New Faster Kernighan-Lin-Type Graph-Partitioning Algorithms*

Shantanu Dutt

Department of Electrical Engineering
University of Minnesota, Minneapolis, MN 55455

Abstract: In this paper we present a very efficient graph partitioning scheme **Quick_Cut** that uses the basic strategy of the Kernighan-Lin (K-L) algorithm to swap pairs of nodes to improve an existing partition of a graph G. The main feature of **Quick_Cut** is a "neighborhood search" strategy that is based on the result (obtained in this paper) that it is not necessary to search more than a certain subset of d^2 node pairs to find the node pair with the maximum swap gain. Here d is the maximum node degree of G. We also use an improved data structure, viz., balanced trees, to store the nodes in the two partitions. Due to the new search strategy and data structure, **Quick_Cut** has a worst-case time complexity of $\Theta(\max(ed, e\log n))$, and an average-case complexity of $\Theta(e\log n)$, where e is the number of edges of G. The K-L algorithm, on the other hand, has a time complexity of $\Theta(n^2\log n)$, where n is the number of nodes of G. Another contribution of this paper is the presentation of efficient node-pair scanning methods for obtaining the best node pair to swap in the K-L algorithm. We have implemented **Quick_Cut**, K-L, and its various enhancements suggested here. The results obtained from using these different algorithms to partition a number of random and other well-defined graphs show that **Quick_Cut** is the fastest, and that it is faster than the K-L algorithm by factors of 5 to 50. Thus **Quick_Cut** can serve as the basis for very fast VLSI partitioning and layout tools.

1 Introduction

Min-cut graph partitioning is a problem that is naturally encountered in VLSI layout of circuits and in packaging and board layout of multiprocessor systems. While laying out circuits on VLSI, the attempt is always to distribute the wires in the circuit uniformly over the wafer, and have as few long wires as possible in order to minimize the total wiring area. To do so, the graph G representing the circuit (wherein the nodes of G are components of the circuits, and the edges of G are the wires of the circuit) is recursively partitioned into two equal size partitions P_1 and P_2, so that the total cost of the edges between P_1 and P_2 is minimized. More specifically, let $G(V, E)$ be a graph with node set $V(G)$ and edge set $E(G)$. There is a positive cost $c(\{v_i, v_j\})$ associated with every edge $\{v_i, v_j\} \in E(G)$ that may, for example, represent the width of the corresponding link. The problem is to partition $V(G)$ into partitions P_1 and P_2 so that $-1 \leq |P_1| - |P_2| \leq 1$, and the cost of the *cut-set* $\sum c(\{v_i, v_j\})$ is minimized, where v_i and v_j belong to different partitions. The net effect of this is that nodes that are densely connected to each other are placed near each other, and thus there are more short wires and fewer long wires in the layout. In a recent survey paper by Shahookar and Mazumder [7], it has been reported that min-cut graph partitioning is the most cost-effective method available for VLSI placement compared to other methods like simulated annealing and genetic algorithms. Efficient min-cut graph partitioning is thus an important area in VLSI layout.

The min-cut graph partitioning problem is NP-complete [4], and polynomial time heuristics have to be used to solve the problem with reasonable effectiveness. A basic heuristic used by most graph-partitioning algorithms is to start with arbitrary partitions $P_1^{'}$ and $P_2^{'}$ whose sizes satisfy the above mentioned constraint, and then to modify the partitions by swapping subsets $S_1 \subset P_1^{'}$, and $S_2 \subset P_2^{'}$, which are not necessarily of equal size, so that the cost of the final partition is reduced, and their size constraint is satisfied [2, 3, 5, 6]. For example, in the Kernighan-Lin (K-L) algorithm [5], $|S_1| = |S_2| = 1$, while in [3] and [6], $|S_1 \cup S_2| = 1$. In other words, the K-L algorithm swaps pairs of nodes, one from each partition, in each move, while the algorithms of Fidducia and Mattheyses [3] and Krishnamurthy [6] move one node at a time from one partition to the other followed by a move of another node in the reverse direction. The time complexity of the Kernighan-Lin algorithm is $\Theta(n^2\log n)$, while that of [3] and [6] is $\Theta(e)$, where n and e are the number of nodes and edges, respectively, of G. However, it should be noted that while the Kernighan-Lin algorithm swaps pairs of nodes, the algorithms of [3] and [6] move only one node at a time between partitions. An algorithm that swaps node

*This work was supported partly by a University of Minnesota Graduate School grant and partly by NSF grant MIP-9210049

pairs gives a better cutset improvement than one that swaps single nodes at a time. Further, the latter two algorithms make the simplified assumption that edge costs are exactly 1; thus they use bucket-sort (a $\Theta(n)$ algorithm) for sorting the nodes according to the gain associated by moving them. If this unit-cost assumption is invalid, then sorting takes time $\Theta(n \log n)$, and the complexity of the and algorithms in [3, 6] becomes $\Theta(n \log n + e)$. In this paper, we present a number of strategies to improve the time complexity of node-pair-swap algorithms.

The K-L algorithm is the quintessential node-pair-swap algorithm, and our speedup strategies improve this basic algorithm. Essentially, we propose techniques for improving the K-L algorithm in three different ways: (1) Limiting the portion of the graph G that needs to be searched to find the best node-pair to swap in a move; (2) Scanning the node pairs in a particular order in this "limited" graph so that the best node pair can be found without scanning the limited graph entirely; and (3) An efficient data structure to store and update the nodes in each partition. The "neighborhood search" technique is used to achieve (1), and a balanced tree data structure is used for (3). These two strategies improve the order notation time complexity of a node-pair-swap algorithm. Two different strategies "column scan" and "diagonal scan" are proposed for achieving goal (2). Both of these strategies will scan the entire limited graph in the worst case, but in practice scan only a small number of node pairs in it. Thus they do not contribute to improving the worst-case time complexity, but make the algorithm faster in practice.

In particular, we present an algorithm called **Quick_Cut** that uses neighborhood search, column scanning and a balanced tree data structure, and whose worst-case time complexity is $\Theta(\max(ed, e \log n))$, where d is the average node degree of G. We also show that the average-case complexity of this new algorithm is $\Theta(e \log n)$. Hence for sparse graphs **Quick_Cut** has a much better time complexity than the K-L algorithm (time complexity $\Theta(n^2 \log n)$, where n is the number of nodes). We have implemented **Quick_Cut**, and it shows a speedup factor of 5 to 50 over the K-L algorithm on a range of different graphs. We have also implemented other combinations of the techniques mentioned above, and have obtained data that show the time improvements achieved by different strategies.

2 The Kernighan-Lin Algorithm

In this section we briefly describe the K-L algorithm. We first introduce a few notations required in the sequel. Given a partition P_1 and P_2 of $V(G)$ define for each $u \in V(G)$, the *external cost* E_u and the *internal cost* I_u of u as follows:

$$E_u = \sum_{v \in P_i} c(\{u,v\}), \text{where} i = 1,2, u \notin P_i, \text{and} \{u,v\} \in E(G)$$

$$I_u = \sum_{v \in P_i} c(\{u,v\}), \text{where} i = 1,2, u \in P_i, \text{and} \{u,v\} \in E(G)$$

We define the D value of u, $D_u = E_u - I_u$ as the gain (reduction in the cost of the cut-set) obtained by moving u from its current partition. Thus if $u \in P_1$ and $v \in P_2$, then it is easy to see that the gain $G_{u,v}$ associated with swapping the pair of nodes (u,v) is $D_u + D_v - 2c(\{u,v\})$ if $\{u,v\} \in E(G)$, and $D_u + D_v$ otherwise.

Assume that there are $n = 2m$ nodes in G, and the initial partitions are P_1 and P_2, with $|P_1| = |P_2| = m$. Let $P_1 = \{u_1, u_2, \ldots, u_m\}$ and $P_2 = \{v_1, v_2, \ldots, v_m\}$. The main data structure of the Kernighan-Lin algorithm is the symmetric cost matrix C, where $C_{u,v} = c(\{u,v\})$ if $\{u,v\} \in E(G)$, and $C_{u,v} = 0$ otherwise. First the D value of each node u is computed using C. Then, that pair of nodes (u_{i_1}, v_{j_1}) is chosen for swapping that has the maximum value of $G_{u_{i_1},v_j}$. Node u_{i_1} is removed from P_1, v_{j_1} is removed from P_2, (u_{i_1}, v_{j_1}) is inserted in an ordered set S of node pairs, and the D value of each node u is updated to reflect the fact that the pair (u_{i_1}, v_{j_1}) has been swapped between the partitions. This procedure is iterated m times until P_1 and P_2 become empty, with the node pair (u_{i_k}, v_{j_k}) inserted in S in the k'th iteration, $1 \le k \le m$. Finally, $S = ((u_{i_1}, v_{j_1}), (u_{i_2}, v_{j_2}), \ldots, (u_{i_m}, v_{j_m}))$. All partial sums $S_k = \sum_{t=1}^{k} G_{u_{i_t},v_{j_t}}$ are computed, and p is chosen so that the partial sum S_p is the maximum. The set of node pairs that are actually swapped are then, $\{(u_{i_1}, v_{j_1}), \ldots, (u_{i_p}, v_{j_p})\}$, so that the maximum gain $G = S_p$ is obtained. This whole process is called a *pass*. A number of passes are made until the maximum gain G obtained is 0. This is a local maxima with respect to the initial partitions P_1 and P_2. Empirical evidence shows that the number of passes required to achieve a local maxima is two to four [5].

During the k'th iteration in a pass, the node pairs (u_{i_k}, v_{j_k}) with the maximum G value is chosen as follows. The nodes remaining in P_i are sorted in decreasing order by their recomputed D values in list L_i. The lists L_1 and L_2 are then scanned noting the maximum gain $G_{\max}$ among all node pairs scanned so far until a pair of nodes (u,v) is encountered such that $D_u + D_v \le G_{\max}$. It is easy to see that if the node pairs are scanned in an order that roughly corresponds to decreasing $D_u + D_v$ value, then we need not scan any pair of nodes beyond (u,v), since the sum of their D values, and hence their gain will be no greater than $D_u + D_v$, and hence no greater than $G_{\max}$. Thus (u_{i_k}, v_{j_k}) is the pair chosen to be inserted in S. The D value of each $u \in P_1$ is updated as $D_u = D_u + 2C_{u,u_{i_k}} - 2C_{u,v_{j_k}}$, while the D_v value of each $v \in P_2$ is updated as $D_v = D_v + 2C_{v,v_{j_k}} - 2C_{v,u_{i_k}}$. The critical question is how to scan the lists L_1 and L_2 simultaneously, so that it is enough to scan till a node pair is found whose $D_u + D_v$ value is no greater than the maximum gain found so far, without sorting all node pairs by the sum of their D values and then scanning such a sorted list of node pairs. This issue is left unanswered in [5], though it is claimed that such a scanning can be done using two sorted lists of nodes (L_1 and L_2) to find the node pair with the maximum gain fairly quickly. We have devised two

```
Procedure Column_Scan(L1, L2)
/* Column_Scan returns the node pair (u, v), where
u ∈ L1, and v ∈ L2 with the largest gain. L1 and L2 are
the list of nodes from partitions P1 and P2 respectively
that are sorted by the D value of the nodes. */
begin
    Let L1 be the ordered list u1, u2, ..., um;
    Let L2 be the ordered list v1, v2, ..., vm;
    Gmax = −∞; /* Gmax represents the maximum
    gain of a node pair found so far */
    for j = 1 to m do
        for i = 1 to m do begin
            if(G_{ui,vj} > Gmax) then begin
            (u, v) = (ui, vj); Gmax = G_{ui,vj};
            end;
            /* If this is node pair corresponding to the
            top element of the current column of the (hy-
            pothetical) scanning matrix M, then check to
            see if the sum of its D values is less than or
            equal to the maximum gain found so far */
            if(i = 1 and D_{u1} + D_{vj} ≤ Gmax) then
            return(Gmax, (u, v));
        endfor
    endfor
    return(Gmax, (u, v));
end /* Procedure Column_Scan */
```

Figure 1: A column scanning method for finding the best pair of nodes to swap in the K-L algorithm.

scanning methods "column scanning" and "diagonal scanning" that are efficient on the average; these are described in the next section.

Finally, we reiterate the time complexity derivation of the K-L algorithm given in [5] by assuming (as in [5]) that the node-pair scanning step in each move or iteration takes $\Theta(m)$ time, where m is the current size of each partition; the worst case time for node-pair scanning is, however, $O(m^2)$. The initial step of computing each D value takes $\Theta(n^2)$ time, since G is represented by an adjacency matrix in [5]. To sort the nodes by their updated D values, takes time $\Theta((m-k)\log(m-k))$ in the k'th iteration, while it takes $\Theta(m-k)$ time to scan the lists L_1 and L_2 to choose the node pair for inserting in S. Finally, updation of the D values takes $\Theta(m-k)$ time. Thus the dominant time complexity of the k'th iteration is $\Theta((m-k)\log(m-k))$. Thus over $m = n/2$ iterations this gives a total time complexity of $\sum_{k=1}^{m}(m-k)\log(m-k) = \Theta(n^2 \log n)$. The final step of choosing p to maximize the partial sum S_p takes time $\Theta(n)$. Thus the complexity of one pass of the K-L algorithm is $\Theta(n^2 \log n)$. If the empirical evidence that a constant number of passes is required is taken into account, then the whole algorithm also has time complexity $\Theta(n^2 \log n)$.

3 Node-Pair Scanning Methods

3.1 Column Scanning

Visualize an $m \times m$ matrix M (which is not actually constructed), with one axis of M being the sorted list L_1 and the other axis being the list L_2. Each element of M then corresponds to node pairs (u, v), with $u \in L_1$ and $v \in L_2$, and its value is $D_u + D_v$. The node pairs are then scanned column-wise in M, and the maximum G value $G_{\max}$ of all node pairs encountered is noted. The maximum $D_u + D_v$ value of each column scanned in this manner is compared to the current $G_{\max}$, and the scanning stops whenever this maximum $D_u + D_v$ is less than or equal to $G_{\max}$. Note that the maximum $D_u + D_v$ value in each column will correspond to the top element in that column, since the nodes along each axis of M is sorted by decreasing D values. The above node-pair scanning method is given formally as algorithm **Column_Scan** in Fig. 1, and illustrated in Fig. 2a. Theorem 1 establishes correctness of **Column_Scan** for finding the maximum gain pair.

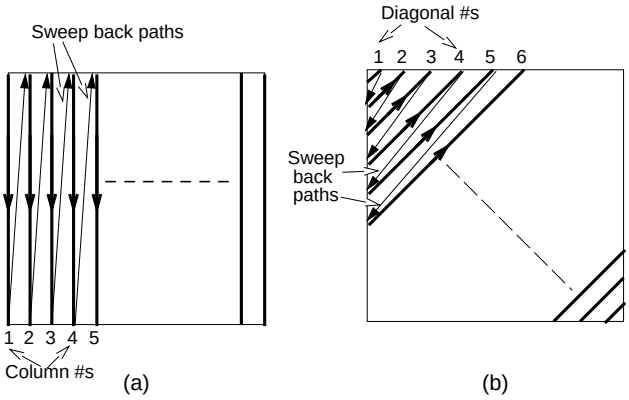

Figure 2: Scanning matrix M: (a) Column scan; (b) Diagonal scan.

Theorem 1 The gain $G_{\max}$ found by **Column_Scan** is the maximum gain among all node pairs from the two partitions P_1 and P_2 of the node set $V(G)$.

Proof: Suppose the scanning stopped at node pair (u_1, v_j) corresponding to the top element $M_{1,j}$ of column j of M. Then, it is easy to see that all unscanned node pairs (u_k, v_j), where $k > i$, in the current column have the sum of their D values no greater than $D_{u_1} + D_{v_j}$; hence their gains cannot be greater than $G_{\max}$. Also, the sum of D values of each node pair in the submatrix of M from column $j+1$ to m and from row 1 to m is no greater than the sum of the D values of the corresponding pair in column j. In other words $D_{u_i,v_k} \leq D_{u_i,v_j}$, when $k > j$. Since $G_{u_1,v_j} \leq D_{u_1,v_j} \leq G_{\max}$, and for all $i > 1$, $G_{u_i,v_k} \leq D_{u_i} + D_{v_j} \leq D_{u_1,v_j}$, it follows that no unscanned node pair (corresponding to the submatrix from column $j+1$ to m and the last $m-1$ elements of the jth column of M) can have a gain greater than $G_{\max}$. □

In the worst case, all the m^2 node pairs will be scanned by **Column_Scan** to find the node pair with the maximum gain, where during the kth iteration of selecting a new node pair to swap $m = (N/2) - k$. However, it is implicitly claimed in [5] that on the average, the node pair with the maximum gain will be found much more quickly, say, in $\Theta(m)$ time, instead

of the worst case of $O(m^2)$, and our use of the K-L algorithm (implemented using **Column_Scan**) on various graphs confirms this observation. The time complexity analysis for the K-L algorithm (given in the previous section) was performed in [5] on the basis of the above assumption.

3.2 Diagonal Scanning

A more efficient way to scan the node pairs is to scan M diagonally instead of column-wise as is done in **Column_Scan**. Since the node pairs with high $D_u + D_v$ values are clustered in the upper left part of M, a scan along the left-to-right diagonals of M starting from its upper left corner is almost like visiting the node pairs in the sorted order of their $D_u + D_v$ values. In a diagonal scan, we keep track of the maximum $D_u + D_v$ value of each diagonal, and stop scanning whenever this value is less than or equal to $G_{\max}$, the maximum gain of all node pairs scanned so far—note that we use the same principle in **Column_Scan** in which we note the $D_u + D_v$ value of the first pair in each column, which is also the maximum $D_u + D_v$ value of that column, and stop when this value is less than or equal to $G_{\max}$. Since the diagonal scanning method scans the node pairs in an approximate sorted order of their $D_u + D_v$ values, we are more likely to stop scanning sooner than in the column scanning scheme. Note that in the latter scheme we scan many node pairs of low $D_u + D_v$ values at the bottom of each column before scanning nodes of higher $D_u + D_v$ values at the top of the next column; hence more node pairs are likely to scanned. The diagonal scanning method is presented as Procedure **Diag_Scan** in Fig. 3, and illustrated in Fig. 2b.

The next theorem establishes the correctness of **Diag_Scan**.

Theorem 2 The gain $G_{\max}$ found by **Diag_Scan** is the maximum gain among all node pairs from the two partitions P_1 and P_2 of the node set $V(G)$.

Proof: There are $2m-1$ left-to-right diagonals in M, and let us number them from 1 to $2m-1$ based on the order in which they are scanned by **Diag_Scan**. Suppose that the scanning stopped at diagonal t, and that the node pair (u_i, v_j) in it has the maximum $D_u + D_v$ value. Let (u_x, v_y) be an arbitrary node pair in diagonal t. It is easy to see that all unscanned node pairs in the xth row to the right of (u_x, v_y), and all unscanned node pairs in column y below (u_x, v_y) have $D_u + D_v$ values that are less than or equal to $D_{u_x} + D_{u_y} \leq D_{u_i} + D_{v_j} \leq G_{\max}$. Hence none of the above unscanned node pairs can have G values greater than $G_{\max}$. If diagonal t belongs to the lower right triangle of M, then we have accounted for all unscanned node pairs and we have proved the theorem.

We now consider the case when t belongs to the upper left triangle of M; see Fig. 4. In this case there will be a number of unscanned node pairs that are neither in the same row or column of some node pair (u_x, v_y) of diagonal t. These node pairs can only belong to completely unscanned columns of M, and these columns lie to the right of the tth column of M that contains the rightmost element (u_1, v_t) of diagonal t,

```
Procedure Diag_Scan(L1, L2)
/* Diag_Scan returns the node pair (u, v), where u ∈
L1, and v ∈ L2 with the largest gain. L1 and L2 are
the list of nodes from partitions P1 and P2 respectively
that are sorted by the D value of the nodes. */
begin
    Let L1 be the ordered list u1, u2, ..., um;
    Let L2 be the ordered list v1, v2, ..., vm;
    Gmax := −∞ ; /* Gmax represents the maximum
    gain of a node pair found so far */
    /* First scan the upper left triangle of the scanning
    matrix M. The scanning is done along left-to-right
    diagonals */
    for i = 1 to m do
        diag_max := −∞ ; /* diag_max stores the max-
        imum Du + Dv value of the current diagonal */
        for j = 1 to i do begin
            if(G_{u_{i−j+1}, v_j} > Gmax) then begin
            (u, v) := (u_{i−j+1}, v_j); Gmax := G_{u_{i−j+1}, v_j};
            end;
            if(D_{u_{i−j+1}} + D_{v_j} > diag_max) then
            diag_max := D_{u_{i−j+1}} + D_{v_j};
        endfor
        if (diag_max ≤ Gmax) then
        return(Gmax, (u, v));
    endfor
    /* Now scan the lower right triangle of M */
    for j = 2 to m do
        diag_max := −∞;
        for i = m downto j do begin
            if(G_{u_i, v_{j+m−i}} > Gmax) then begin
            (u, v) := (u_i, v_{j+m−i}); Gmax := G_{u_i, v_{j+m−i}};
            end;
            if(D_{u_i} + D_{v_{j+m−i}} > diag_max) then
            diag_max := D_{u_i} + D_{v_{j+m−i}};
        endfor
        if (diag_max ≤ Gmax) then
        return(Gmax, (u, v));
    endfor
    return(Gmax, (u, v));
end /* Procedure Diag_Scan */
```

Figure 3: A diagonal scanning method for finding the best pair of nodes to swap in the K-L algorithm.

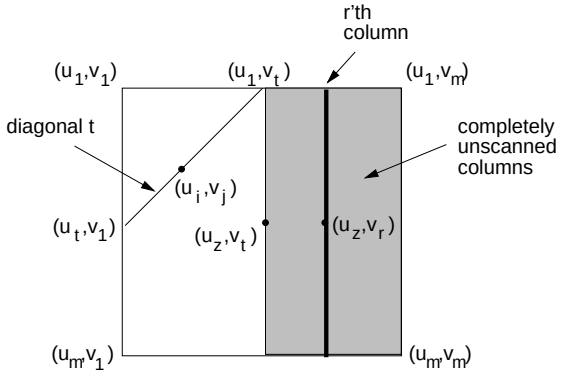

Figure 4: The scanning matrix M: Illustrating the proof of Theorem 2.

where $1 \leq t \leq m-1$. Consider such a completely unscanned column r ($r > t$), and some node pair (u_z, v_r) in it. It is easy to see that $G_{u_z,v_r} \leq D_{u_z} + D_{v_r} \leq D_{u_z} + D_{v_t} \leq D_{u_1} + D_{v_t} \leq D_{u_i} + D_{v_j} \leq G_{\max}$ (recall that node pair (u_i, v_j) has the maximum $D_u + D_v$ value in diagonal t). Hence $G_{\max}$ is the maximum gain among all node pairs in $P_1 \times P_2$. □

4 An Improved Partitioning Algorithm

In this section we propose a K-L-type partitioning scheme whose worst and average-case time complexities are much lesser than those of the K-L algorithm. We define $d = \lceil 2e/n \rceil$ as the average node degree, and assume that the maximum node degree $d_{max} = \Theta(d)$. The dominant time complexity of the Kernighan-Lin algorithm arises from the process of selecting in each iteration the node pair (u_i, v_j) with the maximum gain. In the previous section we described two scanning methods **Column_Scan** and **Diag_Scan** for selecting this node pair that work well in practice, but whose worst-case time complexity is $\Theta(m^2)$, where m is the number of unlocked nodes in each partition. Hence, by themselves they cannot help to reduce the time complexity of a K-L-type partitioning algorithm. In this section, we constructively prove that the worst-case time complexity for finding the node pair with the largest gain can be limited to $\Theta(d^2)$; in other words, we need scan at most d^2 node pairs in the fictitious scanning matrix M to find the maximum-gain pair.

We use an adjacency list for storing G ($\Theta(e)$ creation time) rather than an adjacency matrix ($\Theta(n^2)$ creation time) that was used in [5]; however, this is a minor point and does not contribute significantly to the overall time complexity of either algorithm. We also use AVL trees [1], which are balanced binary search trees, to store the nodes in the two partitions, so that after each swap, each updated neighbor of the swapped nodes can be stored back in $\Theta(\log m)$ time; note that an inorder scan of a binary search tree results in a sorted ascending order of the nodes (according to some key value), while a reverse inorder scan produces a sorted descending order. After describing our new scanning method in terms of the scanning matrix M, we will also describe how this scanning is done when the nodes are stored in AVL trees.

4.1 Neighborhood Search

Neighborhood search is the technique that allows us to limit our node-pair scanning process using either column or diagonal scanning to a subset of d^2 node pairs. These d^2 node pairs correspond to the upper left $d \times d$ submatrix of the scanning matrix M. In what follows, we describe only the column scanning method in conjunction with neighborhood search that effectively yields a "new" scanning method that we call **Compact_Col_Scan**.

This new scanning method can be looked upon as applying the column scanning technique (Fig. 1) to the $d \times d$ submatrix of M consisting of the first d rows and columns of M. Let the m rows of M represent the nodes $(u_1, u_2, \ldots, u_m)$ of P_1, and the columns represent the nodes $(v_1, v_2, \ldots, v_m)$ of P_2 in descending order of their D values. Then, in scanning, say the ith column, it is not necessary to scan any node pair beyond the first node v_l encountered that is not a neighbor of u_i. This is because, for any node v_j, $j > l$, $G_{u_i,v_j} \leq D_{u_i} + D_{v_j} \leq D_{u_i} + D_{v_l} = G_{u_i,v_l} \leq G_i^l$, where G_i^l is the maximum gain among the node pairs in the first l rows of column i, i.e., $G_i^l = \max\{G_{u_i,v_1}, G_{u_i,v_2}, \ldots, G_{u_i,v_l}\}$. Similarly, if u_k is the first node that is not a neighbor of v_1, then it is not necessary to scan beyond column $k-1$. Note that when G is regular both $k-1$ and l are at most d, and when G is not regular they are at most $d_{\max} = \Theta(d)$. Algorithm **Compact_Col_Scan** describes the above scanning technique formally in Fig. 5.

```
Procedure Compact_Col_Scan(L1, L2)
/* Compact_Col_Scan returns the node pair (u, v),
where u ∈ L1, and v ∈ L2 with the largest gain. L1
and L2 are the list of nodes from partitions P1 and P2
respectively that are sorted by the D value of the nodes.
*/
begin
    Let L1 be the ordered list u1, u2, ..., um;
    Let L2 be the ordered list v1, v2, ..., vm;
    k := 1;
    while (uk is a neighbor of v1) and (k < m) do
    k := k + 1;
    Gmax := G_{uk,v1} ; /* Gmax represents the maximum gain of a node pair found so far */
    (u, v) := (uk, v1);
    for j = 1 to k − 1 do begin
        /* Scan only the first k − 1 columns */
        i := 0;
        repeat
            i := i + 1;
            if(G_{vi,uj} > Gmax) then begin
            (u, v) = (vi, uj); Gmax = G_{vi,uj};
            end;
            /* If this is node pair corresponding to the
            top element of the current column of the (hypothetical) scanning matrix M, then check to
            see if the sum of its D values is less than or
            equal to the maximum gain found so far */
            if(i = 1 and Dv1 + Duj ≤ Gmax) then
            return(Gmax, (u, v));
        until (vi is not a neighbor of uj) or (i = m)
    endfor
    return(Gmax, (u, v));
end /* Procedure Compact_Col_Scan */
```

Figure 5: A more efficient column scanning method for finding the pair of nodes with the maximum gain.

The theorem proved below establishes that **Compact_Col_Scan** always chooses the node pair (u_i, v_j) with the maximum gain.

Theorem 3 The gain $G_{\max}$ found by **Compact_Col_Scan** is the maximum gain among all node pairs from the two partitions P_1 and P_2 of the node set $V(G)$. Furthermore, if G is a regular graph, then **Compact_Col_Scan** scans at most $(d+1)^2$ node pairs, otherwise it scans at most $(d_{\max}+1)^2$ node pairs in the scanning matrix M.

Proof: Refer to Fig. 5. Let u_k be the first node encountered when scanning the first row of M that is not a neighbor of v_1 (in the first **while** loop of **Compact_Col_Scan**). Then, it is obvious that $G_{v_1,u_k} =$

$D_{v_1} + D_{u_k} \geq D_{v_1} + D_{u_j} \geq G_{v_1,u_j}$, where $j > k$. Since $D_{v_1} + D_{u_j}$ is the maximum possible gain value among all node pairs in the jth column, this means that the maximum gain among all node pairs is either G_{v_1,u_k} or is the gain of a node pair in the leftmost $m \times (k-1)$ submatrix of M. Note that $G_{\max}$ is initialized to be G_{v_1,u_k} in **Compact_Col_Scan**.

Further, in any column, say the jth one, in M, it is necessary only to scan till the first node v_l that is not a neighbor of u_j, since for $i > l$, $G_{v_i,u_j} \leq D_{v_i} + D_{u_j} \leq D_{v_l} + D_{u_j} = G_{v_l,u_j} \leq G_j^l$, where $G_j^l = \max\{G_{u_j,v_1}, G_{u_j,v_2}, \ldots, G_{u_j,v_l}\}$. The **for** loop of **Compact_Col_Scan** scans the first $k-1$ columns of M by ending each column j's scan when a non-neighbor of u_j is encountered, where $1 \leq j < k$. Hence it follows from the argument in this paragraph and Theorem 1 that **Compact_Col_Scan** finds the maximum gain among all node pairs in the leftmost $m \times (k-1)$ submatrix of M. Also, since $G_{\max}$ is initialized to G_{v_1,u_k}, **Compact_Col_Scan** returns the maximum of G_{v_1,u_k} and the maximum gain in the leftmost $m \times (k-1)$ submatrix. As established above, this represents the maximum gain among all node pairs in M.

We assume for simplicity that G is regular—when G is not than the following argument works by substituting $d_{\max}$ for d. It is easy to see that the first **while** loop scans at most $d+1$ node pairs. Subsequently, in the **for** loop the leftmost $k-1$ columns of M are scanned, where $k-1 \leq d$. The scanning of each column j in the **repeat-until** loop stops at the first non-neighbor of u_j, and hence at most $d+1$ node pairs are scanned in each of the $k-1$ columns. Thus **Compact_Col_Scan** scans at most $d+1+d(d+1) = (d+1)^2$ node pairs. It is straightforward to verify that each node-pair scan takes constant time, and that node-pair scanning has the dominant time complexity in **Compact_Col_Scan**; hence its worst case time complexity is $O(d^2)$ □

Note that though the worst-case time complexity is $O(d^2)$, it is intuitively obvious that **Compact_Col_Scan** is expected to perform much better on the average, even when G is not sparse—the worst-case scenario assumes that all neighbors of v_1 in partition P_1 are clustered at the front of the sorted list L_1, and all neighbors in P_2 of these neighbors of v_1 are clustered at the front of L_2.

It is also possible to use diagonal scan in conjunction with neighborhood search by restricting it to scan in only the $d \times d$ upper left submatrix of M. However, we really do not have M as a real data structure, but use adjacency lists and balanced AVL trees in its place as described in the next subsection. With such a data structure, it is difficult to implement a diagonal scan on the upper left $d \times d$ submatrix of the fictitious scanning matrix M, and we have not done so. We will thus ignore this combination of strategies in the rest of the paper.

4.2 Data Structures

As mentioned earlier, the data structure representing G is an adjacency list. Each node u in G is represented by a *primary structure* $Pr(u)$, which contains the following information about u:

1. Its D value
2. Pointers for the AVL tree (T_1 or T_2) in which $Pr(u)$ is a node
3. A pointer to a list $adj(u)$ of secondary structures representing the neighbors of u in G
4. A marker $m(u)$, whose use will become clear shortly
5. An integer $i(u)$ indicating whether u is in P_1, P_2, or the ordered set S of swapped node pairs.

The list $adj(u)$ is an adjacency list of u that contains information about each neighbor v of u; this information on v is contained in the *secondary structure* $Sc(v)$ of v that is a node of $adj(u)$.

1. The cost $c(\{u,v\})$ of the edge $\{u,v\}$
2. A pointer to the primary structure $Pr(v)$ of v
3. Pointers to the immediately preceding and succeeding secondary structures in $adj(u)$

The nodes of G are numbered from 1 to n and the primary structures are accessed from an array of pointers $N[1..n]$. There are also two AVL-trees T_1 and T_2 in which nodes belonging to P_1 and P_2, respectively, are inserted using their D values as the keys. We will not always spell out in the description of our algorithm how certain computations are done; they follow quite naturally from the above description of the data structure employed.

An informal description of our algorithm **Quick_Cut** follows. All the data structures are initialized correctly. Note that initially we do not explicitly have the partitions P_1 and P_2, but have an integer $i(u)$ in each $Pr(u)$ indicating the partition it belongs to. The D value of each u is first computed by scanning the list $adj(u)$. Each $Pr(u)$ belonging to partition P_i is then inserted into T_i according to its D value, where $i = 1, 2$. Note that a reverse inorder scan[1] of T_i gives us the nodes in P_i in decreasing order according to their D values. This represents the set-up phase of the algorithm; the next series of computations described below are iterated m times.

We now describe how the node pair scanning method described in Algorithm **Compact_Col_Scan** (Fig. 5) is implemented with the nodes in each partition stored in two AVL trees T_1 and T_2. Corresponding to the rightmost node v_1 in T_2, i.e., the node with the highest D value in P_2, T_1 is scanned in reverse inorder until a node u_k that is not a neighbor of v_1 is encountered. Let $Q = \{u_1, u_2, \ldots, u_k\}$ denote the nodes in T_1 scanned in this manner and let G_{v_1} be the maximum of all gains $\{G_{v_1,u_j} : 1 \leq j \leq k\}$; note that $k = O(d)$. The above scanning is done as follows. First $adj(v_1)$ is scanned and all those $Pr(u)$'s of nodes in T_1 that are neighbors of v_1 are marked using $m(u)$. Then, T_1 is scanned reverse inorder until a non-marked node u_k is encountered. Finally, $adj(v_1)$ is scanned again, and all its neighbors in T_1 are unmarked. Let us call this scanning procedure of T_1 for node v_1 as **Rev_Inorder_Scan**(v_1, T_1).

[1] A reverse inorder scan of a binary tree with root r can be recursively defined as visit the right subtree of r in reverse inorder, visit r, and visit the left subtree of r in reverse inorder.

Now for each node u_j in Q, $1 \leq j < k$, **Rev_Inorder_Scan**(u_j, T_2) is executed in order of node index. **Rev_Inorder_Scan**(u_j, T_2) returns the maximum gain G_{u_j}, the node pair it corresponds to. Before invoking **Rev_Inorder_Scan**(u_j, T_2), it is checked whether $D_{u_j} + D_{v_1}$ is less than or equal to the maximum gain $G_{\max}$ of all node pairs scanned so far. If this is true, then the scanning is terminated, otherwise **Rev_Inorder_Scan**(u_j, T_2) is called, and $G_{\max}$ is updated to G_{u_j} if $G_{u_j} > G_{\max}$. At the end of the scanning process, $G_{\max}$, corresponding to the gain of, say, node pair (u_i, v_j), represents the maximum gain of all node pairs. $Pr(u_i)$ and $Pr(v_j)$ are removed from their corresponding trees and inserted as a pair in the ordered set S; S can be implemented as two queues S_1 and S_2, into which the nodes u_i and v_j are inserted, respectively, where $u_i \in P_1$ and $v_j \in P_2$.

It is easy to see that the D values of only the neighbors of u_i and v_j are affected by the swap of the node pair (u_i, v_j). Thus the adjacency lists of u_i and v_j are scanned, and for each neighbor w encountered that is not in S, D_w is updated as explained in the preceding section. Simultaneous with the updation of D_w, $Pr(w)$ is removed from its tree T_i, and reinserted in its correct position in T_i. This whole process is iterated until both trees T_1 and T_2 become empty. Finally, as in the K-L algorithm, p is chosen so that the partial sum $\sum_{k=1}^{p} G_{u_{i_k}, v_{j_k}}$ in S is the maximum. The first p ordered pairs in S are actually swapped, and the data structure updated to reflect the swaps. This corresponds to one pass of **Quick_Cut**. A number of passes are made till the maximum partial sum is zero.

The following theorem establishes the correctness of **Quick_Cut**.

Theorem 4 In every iteration of **Quick_Cut**, the node pair (u_i, v_j) is selected for inclusion in the ordered set S of node pairs, such that $G_{u_i, v_j} = \max\{G_{u,v} : u \in P_1, v \in P_2\}$.

Proof: It is easy to see that the scanning process for node pairs described for **Quick_Cut** and **Rev_Inorder_Scan** is a correct implementation of **Compact_Col_Scan**. Hence the result follows from Theorem 3. □

4.3 Complexity Analysis

We now perform worst- and average-case time complexity analysis of one pass of **Quick_Cut**. It is easy to see that the initialization phase including the computation of the D_u values takes time $\Theta(e)$, while constructing the two AVL trees T_1 and T_2 has complexity $\Theta(n \log n)$. Since **Rev_Inorder_Scan**(u, T_i) stops after encountering the first node in T_i that is not a neighbor of u, its time complexity is $\Theta(d)$ in the worst case. For finding the best node pair to swap in every iteration of **Quick_Cut**, **Rev_Inorder_Scan** is called $\Theta(d)$ times in the worst case; thus the total time complexity of all calls to **Rev_Inorder_Scan** is $\Theta(d^2)$ per iteration in the worst case. Also, in every iteration, it takes $\Theta(d)$ time to update the D_u values of all nodes $w \in adj(u_{i_t}) \cup adj(v_{j_t})$, where the node pair (u_{i_t}, v_{j_t}) is selected for inclusion in S in the t'th iteration. To delete and insert each $Pr(w)$ in its tree T_i takes $\Theta(\log n)$ time. Hence the total cost of all updations done in T_1 and T_2 during any iteration is $\Theta(d \log n)$ in the worst case. Thus we get the total worst-case time complexity of all node-pair swaps and updates in a pass to be $\Theta(n(d^2 + d \log n))$. Finally, choosing p to maximize the partial sum S_p has complexity $\Theta(n)$, and updating all the data structures after swapping the p nodes takes time $\Theta(e)$. The worst-case complexity of one pass of **Quick_Cut** is thus dominated by the complexity of finding the best node pairs; hence the complexity of one pass of **Quick_Cut** is $\Theta(nd \log n)$, when $d = O(\log n)$, and $\Theta(nd^2) = \Theta(ed)$, when $d = \Omega(\log n)$.

The average-case analysis of **Quick_Cut** assumes that G is a sparse random graph with a large number of nodes, as is true for most VLSI circuits. Our main assumption is that given a node u it is equally likely that there is an edge between u and v for all nodes $v \neq u$. Hence, if p is the probability that there is an edge between a given node u and any other node v, then $p = \Theta(d/n)$. It can be seen that the average number of nodes scanned by **Rev_Inorder_Scan**(u, T_i) is

$$\sum_{j=1}^{d} (j(1-p)p^{j-1}) + (d+1)p^d$$

$$= (dp^d(p-1) + 1 - p^d)/(1-p) + (d+1)p^d$$

$$= (1 - p^{d+1})/(1-p)$$

For large n and $d \ll n$, we can assume that $1 - p \approx 1$ and thus also that $1 - p^{d+1} \approx 1$. Thus the average number of nodes scanned by **Rev_Inorder_Scan** is $\Theta(1)$; hence the total complexity of all calls to **Rev_Inorder_Scan** for finding the best node-pair in any iteration is $\Theta(1)$. The average-case complexity of all other parts of **Quick_Cut** remains the same as in the worst case. Hence the average-case complexity of **Quick_Cut** is $\Theta(nd \log n) = \Theta(e \log n)$.

5 Experimental Results

Table 1 shows the timing results for partitioning several graphs, some of which are random, using the basic K-L algorithm, its various improvements suggested in this paper and algorithm **Quick_Cut**. The partitioning quality (cut-set sizes) of all these methods are either the same or very close for all the graphs in Table 1. The times given are those for partitioning the graph completely, i.e., by recursively partitioning each partitioned half until there are at most two nodes remaining in a partition. Furthermore, the times are also averages over four random initial partitions of the same graph into two halves. For the random graphs, the times have also been averaged over five different random graphs with the same number of nodes and edges.

We can summarize the data in Table 1 as follows:

1. Partitioning methods Q-C, DS-Tree and K-L-Tree are more sensitive to the number of edges rather

Graph	CPU Execution Time in Seconds				
	Q-C	**DS-Tree**	**K-L-Tree**	**DS-List**	**K-L**
9-dim. hypercube	4.82	4.95	8.5	48.85	53.02
4-FT 9-cube	17.15	16.25	32.40	36.70	51.70
20×30 mesh	4.43	4.42	10.45	258.65	244.77
Random Graphs					
$n = 200$					
$e = 500$	0.89	0.90	1.34	2.86	3.27
$e = 1000$	1.31	1.33	2.16	2.73	3.38
$e = 1500$	1.64	1.63	2.81	2.92	4.10
$e = 2000$	1.94	1.86	3.43	3.02	6.13
$e = 3000$	2.54	2.43	4.60	3.11	4.87
Average for $n = 200$	1.66	1.63	2.87	2.92	4.35
$n = 400$					
$e = 1000$	2.28	2.34	4.74	19.39	27.93
$e = 2000$	3.01	3.04	6.88	16.27	17.76
$e = 3000$	3.86	3.84	9.46	15.00	19.08
$e = 4000$	4.56	4.47	12.06	15.25	25.24
$e = 6000$	5.84	5.77	15.56	15.53	32.28
Average for $n = 400$	3.91	3.89	9.74	16,29	24.45
$n = 800$					
$e = 2000$	5.74	5.84	18.39	172.32	196.36
$e = 4000$	7.88	7.81	29.20	115.36	149.87
$e = 6000$	9.69	9.27	36.38	108.74	203.84
$e = 8000$	11.31	11.13	45.44	98.70	143.58
$e = 12000$	14.41	14.05	62.58	92.35	137.26
Average for $n = 800$	9.8	9.62	38.39	117.49	166.18
$n = 1000$					
$e = 2500$	7.69	7.82	23.91	308.88	251.15
$e = 5000$	9.89	9.93	31.82	250.45	264.27
$e = 7500$	12.47	12.57	46.61	209.90	261.47
$e = 10000$	15.14	15.14	58.89	202.24	207.55
$e = 15000$	19.94	19.14	80.06	185.94	223.39
Average for $n = 1000$	13.02	12.90	48.26	231.48	241.56

Legend:
Q-C: Algorithm **Quick_Cut**, which uses neighborhood search, column scanning and an AVL tree data structure.
DS-Tree: K-L-type algorithm augmented with diagonal scanning and AVL tree data structure
K-L-Tree: K-L algorithm augmented with an AVL tree data structure
DS-List: K-L-type algorithm augmented with diagonal node-pair scanning
K-L: Original Kernighan-Lin algorithm [5] that uses column scanning and a list data structure
4-FT 9-cube: A 4-fault-tolerant version of a 9-dim. hypercube in which the node degree is $4 \times 9 = 36$
n: number of nodes in the random graph; e: number of edges in it.

Table 1: Comparison of the original K-L algorithm with its various improvements and with the **Quick_Cut** algorithm.

than the number of nodes in the graph. For example, for random graphs with $n = 400$, the times for these tree methods increases (almost linearly) with the number of edges e. This data thus corroborates the $\Theta(ed)$ time complexity of Q-C. Empirically, D-S-Tree also seems to have the same time complexity as Q-C, and so does K-L-Tree to some extent, but with larger constants. On the other hand, DS-List and K-L are sensitive only to the number of nodes, and insensitive to the number of edges. See again, for example, their times for $n = 400$—there is almost no variation as the number of edges change, or at least the small variations in them do not seem to be related to the number of edges. Furthermore, the average times for DS-List and K-L for the different n values definitely show an increase that is at least $\Omega(n^2)$ and seems to be more like $\Theta(n^2 \log n)$ as predicted in [5].

2. Again, consider the average data for random graphs with $n = 400$ nodes—the trends shown in this case are typical of the other n values. The factor decrease in time from K-L to K-L-Tree is 2.5, which is the improvement afforded by having a balanced tree data structure for node storage and updates. Going from K-L-Tree to Q-C, we get a further factor decrease in time of 2.5. This is the improvement obtained by using the neighborhood search strategy. Note that both Q-C and K-L-Tree use column scanning. To estimate the benefits obtained by diagonal scanning of node pairs we compare the times of KL-Tree to DS-tree and notice a factor improvement of 2.5 again. Thus both Q-C and DS-Tree give a factor of 6.25 improvement in time over K-L.

   Similarly, from the average data for $n = 800$ it can be seen that a balanced tree data structure (K-L-Tree) gives a speedup factor of 4.3 over K-L, the use of neighborhood search gives a further speedup by a factor of 3.9 (Q-C vs K-L-Tree), while using diagonal scanning improves the time by a factor of 4.0 (DS-Tree vs K-L-Tree). Thus both Q-C and DS-Tree give a factor improvement in time of about 16 over K-L.

3. There is no significant time difference between Q-C and DS-Tree. In fact, there cannot be any meaningful comparison between the two, since Q-C uses a combination of column scanning and neighborhood search, while DS-tree uses diagonal scanning; both use a balanced tree data structure. It seems logical that the best partitioning method would combine diagonal scanning and neighborhood search. However, such a method is difficult to implement and will most probably have a high overhead. In any case, from all the regular and random graph data it is evident that both Q-C and DS-Tree are faster than K-L by factors of 5 to 50. Note that assuming that node-pair scanning takes $\Theta(n)$ per node pair on the average (the worst-case time is $\Theta(n^2)$), DS-Tree has

a time complexity of $\Theta(n^2)$—this is the same assumption used to obtain the time complexity of $\Theta(n^2 \log n)$ for the K-L algorithm.

6 Conclusions

We presented an algorithm **Quick_Cut** for partitioning a graph G so that the cost of the cut-set of the partition is minimized. **Quick_Cut** uses the same overall strategy as the Kernighan-Lin algorithm, but uses (1) an improved data structure, (2) an efficient node-pair scanning method **Column_Scan** and most importantly (3) the result that only $O(d^2)$ node pairs need to be searched in the worst case to locate the node pair with the maximum gain. These have helped to speed up the partitioning process considerably. The worst-case time complexity of **Quick_Cut** is $\Theta(e \log n)$, when $d = O(\log n)$, and $\Theta(ed)$, when $d = \Omega(\log n)$, while its average-case complexity is $\Theta(e \log n)$. The worst-case complexity of the Kernighan-Lin algorithm is $\Theta(n^2 \log n)$. Hence even in the worst case, **Quick_Cut** is faster for sparse graphs, or specifically, when $d = O((n \log n)^{1/2})$. We implemented **Quick_Cut** as well as the K-L algorithm and its various enhancements proposed here, and used them to partition various random and well-known graphs. The timing results show that **Quick_Cut** is 5 to 50 times faster than the basic K-L algorithm. The DS-Tree method also affords the same performance improvements over the K-L algorithm, although its worst-case complexity of $\Theta(n^2)$ is much worse than that of **Quick_Cut**. Either **Quick_Cut** or DS-Tree can thus serve as the basis for new fast partitioning and layout tools for VLSI circuits.

Acknowledgement

We gratefully acknowledge the contributions of Henry A. Rowley who has written the code for the algorithms presented in this paper, and Deepak Sherlekar who suggested the problem of making the Kernighan-Lin algorithm sensitive to the number of edges in G rather than to the square of the number of nodes in it.

References

[1] A.V Aho, J.E. Hopcroft and J.D. Ullman, "The Design and Analysis of Computer Algorithms", Addison Wesley, Reading, Massachussetts, 1974.

[2] Alfred E. Dunlop and Brian W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, CAD-4(1):92–98, January 1985.

[3] C.M. Fidducia and R.M. Mattheyses, "A linear-time heuristic for improving network partitions", *Proc. Nineteenth Design Automation Conf.*, 1982, pp. 175-181.

[4] M.R. Garey and D.S. Johnson, *Computers and Intractability*, W.H. Freeman and Company, New York, pp. 209-210.

[5] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *Bell System Tech. Journal*, vol. 49, Feb. 1970, pp. 291-307.

[6] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks", *IEEE Trans. on Comput.*, vol. C-33, May 1984, pp. 438-446.

[7] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2), June 1991.