A Framework for the Analysis and Design of Algorithms for a Class of VLSI-CAD Optimization Problems*

C.-J. Shi
Department of Electrical and Computer Engineering
University of Iowa
Iowa City, Iowa 52242, U.S.A.
e-mail: cjshi@eng.uiowa.edu

J. A. Brzozowski
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
e-mail: brzozo@maveric0.uwaterloo.ca

Abstract—A simple mathematical framework, called cluster-cover, is established for several VLSI optimization problems including logic minimization, constrained encoding, multi-layer topological planar routing, application timing assignment for delay-fault testing, and minimization of monitoring logic for BIST enhancement. Two paradigms, prime covering and greedy peeling, are presented for developing both exact and heuristic algorithms. The paradigms capture generally applicable ingredients from previously developed algorithms for individual applications. This makes it possible to re-use established techniques in new problems, and provide new insights into existing problems. The paradigms are simple enough to be amenable to theoretical analysis. Bounds on the performance of greedy peeling are derived; these bounds are applicable to many published heuristics which previously could be evaluated only by benchmarks.

I. INTRODUCTION

Optimization problems arise in almost every phase of VLSI circuit and system design. To handle the ever increasing design complexity and reduce the design cycle time, efficient and reliable algorithms are highly desirable. But the design of such algorithms has been a great challenge. Most VLSI optimization problems are not only large (involve millions of variables and constraints), but also inherently computationally difficult (i.e., NP-hard). Nevertheless, tremendous progress has been made in algorithm development for some of these problems. Perhaps the best example is two-level logic minimization [1]. Exact algorithms have been developed; these algorithms are capable of finding optimum solutions reasonably fast for large problem instances [4, 5, 14, 17]. A natural question, and a primary motivation of this research, is "Is it possible to apply these techniques to other VLSI problems with similar combinatorial structure?"

Besides exact solutions many heuristic algorithms, capable of finding good approximate solutions efficiently, have been developed. Except for some special cases, however, no analysis of the performance of these heuristics has been carried out. In practice, the evaluation of both the efficiency and robustness of heuristic algorithms relies heavily on the use of benchmarks. Since benchmarks represent only a limited class of "real" instances, heuristics well tuned for benchmarks may turn out to be not as efficient and reliable for other practical instances. Hence, the analysis of the performance of various heuristics is practically important. This is our second motivation.

An abstract framework, called cluster-cover, is established in this paper for a class of optimization problems arising in a variety of VLSI applications. These include minimization of two-level combinational logic [1, 18], constrained encoding for the synthesis of sequential logic [20, 21, 23], multi-layer topological planar routing [3], application timing for delay-fault testing [11], and minimization of monitoring logic for built-in self-test (BIST) enhancement [10]. Using our framework, we describe prime covering and greedy peeling as two basic paradigms for developing exact and heuristic algorithms, respectively.

After introducing the cluster-cover framework in Section II, we describe five optimization problems and formulate them in our framework in Section III. In Section IV, we present prime covering and greedy peeling. In Sections V, we derive greedy peeling performance bounds. In Section VI, we use these bounds to justify some experimental results from the literature. Conclusions are made in Section VII.

II. THE CLUSTER-COVER FRAMEWORK

A set system is a pair \((E, \mathcal{P})\), where \(E\) is a finite set, and \(\mathcal{P}\) is defined by an application-specific predicate \(\Pi(P)\), where \(P\) is a variable ranging over the set of all subsets of \(E\). Thus \(\mathcal{P} = \{P \subseteq E \mid \Pi(P)\}\), i.e., a subset \(P \subseteq E\) is in \(\mathcal{P}\) if \(\Pi(P)\) is true.

**Example 1**: The set system defined by \(E = \{1, 2, 3\}\) and \(\Pi(P) : (\sum_{e \in P} e) < 4\) has \(\mathcal{P} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}\}\).

**Example 2**: The set system defined by \(E = \)
\{1, 2, 3\} and \Pi(P) = (\sum_{e \in \mathcal{P}} e) > 2 has \mathcal{P} = \{\{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.

We will be interested in two specific set systems: A subset-closed system\(^1\) is a one satisfying (1) \emptyset \in \mathcal{P}, and (2) \(Y \in \mathcal{P}\) and \(X \subseteq Y\) implies \(X \in \mathcal{P}\). A superset-closed system is one satisfying (1) \(E \in \mathcal{P}\), and (2) \(Y \in \mathcal{P}\) and \(X \supseteq Y\) implies \(X \in \mathcal{P}\). Properties 1 and 2 are called nonemptiness and hereditary properties, respectively. The set systems in Examples 1 and 2 are subset-closed and superset-closed, respectively.

We now present our cluster-cover framework. We have a set \(E\), a predicate \(\Pi\) on the set of subsets of \(E\) that defines a subset-closed system \((E, \mathcal{P})\), and a predicate \(\Gamma\) on the set of subsets of \(\mathcal{P}\) that defines a superset-closed system \((\mathcal{P}, C)\). The set \(E\) is the ground set, \(\Pi\) is a compatibility predicate, the subsets in \(\mathcal{P}\) are clusters, and the elements in a cluster are said to be compatible. The compatibility relation needs not be transitive. The subsets in \(C\) are covers, and \(\Gamma\) is a coverability predicate. We are interested in a coverability predicate defined as follows:

Given \(\alpha, 0 < \alpha \leq 1\), an \(\alpha\)-cover is a set of clusters that contains at least the fraction \(\alpha\) of the ground elements.

In Fig. 1 the ground set \(E\) is \{a, b, c\}; suppose some application-specific predicate \(\Pi\) gives clusters \{a, b\}, \{b, c\}, and all their subsets, including \emptyset. This compatibility relation is not transitive: \{a, b\} and \{b, c\} are clusters, but \{a, c\} is not. If \(\alpha = \frac{1}{2}\), the set of \(\alpha\)-covers is \(C = \{C_1, \ldots, C_5, \ldots\}\), where \(C_1 = \{a, b\}\), \(C_2 = \{a, c\}\), \(C_3 = \{b, c\}\), \(C_4 = \{a, b, c\}\), \(C_5 = \{b, c\}\), and the remaining covers are all the supersets of the first five. The empty cluster is ignored.

![Fig. 1: A cluster-cover example.](image)

The following questions arise frequently. The maximum \(k\)-cluster problem is: Given an integer \(k\), find at most \(k\) clusters that together contain as many ground elements as possible. When \(k = 1\), this is the maximum cluster problem. The minimum \(\alpha\)-cover problem is: Given a fraction \(\alpha\), find an \(\alpha\)-cover that contains as few clusters as possible. When \(\alpha = 1\), this is the minimum cover problem, or the exact cover problem.

The cluster-cover framework is related to two mathematical concepts. The first is a set cover. For a ground set \(E\), and a set \(\mathcal{P}\) of subsets of \(E\), a set cover is a collection of subsets from \(\mathcal{P}\) that together cover all the elements in \(E\). Cluster cover differs from set cover in two aspects: (1) our subsets are implicitly given by an application-specific predicate, and (2) our set of subsets is subset-closed. The second concept is a matroid [6, 13], which is a subset-closed system \((E, \mathcal{P})\) that obeys the combinatorial aspect of the Steinitz exchange principle:

- If \(X, Y \in \mathcal{P}\) and \(|X| < |Y|\), then there exists a \(y \in Y - X\) such that \(X \cup \{y\} \in \mathcal{P}\).

We do not require this condition to hold in our framework.

The cluster-cover framework models the structure of many applications. The maximum \(k\)-cluster and the minimum \(\alpha\)-cover problems provide a framework for many optimization problems, such as those discussed next.

### III. Five Cluster-Cover Problems

Five applications are now cast into our framework. Logic minimization and constrained encoding are well understood; multi-layer topological planar routing and application timing for delay-fault testing are newer and less explored, and monitoring-logic design for BIST enhancement is very recent.

**Problem 1: Logic Minimization**

Combinational logic minimization [1, 18] is a problem of minimizing the cost of the circuit implementing a Boolean function. A Boolean function \(f\) of \(n\) variables is usually \((incompletely)\) specified by its on-set, the set of input values \(x \in \{0, 1\}^n\) such that \(f(x) = 1\), and its off-set, the set of input values such that \(f(x) = 0\). Each element in \{0, 1\}^n is a minterm. Let the on-set be the ground set of minterms. A set of minterms contained in a single cube in \{0, 1\}^n that does not have any minterms from the off-set is a cluster; such a cube is an implicant of \(f\). The exact cover problem is of great practical interest [1, 18].

To illustrate this, consider a Boolean function with on-set \{x_1x_2, x_3x_5\} and off-set \{x_1x_2, x_3x_5\}. Then \{x_1x_2, x_3x_5\} is a cluster, because both minterms are included in implicant \{x_1\}. This cluster is also the minimum-cost cover.

**Problem 2: Constrained Encoding**

This fundamental problem [21, 23] arises in race-free state assignment for asynchronous sequential machines, in delay-free realizations of asynchronous machines without essential hazards, in optimum state assignment for synchronous machines, and in PLA decomposition. Here, the so-called dichotomy, i.e., a pair of disjoint subsets of a given set of "states," is a ground element. For example, \(a = \{(1, 2), \{3\}\}, b = \{(2), \{3\}\}, \) and \(c = \{(1), \{2, 3\}\}\) are dichotomies on the set \(\{1, \ldots, 4\}\). Two dichotomies are compatible if no two states appearing in a single subset of one dichotomy appear in different subsets of the other. For example, \(a\) and \(b\) are compatible pairs, but
(a, c) is not. A set of mutually compatible dichotomies is a cluster, which can also be represented by a dichotomy. For example, a and b can be represented by \{1, 2\}, \{3, 4\}. The structure of this example is illustrated in Fig. 1. The maximum cluster problem is to find a dichotomy that includes as many ground dichotomies as possible. The minimum cover problem is to find a minimum number of dichotomies that cover all the ground dichotomies. Both problems are of interest to sequential logic synthesis [21].

**Problem 3: Topological Planar Routing**

Fig. 2 has two rows of terminals marked by numbers. Terminals with the same number form a net, and each net is a ground element. Two nets are compatible if they can be routed in a plane without crossing. For example, nets 1, 2 and 3 (solid) in Fig. 2 form a cluster. Nets 4 and 5 (dashed) form another cluster. The maximum cluster problem is to find a maximal set of nets that can be routed in one plane. The minimum cover problem is to find the minimum number of planes such that all the nets can be routed without crossing. Both problems are of practical interest [3].

![Fig. 2: Example of multi-layer topological routing.](image)

**Problem 4: Application Timing Assignment**

The combinational circuit of Fig. 3 has two primary inputs (I1 and I2), two primary outputs (O1 and O2), and five gates (G1 to G5) [11]. Associated with each gate and wire is a delay: e.g., G1 has delay 0, and the wire from I1 to G1 has delay 10. Delay-fault testing involves a sequence of test patterns at the primary inputs. Let \(T_i\) be the time of applying a signal to input \(i\). For example, we may have the application timing assignment \(T_{I1} = 0\) and \(T_{I2} = 5\). For a timing assignment we denote by \(T_j\), the latest time for signals to arrive at output \(j\) from all the inputs. \(T_{O1} = \max(0 + 10 + 0 + 5 + 5 + 5 + 5 + 0 + 10 + 5) = 25\), and \(T_{O2} = \max(0 + 10 + 0 + 10 + 5, 5 + 5 + 5 + 0 + 5 + 3 + 2 + 5) = 25\).

For a path from input \(i\) to output \(j\) the time allowed for signal propagation is \(T_j - T_i\), which may differ from the total delay of that path; this difference is the slack of the path. (Path I1→G1→G4→O1 has \(T_{I1} = 0\) and \(T_{O1} = 25\). Since its delay is only 20, it has slack 5.) We define the delay slack for each delay as the minimum of the path slacks of all input-to-output paths that include this delay. (The wire delay between G1 and G4 has slack 5.)

For timing assignment \(T_{I1} = 0\) and \(T_{I2} = 5\), only the wire delay between G1 and G4 has nonzero slack (5), and the sum of all the delay slacks is 5. For timing assignment \(T_{I1} = T_{I2} = 0\), we have \(T_{O1} = 20\), and \(T_{O2} = 25\). Now path I2→G2→G3→G5→O2 has slack 5. Also the wire delays between G2 and G3, and G3 and G5, have nonzero slack (5). The sum of the delay slacks is 10.

A key observation in [11] is that delay slacks affect the size of the delay fault detectable by any test set. To improve the quality of delay-fault testing, we need an application timing assignment that keeps delay slacks as small as possible. However, the slack of a delay \(d\) cannot be made arbitrarily small; the length of a longest input-to-output path minus the length of a longest input-to-output path including delay \(d\) forms a slack lower bound for \(d\).

The multiple-test application timing assignment problem is as follows: Given a fraction, \(\alpha\), \(0 < \alpha < 1\), find the smallest number of application timing assignments such that, for \(\alpha\) of the delays, there exists at least one timing assignment for each delay that enables it to achieve its slack lower bound. The single-test application timing assignment problem is to find an assignment such that as many delays as possible achieve their slack lower bounds.

In our framework, we take the circuit delays as ground elements. Then a subset of delays forms a cluster if all the delays in the subset can achieve their slack lower bounds under a single timing assignment; this is called a consistent-tight set in [11]. The multiple-test application timing assignment problem is the maximum \(k\)-cluster problem, whereas the single-test application timing assignment problem is the minimum \(\alpha\)-cover problem.

![Fig. 3: Example of application timing assignment.](image)

**Problem 5: Monitoring Logic Design for BIST**

Built-in self-test (BIST) is widely used in VLSI. A typical BIST structure is shown in Fig. 4, and consists of a test generator, a combinational circuit under test (CUT), and a multiple-input signature analyzer. The test generator produces a test sequence for the CUT. The signature analyzer compresses the output sequence of the CUT into a signature, and compares it to the correct signature. If the two signatures differ, the CUT is faulty. However, if the CUT is faulty, the signature may be correct because of data compression; this is called aliasing.

To overcome aliasing, [10] proposes a BIST structure of Fig. 4. The output \(y\) is the first bit of the signature; we call it the key. If the CUT is correct, the value of the key is \(f_0(x)\). The test sequence includes all input combinations; \(f_0(x)\) is not. A set of mutually compatible dichotomies is a cluster, which can also be represented by a dichotomy. For example, a and b can be represented by \{1, 2\}, \{3, 4\}. The structure of this example is illustrated in Fig. 1. The maximum cluster problem is to find a dichotomy that includes as many ground dichotomies as possible. The minimum cover problem is to find a minimum number of dichotomies that cover all the ground dichotomies. Both problems are of interest to sequential logic synthesis [21].

\(T_{I1} = T_{I2} = 0\), we have \(T_{O1} = 20\), and \(T_{O2} = 25\). Now path I2→G2→G3→G5→O2 has slack 5. Also the wire delays between G2 and G3, and G3 and G5, have nonzero slack (5). The sum of the delay slacks is 10.

A key observation in [11] is that delay slacks affect the size of the delay fault detectable by any test set. To improve the quality of delay-fault testing, we need an application timing assignment that keeps delay slacks as small as possible. However, the slack of a delay \(d\) cannot be made arbitrarily small; the length of a longest input-to-output path minus the length of a longest input-to-output path including delay \(d\) forms a slack lower bound for \(d\).

The multiple-test application timing assignment problem is as follows: Given a fraction, \(\alpha\), \(0 < \alpha < 1\), find the smallest number of application timing assignments such that, for \(\alpha\) of the delays, there exists at least one timing assignment for each delay that enables it to achieve its slack lower bound. The single-test application timing assignment problem is to find an assignment such that as many delays as possible achieve their slack lower bounds.

In our framework, we take the circuit delays as ground elements. Then a subset of delays forms a cluster if all the delays in the subset can achieve their slack lower bounds under a single timing assignment; this is called a consistent-tight set in [11]. The multiple-test application timing assignment problem is the maximum \(k\)-cluster problem, whereas the single-test application timing assignment problem is the minimum \(\alpha\)-cover problem.

The multiple-test application timing assignment problem is as follows: Given a fraction, \(\alpha\), \(0 < \alpha < 1\), find the smallest number of application timing assignments such that, for \(\alpha\) of the delays, there exists at least one timing assignment for each delay that enables it to achieve its slack lower bound. The single-test application timing assignment problem is to find an assignment such that as many delays as possible achieve their slack lower bounds.

In our framework, we take the circuit delays as ground elements. Then a subset of delays forms a cluster if all the delays in the subset can achieve their slack lower bounds under a single timing assignment; this is called a consistent-tight set in [11]. The multiple-test application timing assignment problem is the maximum \(k\)-cluster problem, whereas the single-test application timing assignment problem is the minimum \(\alpha\)-cover problem.

\(\alpha\)The formulation in [11] is slightly different.
If $y$ predicts the correct value of the key with the aid of two Boolean function $f$, hence $f_y$ hence:

1. $y_0 = 0$ implies $f_0 = 1$. 
2. $y_1 = 0$ implies $f_0 = 0$. 

3'. The on-sets of $y_0$ and $y_1$ are such that, for each $f_i, i = 1, \ldots, l$, there exists at least one input pattern $x$ for which the output $y_0(x)f_i(x) + y_1(x)f_i(x)$ of the monitoring circuit is 1.

If $f_i(x) \neq f_0(x) = 0$, we say that $x$ 0-distinguishes $f_i$.
If $f_i(x) \neq f_0(x) = 1$, we say that $x$ 1-distinguishes $f_i$.
Condition 3' can be replaced by 3, 4, and 5.

3. Each $f_i, i = 1, \ldots, l$, is either 0-distinguishable or 1-distinguishable at least one pattern $x$.
4. $y_0 = 1$ for all 0-distinguishing patterns.
5. $y_1 = 1$ for all 1-distinguishing patterns.

The new logic minimization problem here is: Given $f_i, i = 0, \ldots, l$, design a minimum-cost circuit with outputs $y_0$ and $y_1$ that satisfies 1–5.

Table 1 [10] shows the correct function $f_0$, and three faults $f_1$, $f_2$, and $f_3$ undetectable because of aliasing. Fault $f_1$ is 0-distinguishable by 001 and 1-distinguishable by both 110 and 011; faults $f_2$ and $f_3$ are 0-distinguishable by 001 and 000, respectively.

An input pattern $x$ can 0-distinguish a set of faults, or 1-distinguish them, but not both. Thus $y_0(x) = 1$ implies $y_1(x) = 0$, and $y_1(x) = 1$ implies $y_0(x) = 0$. According to the terminology in logic synthesis, the on-set of $y_0$ is the set of 0-distinguishing patterns, and the on-set of $y_1$ is the set of 1-distinguishing patterns. Hence, the set of implicants of $y_0$ is disjoint from that of $y_1$. This problem is illustrated in Fig. 5: Each minterm distinguishes a set of faults; each implicant includes a set of minterms; and a cover is a set of implicants.

Because the relation between faults and minterms is easily obtained from the given table in linear time, the problem can be simplified so that it fits our framework. The set of faults constitutes the ground set. A cluster is a set of faults that can be distinguished by minterms included in one implicant. A cover is a set of clusters that covers all the faults. Now the problem considered in [10] is to find a cover that contains as few clusters as possible.

<table>
<thead>
<tr>
<th>Test pattern $x$</th>
<th>$f_0$</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Fig. 4: An augmented BIST structure.](image)

![Fig. 5: Three-level hierarchy for monitoring-logic.](image)

![Fig. 6: Cluster-cover for the monitoring-logic example.](image)
Prime Covering versus Greedy Peeling

IV. Prime Covering versus Greedy Peeling

We now describe two paradigms for the optimization problems with the cluster-cover structure. We categorize the heuristics developed for each problem of the previous section.

A. Prime Covering

Prime covering is an exact approach to solving the minimum \( \alpha \)-cover problem. It consists of two stages, corresponding to the clusters and covers of Fig. 1. First, the set of clusters is generated. Because it is subset-closed, only maximal clusters—for which the addition of one more ground element will violate the predicate—need to be found. Maximal clusters are called primes.

In the second stage the standard set-cover problem is solved by the branch-and-bound method, in which the solution space is successively partitioned. A cluster is selected—this is called branching—and the problem is examined, first assuming that the cluster is in the minimum cover, and then that it is not. Bounding refers to generating lower bounds that can be used to prune the search space.

To obtain a good lower bound for the objective function at each branch, either a compatibility graph or a conflict graph can be used. A compatibility (conflict) graph has ground elements as vertices. An edge joins two vertices if the corresponding ground elements are compatible (incompatible). The number of vertices in the maximum independent set in the compatibility graph (which equals the number of vertices in the maximum clique in the conflict graph) is a lower bound on the size of the minimum cover. In practice, finding the maximum clique is easier than finding the maximum independent set [18].

The choice of the branching cluster is based on the intuition that ground elements present in only a few clusters are "hard" to cover. To measure this "hardness," each element is assigned a weight—the reciprocal of the number of clusters in which it appears. The weight of a cluster is the total weight of all its elements. The cluster that appears in the independent set and is of maximum weight is chosen as the branching cluster.

Prime covering can be converted into a greedy heuristic by not using backtracking. The solution obtained is then an upper bound for the minimum cover.

The set-cover problem is NP-hard [9]. In the worst case, the number of branches needed for finding an optimum solution is exponential in the number of primes. The version of the algorithm without backtracking, or even greedy peeling, can be used for set covering. However, the number of primes can be exponential in the number of ground elements. Furthermore, the construction of primes is usually time- and space-consuming for problems defined by complicated compatibility predicates. Hence, prime covering is normally used as a paradigm for finding optimum solutions to relatively small problems or those for which the clusters can be easily constructed.

B. Greedy Peeling

Greedy peeling solves both the minimum \( \alpha \)-cover and the maximum \( k \)-cluster problems in a unified manner. It constructs directly the clusters needed in the final solution. Its description in terms of our framework is given in Fig. 7. We have a ground set \( E \) and an integer \( k \). The initial solution is an empty cover \( C \) (line 2). The core of greedy peeling is a subroutine that solves the maximum \( l \)-cluster problem \( (l \leq k; \text{typically } l = 1) \). The algorithm iterates \( \lceil \frac{k}{2} \rceil \) times (lines 3 to 6). At each iteration, the subroutine is first invoked to find a set \( P_i \) of \( l \) clusters and the subset \( E_i \) of elements covered by \( P_i \) (step: solve); then the elements covered by \( P_i \) are removed from \( E \) (step: peel-off); and, finally, the newly found set \( P_i \) of clusters is added to the solution (step: augment). The algorithm returns the computed cover \( C \) and the set of elements not covered by \( C \).

\[
\text{GREEDY.PEELING} (E, k)
\]

1. \( E' \leftarrow E \)
2. \( C \leftarrow \{ \} \)
3. for \( i = 1 \) to \( \lceil \frac{k}{2} \rceil \) do
   4. \( (P_i, E_i) \leftarrow \max \text{-} l \text{-cluster on } E' \) /*solve*/
   5. \( E' \leftarrow E' - E_i \) /*peel-off*/
   6. \( C \leftarrow C \cup P_i \) /*augment*/
7. return \((C, E')\)

Fig. 7: Paradigm of greedy peeling.

**Theorem 1** Greedy peeling yields an optimum solution for the maximum \( k \)-cluster problem, if the compatibility predicate \( \Pi \) satisfies one of the following properties:

1. Transitivity: For any \( x, y, z \in E \), \( \Pi(\{x, y\}) \) and \( \Pi(\{y, z\}) \) imply \( \Pi(\{x, z\}) \).
2. Steinitz exchange property: If \( X, Y \in \mathcal{P} \) and \( |X| < |Y| \), then there is a \( y \in Y - X \) such that \( X \cup \{y\} \in \mathcal{P} \).

In general, greedy peeling is a heuristic for finding approximate solutions. Often, step solve is approximated by a heuristic. We now describe two techniques for improving the solution quality of greedy peeling. First, one can build local search on top of greedy peeling; this gives
rise to iterative greedy peeling, which works as follows: Suppose we find a solution by greedy peeling after \(k\) iterations. The \(k\)th iteration was introduced, since there is a nonempty set \(E'\) of ground elements not covered by the first \(k-1\) iterations. The set \(E'\) thus appears to be "hard" to cover. Therefore, we start a new run of greedy peeling by "insisting" that the first iteration peel off \(E'\). This can be done by using a modified local search heuristic. The other technique is to prevent the peeling from being too greedy by imposing certain problem-specific criteria, e.g., balance criteria in constrained encoding [21, 23].

Note that greedy peeling can be applied to the second stage of prime covering for solving the set-cover problem, as mentioned earlier. But the chief advantage of greedy peeling over prime covering is the avoidance of the complicated and time-consuming process of cluster generation.

C. Taxonomy of Cluster-Covering Heuristics

We classify various heuristics developed previously for some VLSI optimization problems that fit our framework as prime covering or greedy peeling. Prime covering includes algorithms for combinational logic minimization [1, 18] and constrained encoding [7, 20, 23]. Included in the category of greedy peeling are the constrained encoding algorithm of [21], the topological planar routing algorithm of [3], the application timing algorithm of [11], and the monitoring-logic optimization algorithm of [10].

V. Performance of Greedy Peeling

The quality of the solution by greedy peeling will be measured by the performance ratio of the greedy peeling result to the result of an optimal solution.

**Theorem 2** Suppose that the maximum \(l\)-cluster problem can be solved with a performance ratio \(\epsilon\), and let \(r = \left(\frac{1}{2}\right)^l\); then the performance ratio of greedy peeling for the maximum \(k\)-cluster problem is

\[
\gamma \geq 1 - \left(1 - \frac{e}{r}\right)^r.
\]

We note that \(\gamma\) has the following properties:

- When \(r = 1\), then \(\gamma = \epsilon\).
- \(\gamma\) is a decreasing function. Since \(\lim_{x \to \infty} (1 - \frac{e}{x})^x = \left(\frac{1}{e}\right)^e\), \(\gamma\) is bounded by \(1 - \left(\frac{1}{e}\right)^e\). Note that \(0 < \epsilon < 1\).
- When \(\epsilon = 1\), then \(\gamma\) is bounded by \(1 - \frac{1}{e} = 0.632\).
- The derivative of the performance ratio with respect to \(\epsilon\) is \(\frac{\partial \gamma}{\partial \epsilon} \geq \left(1 - \frac{e}{r}\right)^{r-1}\). If \(r = 1\), then \(\frac{\partial \gamma}{\partial \epsilon} = 1\). If \(r \to \infty\), then \(\frac{\partial \gamma}{\partial \epsilon} \geq e^{-\epsilon}\).
- The derivative of the performance ratio with respect to \(r\) is \(\frac{\partial \gamma}{\partial r} \geq \frac{\epsilon}{(r^2)(1 - \frac{e}{r})^2} \ln(1 - \frac{e}{r})\).

Theorem 2 is based on a generalization of the performance analysis of a greedy heuristic for topological planar routing [3] to problems of the cluster-cover structure. We have extended it in two directions: each peeling has a performance ratio \(\epsilon\) (instead of 1), and each peeling may take \(l \geq 1\) clusters. Theorem 2 establishes that, if a maximum cluster can be obtained, greedy peeling has a guaranteed performance ratio \(\gamma\) (at least 0.632) for solving the maximum \(k\)-cluster problem. That is, the number of elements covered by \(k\) clusters found by greedy peeling is at least 63.2% of the number covered by the optimal solution. Note that \(0 \leq \gamma \leq 1\) for the maximization problem.

**Theorem 3** The performance ratio \(\gamma\) of greedy peeling for the minimum cover problem is bounded from above by \(\log |P|\), where \(P\) is the largest cluster.

Theorem 3 is a direct adaption of the work of [12], [15] and [2]. It states that, if we insist on covering all the ground elements by greedy peeling, then the number of clusters used may be \(\log |P|\) times the number of clusters actually needed. Note that here \(1 \leq \gamma \leq \infty\) for the minimization problem.

In view of a very recent discovery in [16] that, for any \(0 < c < 1/4\), the set-cover problem cannot be approximated within ratio of \(c \log |E|\) in polynomial time unless \(NP \subseteq DTIME([E]^{\text{polylog}[|E|]})\)—a very unlikely event—the greedy peeling heuristic is likely the best-possible approximation algorithm for the optimization problems of the cluster-cover structure. This means that, though sophisticated heuristics (e.g., based on prime covering) may sometimes produce better results, they are likely to have the same asymptotic performance as simple greedy peeling.

VI. Justification of Previous Empirical Results

We examine some experimental results reported in the literature on the use of greedy peeling in two VLSI design applications. In particular, we show how our analysis is confirmed by empirical evidence.

Table 2 shows experimental results of [11] for application timing assignment. The parameter \(\beta\) in the table is \(1/\alpha\). The authors make the following observation [11]: For some examples, there are relatively large differences in the number of assignments required for \(\beta = 1.1\) and \(\beta = 1\). In other words, while the slack lower bound of a large fraction of the fault sites is achieved by the first few assignments, a large number of new assignments may be needed to achieve the lower bound for the remaining few sites. Clearly, Theorem 2 indicates that the first few assignments can achieve the slack lower bound of a large fraction of the fault sites. Theorem 3, combined with the observation that greedy peeling is likely the best-possible approximation algorithm, implies that a large number of assignments (\(\log |P|\) times the number of assignments actually needed, where \(|P|\) is the size of the largest cluster) may be needed by greedy peeling to achieve the slack lower bound for the remaining few sites.

Another remark in [11] is that the number of assignments does not seem to grow with the size of combina-
Table 2: Results of application timing assignments.

<table>
<thead>
<tr>
<th>circuit</th>
<th>#edges</th>
<th>#fault sites</th>
<th>#assignments to achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>225</td>
<td>432</td>
<td>1</td>
</tr>
<tr>
<td>c499</td>
<td>1312</td>
<td>499</td>
<td>3</td>
</tr>
<tr>
<td>c4880</td>
<td>419</td>
<td>880</td>
<td>4</td>
</tr>
<tr>
<td>c1355</td>
<td>1312</td>
<td>1355</td>
<td>5</td>
</tr>
<tr>
<td>c1908</td>
<td>807</td>
<td>1908</td>
<td>4</td>
</tr>
<tr>
<td>c2670</td>
<td>1143</td>
<td>2746</td>
<td>3</td>
</tr>
<tr>
<td>c3540</td>
<td>724</td>
<td>3540</td>
<td>5</td>
</tr>
<tr>
<td>c5315</td>
<td>2978</td>
<td>5315</td>
<td>5</td>
</tr>
<tr>
<td>c6288</td>
<td>784</td>
<td>6283</td>
<td>5</td>
</tr>
<tr>
<td>c7522</td>
<td>3544</td>
<td>7553</td>
<td>4</td>
</tr>
</tbody>
</table>

The following two open questions are also posed in [11]:

- Can the TAT multiple heuristic be improved to bring the number of assignments for $\beta = 1$ closer to that for $\beta = 1.1$? Is there a good lower bound for the number of assignments necessary to achieve $\beta = 1$? For the first question, from Theorem 3 and the observation that greedy peeling is likely the best possible approximation algorithm, it is very unlikely that the heuristic can be improved to bring the number of assignments for $\beta = 1$ closer to that for $\beta = 1.1$ in such a way that the heuristic still runs in polynomial time. For the second question, the prime covering paradigm provides an optimal solution for the number of assignments necessary to achieve $\beta = 1$.

The same observations have been made regarding experimental results on multi-layer topological planar routing (Table 3) [3]. First, we can have a planar routing for the majority of nets. (...) Given a relatively large number of routing layers (say, more than four layers), we can route most of the nets without vias. Second, insisting on planar routing for all the nets is very costly, i.e., it requires a large number of routing layers. Although we can have planar routing for over 60% of the nets in the first five layers, we need 4-13 layers to route the remaining 20-40% of the nets. Therefore, it is unrealistic to insist on planar routing for all the nets.

We note that the reason why it is unrealistic to insist on planar routing for all the nets may come from the characteristic of the greedy peeling heuristic, not from the property of the original problem. Greedy peeling may result in a large number of layers, even though the number of layers actually needed is not very large. It will be interesting to apply prime covering to find the exact number of layers needed and to validate this observation experimentally.

Table 3: Results of multi-layer topological routing.

<table>
<thead>
<tr>
<th>circuit</th>
<th>1L</th>
<th>2L</th>
<th>3L</th>
<th>4L</th>
<th>5L</th>
<th>total layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus</td>
<td>41%</td>
<td>58%</td>
<td>70%</td>
<td>79%</td>
<td>83%</td>
<td>9</td>
</tr>
<tr>
<td>ex1</td>
<td>38%</td>
<td>52%</td>
<td>66%</td>
<td>76%</td>
<td>80%</td>
<td>9</td>
</tr>
<tr>
<td>ex3a</td>
<td>45%</td>
<td>59%</td>
<td>68%</td>
<td>75%</td>
<td>79%</td>
<td>11</td>
</tr>
<tr>
<td>ex3b</td>
<td>31%</td>
<td>53%</td>
<td>68%</td>
<td>75%</td>
<td>79%</td>
<td>10</td>
</tr>
<tr>
<td>ex3c</td>
<td>37%</td>
<td>55%</td>
<td>62%</td>
<td>70%</td>
<td>75%</td>
<td>13</td>
</tr>
<tr>
<td>ex4b</td>
<td>41%</td>
<td>57%</td>
<td>68%</td>
<td>75%</td>
<td>81%</td>
<td>13</td>
</tr>
<tr>
<td>ex5</td>
<td>31%</td>
<td>48%</td>
<td>59%</td>
<td>70%</td>
<td>83%</td>
<td>9</td>
</tr>
<tr>
<td>ex5b</td>
<td>31%</td>
<td>48%</td>
<td>60%</td>
<td>71%</td>
<td>79%</td>
<td>11</td>
</tr>
<tr>
<td>deut</td>
<td>23%</td>
<td>38%</td>
<td>50%</td>
<td>58%</td>
<td>63%</td>
<td>18</td>
</tr>
</tbody>
</table>

VII. Concluding Remarks

This paper has established that the widely studied two-level logic minimization shares the same combinatorial structure—cluster-cover—as many other VLSI-CAD optimization problems. All these problems can be cast into two meta-problems, called maximum cluster and minimum cover. They only differ in the definition of the compatibility predicate.

Prime covering is a paradigm for finding exact solutions. Remarkable progress has been made recently towards the use of prime covering to solve very large logic minimization problems [4, 5, 14, 17]. Our work provides, for the first time, a systematic foundation for applying these techniques to other cluster-cover problems.

A simple heuristic called greedy peeling is characterized in this paper. We derived certain conditions for a compatibility predicate which guarantee that greedy peeling produces optimum solutions. We have obtained theoretical results on the performance of greedy peeling, and applied them successfully to explain experimental results reported in the literature.

References


