Partitioning of VLSI circuits using a reduced connectivity matrix

Wilsin Gosti
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Engineering Commons, and the Electrical and Computer Engineering Commons

Recommended Citation

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Partitionig of VLSI circuits using a reduced connectivity matrix

by

Wilsin Gosti

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Signatures have been redacted for privacy

Iowa State University
Ames, Iowa
1990
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Graph Representation of VLSI Circuits</td>
<td>4</td>
</tr>
<tr>
<td>2. PARTITIONING ALGORITHM</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Partitioning Problem</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Partitioning Phase</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Reduced connectivity matrix (M)</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Cut index</td>
<td>13</td>
</tr>
<tr>
<td>2.2.3 Permutation of rows</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4 Selecting partitions</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Optimization Phase</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Global Algorithm</td>
<td>28</td>
</tr>
<tr>
<td>2.5 Complexity of Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>2.5.1 Reduced connectivity matrix and auxiliary matrix</td>
<td>29</td>
</tr>
<tr>
<td>2.5.2 Cut index</td>
<td>32</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.5.3 Permutation of rows</td>
<td>33</td>
</tr>
<tr>
<td>2.5.4 Selecting partitions</td>
<td>34</td>
</tr>
<tr>
<td>2.5.5 Optimization</td>
<td>37</td>
</tr>
<tr>
<td>2.5.6 Total complexity</td>
<td>40</td>
</tr>
<tr>
<td>3. rcmp IMPLEMENTATION</td>
<td>42</td>
</tr>
<tr>
<td>3.1 X Window System</td>
<td>42</td>
</tr>
<tr>
<td>3.2 Overview</td>
<td>44</td>
</tr>
<tr>
<td>3.3 Menu Commands</td>
<td>47</td>
</tr>
<tr>
<td>3.4 Implementation</td>
<td>51</td>
</tr>
<tr>
<td>3.4.1 Opening the BorderWindow, DrawWindow, and menu boxes</td>
<td>52</td>
</tr>
<tr>
<td>3.4.2 Node and branch</td>
<td>52</td>
</tr>
<tr>
<td>3.4.3 Undo, delete, and wipe</td>
<td>54</td>
</tr>
<tr>
<td>3.4.4 Zooming</td>
<td>58</td>
</tr>
<tr>
<td>3.4.5 Save and load</td>
<td>61</td>
</tr>
<tr>
<td>3.4.6 Part</td>
<td>62</td>
</tr>
<tr>
<td>3.5 X Window Functions</td>
<td>62</td>
</tr>
<tr>
<td>4. EXPERIMENTAL RESULTS</td>
<td>66</td>
</tr>
<tr>
<td>5. CONCLUSIONS</td>
<td>76</td>
</tr>
<tr>
<td>5.1 Recommendations</td>
<td>77</td>
</tr>
<tr>
<td>6. BIBLIOGRAPHY</td>
<td>79</td>
</tr>
<tr>
<td>6.1 Additional References</td>
<td>80</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 2.1: Sorted list of Example 2.1 .............................. 14
Table 2.2: Cut indices of Example 2.2 .............................. 17
Table 2.3: List of all cut indices of Example 2.4 ................. 21
Table 2.4: List of acceptable cut indices of Example 2.4 ......... 21
Table 2.5: List of acceptable cut indices of Example 2.5 ......... 24
Table 2.6: Interface nodes of the first partitioning point of Example 2.5 25
Table 2.7: Interface nodes of the second partitioning point of Example 2.5 25
Table 2.8: Gains for the interface nodes of Example 2.6 .......... 27
Table 2.9: List of acceptable cut indices with their nodes of Example 2.7 34

Table 4.1: Results of Example 4.1 ................................. 68
Table 4.2: Results of Example 4.2 ................................. 70
Table 4.3: Results of Example 4.3 ................................. 71
Table 4.4: Results of Fig. 4.8 and Fig. 4.9 ....................... 71
Table 4.5: Comparison ................................................. 74
LIST OF FIGURES

Figure 1.1: Branch tearing algorithm ........................................ 3
Figure 1.2: Node tearing algorithm ........................................... 4
Figure 1.3: Complete graph representation of VLSI circuits ............... 6
Figure 1.4: Transforming VLSI circuits into graphs ....................... 7

Figure 2.1: Graph of Example 2.1 ............................................ 14
Figure 2.2: Graph of Example 2.4 ........................................... 22
Figure 2.3: Partitioned graph of Example 2.4 ............................... 23
Figure 2.4: Optimized result of Example 2.5 ............................... 26
Figure 2.5: Graph of Example 2.6 before optimized ....................... 27
Figure 2.6: Optimized result of Example 2.6 ............................... 28
Figure 2.7: Optimization .................................................... 39

Figure 3.1: Empty window ................................................... 45
Figure 3.2: A filename.crd file .............................................. 47
Figure 3.3: A filename.net file .............................................. 48
Figure 3.4: Dialog window .................................................. 49

Figure 4.1: Graph of Example 4.1 without optimization ................. 67
Figure 4.2: Graph of Example 4.1 with optimization ..................... 67
DEDICATION

To my parents,
Siti Lian and Patrice Pohar Gosti,
and my family.
ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and thanks to my major professor, Dr. Marwan M. Hassoun, for his invaluable guidance, encouragement, technical advice, and financial assistance throughout this research. Thanks are extended to Dr. Charles T. Wright and Dr. Gurpur M. Prabhu for their suggestions and being on my committee.

I would also like to thank Dr. David Fernandez-Baca for his advice and suggestions on graph algorithms.
A heuristic algorithm for partitioning of VLSI circuits which attempts to minimize the number of branches cut between partitions is presented in this thesis. The algorithm uses a reduced connectivity matrix and the concept of cut index. The first phase of the algorithm, which is called the partitioning phase, finds the cut lines of the graph, and the second phase of the algorithm, which is called the optimization phase, optimize the branches cut by the partitions. The required computation time is shown to be bounded by $O(n^2)$, where $n$ is the number of nodes in a graph. The proposed algorithm is currently implemented using X Window System. The program is called Reduced Connectivity Matrix Partitioning, abbreviated as rcmp. A universal netlist format used in the rcmp allows it to interface to most existing VLSI CAD tools. Experimental results show that the proposed algorithm yields optimal solutions in most cases and near optimal solutions in some cases.
1. INTRODUCTION

Integrated circuits have been increasing in the number of components that can fit in a unit area to such a point that CAD tools are becoming necessary to aid in design and layout. Since design cost is proportional to the design time and area, fast CAD tools that minimize the area required for a design are very desirable. Three major problems in CAD tools are floorplanning, placement, and routing. Fast and efficient algorithms for all of these problems are necessary to come up with good CAD tools. In all of these problems, partitioning is often used to minimize delay and area. In floorplanning, hierarchical partitioning is used to decompose a circuit before the circuit blocks are placed [1]. In placement, partitioning is often used to find strongly connected subcircuits. A placement algorithm can then place those components together, minimizing both delay and routing area [2]. Partitioning is often used in circuit simulation problems as well. A large circuit is often partitioned into smaller subcircuits. These subcircuits are analyzed independently, and the solutions are finally merged to get the solution to the original circuit. Partitioning minimizes the simulation time, and makes the analysis of large circuits possible [3] [4] [1].

Because partitioning belongs to the class of NP-complete problems [5], only heuristic approaches are possible. There have been a number of such approaches proposed. Several of these approaches are:
• Kernighan and Lin [6]: This approach starts with any arbitrary two partitions. A series of interchanges of a subset of each partition is performed to minimize the external cost of these two partitions. This 2-way partition procedure can be extended to perform a k-way partition procedure where a graph is partitioned into k subgraphs.

• Charney and Plato [7]: This approach is based on the use of an electrical analogue. A network of interconnecting components is divided into groups of components so that the sum of the square of each interconnecting wire length is minimum.

• Russo, Oden, and Wolff [8]: This approach is also called the group generation and group allocation algorithm. All circuit blocks are assigned into functional groups. These groups are then assigned to modules or partitions to minimize the interconnection wires or pins.

• Rutkowski [9]: This approach uses the concept of partitioning the fundamental loop matrix for a priori chosen tree. First, loop indices are calculated for all rows (trees) in the fundamental loop matrix. This matrix is iteratively permuted and partitioning is decided from the final permuted fundamental loop matrix.

• Hassoun [3] [4]: This approach uses a similar method as that by Rutkowski. In addition to loop indices, tearing indices and branch indices are introduced to optimize either the sizes of the partitions or the number of tearing nodes on path.

• Sangiovanni-Vincentelli, Chen, and Chua [1]: This approach is based on the
The concept of a contour tableau which consists of an array of three columns: iterating set, adjacent set, and contour number. The partitioning is decided from this contour tableau.

The partitioning algorithm proposed in this thesis is a branch tearing algorithm. A branch tearing algorithm is a graph partitioning algorithm where subgraphs are separated by branches cut by the algorithm. The other type of graph partitioning is node tearing, where subgraphs are separated by nodes cut by the algorithm. Fig. 1.1 shows an example of a branch tearing, and Fig. 1.2 shows an example of a node tearing.

The proposed algorithm uses undirected graphs as inputs. For this reason, VLSI circuits need to be represented as graphs. The following section illustrates the graph representation of VLSI circuits.
1.1 Graph Representation of VLSI Circuits

VLSI circuits consist of interconnected circuit modules which can be characterized as:

1. low level or primitive modules, like resistors, capacitors, and transistors,

2. medium level modules, like bit adders, counters and multipliers in standard cell, gate array, and custom libraries, and

3. high level modules, which can be subsystems like array multipliers, systolic processors, and ALUs.

Among these circuit modules, the interconnections are called nets. A collection of nets forms a netlist. The definitions of both net and netlist are stated as follows:
Definition 1.1 A net is a set of interconnections which are to be electrically common [10].

Definition 1.2 A netlist is an interconnection of circuit modules that represent the implementation of the design. The connections represent logical connections among the modules instead of physical placement [2].

As stated above, a VLSI circuit consists of circuit modules and nets. The algorithm that generates the graph representation of the circuit depicts each circuit module and each net according to the following rules:

- A circuit module is represented as a node in its corresponding graph. If a VLSI circuit has $n$ modules, then its graph will have $n$ nodes.

- A net can be represented in several ways. The most complete and representative form would be to fully model the interconnections between its nodes. This is shown in Fig. 1.3. The overhead of such a representation for a net with $n$ modules is:

$$b_{complete} = \frac{n(n - 1)}{2}$$  \hfill (1.1)

branches (a complete graph) [11]. Since each branch contributes an entry to the reduced connectivity matrix ($M$), which is the representation of the graph on which the partitioning is performed (see Chapter 2), the sparsity of the matrix is greatly reduced. An alternative way is a simpler representation which preserves the sparsity of the $M$ matrix. If a net consists of terminals from only 2 modules, it is represented as a branch connecting both nodes corresponding to the two modules. However, if a net consists of terminals from more than
2 modules, it is represented as two branches with a branch connecting every node on the net to two other nodes also on the net (Fig. 1.4). For the case where any two nodes in the resulting graph have \( w \) parallel branches between them, they are replaced by one branch with an assigned weight of \( w \). The space complexity for the latter representation is \( O(n) \), as opposed to that of the former representation, \( O(n^2) \).

Fig. 1.4 illustrates examples of how to represent VLSI circuits into undirected graphs.

After the VLSI circuit has been represented as a graph, the partitioning algorithm is applied.

For software packages as large as VLSI CAD tools, graphic interfaces are usually included to relieve the users from having to interact with character-based interface which is harder to use and less user-friendly. For this reason, the partitioning algorithm proposed in this thesis has been implemented in X Window System [12], a window system which has gained a lot of acceptance among the academic and business worlds. This implementation is called Reduced Connectivity Matrix Parti-
Figure 1.4: Transforming VLSI circuits into graphs
tioning, abbreviated as *rcmp*. *rcmp* uses a netlist as its input and output, since it will enable the partitioning algorithm to be interfaced with almost any existing set of CAD tools.

The algorithm proposed in this thesis uses the reduced connectivity matrix and the concept of a cut index as tools for partitioning. Chapter 2 presents the partitioning algorithm, where the problem of partitioning is stated, and the complete algorithm, which consists of a partitioning phase and an optimization phase is discussed. The complexity of this algorithm is also analyzed in Chapter 2. X Window System implementation of the *rcmp* is discussed in Chapter 3. A number of experimental results are shown in Chapter 4. Chapter 5 states the conclusions and recommendations for future work.
2. PARTITIONING ALGORITHM

After a VLSI circuit has been represented as a graph, the algorithm proceeds by partitioning the graph. This chapter starts by clearly stating the partitioning problem. It continues with the discussion of the first phase of the partitioning algorithm, which is the partitioning phase. The second phase of the algorithm, which is the optimization phase is added because no significant overhead is needed to improve the solution. This optimization phase is also covered in this chapter. The next section outlines the whole algorithm in pseudocode. This chapter closes by analyzing the complexity of the algorithm.

2.1 Partitioning Problem

The following definition states the partitioning problem, the partitioning criteria, and the optimal solution that the algorithm attempts to find.

**Definition 2.1** Let \( G = (N, B) \) denote an undirected graph, where \( N \) is a set of nodes and \( B \) is a set of branches. Let \( n \) be the number of nodes in \( N \), and \( b \) be the number of branches in \( B \). For each branch \( j (j = 1, \ldots, b) \), there is an associated weight \( w_j \) assigned to it. The problem is to partition \( G \) into \( g \) subgraphs \( G_k = (N_k, B_k), k = 1, \ldots, g \) such that

1. \( g \leq g_{\text{max}} \), where \( g_{\text{max}} \) is the maximum number of subgraphs allowed,
2. $n_k \geq n_{\text{min}}, k = 1, \ldots, g$, where $n_{\text{min}}$ is the specified minimum number of nodes that $G_k$ must have,

3. $\cup N_k = N$ and $N_i \cap N_j = \emptyset$ for $i \neq j$, and

4. the sum of the weights of the branches connecting $G_i$ to $G_j$ is minimum, for $j = 1, \ldots, g$ and $j \neq i$.

2.2 Partitioning Phase

This phase of the algorithm involves finding the optimal cuts on a graph. The step-by-step procedure for this phase for a given graph is:

1. Build a reduced connectivity matrix $M$ (see Definition 2.3 below) for this graph.

2. Calculate the cut index of each row of $M$.

3. Permute rows of $M$.

4. Select the partitioning points.

Steps 2 and 3 need to be iterated until a permuted matrix is found and the whole procedure can be recursively executed if necessary. Each of these steps is explained in more detail below.

2.2.1 Reduced connectivity matrix ($M$)

This section starts by defining what an independent set of a graph $G$ is. The significance of this independent set will be clear later.
Definition 2.2  An independent set $R$ in an undirected graph $G = (N, B)$ is a set of nodes no two of which are connected [13].

The following defines what a reduced connectivity matrix is:

Definition 2.3 For a graph $G = (N, B)$, an independent set $R$ of $G$ of size $r$, and the remaining nodes $C$ of $G$ of size $c$, the reduced connectivity matrix $M$ is an $r \times c$ matrix

$$M = \{m_{ij}\} \quad (2.1)$$

where $m_{ij}$ is equal to the weight of the branch whose nodes are node $i$ in $R$ and node $j$ in $C$, 0 if no such branch exists. $R$ is called the row-nodes, and $C$ is called the column-nodes.

Observe that there is a need to find the independent set of the graph $G$, which is the graph representation of the VLSI circuit to be partitioned. For this purpose, an algorithm has been devised to find this independent set $R$ in logarithmic time.

For the algorithm described below, remember that $R$ and $C$ are initially empty. The following is the steps of the independent set finding algorithm:

1. Sort the nodes in $N$ in the increasing sum of weights of branches incident on each node. Call this sorted list $L$.

2. Let $N_i$ be the first node in $L$.

3. Append $N_i$ to $R$ if $N_i$ is not already in $R$ or $C$; otherwise, go to Step 5.

4. Append all neighboring nodes of $N_i$ to $C$.

5. Advance $N_i$ to the next node in $L$. 
6. If \( L \) is not empty, go to Step 3.

The result of this algorithm will be an independent set of \( G \) stored in \( R \) and all remaining nodes of \( G \) stored in \( C \).

This matrix \( M \) basically describes the connectivity between \( R \) and \( C \); however, it does not fully describe the connectivity of \( G \). To fully describe \( G \), an auxiliary matrix \( A \) is needed to describe the connectivity among nodes in \( C \), and it is defined as follows:

\textbf{Definition 2.4} The auxiliary matrix \( A \) for a graph \( G \) and a set of \( c \) chosen column-nodes \( C \) is a \( c \times c \) matrix

\[
A = \{a_{ij}\}
\]

(2.2)

where \( a_{ij} \) is equal to the weight of the branch whose end nodes are column-node \( i \) and column-node \( j \), 0 if no such branch exists.

\( M \) together with \( A \) fully describe \( G \), because a similar auxiliary matrix to describe the connectivity among nodes in \( R \) is not needed. The reason it is true is that every node in \( R \) is not an adjacent node to any other node in \( R \) since \( R \) is an independent set.

\textbf{Example 2.1} The sorted list \( L \) for the graph in Fig. 2.1 is shown in Table 2.1. \( b_i \) is the number of branches of node \( i \). Its \( M \) matrix is
Its matrix $A$ is

$$
\begin{bmatrix}
0 & 3 & 13 & 9 & 2 & 5 & 7 & 10 & 12 \\
4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
14 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
8 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
11 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}
$$

2.2.2 Cut index

The partitioning algorithm is done by permuting rows of $M$ and then partitioning it into the submatrices $M_1, ..., M_g$ as follows:
Figure 2.1: Graph of Example 2.1

Table 2.1: Sorted list of Example 2.1

<table>
<thead>
<tr>
<th>$i$</th>
<th>$b_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
where each submatrix $M_i$ defines a partition or a subgraph $G_k$ of $G$. The permutation of rows of $M$ and the selection of submatrices are based on the concept of cut index. This cut index was first proposed by Hassoun in [3] as tearing index. The tearing index concept has been modified accordingly to suit the branch tearing algorithm proposed in this thesis as opposed to the node tearing algorithm suggested by Hassoun and Lin.

Associated with each node in $R$, there is a cut index $c_i$. The cut index of row $i$ of $M$, $c_{ii'}$, may be defined as follows:

**Definition 2.5** The cut index $c_{ii'}$ of the $i^{th}$ row of $M$ is the number of branches shared between the subgraph defined by the first through $i^{th}$ row and the rest of the graph.

So the cut index says that if the first $i$ rows of $M$ were joined together to form a partition (subgraph) of $G$, then the partition will have $c_{ii'}$ tearing branches. The computation of $c_{ii'}$ is illustrated by the following example:

**Example 2.2** The graph of Fig 2.1 is used again in this example. Its permuted matrix $M_p$ and matrix $A$ are:
To calculate $c_{i1}$, notice that the only non-zero entries in row 1 are in column 13, 10, and 12. The sum of the entries in these columns from row 2 to row 15 excluding row 9, 14, and row 15 is 4. Therefore $c_{i1} = 4$. For $c_{i2}$, the non-zero entries in row 1 and row 2 are in column 13, 9, 10, and 12. The sum of the entries in these four columns from row 3 to row 15 excluding row 9, 10, 14, and 15 is 3. Therefore $c_{i2} = 3$. $c_{i3}, c_{i4}, c_{i5}$, and $c_{i6}$ are calculated in the same way. The results are shown in Table 2.2.
Table 2.2: Cut indices of Example 2.2

<table>
<thead>
<tr>
<th>i</th>
<th>$c_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Since $A$ is usually a larger matrix than $M$, and its function is just to calculate $c_i$, a list of branches describing the connectivity of column-nodes $C$ is used to replace $A$ in the implementation. The sparsity of matrices $M$ and $A$ is discussed in the Complexity of Algorithm section in this chapter.

2.2.3 Permutation of rows

The cut index is used as a criterion to decide which rows to swap. The row with the minimum $c_{i_1}$ is permuted to row 1. Next, $c_{i_2}$ is calculated for the remaining rows, and the row with the minimum $c_{i_2}$ is permuted to row 2. This procedure is repeated until all rows are permuted. The resulting matrix is the permuted matrix $M_p$. Heuristically, $M_p$ will have a local minimum $c_{i_i}$ at row $i$ where the number of tearing branches is optimal or near optimal if $G$ is partitioned at row $i$. Row $i$ is called the partitioning point.

**Definition 2.6** The partitioning points of a permuted matrix $M_p$ are the rows of $M_p$ which cut indices are local minimum among all the cut indices in $M_p$.

The following example illustrates how to get $M_p$ from $M$:
Example 2.3 Again, consider the graph of Fig. 2.1 and the \(M\) matrix from Example 2.1:

\[
\begin{array}{cccccccccc}
0 & 3 & 13 & 9 & 2 & 5 & 7 & 10 & 12 & ci_1 \\
4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\
14 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 5 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 5 \\
6 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 6 \\
8 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 6 \\
11 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 4 \\
\end{array}
\]

**Step 1:** Calculate \(ci_1\) for all rows and swap the row with minimum \(ci_1\) with row 1.

\[
\begin{array}{cccccccccc}
0 & 3 & 13 & 9 & 2 & 5 & 7 & 10 & 12 & ci_2 \\
11 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
14 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 3 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 9 \\
6 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 10 \\
8 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 4 \\
4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \\
\end{array}
\]

**Step 2:** Calculate \(ci_2\) for row 2, 3, 4, and 5 and swap the row with minimum \(ci_2\) with row 2.
Step 3: Continue with $ci_3$, and $ci_4$. The resulted $M_p$ is

$$
\begin{array}{cccccccccc}
0 & 3 & 13 & 9 & 2 & 5 & 7 & 10 & 12 & ci_3 \\
11 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
14 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
8 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
$$

which is the $M_p$ of Example 2.2.

2.2.4 Selecting partitions

Based on the cis of the permuted matrix $M_p$, two or more partitions are selected. A subgraph $G_k$ defined by the submatrix $M_k$ consists of all row-nodes in $M_k$ and all column nodes that have at least 1 non-zero entry in one of the rows of $R$ in $M_k$ and not covered by subgraphs $G_1, \ldots, G_{k-1}$. 
Let $n_{min}$ define the minimum number of nodes that have to be present in a partition. From the cis of $M_p$, let $c_{istart}$ and $c_{iend}$ be defined as:

\begin{align*}
\text{c}_{istart} & = \text{c}_{ifirst} \\
\text{c}_{iend} & = \text{c}_{ilast}
\end{align*}

where

\begin{align*}
\text{c}_{ifirst} & = \text{first } c_i \text{ in } M_p \text{ that covers at least } n_{min} \text{ nodes} \\
\text{c}_{ilast} & = \text{last } c_i \text{ in } M_p \text{ that leaves at least } n_{min} \text{ nodes}
\end{align*}

Using the $c_{istart}$ and $c_{iend}$, a list of acceptable cis can be defined as follows:

Definition 2.7 A list of acceptable cis, $I$, is the set of all local minimum cis of $M_p$ between $c_{istart}$ and $c_{iend}$.

From this list, the smallest $c_i$ is called $c_{imin}$.

If partitioning were performed at the row defined by $c_{imin}$ only, the algorithm would be a bisection algorithm. However, an attempt is made to partition the graph $G$ into as many subgraphs as possible without exceeding $g_{max}$, the maximum number of subgraphs allowed. For this purpose, another partitioning criterion is introduced, which is referred to as tolerance $tol$. The idea is to partition $G$ at rows defined by $c_i < c_{imin} + tol$. However, only rows with local minimum $c_i$ are compared to $c_{imin} + tol$ to be considered for partitioning points. These points will provide better partitions since these points are the local minimum of the number of branches cut by the partitions.

The following example illustrates $n_{min}$, $c_{imin}$, and $tol$ mentioned above:
Example 2.4 Consider the graph in Fig. 2.2. The number of nodes in this graph is equal to 46. Let $n_{\text{min}} = 10$, $\text{tol} = 2$, $g_{\text{max}} = 3$, and the number of nodes corresponding to $ci_i$ be called $n_i$. The cis of its permuted matrix are shown in Table 2.3. Because $n_{\text{min}} = 10$, the list of acceptable cis ranges from $ci_4 = 8$ and $n_4 = 12$ to $ci_{13} = 6$ and $n_{13} = 36$. The list of acceptable cis ($I$) is shown in Table 2.4.

Table 2.3: List of all cut indices of Example 2.4

<table>
<thead>
<tr>
<th>$i$</th>
<th>$ci_i$</th>
<th>$n_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>31</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 2.4: List of acceptable cut indices of Example 2.4

<table>
<thead>
<tr>
<th>$i$</th>
<th>$ci_i^a$</th>
<th>$n_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>36</td>
</tr>
</tbody>
</table>

$^a$When 2 or more cis are the same consecutively, the second and the rest of the cis are considered for partitioning in the $rcmp$ implementation.
From this list, $c_{i_{\text{min}}}$ is 5. With $tol = 2$, only the cis number 12 and 13 are considered. The first partitioning point is set at $ci_6$. If a partitioning point is set at $ci_{12}$, $ci_{13}$ is automatically not a partitioning point since the number of nodes between $ci_{12}$ and $ci_{13}$ is only 3 ($< 10$). The partitioned graph is shown in Fig. 2.3.

2.3 Optimization Phase

By observation from Example 2.4, the partitioning phase gives a result close to the optimal solution. However, the optimal solution would have been reached if node 19 had been moved from partition I to partition II, node 15 from partition II to partition I, and node 36 from partition III to partition I. This section deals with this optimization procedure and is therefore called the optimization phase. This optimization is a modified version of the Kernighan and Lin algorithm in [6].

Let the nodes whose branches consist of at least one tearing branch be called
interface nodes. Associated with each interface node, there are an external cost and an internal cost. An external cost $E_i$ of an interface node $i$ is the sum of the weights of the tearing branches incident on $i$, and its internal cost $I_i$ is the sum of the weights of all other branches incident on $i$. Let the gain $D_i$ be the difference between $E_i$ and $I_i$ (i.e., $D_i = E_i - I_i$) Mathematically, $E_i$, $I_i$, and $D_i$ are:

$$E_i = \sum_{j=1}^{x} w_j$$

$$I_i = \sum_{j=1}^{t} w_j$$

$$D_i = E_i - I_i$$

where

$x$ = number of branches of interface node $i$ which are cut branches

$t$ = number of branches of interface node $i$ which are not cut branches
Table 2.5: List of acceptable cut indices of Example 2.5

<table>
<thead>
<tr>
<th></th>
<th>( c_{i_j} )</th>
<th>( n_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>36</td>
</tr>
</tbody>
</table>

The optimization process is done by moving interface nodes with positive gains to the appropriate partitions. If the gain \( D_i \) of an interface node \( i \) is positive, then it implies that moving \( i \) from the partition where it is to the other partition will reduce the number of cut branches by \( D_{ii} \); however, if the gain \( D_i \) is negative, then it implies that moving \( i \) from the partition where it is to the other partition will increase the number of cut branches by \( D_i \). This optimization is not done after selecting all the partitions, however. It is done with the selection of partitions simultaneously. The following example illustrates how the optimization process is done:

**Example 2.5** Using the same graph as in Example 2.4. The list of acceptable \( cis \) (I) is shown again in Table 2.5.

First, a partitioning point is set at \( ci_6 = 5 \) and \( n_6 = 17 \). The interface nodes of Fig 2.3 are tabulated in Table 2.6.

Note that only node 18 and 35 have positive gains. The node that has the largest \( D_i \) is optimized first. In this case, since \( D_{18} \) and \( D_{35} \) are the same, either one of them can be considered first. Let node 18 be the first to be moved from the rest of the nodes to partition I. \( D_{35} \) is then updated, and optimized by moving it from partition III to partition I. The second partitioning point is set at \( ci_{12} = 5 \) and \( n_{12} = 33 \). The interface nodes are tabulated in Table 2.7. Since \( D_{17} \) is positive, node 17 is moved from partition II to partition III.
Table 2.6: Interface nodes of the first partitioning point of Example 2.5

<table>
<thead>
<tr>
<th>Interface Nodes (i)</th>
<th>$E_i$</th>
<th>$I_i$</th>
<th>$D_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 2.7: Interface nodes of the second partitioning point of Example 2.5

<table>
<thead>
<tr>
<th>Interface Nodes (i)</th>
<th>$E_i$</th>
<th>$I_i$</th>
<th>$D_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>
Figure 2.4: Optimized result of Example 2.5

The optimized solution is shown in Fig. 2.4.

This optimization steps are only applied to optimize the interface nodes that have positive gain. It does not perform a successive optimization. The nodes with gains becoming positive after certain nodes have been moved between partitions are not considered for optimization. This is sufficient since the goal of the optimization phase is to reduce the already acceptable number of tearing branches. The following example illustrates the idea:

**Example 2.6** Assume that a graph has been partitioned into 2 partitions, $G_1$, and $G_2$. The interface nodes of partition $G_1$ and the cut line are shown in Fig. 2.5. The gains for interface nodes 1, 2, and 3 are shown in Table 2.8. The optimization algorithm will move node 1 from partition $G_1$ to partition $G_2$, as shown in Fig. 2.6. Node 2 will not be moved because it had a gain of -1 initially.
Table 2.8: Gains for the interface nodes of Example 2.6

<table>
<thead>
<tr>
<th>i</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Figure 2.5: Graph of Example 2.6 before optimized
2.4 Global Algorithm

Essentially, the partitioning algorithm can be outlined in the following steps:

Step 1: Sort the nodes in the order of increasing number of branches incident on them.

Step 2: Build matrix $M$ and auxiliary matrix $A$ to represent the connectivity of all nodes.

Step 3: Permute matrix $M$ based on the concept of cut index to get the permuted matrix $M_p$.

Step 4: Select partitioning points and perform the optimization simultaneously based on the partitioning criteria $n_{min}$, $g_{max}$, and $tol$.

Step 5: If the number of partitions is less than $g_{max}$, recursively run the algorithm on the partitions (subgraphs) until $g_{max}$ is reached or all cut indices are
greater than $c_{\text{min}} + \text{tol}$.

The algorithm can be stated as follows:

```plaintext
recur(M, g_{max})
{
    /* Sort nodes and build M matrix */
    build(M);
    /* Calculate $c_{is}$ for each row and obtain $M_{p}$ */
    calc_index(M);
    /* Select partitions and optimize */
    num_part = select_part(M, g_{max})
    if ((num_part == g_{max}) || (num_part == 1)) return;
    /* For each partition, update $g_{max}$ and call this routine again */
    for (each partition) {
        - update $g_{max}$
        recur(M_k, g_{max});
    }
} /* End of recur */
```

## 2.5 Complexity of Algorithm

### 2.5.1 Reduced connectivity matrix and auxiliary matrix

The reduced connectivity matrix $M$ has $r$ rows and $c$ columns. Because each node only appears once in either $R$ or $C$, it follows that $r + c = n$. $M$ will have at
most

\[
    \text{number of elements in } M = \frac{n}{2} \times \frac{n}{2} = \frac{n^2}{4}
\]  

(2.9)
elements, that is, if \( r = \frac{n}{2} \) and \( c = \frac{n}{2} \). The auxiliary matrix \( A \) can be represented by a list of branches that are not covered by \( M \). If a graph is complete, then its independent set consists of only one element, which means that there is only one row in \( M \), and there are \( n - 1 \) columns in \( M \). This is the case when \( A \) has the most elements, which are equal to

\[
    \text{number of elements in } A = (n - 1)(n - 2)/2
\]  

(2.10)
elements. The upper bound of the total space \( S \) required for storing the graph to be partitioned in the worst case is less than

\[
    S < \frac{n^2}{4} + (n - 1)(n - 2)/2 = 3n^2 - 6n + 4/4
\]  

(2.11)
However, since \( M \) and \( A \) matrices are very sparse in practice, only nonzero elements need to be stored. The number of nonzero entries in \( M \) and \( A \) is equal to the number of branches in the graph. Thus, the total space actually required by the partitioning algorithm to store a graph is equal to \( b \), the number of branches in that graph. If it is compared with the size of connectivity matrix, which is \( 2b \), it is half that size.

Let the algorithm to build matrix \( M \) be called \texttt{buildM()}\), which is presented below:

\begin{verbatim}
buildM()
    L = sort(N)  /* sort all nodes */
    i = first node in L
    while (L is not empty) {
        ...
    }
\end{verbatim}
if (i is not in C) {
    add i to R;
    append all adjacent nodes of i to C, and build a node in M for each node appended;
}
i = next node in L;
}

This algorithm takes $O(n^2)$ time to complete, where $n$ is the number of nodes in the graph $G$.

The algorithm for building $A$ is as follows:

buildA()
{
    i = first node in C;
    j = next of i;
    while (i is not NULL) {
        while (j is not NULL) {
            if (j is adjacent to i)
                build a branch-node connecting i and j;
                j = next of j;
        }
        i = next of i;
        j = next of i;
    }
}
This algorithm also takes $O(n^2)$ time to complete.

2.5.2 Cut index

It is shown in the permutation of rows section in this chapter that cut index $c_{i,i}$ is calculated for row $i, i+1, \ldots, r$, where row $r$ is the last row in $M$. $c_{1,1}$ is calculated for all $r$ rows; $c_{i,2}$ is calculated for all $r-1$ rows except the first row; $c_{i,3}$ is calculated for all $r-2$ rows except the first two rows; and so on. The total number of cut index calculations is

$$y = r + (r - 1) + (r - 2) + \cdots + 2 = \frac{r(r + 1)}{2} - 1 \quad (2.12)$$

The cut index for row $i$, $c_{i,i}$, can be calculated from the following equation:

$$c_{i,i} = c_{i,i-1} - \sum_{j=1}^{u} b_{ij} + \sum_{j=1}^{u} b_{ij} d_{j} f_{j} + \sum_{j=1}^{u} \sum_{k=1}^{v} (a_{jk} \& 1) d_{j} \quad (2.13)$$

where

- $u$ = number of nonzero elements in row $i$
- $v$ = number of branches connecting a column-node to any other column-node
- $b_{ij}$ = row $i$ and column $j$ entry of $M$
- $d_{j}$ = 0, if a node corresponding to the $j$th column is present in subcircuits permuted into first $(i-1)$ rows of $M$
- $= 1$, otherwise [9]
The computational complexity of calculating a single cut index, \( c_i \), is defined by \( u \) and \( v \). Practically, on the average, \( u \) and \( v \) will be small constants, since they are at most equal to the number of branches to which a node is connected or the degree of connectivity of a node (a module in VLSI circuits). Stated differently, \( u \) and \( v \) are numbers representing nonzero entries in a row of a very sparse matrix. Therefore, the complexity of calculating \( c_i \) does not depend upon \( r \) and \( c \).

Thus, all cut indices can be calculated in

\[
y = r(r + 1)/2 - 1
\]

(2.14)

time or \( O(n^2) \) time.

Because \( cis, ds, \) and \( fs \) are linear lists, the space complexity for storing \( cis, ds, \) and \( fs \) is linear.

### 2.5.3 Permutation of rows

There are at most \( r \) permutations in this algorithm. To permute a row, \( ci_{min} \) is needed from a list of \( cis \). This \( ci_{min} \) can be calculated at the same time as the list of \( cis \) are calculated. From this \( ci_{min} \), it takes \( O(1) \) time to permute a row. Thus, total
Table 2.9: List of acceptable cut indices with their nodes of Example 2.7

<table>
<thead>
<tr>
<th>i</th>
<th>c_i</th>
<th>n_i</th>
<th>Acc_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>{0,1,3,4,5,6,7,8,9,0,11,18}</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>17</td>
<td>Acc_4 + {2,12,13,14,35}</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>33</td>
<td>Acc_6 + {15,16,17,19,20,21,22,23,24,25,26,27,28,29,30,31}</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>36</td>
<td>Acc_{12} + {43,44,45}</td>
</tr>
</tbody>
</table>

The time needed to perform all permutations is $O(r)$ or $O(n)$, the upper bound because $n > r$. Since permutations can be done by swapping pointers, $O(1)$ extra space is required.

2.5.4 Selecting partitions

After all rows are permuted, the resulting matrix is $M_p$. From the Permutation of Rows section, it is shown that all $c_i$s are found as soon as $M_p$ is built. From all these $c_i$s, a list of acceptable cut indices $I$ are selected according to the Definition 2.7. Since there are $r$ rows in $M$, there are at most $r$ elements in $I$. For each $c_i$ in $I$, a list of nodes defined by the subcircuit permuted into the first $(i-1)$ rows of $M_p$ needs to be stored to keep track of all nodes that are in one partition if the cut is performed at row $i$. Let this list be called $Acc$.

**Example 2.7** Consider the list of acceptable $c_i$s ($I$) of Example 2.4. This $I$ is listed in Table 2.9 together with its list of the nodes stored ($Acc$).

The total space to store this $I$ and its corresponding list of nodes $Acc$ in the worst case is equal to $r$ (number of rows in $M$) multiplied by $n/2$ (average number of nodes stored for each $c_i$ in $I$) or $O(n^2)$.

The following is the algorithm for selecting $I$ and building $Acc$:
buildIO()
{
    find first row \( p \) in \( M_p \) with number of nodes permuted into first \( (p - 1) \) rows of \( M_p \) \( \geq n_{\min} \), where \( n_{\min} \) is equal to the minimum number of nodes to be present in one partition;
    \( last_i = p; \)
    for \( i := p \) to \( r \) {
        if ((\( n \) - (number of nodes permuted into first \( i \) rows)) < \( n_{\min} \))
            break;
        if (\( c_i \) \( > \) \( c_{i+1} \))
            continue;
        /* \( c_i \) is an acceptable \( c_i \) */
        append \( c_i \) to \( I \);
        buildAcc(\( last_i, i \));
        \( last_i = i; \)
    }
}

buildAcc(\( last_i, i \))
{
    copy all nodes from Acc[\( last_i \)] to Acc[i];
    for \( j := last_i \) to \( i \) {
        append row-node \( j \) to Acc[i];
        append all column-nodes adjacent to row-node \( j \) to Acc[i];
    }
}
this step can be done in linear time because there is a pointer from m\_jl to column-node l if column-node l is connected to row-node j */
}
}

buildAcc() takes linear time to complete execution, proportional to the number of nodes in Acc[i]. The loop in buildI() will execute at most r times, since the number of rows in M\_p is r. Thus, buildI() takes O(n^2) time to execute.

The next step is selecting partitions from I and Acc. The idea is to select the first g partitions where g \leq gmax, and each partition has n\_k nodes where n\_k \geq n\_min. The algorithm below states it in more detail

select()
{
    n\_part = 0;
    N\_part initially empty; /* Dummy storage */
    for i := 1 to s { /* s = number of cis in I */
        if ((ci of Acc[i] \leq cim\_in + tol) and
            (number of nodes in Acc[i] - n\_part \geq n\_min)) {
            build one partition from Acc[i] - N\_part;
            add Acc[i] to N\_part;
            n\_part = number of nodes in N\_part;
        }
    }
    if (number of partitions > gmax)
repeat the whole algorithm for all partitions;

while (number of partitions > gmax)
    combine two partitions with the most number of cut branches;

The procedure to build one partition in this algorithm takes linear time, i.e.,
proportional to the number of nodes in the partition. The first loop in this algorithm
therefore takes

\[ T(n) = s \times \text{number of nodes in Acc}[i] \]  

(2.15)

or \( O(n^2) \) time to complete. The recursive part of this algorithm is executed last,
whether or not optimization is performed. The complexity of this recursive part will
be analyzed in the Total Complexity section, since it is the total complexity of the
algorithm.

This algorithm needs \( O(n) \) space for storing each partition, or \( \text{num} \) (number of
partitions) \( \times \) \( O(n) \) for \( \text{num} \) partitions. At the worst case, \( O(n^2) \) space is required.

2.5.5 Optimization

The following formulas are used to determine if a node is an interface node:

\[ e_j = \sum_{k=1}^{v} (a_{jk} \& 1) \]  

(2.16)

for nodes that are included in subcircuits permuted into first \((i-1)\) rows of \( M \), and

\[ e_j = \sum_{k=1}^{v} (a_{jk} \& 1) + f_j \]  

(2.17)

for nodes that are not included in subcircuits permuted into first \((i-1)\) rows of \( M \).
If $e_j > 0$, then the corresponding node is an interface node; otherwise, it is not. The time required for these calculations is constant for each node in column-node $C$, again because $v$ is a small constant. Thus, to find all interface nodes, $O(c)$ time is required. These calculations are not needed for row-nodes $R$ because $R$, which is an independent set, can not be interface nodes.

An interface matrix is built after all interface nodes have been found.

**Definition 2.8** An interface matrix $T$ of a partition defined by the subcircuit permuted into first $(i-1)$ rows of $M$ is an $l \times m$ matrix whose rows are built from all interface nodes that are not part of this subcircuit, and whose columns are built from all interface nodes that are part of this subcircuit with

$$
T = \{t_{ij}\}
$$

where $t_{ij}$ is equal to the weight of the branch whose end nodes are interface nodes $i$ and $j$, it is equal to zero if no such branch exists.

The time needed to build this matrix $T$ is $O(n(l + m))$ since for each node in $T$ which has $(l + m)$ nodes, a scan through all the $n$ nodes is necessary to find out if it is connected to some node. In terms of $n$, $O(n(l + m))$ is proportional to $O(n^2)$ although it is much smaller than $n^2$ for large $n$.

This interface matrix is built so that the optimization is correctly performed. Consider the case of Fig. 2.7. The interface matrix for this partitioning result has an entry of 1 in column $a$ and row $b$ as shown below.
Assume that the partitioning algorithm will move node a from partition I to partition II initially. The entry $T[b, a]$ is changed from 1 to -2 because branch $l$ would be counted as a gain twice if $b$ were moved from partition II to partition I. When considering $b$ for optimization, the gain calculation will be

$$gain = external\ cost - internal\ cost + T[b, a] = 2 - 1 - 2 = -1$$

(2.19)

Thus, $b$ will not be moved because $gain \leq 0$.

The algorithm to do the optimization is:
while (gain > 0) {
    gain = 0;
    for (each node in columns of T)
        calculate colgain[i];
    for (each node in rows of T)
        calculate rowgain[i];
    gain = max(rowgain[i],colgain[i]);
    optimize row or column with maximum gain;
    update matrix T;
}

The while loop will execute at most \((l + m)\) times, and each time \((l + m)\) steps are executed. Thus, this algorithm takes \(O((l + m)^2)\) or \(O(n^2)\). The space required for optimization phase is obviously equal to the space required by \(T\) or \(O(n^2)\).

2.5.6 Total complexity

From the discussion in this section, the time complexity of the partitioning algorithm to execute one time is:

\[
T_1(n) = O(n^2) \quad \text{(building M and A)}
\]

\[
+ O(n^2) \quad \text{(calculating all cut indices)}
\]

\[
+ O(n) \quad \text{(permutating rows of M)}
\]

\[
+ O(n) \quad \text{(selecting partitions of M)}
\]

\[
+ O(n^2) \quad \text{(optimizing partitions of M)}
\]
which is equal to \( O(n^2) \). The total time complexity for the algorithm to run recursively is:

\[
T(n) = qT(n/q) + T_1(n) \tag{2.20}
\]

which is equal to:

\[
T(n) = O(n^2) \tag{2.21}
\]

where \( q \) is an arbitrary constant.

The total space complexity is:

\[
S(n) = O(n^2) \quad \text{(space required for M and A, which is } O(b),
\text{ and } b = O(n^2) \text{ in the worst case, i.e. for complete graphs)}
\]

\[
+ O(n) \quad \text{(storing cut indices)}
\]

\[
+ O(1) \quad \text{(permutations of rows)}
\]

\[
+ O(n^2) \quad \text{(storing subcircuits in selecting partitions)}
\]

\[
+ O(n^2) \quad \text{(storing T in optimization phase)}
\]

which is equal to \( O(n^2) \).
3. **rcmp IMPLEMENTATION**

This chapter talks about the implementation of the partitioning algorithm discussed in the previous chapter using X Window System as the graphic interface. First, the history and advantages of using X Window System, as well as a little introduction about X Window System is presented. Next, it gives an overview of *rcmp* (Reduced Connectivity Matrix Partitioning) which is the software implementation of the partitioning algorithm. The next section explains in more detail the implementation of *rcmp*. A list of X Window routines that are used in the *rcmp* is briefly explained in the last section of this chapter.

### 3.1 X Window System

X was born of necessity in 1984 [14]. MIT officials were faced with problems of needing a decent display environment for debugging multiple distribution processes. A project called Project Athena was formed, an undergraduate education program sponsored by Digital and IBM that would ultimately populate the campus with thousands of workstations.

The project started with a prototype window system called *W* that run under Stanford’s *V* operating system. In May 1984, Bob Scheifler replaced the synchronous protocol of *W* with an asynchronous protocol and replaced the display lists with
immediate mode graphics. Without any particular thought about the name, and because the familial resemblance to W was still strong at that date, Bob Scheifler called the result X. Much later, when the name became a serious issue, X had already stuck and was used by too many people to permit a change.

In January of 1986, Digital announced the VAXstation-II/GPX which was the first commercial X implementation. Over the next few months, many other corporations, such as Hewlett Packard, were basing products on version 10, and groups at universities and elsewhere were porting X to other displays and systems, including Apollo Computer and Sun Microsystems workstations.

In September 1987, version 11 release 1 of X was released. The MIT X Consortium, an open organization funded by participants, was created in January 1988 to support and control the development and evolution of the system. The second release of version 11 was available on March 1, 1988 [14].

X is a network-transparent and device-independent window system. Network transparent means that an application program can use other application programs scattered throughout the network without having to worry about the underlying network architecture, as if they were running on the same machine. Device-independent means that an application program need not be rewritten, recompiled, or even relinked to work with new display hardware.

As graphic tools, X provides all the drawing facilities that a window system would provide, such as generating multifont text, and drawing two-dimensional graphics in rectangular windows. Windows can contain subwindows within them, to arbitrary depths. They can also overlap each other.

An application program or a client sends a request to the X server for any of the
services. X server is a program running on the machine providing network services of the window system. X server uses the interprocess communication protocol to send information back to the client in response to various requests, forwards any input (entered with a keyboard or a mouse) to the appropriate clients for processing, and handles output from the clients to the display. The services, inputs, and outputs are invoked by the clients by calling the client libraries. Xlib is the library for the C programming language.

The main reason why the partitioning algorithm proposed in this thesis is implemented using X Window System is that X has been pretty much standardized. It means that an application program written for one machine does not need to be rewritten for other machines (device-independent). Moreover, the X Window System has been endorsed as a graphic windowing environment by several large computer corporations that formed the X Consortium, such as Apollo Computer, Apple Computer, AT&T, Digital Equipment Corp., Fujitsu, Hewlett Packard, IBM, etc.

3.2 Overview

The partitioning algorithm explained in the previous chapter has been implemented in a software package called Reduced Connectivity Matrix Partitioning, abbreviated as \textit{rcmp}. It is implemented on an HP9000/350CH workstation using X Windows as the graphic interface. It is written in C language.

The \textit{rcmp} package can be used to draw graphs and to load drawn graphs, where the nodes of the graphs are drawn as circles and the branches are drawn as line segments. Inputs are read from a keyboard and a mouse, and outputs are displayed on a screen. The \textit{rcmp} package is started by typing \textit{rcmp} at the command prompt in
Figure 3.1: Empty window

X window. When \texttt{rcmp} is started, a window that looks like Fig. 3.1 will appear at the top left corner of the screen. This window has a drawing window called DrawWindow and a menu list. The following describes what each menu box means and terms used in \texttt{rcmp}:

- **BorderWindow**: the window which contains the DrawWindow and the menu list

- **DrawWindow**: The window where a graph is drawn

- **Save**: Save the current graph drawing into a file

- **Load**: Load a drawn graph into DrawWindow

- **Branch**: Draw a branch in the DrawWindow

- **Node**: Draw a node in the DrawWindow
• **Undo**: Undo a drawing or a deletion

• **Delete**: Delete a node or a branch

• **Wipe**: Delete everything from the DrawWindow

• **In**: Zooming-in command

• **Out**: Zooming-out command

• **Full**: Bring the whole graph to appear in the DrawWindow

• **Part**: Start partitioning the graph

• **Quit**: Exit from **rcmp**

• **Click**: Press and release a button of the mouse

Each of the command on the menu will be explained separately.

The **rcmp** package uses two files, the *filename.crd* file and the *filename.net* file. Both of these files are written when the **Save** command is executed. The file *filename.crd* is read by **rcmp** when executing **Load** command. The format for *filename.crd*, the coordinate file, is:

```
node-number   x-coord   y-coord
:  
branch-number x1-coord y1-coord x2-coord y2-coord
:  
```

and the format for *filename.net*, the netlist file, is:
3.3 Menu Commands

The commands in the BorderWindow are explained below:

- **Save:** The Save command saves the graph in the DrawWindow into two files. Clicking in this Save box will bring up a dialog window (Fig. 3.4) asking for

\[
\begin{align*}
N0 & \quad 210 \quad 190 \\
N1 & \quad 170 \quad 290 \\
N2 & \quad 250 \quad 290 \\
N3 & \quad 210 \quad 390 \\
N4 & \quad 410 \quad 190 \\
N5 & \quad 370 \quad 290 \\
N6 & \quad 450 \quad 290 \\
N7 & \quad 410 \quad 390 \\
B0 & \quad 210 \quad 190 \quad 170 \quad 290 \\
B1 & \quad 170 \quad 290 \quad 210 \quad 390 \\
B2 & \quad 210 \quad 390 \quad 250 \quad 290 \\
B3 & \quad 250 \quad 290 \quad 210 \quad 190 \\
B4 & \quad 210 \quad 190 \quad 410 \quad 190 \\
B5 & \quad 410 \quad 190 \quad 370 \quad 290 \\
B6 & \quad 370 \quad 290 \quad 410 \quad 390 \\
B7 & \quad 410 \quad 390 \quad 450 \quad 290 \\
B8 & \quad 450 \quad 290 \quad 410 \quad 190 \\
B9 & \quad 170 \quad 290 \quad 250 \quad 290 \\
B10 & \quad 370 \quad 290 \quad 450 \quad 290 \\
B11 & \quad 210 \quad 390 \quad 410 \quad 390
\end{align*}
\]

Figure 3.2: A filename.crd file

node-number   list of branch-numbers incident to node-number

Fig. 3.2 shows an example of the file filename.crd, and Fig. 3.3 shows an example of the file filename.net.
the name of the files where the graph is to be saved. Typing in the file name and clicking in the OK box will save the graph. Clicking in the Cancel box will cancel the saving process. One file will have a .crd extension and the other file will have a .net extension.

• **Load**: This command loads a file with either .net or .crd extension into the DrawWindow. Clicking in the Load box will bring up a dialog window asking for the name of the file from which the graph is to be loaded from. Clicking in the Cancel box will cancel the loading process. Typing in the file name and clicking in the OK box will bring up another dialog window asking if the file is a netlist file or a coordinate file. Typing in either netlist or coordinate and clicking in the OK box will load the file. Clicking in the Cancel box will cancel the loading process.

• **Node**: Clicking in the Node box changes the DrawWindow cursor into a cross-hair cursor. Clicking the mouse anywhere in the DrawWindow after this will draw a node in the DrawWindow and will automatically number it. A node

```
0 0 3 4 ;
1 0 1 9 ;
2 2 3 9 ;
3 1 2 11 ;
4 4 5 8 ;
5 5 6 10 ;
6 7 8 10 ;
7 6 7 11 ;
```

Figure 3.3: A filename.net file
can only be present at certain grid positions. Drawing more than one node at a grid position will cause an error and the new node will not be drawn. A node is automatically connected to a branch if it is drawn at the grid position where either end of the branch is.

- **Branch**: Clicking in the **Branch** box changes the DrawWindow cursor into a *top-left-arrow* cursor. A branch is drawn by pressing and holding down a button of the mouse to draw one end of the branch and sliding the mouse and releasing it to draw the other end of the branch. A branch is automatically numbered when it is drawn. An end of a branch can only be present at certain grid positions. Drawing the same branch more than once will cause an error and the new branch will not be drawn. A branch is automatically connected to a node if either end of that branch is drawn at the same grid position as the node is.
• **Delete:** This command deletes a node or a branch. Clicking in the **Delete** box will change the DrawWindow cursor into a *pirate* cursor. Moving the cursor to a node and clicking a button of the mouse will delete that node. Moving the cursor to anywhere of a branch and clicking a button of the mouse will delete that branch. If the mouse is clicked at an end of a branch where it meets an end of another branch, a pop-up window will appear asking for the branch number of the branch to be deleted. Typing in the branch number and clicking in the **OK** box will delete the branch. Clicking in the **Cancel** box will cancel the deletion.

• **Undo:** This command undoes the last 10 commands which can be: drawing a node, drawing a branch, or deletion. It can not undo an undo. Every click in the **Undo** box undoes the most recent command. Undoing the last 11th command will bring up a pop-up window displaying an error message.

• **Wipe:** Clicking in the **Wipe** box will erase the whole current graph in the DrawWindow. This **Wipe** command can not be undone.

• **In:** This is a zooming-in command which is used to enlarge a graph or part of a graph. Clicking in the **In** box will change the DrawWindow cursor into a *sizing* cursor. Moving this *sizing* cursor to the part of the graph that is to be enlarged, drawing a rectangle around it by pressing any button of the mouse at the top left corner of the rectangle, sliding it while holding down the button and releasing it at the opposite corner of the rectangle will bring the area in the rectangle to be redrawn and to occupy the whole DrawWindow.
• **Out**: This is a zooming-out command which is used to reduce the size of the current content of the DrawWindow into half of its current size. Clicking in the **Out** box will zoom-out the graph in the DrawWindow.

• **Full**: Clicking in the **Full** box will bring the graph that is currently in the DrawWindow to fill the DrawWindow.

• **Part**: This command is used to partition the graph currently in the DrawWindow. Clicking in the **Part** box will bring up a dialog window, asking for the partitioning parameters. \( \text{min} \) is the minimum number of nodes acceptable in one partition, \( \text{num} \) is the number of partitions requested, \( \text{tol} \) is the number of extra connections that is acceptable after the minimum number of connections found by the algorithm, \( \text{cascaded} \) is the parameter to specify if the connections cut by a partition will still be taken into account in calculating the number of connections in the next partition, 1 for \text{yes} \) and 0 for \text{no}, and \( \text{optimized} \) is to specify whether optimization algorithm is to be applied to the partitioning algorithm, again 1 for \text{yes} \) and 0 for \text{no}. Typing in all parameters and clicking in the **OK** box will start partitioning process. Clicking in the **Cancel** box will cancel partitioning process. When partitioning is done, the graph will be redrawn in colors, one for each partition.

### 3.4 Implementation

This section explains the graphic interface of \( rcmp \) which uses the X Window System. It describes how some of the X functions are used in \( rcmp \) and it also describes the algorithm for each command. The X system functions used in \( rcmp \)
can be found in the next section. For a complete list of X functions, refer to [12].

rcmp is implemented to provide the users with the ability to draw an arbitrary graph, partition that graph and save it. It allows the users to load a predrawn graph. And it outputs a circuit netlist so that it can be used as an input to other VLSI software. rcmp is started by first opening a BorderWindow, which contains all the menu boxes and a DrawWindow. Then for each menu box, a corresponding application program is written.

3.4.1 Opening the BorderWindow, DrawWindow, and menu boxes

Before a window can be opened, there should have been a connection to the X server. To establish a connection, the call to the X system function XOpenDisplay() is necessary. After a connection to the X server has been established, all windows are opened by calling the X function XCreateWindow(). X Window System requires a graphics context (GC) associated with each window for that window to be able to draw anything. A GC is a data structure X Window System uses to control the drawing of a window. For each window in rcmp, a call to XCreateGC() creates a GC associated with the window.

3.4.2 Node and branch

The size of the DrawWindow is determined in pixels. Let the pixel at the top left corner of the DrawWindow be of coordinate (0,0). Pixels to the right of (0,0) have an increasing x-coordinate, and pixels to the bottom of (0,0) have an increasing y-coordinate. These pixels are divided in a grid system such that only pixels with
coordinates

\[(x_{ad}, y_{ad}) = \left(\frac{x}{\text{nodeDiameter}} - \frac{\text{nodeDiameter}}{2}, \frac{y}{\text{nodeDiameter}} - \frac{\text{nodeDiameter}}{2}\right)\] (3.1)

can be drawn nodes and branches. This coordinate will be referred to as *adjusted coordinate* in the proceeding discussion. If the cursor is moved to a pixel \(i\) and clicked to draw a node or a branch, then a node or a branch will be drawn on a pixel with adjusted coordinate that is closest to pixel \(i\). A data structure is built for each node and branch. The data structure for nodes and branches are:

typedef struct Dnode_struct {
    int x,y; /* node coordinate when it is first drawn */
    int zoomX,zoomY; /* node coordinate in the current drawing */
    int num; /* node number */
    struct arc_struct *branchList; /* list of branches connected to this node */
    struct Dnode_struct *leftNode; /* left pointer */
    struct Dnode_struct *rightNode; /* right pointer */
} Node;

typedef struct Dbranch_struct {
    int x1,y1,x2,y2; /* branch coordinate when it is first drawn */
    int zoomX1,zoomY1,zoomX2,zoomY2; /* branch coordinate in the current drawing */
    int num; /* branch number */
}
struct Dbranch_struct *leftBranch;  /* left pointer */
struct Dbranch_struct *rightBranch;  /* right pointer */
}

typedef struct arc_struct {
    Branch *branch;  /* pointer to a branch */
    struct arc_struct *next;  /* next pointer */
} arcList;

All nodes and branches are hashed to a table with NUM_BUCKETS (=512) buckets called hashTable [15]. The hashing key is the adjusted coordinate and the hashing function is

\[ h(x,y) = (((x/nodeDiameter) \& 30) \ll 3)(((y/nodeDiameter) \& 30) \gg 1) \] (3.2)

3.4.3 Undo, delete, and wipe

There are often times where a node or a branch needs to be deleted and there are cases when a node or a branch needs to be undrawn or restored. The following command have been implemented to perform such tasks

3.4.3.1 Delete To delete a node, the cursor has to be moved inside of the node. To make using remp easier, however, the cursor can be clicked within 5 pixels away from a branch to delete that branch. When the command is the delete command, a click in the DrawWindow at \((x,y)\) will invoke the delete routine. The adjusted coordinate \((x_a,y_a)\) is hashed to get \(h_a\). Then it follows the following algorithm:
delete(x,y)
{
    calculate (x_a, y_a);
    if (hashTable(h_a) is empty) {
        if (there is any branch that passes a pixel (x_p, y_p)
            such that (x == x_p) and (y-y_p <= 5))
            delete that branch, attach it to the undoList for undoing purpose;
        /* do not deallocate it because it is still needed to undelete this branch */
    }
    else { /* hashTable[h_a] is not empty */
        r = first record in hashTable[h_a];
        while (r != NULL) {
            if ((r is a node) and (coordinate(r) == (x_a, y_a))) {
                delete this node, attach it to the undoList for undoing purpose;
                /* do not deallocate it because it is still needed to undelete this node */
            }
            r = next record;
        }
        if (there is no node record in hashTable[h_a] and there are more than one branch record in hashTable[h_a])
            ask for branch number using a pop-up window, delete
that branch, and attach it to the undoList for undoing purpose;
/* do not deallocate it because it is still needed to undelete this branch */
else /* there is only one branch in hashTable[h] */
delete that branch, attach it to the undoList for undoing purpose;
/* do not deallocate it because it is still needed to undelete this branch */

3.4.3.2 **Undo** Every click in the **Undo** box undoes the most recent event. An event is a drawing of a node or a branch or a deleting of a node or a branch. Undoing can go 10 levels deep. It means undo can be done 10 times for the last 10 events. To keep track of the last 10 events, a data structure is needed:

typedef struct {
  int type;       /* type of undo */
  union {
    Branch *branch;
    Node *node;
  } uval;
} Undo;
Undo *undoList;  /* circular list */
int curUndo;  /* circular pointer */

where type can only be EMPTY (-1), BRANCH (1), NODE (2), UNDEL_BRANCH (3), or UNDEL_NODE (4). The following is the algorithm for undo:

undo()
{
    if (undoList is empty)
        return;

    switch (undoList[curUndo].type) {
        case NODE:
            - delete and deallocate
              undoList[curUndo].uval.node
              /* because we can not undo an undo */
            - change undoList[curUndo].type from NODE to EMPTY
            - circularly advance curUndo
        case BRANCH:
            - delete and deallocate
              undoList[curUndo].uval.branch
              /* because we can not undo an undo */
            - change undoList[curUndo].type from BRANCH to EMPTY
            - circularly advance curUndo
case UNDEL_NODE:
    - append undoList[curUndo].uval.node back to nodeList
    - hash it back to hashTable
    - change undoList[curUndo].type from UNDEL_NODE to EMPTY
    - circularly advance curUndo

case UNDEL_BRANCH:
    - append undoList[curUndo].uval.branch back to branchList
    - hash it back to hashTable
    - change undoList[curUndo].type from UNDEL_BRANCH to EMPTY
    - circularly advance curUndo

3.4.3.3 Wipe A click in the Wipe box will erase all the data structure build for the current graph in the DrawWindow. It deallocates all nodes, branches, and hashTable entries.

3.4.4 Zooming

Zooming is essential when part of a graph is needed to be enlarged. There are 3 kinds of zooming in rcmp: IN, OUT, and FULL. IN is to enlarge a graph or part of a
graph, OUT is to lessen the size of the content of the current DrawWindow by half, and FULL is to redraw the DrawWindow such that the current graph is drawn to take as much space as possible of the DrawWindow. A variable zoomRatio is maintained to keep track of the size of the current drawing. Two other variables, offsetX and offsetY, are maintained to keep track of the point on the graph that is to be drawn at the upper left corner of the DrawWindow. The record fields x, y, and x1, y1, x2, y2 of the data structure Node and Branch as seen in Node and Branch part of this section maintain the coordinates of the nodes and branches when they are initially drawn. Loading of a graph will use these fields to draw its nodes and branches in the DrawWindow. Their fields zoomX, zoomY, and zoomX1, zoomY1, zoomX2, zoomY2 maintain the coordinates of their current drawing in the DrawWindow.

3.4.4.1 IN To zoom in, a rectangle s is drawn around the part of the graph that is needed to be enlarged. After s is drawn, the zoomRatio is adjusted by the following formula:

\[
TEMP = \min(\text{width of DrawWindow}/\text{width of } s, \frac{\text{height of DrawWindow}}{\text{height of } s})
\]

\[
\text{zoomRatio} = \text{zoomRatio} \times TEMP
\]

(3.3) \hspace{1cm} (3.4)

TEMP is to ensure that everything in s is visible in the new drawing. The width and height of s are adjusted such that s is congruent to the DrawWindow. From the adjusted width and height, offsetX and offsetY are updated. The fields, zoomX, zoomY, zoomX1, zoomY1, zoomX2, and zoomY2, are calculated from zoomRatio, offsetX, and offsetY by

\[
\text{zoomX} A = (\text{zoomX} A - \text{offset} X) \times \text{zoomRatio}
\]

(3.5)
\[ zoom_{YB} = (zoom_{YB} - offset_Y) \times zoomRatio \] \hspace{1cm} (3.6)

The whole graph is then redrawn using these calculated coordinates.

3.4.4.2 **OUT** To zoom out, no rectangle needs to be drawn. However, an imaginary rectangle \( s \) is used by \( rcmp \) of size \( 1/2 \) width of the DrawWindow \( \times \) \( 1/2 \) height of the DrawWindow. Using this imaginary \( s \), the same calculations as in **IN** proceeds.

3.4.4.3 **FULL** **FULL** is merely bringing the whole graph to occupy the whole DrawWindow. For this purpose, there are four additional variables needed to be maintained: \( \text{smallestX} \), \( \text{smallestY} \), \( \text{largestX} \), and \( \text{largestY} \). \( \text{smallestX} \) and \( \text{smallestY} \) are the values of smallest x-coordinate and y-coordinate of all nodes and branches of the current graph, respectively. And \( \text{largestX} \) and \( \text{largestY} \) are the values of the largest x-coordinate and y-coordinate. These four variables are updated every time a node or a branch is drawn. Thus it only takes constant time to update them. From these four variables, an imaginary rectangle \( s \) is drawn with the size

\[
\text{size of } s_{\text{imag}} = (\text{largestX} - \text{smallestX} + 2 \times \text{DEFAULT_OFFSET}) \\
\times (\text{largestY} - \text{smallestY} + 2 \times \text{DEFAULT_OFFSET})
\] \hspace{1cm} (3.7)

where \( \text{DEFAULT_OFFSET} \) is the number of pixels around the edge of the DrawWindow that is not to be drawn if **FULL** is selected. From here, the same calculation as in **IN** proceeds.
3.4.5 Save and load

3.4.5.1 Save When Save is selected, a dialog window will appear and ask for the name of the file to be saved. remp will create 2 files and adds .crd extension to one of them and .net extension to the other. The net list file filename.net is provided so that it can be used to interface with some other software in the future.

3.4.5.2 Load remp can load both netlist files and coordinate files. If the file name provided by the user does not have any extension, remp will automatically add .crd extension if it is a coordinate file and .net extension if it is a netlist file. The content of the DrawWindow is then replaced by the graph drawn from either filename.crd or filename.net.

If the input file is a coordinate file, remp reads all nodes and branches together with their coordinates. Then the data structure for this input graph is built. From the coordinates of nodes and branches, remp draws the graph in the DrawWindow.

If the input file is a netlist file, coordinates have to be assigned to the nodes and branches of the netlist. The algorithm for doing this is called graph drawing algorithm. The graph drawing algorithm used in remp can be found in [16]. This algorithm is applied after the input netlist has been partitioned using the algorithm proposed in this thesis. Let the subroutine for the graph drawing algorithm be called kamada_kawai(). The algorithm for loading a netlist file, loadNetlist(), is listed in pseudocode below:

```plaintext
loadNetlist()
{
    readNetlist(netlist);
}
```
partition(netlist, subnetlist);
for (all partitions in subnetlist)
    kamada_kawai();
}

3.4.6 Part

When this window is selected, rcmp calls the partition subroutine. This subroutine returns a list of cut branches and partitions of the graph. rcmp then draws a dot on each cut branch and colors each partition differently using the X system build-in function XFillArc().

3.5 X Window Functions

The following X Window System functions are used in rcmp. They are some of the functions in the Xlib client libraries. All of these functions can be found in [12]. Here is the alphabetical listing of the functions:

- **BlackPixel()**: Get "black" pixel value.

- **XAllocColor()**: Allocate read-only color cell.

- **XCheckWindowEvent()**: Remove next event that matches both a window and an event mask.

- **XClearWindow()**: Clear entire area within a window.

- **XCloseDisplay()**: Close display.
• `XCreateFontCursor()`: Create cursor from a standard font.

• `XCreateGC()`: Create new graphics context that is usable on a screen with a depth of drawable.

• `XCreateWindow()`: Create unmapped window and set its attributes.

• `XDefineCursor()`: Define which cursor will be used in a window.

• `XDestroySubwindows()`: Destroy all subwindows of a window.

• `XDrawArc()`: Draw arc in a given drawable.

• `XDrawLine()`: Draw line between two points in a given drawable.

• `XDrawString()`: Draw 8-bit characters in a given drawable.

• `XFillArc()`: Fill arc in a given drawable.

• `XFillArcs()`: Fill multiple arcs in a given drawable.

• `XFillRectangle()`: Fill rectangle in a given drawable.

• `XFlush()`: Flush output buffer.

• `XFreeCursor()`: Free cursor.

• `XFreeFont()`: Unload fond and free the storage used by the font structure.

• `XFreeGC()`: Free given graphics context.

• `XGetErrorText()`: Get textual descriptions of the specified error code.

• `XLoadQueryFont()`: Perform XLoadFont and XQueryFont in a single operation.
• *XLookupColor()*: Look up the name of a color.

• *XLookupString()*: May key event to an ASCII string.

• *XMapWindow()*: Map window.

• *XNextEvent()*: Get next event and remove it from queue.

• *XOpenDisplay()*: Open connection to the X server controlling a display.

• *XParseGeometry()*: Parse standard window geometry strings.

• *XQueryPointer()*: Get root window the pointer is currently on and the pointer coordinates relative to the root’s origin.

• *XResizeWindow()*: Change window’s size without changing upper left coordinate.

• *XSelectInput()*: Select events to be reported to the client application.

• *XSetBackgroundColor()*: Set background of a given graphics context.

• *XSetClassHint()*: Set class of a window.

• *XSetClipMask()*: Set clip mask of a graphics context.

• *XSetClipRectangles()*: Set clip mask of a graphics context to the specified list of rectangles.

• *XSetErrorHandler()*: Set error handler.

• *XSetFont()*: Set current font of a graphics context.
- `XSetForeground()`: Set foreground of a given graphics context.

- `XSetIconName()`: Set name to be displayed in a window’s icon.

- `XStoreName()`: Name window.

- `XSetNormalHints()`: Set size hints.

- `XSetWMHints()`: Set window manager hints.

- `XTextWidth()`: Determine width of an 8-bit character string.

- `WhitePixel()`: Get "white" pixel value.
4. EXPERIMENTAL RESULTS

The proposed partitioning algorithm has been tested for a number of graphs. Some of these graphs can be found in [17]. This chapter presents several examples, illustrates the effects of changing some of their partitioning parameters, and plots the running time of the algorithm. For all the tables shown in this chapter:

- \( g \) is the number of partitions resulting from the algorithm,
- \( n_{min} \) is the minimum number of nodes that have to be present in every partition,
- \( tol \) is the internal partitioning parameter,
- \( cascaded \) is equal to one if the number of cut branches of a partition is also accounted in the number of cut branches of its neighboring partition; otherwise it is equal to zero,
- \( optimized \) is equal to one if optimization phase is applied; otherwise, it is equal to zero, and
- \( t \) is the total number of branches cut by all partitions.

**Example 4.1** This example illustrates the advantage of including the optimization phase in the algorithm. The graph of this example which is partitioned without
performing the optimization is shown in Fig. 4.1. The solution after performing the optimization is shown in Fig. 4.2. Table 4.1 tabulates the results of this example where I is the result without the optimization phase; and II is the result with the optimization phase. In this example, the optimization reduces $t$ by 1.

**Example 4.2** This example illustrates the effects of changing the $tol$. Fig. 4.3 shows a partitioned graph with $tol = 1$. For $tol = 2$, the partitioned graph is shown in Fig. 4.4. Fig. 4.5 shows the result for $tol = 3$. For all of these $tol$ values, the results are tabulated in Table 4.2. In the table, I is the result for $tol = 1$. II is the result for
Table 4.1: Results of Example 4.1

<table>
<thead>
<tr>
<th>$n_{min}$</th>
<th>$g_{max}$</th>
<th>$tol$</th>
<th>cascaded</th>
<th>optimized</th>
<th>$g$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>II</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 4.3: Graph of Example 4.2 for $tol = 1$

tol = 2, and III is the result for $tol = 3$. Notice that when $tol = 1$, the graph can only be partitioned into two subgraphs. When $tol = 2$, the graph is partitioned into three optimal subgraphs. When $tol = 3$, the graph is partitioned into three subgraphs, and the solution is close to optimal.

Example 4.3 This example illustrates the effects of assuming whether the circuit is cascaded or not. The partitioned result for the graph which is not assumed to be cascaded is shown in Fig. 4.6. The partitioned result for the same graph which is assumed to be cascaded is shown in Fig 4.7. Table 4.3 tabulates the results, where I is the result where no cascading is assumed, and II is the result where cascading is
Figure 4.4: Graph of Example 4.2 for $tol = 2$

Figure 4.5: Graph of Example 4.2 for $tol = 3$
Table 4.2: Results of Example 4.2

<table>
<thead>
<tr>
<th></th>
<th>$n_{\min}$</th>
<th>$g_{\max}$</th>
<th>tol</th>
<th>cascaded</th>
<th>optimized</th>
<th>$g$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>12</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>II</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>III</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 4.6: Graph of Example 4.3 without cascading

Assumed. As can be seen that for this graph, II gives a better result.

The rest of the graphs which are tested by the partitioning algorithm are shown in Fig. 4.8 and Fig. 4.9. The results are tabulated in Table 4.4. As can be seen, in most cases the optimal solutions are reached; while in some cases, the results close to the optimal solutions are reached.

The proposed algorithm showed comparable, and superior solutions in some cases, to those of Park and Park’s algorithm [17]. Table 4.5 shows the comparison between the solutions of the proposed algorithm (A) and those of the Park and Park’s
Figure 4.7: Graph of Example 4.3 with cascading

Table 4.3: Results of Example 4.3

<table>
<thead>
<tr>
<th>Fig.</th>
<th>$n_{min}$</th>
<th>$g_{max}$</th>
<th>tol</th>
<th>cascaded</th>
<th>optimized</th>
<th>g</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>II</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.4: Results of Fig. 4.8 and Fig. 4.9

<table>
<thead>
<tr>
<th>Fig.</th>
<th>$n_{min}$</th>
<th>$g_{max}$</th>
<th>tol</th>
<th>cascaded</th>
<th>optimized</th>
<th>g</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8a</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4.8b</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4.8c</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4.8d</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4.8e</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>4.9a</td>
<td>13</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4.9b</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4.9c</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4.9d</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4.9e</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
Figure 4.8: Examples
Figure 4.9: Examples
Table 4.5: Comparison

<table>
<thead>
<tr>
<th>Fig.</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4.9b</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4.9a</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4.7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>4.4</td>
<td>3</td>
<td>14</td>
</tr>
</tbody>
</table>

algorithm (B).

The time complexity of this algorithm is $O(n^2)$. The run times of several graphs with different number of nodes are plotted in Fig. 4.10. The $n^2$ curve is also plotted. $n$-axis is for the number of nodes in the graphs, and $t$-axis is for the run time in milliseconds.
Figure 4.10: Run time of the proposed algorithm
5. CONCLUSIONS

An algorithm for partitioning of VLSI circuits using a reduced connectivity matrix has been developed. The proposed algorithm is a new algorithm for partitioning of VLSI circuits which uses a reduced connectivity matrix and the concept of a cut index. It is a heuristic algorithm that attempts to find an optimal solution in the worst case time complexity of $O(n^2)$. The space complexity has also been shown to be $O(n^2)$. Several examples have been tested. Most of the results are optimal, and some of them are near optimal. Out of the graphs tested by the algorithm, about 90% of them reached optimal solution.

The partitioning algorithm proposed in this thesis consists of two phases, which are:

1. Partitioning phase: This phase of the algorithm first reads a graph representation of a VLSI circuit, then it finds an independent set in the graph. From the independent set, the reduced connectivity matrix $M$ is built. After that, the rows of $M$ are permuted based on the concept of cut index. The idea is to group tightly coupled subcircuits together. The resulting matrix is called the permuted matrix $M_p$. Based on the concept of cut index, $M_p$ is partitioned. The goal is to find rows of $M_p$ with local minimum cut indices while considering the size of the partitions. The result of this phase is a set of partitions.
2. Optimization phase: This phase of the algorithm attempts to optimize the branches cut by the partitioning phase. Each of the cut branches has an external cost and an internal cost. The optimization phase searches for the branch with the maximum gain, which is the external cost minus the internal cost, and performs the optimization on it while considering the size of the partitions.

The proposed algorithm has been implemented in a software package (*rcmp*) using the C programming language. The program, which is referred to as Reduced Connectivity Matrix Partitioning, abbreviated as *rcmp*, uses X Window System as its graphic interface. It allows the users to draw arbitrary graphs, input graphs, output netlists, and input netlists. The *rcmp* package has been implemented using a universal netlist format that allows it to be interfaced with most existing VLSI CAD tools.

5.1 Recommendations

A common approach to solving partitioning problems is to devise an algorithm that will perform the partitioning on general graphs for a multitude of general applications. A better approach to the problem is to develop a customize algorithm to optimally or near optimally solve a specific class of problems. The algorithm proposed in this thesis can be modified to solve these different classes of problems. The main interest here is the VLSI applications. Even within this application, several classes of problems can be identified, such as:

- analog circuits,
- digital circuits,
• repetitive circuits, and

• cascaded circuits.

Two classes have been considered: cascaded circuits and general circuits. More research could be done on exploring the properties of the other classes mentioned above, and tailoring the algorithm to solve them.
6. BIBLIOGRAPHY


6.1 Additional References


