)$ if all l buckets have to be newly allocated.

5. Description of the Algorithm

Following is a description of the algorithm for s-way network partitioning.

After the network is initialized and a starting partition is obtained, passes are performed until no more improvement in cutset size results. Network initialization consists of reading in a description of the network and initializing the lists of cells associated with each net and the lists of nets associated with each cell. The network description may for instance be in the form of a list of net numbers, each followed by the numbers of the cells in the net. A partition P is assumed to be an array indexed by cell number specifying which segment each cell should be in. The starting partition is assumed to be read in from a file.

Partition_Network
1) do network initialization
2) obtain a starting partition P

3) repeat
    call Init_Partition to initialize partition P
    call Do_Pass to perform a pass and return new partition P
  until there is no improvement in cutset size

Initializing the partition consists of assigning each cell to its corresponding segment and then initializing the gain values and gain structures for the partition.

Init_Partition(P)
  1) for each cell C
      let A_k be the segment specified by P for C
      put C in A_k
      for each net N incident on C
        increment $\varphi_{A_k}(N)$
        increment $\beta_{A_k}(N)$
      end for
    end for
  2) for each net N
      for each segment A_k
        if $\beta'_{A_k}(N) \leq l$ and $\beta_{A_k} > 0$
          for each cell C on net N
            call Update_Gain(C,A_k,N)
          end for
        end if
      end for
    end for
  3) for each cell C
      let A_k be the segment to which C belongs
      for each segment $A_i \neq A_k$
        create gain node for C's gain in moving to A_i
        insert gain node in appropriate gain structure
      end for
    end for
  4) initialize maxgain pointer heap

An explanation for the code in Update_Gain and Reverse_Update_Gain was provided in a preceding section.

Update_Gain(C,A_k,N)
  let A_j be the segment to which C belongs
  if $A_j \neq A_k$

set i to $\beta'_{A_k}(N)$
increment ith level gain for moving cell C to A_k
else if $\beta'_{A_k}(N) < l$
set i to $\beta'_{A_{k(N)}}+1$
for each segment $A_h \neq A_j$
decrement ith level gain for moving cell C to A_h
end for
end if

Reverse_Update_Gain(C,A_k,N)
let A_j be the segment to which C belongs
if $A_j \neq A_k$
set i to $\beta'_{A_k}(N)$
decrement ith level gain for
moving cell C to A_k
else if $\beta'_{A_k}(N) < l$
set i to $\beta'_{A_k}(N)+1$
for each segment $A_i \neq A_j$
increment ith level gain for
moving cell C to A_i
end for
end if

Each pass consists of repeatedly moving a free cell with highest gain until no more moves are possible. Once moved a cell becomes locked. The partition with the lowest cutset which was found during the pass is returned as the new partition.

In order to keep track of the best partition a list is maintained of the moves which are performed during the pass. Together with each move a record is kept of the gain in cutset size which has been obtained from the beginning of the pass up to that move. A pointer is kept to the move with the largest such gain; this pointer is updated as necessary. At the end of the pass, all moves up to the pointer are applied to the partition in effect at the beginning of the pass. This new partition is returned as the result of the pass.

Do_Pass(P) {P is the current partition}
1) set Move_array to be empty
set Gain_array to be empty
set Best_gain_pointer to 0
2) While (a move is possible)
a) get Nextmove from gain structures
b) call Make_Move to perform Nextmove

```
        c) add Nextmove to Move_array
           add new gain to Gain_array
           if new value in Gain_array is better than the one pointed
                              to by Best_gain_pointer
           update Best_gain_pointer
      end While
   3) use moves in Move_array up to Best_gain_pointer to update P
      return(P)
```

Following are the instructions for Make_Move. The criteria for updating gain values for the affected cells were given in a previous section.

```
Make_Move(Nextmove)
  1) C  = Nextmove.cell
     A_j = Nextmove.source
     A_k = Nextmove.target
     lock cell C and remove from gain structures
  2) for each net N connected to C
     a) for each segment A_h
          if β'_{A_h}(N) ≤ l and β_{A_h}(N)>0
            for each free cell D on net N, D ≠ C
              call Reverse_Update_Gain(D,A_h,N)
              update D's gain nodes
            end for
          end if
        endfor
     b) update φ_{A_j}(N), λ_{A_k}(N), β_{A_j}(N), and β_{A_k}(N)
     c) if β'_{A_k}(N) ≤ L and β_{A_k}(N) > 0
          for each free cell D on net N, D ≠ C
            call Update_Gain(D,A_k,N)
            update D's gain nodes
          end for
        end if
     end for
```

6. Complexity Analysis

We will now show that the algorithm described in the preceding section has time complexity $O(lms(\log s+l+p))$.

Proposition 4: During each execution of Do_Pass, the inner loops of Make_Move where Update_Gain and Reverse_Update_Gain are called are executed at most $l+1$

times for each net N in the network.

Proof: Note that these loops are performed only when a cell on net N is moved and one of the old or new values of β' for N are less than or equal to l.

If the first l cell moves when the conditions for executing the loops are met include two moves to two different segments, then the net N will become locked after these two moves. Its β' values will all equal ∞ and will no longer change during the pass; hence the conditions for the inner loops to be performed will no longer be met during the pass.

Suppose on the contrary that there are l moves to a single segment A_k involving cells on net N where at least some of the β' values are less than or equal to l. Note that in fact, after the first such move, all β' values for N will be ∞ except for $\beta'_{A_k}(N)$, which will decrease by 1 at each move. Hence, since the conditions are not met while $\beta'_{A_k}(N)>l$, the conditions for executing the inner loops will be met at most l times, after which $\beta'_{A_k}(N)$ will be 0. At that time all cells on net N will be in segment A_k and any further move of a cell on net N will take a cell to another segment, thus locking the net and setting β'_{A_k} to ∞. Thereupon the conditions for executing the loop will no longer be met. □

Proposition 5: The algorithm described in the preceding section has time complexity $O(lms(\log s+p+l))$.

Proof: Network initialization, step 1 of the main procedure in the algorithm, is independent of the number of segments and may be performed in $O(m)$ time, if the network description read in consists of net lists or cell lists. Step 2 may clearly be performed in $O(c)\leq O(m)$ time.

Following [2], [3], and [4], we will assume that the number of times the loop in step 3 is performed (i.e. the number of passes performed by the algorithm) is $O(1)$.

In the procedure Init_Partition, step 1 is performed in $O(m)$ time. Each call to Update_Gain requires either $O(s)$ or constant time, depending on whether the input cell C is in the input segment A_k. Hence step 2 of Init_Partition should take $O(ms)$ time. From Proposition 3, we know each insertion of a gain node takes $O(\log s+l)$ time. So step 3 of Init_Partition requires $O(cs(\log s+l))$ time. Step 4 of Init_Partition will take $O(s^2) \leq O(ms)$ time. Hence the Init_Partition procedure requires at most $O(ms(\log s+l))$ time.

We will now examine the complexity of Do_Pass. Step 1 takes constant time. Since a cell becomes locked after it is moved, the while loop in step 2 is performed at most c times. Choosing the next move requires $O(s\log s)$ time. Hence all move selections during execution of the while loop can be performed in $O(cs\log s)\leq O(ms\log s)$ time.

Step 2b in Do_Pass consists of calls to Make_Move. Step 1 of Make_Move can be performed in $O(s(\log s+p+l))$ time since each free cell has a node in only $s-1$ of the

gain structures, corresponding to each of the $s-1$ directions in which it can move. Hence, during all calls of Make_Move in a pass, the time spent in step 1 will be $O(cs(\log s+p+l)) \leq O(ms(\log s+p+l))$. The outer for loop in step 2a of Make_Move is executed s times. However, by Proposition 4, the inner loop is executed at most $l+1$ times for each net in the network during a single pass. Moreover, Reverse_Update_Gain takes $O(s)$ or constant time depending on whether the input cell C belongs to the input segments A_k. The updating of a gain node, which takes $O(\log s+p+l)$ time, is done only each time a gain is updated in Reverse_Update_Gain. Hence step 2a will take $O(lms(\log s+p+l))$ time during the entire pass. Similar arguments may be made to show that step 2c takes $O(lms(\log s+p+l))$ time during one execution of Do_Pass. Step 2b requires constant time each time it is performed, and it is performed at most $O(m)$ times during a pass. So we have shown that calls to Make_Move take up $O(lms(\log s+p+l))$ time during one execution of Do_Pass.

Step 2c in Do_Pass takes constant time each time it is executed. So step 2 of Do_Pass has time complexity $O(lms(\log s+p+l))$. Finally it is easy to see that step 3 of Do_Pass has complexity $O(c)$. Hence Do_Pass can be performed in time $O(lms(\log s+p+l))$.

This completes the proof that the entire algorithm may be performed in time $O(lms(\log s+p+l))$. □

7. Experimental Results

A version of the algorithm has been implemented in C. Experiments were performed using nets artificially created to have a certain distribution of net sizes. As was done in [4], we randomized arbitrary choices and performed a number of runs. Figure 4 shows the results of these runs applied to a net having 300 cells, 300 nets, with $q=8$ and $p=9$. The net sizes come from a uniform distribution between 2 and 8. Runs were made for $s=2,3,4,5$ and $l=1,2,3,4$. Each line shown in the table corresponds to 40 runs, and gives the average, minimum, and maximum final cutset sizes obtained from the runs.

No Segments	Level	Average Cutset	Minimum Cutset	Maximum Cutset
2	1	161.44	154	170
2	2	156.45	150	164
2	3	157.87	154	167
2	4	156.60	153	166
3	1	188.92	179	197
3	2	181.45	174	189
3	3	180.12	173	186
3	4	178.12	171	184
4	1	216.22	207	225
4	2	193.85	187	204
4	3	188.07	183	194
4	4	189.27	183	195
5	1	224.22	217	232
5	2	196.90	191	204
5	3	191.80	187	200
5	4	193.17	188	199

Figure 4

The following observations can be made. For each s there seems to be a level after which no more or negligible improvement is obtained. This level appears to be 2 for $s=2$ and 3, and 3 for $s=4$ and 5. Moreover, the improvement obtained, up to this level, is greater for greater number of segments. This is shown more clearly in Figure 5, where the average normalized cutset size is plotted against level for each s. Normalized cutset size is defined to be $\frac{ct-ct_{\min}}{ct_{\max}-ct_{\min}}$, where $ct_{\min}$ and $ct_{\max}$ are the smallest and largest cutset sizes obtained for all of the runs involving the given number of segments s, and ct is the average cutset for each different l and s. Note that the slopes of the curves get steeper as the number of segments increases, indicating the greater effect of higher levels for greater number of segments.

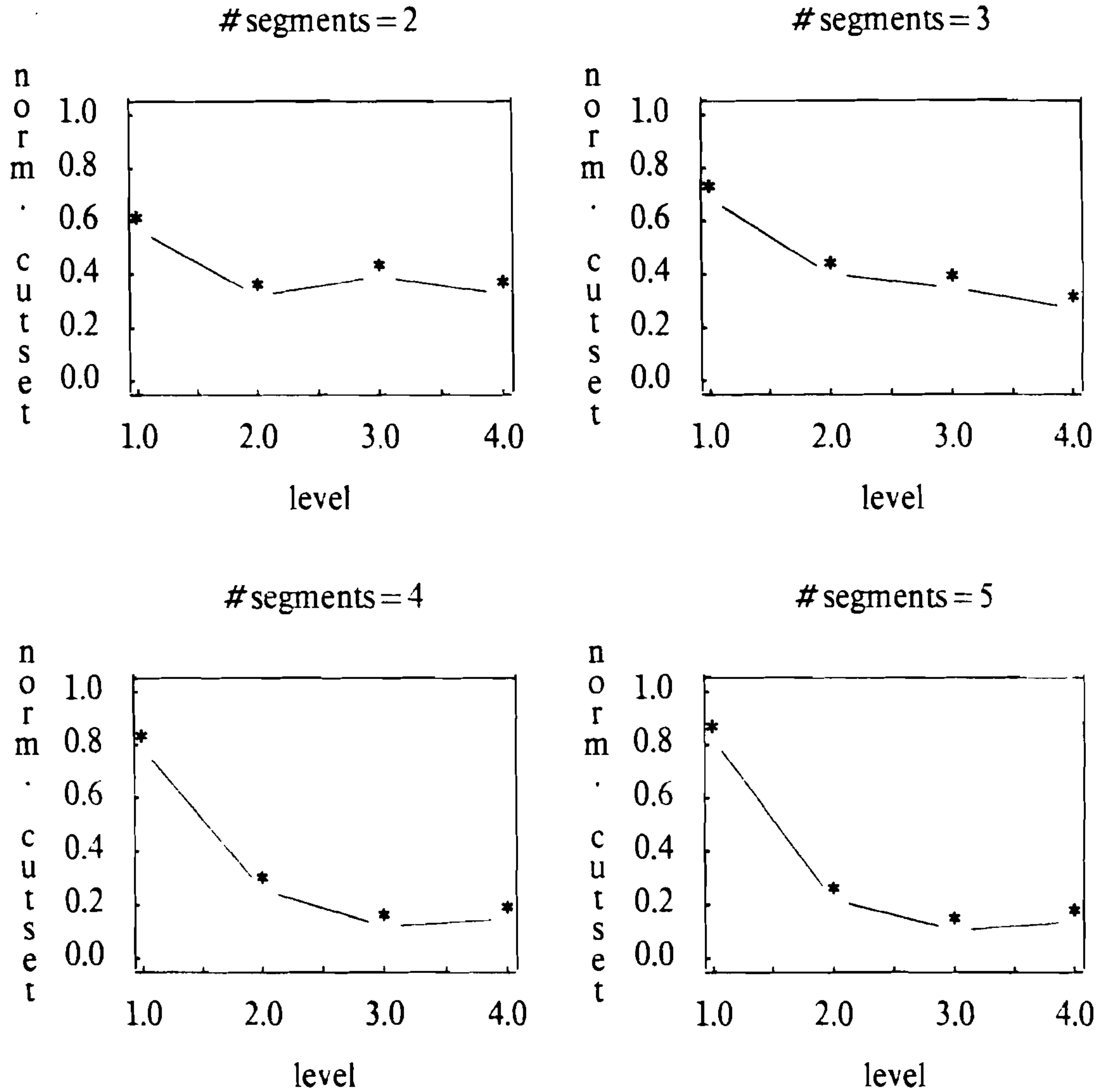


Figure 5

In [4] a method is presented for choosing the optimal number of levels, l, to use for a particular network. Briefly, it is assumed that the $(2p+1)^l$ possible gain vector values are uniformly distributed over the c cells, and l is chosen such that exactly one cell will tend to have a gain equal to the maximum at any time; this is done by setting $1 = \frac{(2p+1)^l}{c}$, which implies $l = \frac{\log c}{\log (2p+1)}$.

Generalizing the formula to an arbitrary number of segments, we obtain $l=\frac{\log c+\log(s-1)}{\log(2p+1)}$ since each cell will have $s-1$ gain vectors corresponding to moves to each of the other $s-1$ segments. This indicates a slight increase in the optimal level with increasing number of segments. We computed this formula for our network for $s=2,3,4,5$, and obtained the values 1.94, 2.17, 2.31, and 2.41, respectively. These are fairly consistent with the experimental results for $s=2$ and 3 but the slight increase in the numbers does not seem to reflect the increased usefulness of level 3 for $s=4$ and 5.

Looking at the situation from another point of view, we may consider the following probabilistic analysis. Consider a net of size r. Assume that each cell on the net has equal probability $\frac{1}{s}$ of being in any one of the s segments, thus inducing a multinomial distribution for the cell locations. Let K be the minimum number of cells that would have to be moved in order to put all cells in a single segment and thus remove the net from the cutset. K is thus the minimum positive level gain that this net could induce on any of its cells. For $0 \leq k \leq r$, the probability that $K \geq k$, i.e. the probability that at least k cells will have to be moved, equals the probability that no segment contains more than $r-k$ of the cells. This is because the way to move the least number of cells is to put all cells in the segment which contains initially the largest number of cells.

Now the probability that no segment contains more than j cells is

$$P1(r,s,j) = \sum_{\substack{0 \leq r_i \leq j \\ r_1+r_2+\cdots+r_s=r}} \frac{r!}{r_1!\ldots r_s!}\left(\frac{1}{s}\right)^r$$

$$= \sum_{0 \leq r_1 \leq j} \frac{r!}{r_1!(r-r_1)!}\left(\frac{s-1}{s}\right)^{r-r_1}\left(\frac{1}{s}\right)^{r_1} \sum_{\substack{0 \leq r_i \leq j \\ r_2+r_3+\cdots+r_s=r-r_1}} \frac{(r-r_1)!}{r_2!\ldots r_s!}\left(\frac{1}{s-1}\right)^{r-r_1}$$

$$= \sum_{0 \leq r_1 \leq j} \frac{r!}{r_1!(r-r_1)!}\left(\frac{s-1}{s}\right)^{(r-r_1)}\left(\frac{1}{s}\right)^{r_1} P1(r-r_1,s-1,j)$$

for $s>1$, while for $s=1$,

$$P1(r,1,j) = 0 \text{ if } j<r$$

$$P1(r,1,j) = 1 \text{ if } j \geq r$$

Then the probability that at least k cells will have to be moved is

$$P2(r,s,k) = P1(r,s,r-k)$$

Using the above formulas values were computed for $P2(r,s,k)$ for $2 \leq r \leq 8$, $2 \leq s \leq 5$, $0 \leq k \leq r$. It was found that for all cases $P2(r,s+1,k) > P2(r,s,k)$. Moreover the expected number of cells that would have to be moved also increases with increasing s. This also indicates that for each level l, increasing the number of segments increases the

probability that gains of level l will be required. Of course the above analysis does not hold unless the uniform cell distribution assumption is a good approximation to the cell distribution in a particular network partition.

Finally, we computed the expected number of cells that would have to be moved to remove a net from the cutset, assuming a uniform distribution of net sizes between 2 and 8. The numbers we obtained are 1.63, 2.23, 2.56, and 2.77 for $s=2,3,4,5$, respectively. These numbers seem to give a slightly better approximation to the best level to use for our particular network.

8. Conclusions

We have presented an adaptation of the network partitioning algorithm in [4] to multiple-way partitioning, whose time complexity is $O(lms(\log s+p+l))$. As the original algorithm has complexity $O(lm(p+l))$ this represents only a linear increase in s in the majority of cases. Through both theoretical and experimental results we have indicated the increasing usefulness of higher levels for increasing number of segments.

One area for further investigation involves finding an effective way of choosing the level to be used based on both the network parameters and the number of segments in the partition. Further work along the lines of the probabilistic analysis in a previous section may prove useful in this regard.

9. Acknowledgments

The author would like to thank Mandayam Srinivas for his support and very helpful suggestions and comments on various drafts of this paper. Thanks are also due to Balakrishnan Krishnamurthy for useful ideas and discussions on which some of the modifications to the gain data structures are based.

[1] M.R. Garey and D.S. Johnson, *Computers and Intractability*, Freeman, San Francisco, CA., 1979.

[2] B.W. Kernighan and S. Lin, " An Efficient Heuristic Procedure for Partitioning Graphs", *Bell Systems Technical Journal 49*, (February 1970), 291-307.

[3] C.M. Fiduccia and R.M. Mattheyses, " A Linear-time Heuristic for Improving Network Partitions", *Proc. 19th Design Automation Conference*, , 1982, 175-181.

[4] B. Krishnamurthy, " An Improved Min-Cut Algorithm for Partitioning VLSI Networks", *IEEE Transactions on Computers 33*, (May 1984), 438-446.

[5] D.G. Schweikert and B.W. Kernighan, " A Proper Model for the Partitioning of Electrical Circuits", *Proc. 9th Design Automation Workshop*, , June 1979, 57-62.

[6] E.R. Barnes, " An Algorithm for Partitioning the Nodes of a Graph", *IBM Watson Research Center Department of Computer Science*, , February 1981.

[7] T. Bui, S. Chaudhuri, T. Leighton, and M. Sipser, " Graph Bisection Algorithms With Good Average Case Behavior", *Proceedings of the 25th annual Annual Symp. on Foundations of Computer Science*, , October 1984, 181-191.

[8] M.K. Goldberg, M. Burstein, " Heuristic Improvement Technique for Bisection of VLSI Networks", *IBM Watson Research Center*, , July 1983.

[9] M. Burstein, " Partitioning of VLSI Networks", *IBM Watson Research Center*, , November 1981.

[10] M.A. Breuer, " A Class of Min-Cut Placement Algorithms", *Proc. 14th Design Automation Conference*, , 1977, 284-290.

[11] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.