Dynamics of Continuous, Discrete and Impulsive Systems
Series **A:** *Mathematical Analysis* **11 (2004)** 339-367

HYPERGRAPH PARTITIONING TECHNIQUES

Dorothy Kucar, Shawki Areibi[2] and Anthony Vannelli

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario N2L 3G1

[2]School of Engineering
Guelph University, Guelph, Ontario N1G 2W1

Abstract. Graph and hypergraph partitioning are important problems with applications in several areas including parallel processing, VLSI design automation, physical mapping of chromosomes and data classification. The problem is to divide the vertices of a hypergraph into k similar–sized blocks such that a given cost function defined over the hypergraph is optimized. The intent of this paper is to expose our readers to the theoretical and implementation issues of hypergraph partitioning techniques.

Keywords. Hypergraph Partitioning, Iterative Improvement, Multilevel Techniques, ILP Model, Eigenvector Methods, Simulated Annealing, Genetic Algorithms, Tabu Search

1 Introduction

Partitioning is used to divide a hypergraph into smaller, more manageable blocks while minimizing the number of connections between the blocks, called *cutnets*. Vertices that are strongly "related" are assigned to the same block; vertices that are weakly "related" are assigned to different blocks. The relationship between vertices is determined by the number of, or the weight of edges connecting the vertices. From this description, it is evident that partitioning is a generic tool applicable to problems in a wide variety of scientific disciplines:

- **Large–Scale Dynamical Systems**: Given a large dynamical system of the form

$$\mathbf{z}' = \hat{\mathbf{A}}\mathbf{z} + \hat{\mathbf{B}}\mathbf{u}$$

  where $\mathbf{z}$ represents the *state* of the system and $\mathbf{u}$ the input to the system: the goal is to decouple the directed graph associated with the system into smaller, highly–connected subgraphs. The system takes on a block–triangular structure where the blocks correspond to strongly–connected structures. It is then computationally easy to determine whether the blocks are individually stable, hence whether the overall system is stable [52, 51].

- **Large–scale matrix computations**: The goal is to minimize communication (edges) between processors (blocks) so that computations on each processor are nearly independent of computations on other processors. This computational model is used in the solution of discrete partial differential equations arising from a variety of scientific applications.

  Linear systems of the type $\mathbf{Ax} = \mathbf{b}$ are solved by factoring $\mathbf{A}$ into a product of lower and upper triangular matrices; the system $\mathbf{LUx} = \mathbf{b}$ is solved relatively easily. The order in which $\mathbf{L}$ and $\mathbf{U}$ are formed determines the sparsity of $\mathbf{L}$ and $\mathbf{U}$. Partitioning can be used to reorder $\mathbf{A}$ so as to minimize the non–zero entries of $\mathbf{L}$ and $\mathbf{U}$.

- **VLSI Physical Design**: In the placement stage, logic gates (modules) are grouped according to net connectivity and highly–connected modules are placed on a two–dimensional grid in proximity to one another so as to minimize overall net usage and reduce wiring congestion. The minimization of net usage and wiring congestion is accomplished through a combination of netlist partitioning, terminal propagation and the judicious use of quadratic optimization solvers. More thorough descriptions of VLSI placement techniques are found in [14, 39]

Due to the ever–increasing sizes of data sets involved in the above areas, partitioning will continue to retain its importance of the tools used in computer science. In this work, we are interested primarily in netlist partitioning applied to problems from VLSI design, although the problem formulations can be easily adapted to formats present in other research areas. The intent of this paper is to expose our readers to the theoretical and implementation issues of hypergraph partitioning techniques.

The rest of this paper is structured as follows: **Section 2** defines partitioning and introduces relevant notation. **Section 3** discuses iterative improvement methods that move individual vertices to different blocks if the cost function improves. **Section 4** covers advanced search heuristics such as GRASP, Simulated Annealing, Tabu Search and Genetic Algorithms. **Section 5** discuses relevant developments in multilevel methods in which clustering is combined with iterative improvement. **Section 6** discuses mathematical programming formulations of hypergraph partitioning starting with a general integer linear programming formulation, which forms the basis of approaches that use eigenvectors. **Section 7** describes testing methodologies and compares the various methods with respect to run–time and solution quality.

2 Preliminaries

Although this paper deals with the general problem of hypergraph partitioning, we will use examples from the Very Large Scale Integrated (VLSI) circuit

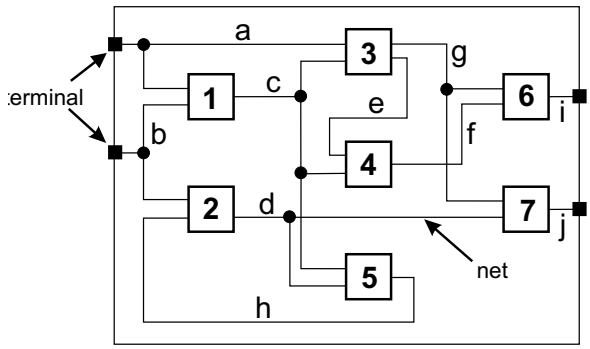


Figure 1: *Graphical Representation of a Netlist*

domain. A circuit is a collection of logic gates connected to one another by wires at the same electrical potential called *signal nets*. The points at which signal nets come into contact with logic gates are called *pins*. If a pin connects gates to areas outside circuit block, it is referred to as a *terminal*. Gate connectivity information is provided in the form of a *netlist* which contains net names followed by the names of gates they are connected to. A schematic identifying the various terminology is shown in Figure 1. Once the physics of circuit operation is abstracted away, a circuit becomes a purely combinatorial structure. The combinatorial and highly "visual" nature of VLSI physical design problems lends itself quite nicely to formulations involving graphs or matrices. Most often, a circuit is represented as a hypergraph where logic gates are weighted vertices (the weight is typically proportional to the number of ports) and nets are hyperedges. A hypergraph is the most natural representation of a circuit because, in a circuit, more than two vertices may be connected by the same signal net. In this section, we assume a hypergraph comprises of $|V|$ vertices, $|E|$ hyperedges, P pins and K blocks. Although a hypergraph is the natural representation of a circuit, it is very difficult to work with. Often, a hypergraph is represented as clique graph.

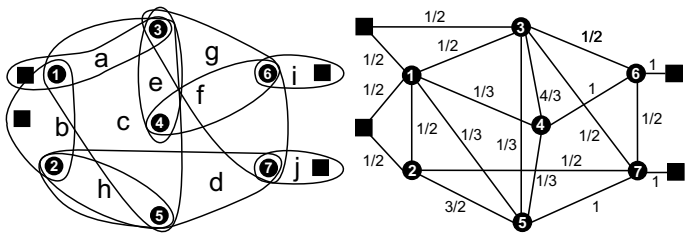
Definition 1 *A* **clique** *is a subgraph of graph $G(V, E)$ in which every vertex is adjacent to every other vertex.*

In the clique representation of a hypergraph, each vertex induces an edge, thus a net on $|V_i|$ vertices will be represented by $\begin{pmatrix} |V_i| \\ 2 \end{pmatrix}$ edges. In the generic clique model, the weight of each edge is

$$\frac{1}{|V_i| - 1}$$

although, many other models have been proposed [2], but none model the cut properties of a net exactly [32] in the sense that regardless of how vertices

in the clique are partitioned, the number of cuts should add up to 1. For example: a net on 4 vertices is represented by an equivalent 6–edge complete graph. To cut off one vertex from the rest requires cutting 3 edges, so the weight should be 1/3 (for a total edge weight of 1). However, to cut off two vertices from the rest requires cutting four edges (i.e. the edge weight should be 1/4) so there is clearly an inconsistency in this weighting scheme. Hypergraph and clique graph representations of the hypergraph in Figure 1 are depicted in Figure 2



(a) Hypergraph (b) Clique Graph

Figure 2: *Netlist Representations*

Given that a circuit is a fairly simple combinatorial object, it can also be represented by different types of matrices:

Definition 2 *Given a graph on $|V|$ vertices, the adjacency matrix,* $\mathbf{A}$*, is defined as*

$$a_{ij} = \begin{cases} w_{ij} & \text{if vertices } i \text{ and } j \text{ are adjacent} \quad i \neq j \\ 0 & \text{otherwise} \end{cases}$$

Definition 3 *Given a vector containing the degrees of $|V|$ vertices, the diagonal degree matrix,* $\mathbf{D}$*, is defined as:*

$$d_{ii} = \sum_{j=1}^{|V|} a_{ij}$$

Definition 4 *Given a graph on $|V|$ vertices, the corresponding $|V| \times |V|$ Laplacian matrix,* $\mathbf{L}$ *is given by* $\mathbf{L} = \mathbf{D} - \mathbf{A}$

If the graph is connected then one of the eigenvalues is 0 and the rest lie in the interval $(0, 1)$. In general the number of 0 eigenvalues equals the number of connected components in the graph. Lastly, we introduce some definitions which come in useful when we discuss hypergraph partitioning:

Definition 5 *A* **partitioning**, $f(P_K)$ *of* $|V|$ *vertices into* K *blocks,* $\{V_1, \ldots, V_K\}$ *is given by* $V_1 \cup V_2 \cup \cdots \cup V_K = V$ *where* $V_i \cap V_j = \emptyset$ *and* $\alpha|V| \leq |V_i| \leq \beta|V|$ *for* $1 \leq i < j \leq K$ *and* $0 \leq \alpha, \beta \leq 1$.

Definition 6 *Edges which have vertices in multiple blocks belong to the* **cutset** *of the hypergraph. Reducing the size of the cutset,* $f(P_K)$ *is one of the principal objective of partitioning.*

2.1 Solving the Partitioning Problem

One of the most interesting features of partitioning is that even the simplest case—graph bisection—is NP–complete [20] thus hypergraph partitioning is at least as hard. Some practical consequences of this fact are that the number of problem solutions grows exponentially with the problem size and that there may be multiple optimal solutions (although the ratio of optimal solutions to all solutions is nearly zero). The reality that problem sizes are prohibitively large imposes fairly strict criteria on possible approximation algorithms: the run–time complexity should be linear or log–linear with the size of the problem.

Currently, the most popular partitioning techniques are multilevel techniques that use interchange heuristics introduced by Kernighan/Lin [37] and Fiduccia/Mattheyses [19] or eigenvector–based partitioners [46, 48, 27] . These algorithms are included in partitioning packages such as hMetis [35], Chaco [27] and MLPart, [3]. These techniques have been studied and improved extensively over the past three decades. Current implementations have very modest run–time complexities and produce optimal or near–optimal results.

2.2 Partitioning Objectives and Constraints

In partitioning, we typically want to minimize the number of cutnets. More formally, if $E(C_i)$ represents hyperedges that belong to the set of cutnets, then the objective is:

$$\text{minimize } f(P_K) = \sum_{i=1}^{K} |E(C_i)|$$

Two types of constraints may be present: the first constraint ensures that each block has approximately the same number of vertices, or else, that it occupies approximately the same area. A typical balance constraint for bisection is:

$$b = \frac{|V_1|}{|V|}$$

with

$$0.45 \leq b \leq 0.55$$

For K–way partitioning, some researchers [36] use

$$\frac{|V|}{100K} \leq |V_i| \leq \frac{100|V|}{K}$$

The second type of constraint takes into account that some vertices may be fixed in a specific partition. The presence of fixed terminals actually adds some convexity to this otherwise highly non–convex problem, rendering it significantly easier (and faster) to solve [16]. In [1], the authors point out that the presence of fixed vertices requires fewer starts in an iterative improvement–based partitioner to approach a "good" solution.

3 Iterative Improvement Methods

In this section, we will outline the most significant developments in the field of iterative improvement partitioners. Iterative improvement partitioners initially fix nodes to arbitrary blocks. This is followed by a process of assigning nodes to different blocks if the connectivity between blocks is reduced. The goal in the end is to find an assignment of nodes such that the connectivity between blocks is globally minimal.

3.1 Kernighan–Lin

Kernighan and Lin's work was the earliest attempt at moving away from exhaustive search in determining the optimal cutset size subject to balance constraints. In their paper [37], they proposed a $O(|V|^2 \log |V|)$ heuristic for graph bisection based on exchanging pairs of vertices with the highest *gain* between two blocks, V_1 and V_2. They define a gain of a pair of vertices as the number of edges by which a cutset would decrease if vertices x and y were to be exchanged between blocks. Assuming a_{ij} are entries of a graph adjacency matrix, the gain is given by the formula:

$$g(v_x, v_y) = \left(\sum_{v_j \notin V_1} a_{xj} - \sum_{v_j \in V_1} a_{xj} \right) + \left(\sum_{v_j \notin V_2} a_{yj} - \sum_{v_j \in V_2} a_{yj} \right) - 2a_{xy}$$

The terms in parentheses count the number of vertices which have edges entirely within one block minus the number of vertices which have edges connecting vertices in the complementary block.

The procedure works as follows: vertices are initially divided into two sets, V_1 and V_2. A gain is computed for all pairs of vertices (v_i, v_j) with $v_i \in V_1$ and $v_j \in V_2$; the pair of vertices (v_x, v_y) with the highest gain are selected for exchanged; v_x and v_y are then removed from the list of exchange candidates. The gains for all pairs of vertices $v_i \in (V_1 - \{v_x\})$ and $v_j \in (V_2 - \{v_y\})$ are recomputed and the pair of vertices resulting in the highest gain are exchanged. The process continues until $g(v_x, v_y) = 0$, at which point, the

algorithm will have found a local minimum. The algorithm can be repeated to improve upon the current local minimum. Kernighan and Lin observe that two to four passes are necessary to obtain a locally optimal solution.

Kernighan and Lin also make the observation that using exhaustive search, partitioning $|V|$ vertices into 2 blocks of size $\frac{|V|}{2}$ results in

$$\frac{1}{2}\left(\begin{array}{c} |V| \\ \frac{|V|}{2} \end{array} \right)$$

configurations, which is a very large number indeed. As an example, they show that a specific 32–vertex graph has 3–5 global optima, thus a random configuration has a 1 in 10^7 chance of being an optimal solution.

In [50], Schweikert and Kernighan introduce a model which deals with hypergraphs directly. They point out that the major flaw of the (clique) graph model is to exaggerate the importance of nets with more than two connections. After partitioning, vertices connecting large nets tend to end up in the same block, whereas vertices connected to two point nets are split up between the blocks. They combine their hypergraph model in the Kernighan–Lin partitioning heuristic and obtain much better results on circuit partitioning problems.

3.2 Fiduccia–Mattheyses

The Fiduccia–Mattheyses [19] method is a linear–time ($O(P)$), per pass, hypergraph bisection heuristic. Its impressive run–time is due to an clever way of determining which vertex to move based on its gain and to an efficient data structure called a *bucket list*

The gain, $g(i)$, of a vertex is defined as the number of nets by which a cutset would decrease if vertex i were to be moved from the current block to the complementary block. Fiduccia and Mattheyses also define a balance criterion given as

$$b = \frac{|V_1|}{|V_1| + |V_2|}$$

Once a vertex is moved—subject to satisfying the balance criterion—it is locked in the complementary block for the remainder of the pass. Vertices can be moved until they all become locked or until the balance criterion prevents further moves. Once a vertex i is moved, the gains of a *specific set* of its free neighbors are recomputed. This set is composed of vertices incident on nets which are either entirely within the starting block, or nets which have only vertex in the starting block (and other vertices in the complementary block).

The gains are maintained in a $[-P_{max} \ldots P_{max}]$ bucket array whose ith entry contains a doubly–linked list of free vertices with gains currently equal to i. The maximum gain, P_{max}, is obtained in the case of a vertex of degree

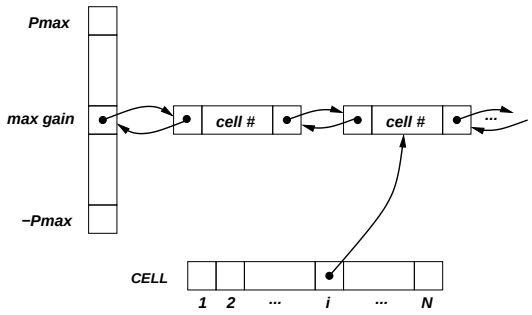


Figure 3: *Bucket List*

P_{max} (i.e. a vertex which is incident on P_{max} nets) that is moved across the block boundary and all of its incident nets are removed from the cutset.

The free vertices in this list are linked to vertices in the main vertex array so that during a gain update, vertices are removed from the bucket list in constant time (per vertex). Superior results are obtained if vertices are removed and inserted from the bucket list using a Last–In–First–Out (LIFO) scheme (over First–In–First–Out or random schemes) [24]. The authors of [24] speculate that a LIFO implementation is better because vertices that naturally cluster together will tend to be listed sequentially in the netlist.

3.3 Improvements on FM

In this section, we discuss four improvements to the original FM implementation which have helped increase the acceptance of FM, until recently, as the most widely–used partitioning technique. One improvement to standard FM is Krishnamurthy's lookahead scheme [41], in which the future implications of moving a vertex that is currently in the cutset, are considered in the gain computation. Sanchis [49] extends the FM concept to deal with multi–way partitioning. Her method computes multi–way partitionings directly by constructing $K(K-1)$ bucket lists (because vertex 1 can move to blocks $2, 3, \ldots K$, vertex 2 can move to blocks $1, 3, \ldots, K$, etc..). One pass of the algorithm consists of examining moves which result in the highest among all $K(K-1)$ bucket lists and the move with the highest gain and one that preserves the balance criterion is selected. After a move, all $K(K-1)$ bucket lists are updated. Dutt and Deng [15] observe that iterative improvement engines such as FM or Look–Ahead do not identify clusters adequately, and consequently, miss local optimal with respect to the number of cut hyperedges. Their method, called CLIP, identifies and moves entire clusters of vertices across the cutline. Cong and Lim [13] envisage K–way partitioning as concurrent bipartitioning problems where vertices are moved between two specific clusters of vertices (each cluster contains a handful of vertices). This

is different from traditional FM partitioning where candidate vertices are initially selected from the entire set of vertices.

4 Advanced Search Heuristics

Advanced search heuristics comprises techniques that simulate naturally–occurring process including *GRASP*, *Simulated Annealing*, *Genetic Algorithms* and *Tabu Search*. These techniques have been used extensively to solve difficult combinatorial problems thanks to their uphill–climbing capabilities.

The motivation for the Simulated Annealing [38] algorithm comes from an analogy between the physical annealing of solids and combinatorial optimization problems. Simulated Annealing is widely recognized as a method for determining the global minima of combinatorial optimization problems. Tabu Search finds some of its motivation in attempts to imitate "intelligent" processes [47], by providing heuristic search with a facility that implements a kind of "memory". Tabu search has been applied across a wide range of problem settings in which it consistently has found better solutions than other methods. GRASP is a random adaptive simple heuristic that intelligently constructs good initial solutions in an efficient manner. Genetic Algorithms on the other hand manipulate possible solutions to the decision problem in such a way that resembles the mechanics of natural selection and offers a number of advantages, the most important being the capability of exploring the parameter space.

4.1 GRASP

GRASP is a greedy randomized adaptive search procedure that has been successful in solving many combinatorial optimization problems efficiently [18, 9]. The GRASP methodology was developed in the late 1980s, and the acronym was coined by Feo [17]. Each iteration consists of a construction phase and a local optimization phase. The key to success for local search algorithms consists of the suitable choice of a neighborhood structure, efficient neighborhood search technique, and the starting solution. The GRASP construction phase plays an important role with respect to this last point, since it produces good starting solutions for local search. The construction phase intelligently constructs an initial solution via an adaptive randomized greedy function. Further improvement in the solution produced by the construction phase may be possible by using either a simple local improvement phase or a more sophisticated procedure in the form of Tabu Search or Simulated Annealing.

The GRASP algorithm starts with a construction phase followed by a local improvement phase. The implementation terminates after a certain number of phases or runs have passed. The construction phase is iterative,

greedy and adaptive in nature. It is *iterative* because the initial solution is built by considering one element at a time. The choice of the next element to be added is determined by ordering all elements in a list. The list of the best candidates is called the restricted candidate list (RCL). It is *greedy* because the addition of each element is guided by a greedy function. The construction phase is *randomized* by allowing the selection of the next element added to the solution to be any element in the RCL. Finally, it is *adaptive* because the element chosen at any iteration in a construction is a function of those previously chosen.

4.2 Simulated Annealing

Simulated Annealing **(SA)** searches the solution space of a combinatorial optimization problem, with the goal of finding a solution of minimum cost value. The motivation for the Simulated Annealing algorithm comes from an analogy between the physical annealing of solids and combinatorial optimization problems [38]. Physical annealing refers to the process of finding low energy states of a solid by initially melting the substance, and then lowering the temperature slowly, spending a long time at temperatures close to the freezing point. In the analogy, the different states of the substance correspond to the different feasible solutions of the combinatorial optimization problem, and the energy of the system corresponds to the function to be minimized.

A simple descent algorithm is analogous to reducing temperature is reduced quickly so that only moves which result in a reduction of the energy of the system are accepted. In Simulated Annealing [8], the algorithm alternatively attempts to avoid becoming trapped in a local optimum by accepting neighborhood moves which increase the value of *(f)* with probability $e^{(-\delta/T)}$ (called the *acceptance function*) where T is a control temperature in the analogy with physical annealing. The acceptance function implies that small increases in f are more likely to be accepted than large increases, and that when T is high, most moves will be accepted, but as T approaches zero most uphill moves will be rejected. The algorithm is depicted in Figure 4

In the body of the algorithm, the outer loop adjusts the temperature and the inner loop determines how many neighborhood moves are to be attempted at each temperature. The determination of the initial temperature, the rate at which the temperature is reduced, the number of iterations at each temperature and the criterion used for stopping is known as the *annealing schedule* [30]. The choice of annealing schedule has an important bearing on the performance of the algorithm [45].

4.2.1 Annealing Schedule

The Simulated Annealing algorithm, in its original formulation converges with probability **one** to a globally minimal configuration in either one of the

```
                    SIMULATED ANNEALING
current_solution ← initial_solution
current_cost ← evaluate(current_solution)
T ← T_initial
While (T ≥ T_final)
        for i = 1 to iteration(T) /* neighborhood moves */
                new_solution ← move(current_solution)
                new_cost ← evaluate(new_solution)
                Δcost ← new_cost - current_cost
                if(Δcost ≤ 0  OR  e^(-Δcost/T) > random())
                        /* accept new solution */
                        current_solution ← new_solution
                        current_cost ← new_cost
                EndIf
        EndFor
        T ← next_temp(T)
EndWhile
```

Figure 4: A Simulated Annealing Algorithm

following cases [42]:

❶ for each value of the control parameter, T_k, an infinite number of transitions is generated and $\lim_{k\to\infty} T_k = 0$. (the homogeneous algorithm);

❷ for each value T_k one transition is generated and T_k goes to zero not faster than $O([\log k]^{-1})$ (the inhomogeneous algorithm).

In any implementation of the algorithm, asymptotic convergence can only be approximated. The number of transitions for each value T_k, for example, must be finite and $\lim_{k\to\infty} T_k = 0$ can only be approximated in a finite number of values for T_k. Due to these approximations, the algorithm is no longer guaranteed to find a global minimum with probability one.

Initial Value of the Control Parameter

The initial value of T is determined in such a way that virtually all transitions are accepted, i.e., T_0 is such that $exp(-\delta cost_{ij}/T_0) \simeq 1$ for almost all i and j. The following empirical rule is proposed: choose a large value for T_0 and perform a number of transitions. If the acceptance ratio χ, defined as the number of accepted transitions divided by the number of proposed transitions, is less than a given value χ_0 (in [38] $\chi_0 = 0.8$), double the current value of T_0. Continue this procedure until the observed acceptance ratio exceeds χ_0.

Final Value of the Control Parameter

A stopping criterion, determining the final value of the control parameter, is either determined by fixing the number of values T_k, for which the algorithm is to be executed, or by terminating execution of the algorithm if the last configuration of consecutive Markov chains are identical for a number of chains.

Number of Iteration per Temperature

The simplest choice for L_k, the length of the k^{th} Markov chain, is a value depending (polynomial) on the size of the problem.

Decrement of the Control Parameter

A frequently used decrement rule is given by $c_{k+1} = \alpha \times c_k,\ k = 0, 1, 2, ...$ where α is a constant close to 1.

4.3 Genetic Algorithms

Genetic Algorithms **(GA's)** are a class of optimization algorithms that seek improved performance by sampling areas of the parameter space that have a high probability for leading to good solutions [53]. The algorithms are called genetic because the manipulation of possible solutions resembles the mechanics of natural selection. These algorithms which were introduced by Holland [29] in 1975 are based on the notion of propagating new solutions from parent solutions, employing mechanisms modeled after those currently believed to apply in genetics. The best offspring of the parent solutions are retained for a next generation of mating, thereby proceeding in an evolutionary fashion that encourages the survival of the fittest. As an optimization technique, Genetic Algorithms simultaneously examine and manipulate a set of possible solutions. Each candidate solution is represented by a string of symbols called a chromosome. The set of solutions P_j, is referred to as the population of the j^{th} generation. The population evolves for a prespecified total number of generations under the application of evolutionary rules called *Genetic Operators*.

4.3.1 Characteristics of Genetic Search

There are many characteristics of Genetic Algorithms which qualify them to be a robust based search procedure [4]. The first feature of Genetic Algorithms is that they are characterized to climb many peaks in parallel. Thus, the probability of finding a false peak is reduced over methods that proceed form point to point in the decision space. Secondly, the operators make use of a coding of the parameter space rather than the parameters themselves. Only objective function information is used, this results in a simpler implementation. Finally, although the approach has a stochastic flavor, it makes use of all the information that has been obtained during the search, and permits the structured exchange of that information [23].

4.3.2 Main Components of Genetic Search

There are essentially four basic components necessary for the successful implementation of a Genetic Algorithm. At the outset, there must be a code or

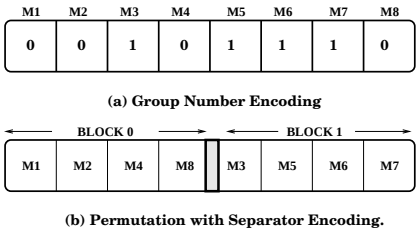


Figure 5: Representation schemes

scheme that allows for a bit string representation of possible solutions to the problem. Next, a suitable function must be devised that allows for a ranking or fitness assessment of any solution. The third component, contains transformation functions that create new individuals from existing solutions in a population. Finally, techniques for selecting parents for mating, and deletion methods to create new generations are required.

Representation Module

In the original GA's of Holland [29], each solution may be represented as a string of bits, where the interpretation of the meaning of the string is problem specific. As can be seen in Figure 5a, one way to represent the partitioning problem is to use *group-number encoding* where the j^{th} integer $i_j \in \{1, \ldots, k\}$ indicates the group number assigned to object j. This representation scheme creates a possibility of applying standard operators [44]. However an offspring may contain less than k groups; more-over, an offspring of two parents, both representing feasible solutions may be infeasible, since the constraint of having equal number of modules in each partition is not met. In this case either special *repair heuristics* are used to modify chromosomes to become feasible, or *penalty functions* that penalize infeasible solutions, are used to eliminate the problem. The second representation scheme is shown in Figure 5b. Here, the solution of the partitioning problem is encoded as $n + k - 1$ strings of distinct integer numbers. Integers from the range $\{1, .., n\}$ represent the objects, and integers from the range $\{n+1, \ldots, n+k-1\}$ represent separators; this is a *permutation with separators* encoding. This representation scheme leads to 100% feasible solutions [44], but requires more computation time due to the complexity of the unary operator involved.

Evaluation Module

Genetic Algorithms work by assigning a value to each string in the population according to a problem-specific *fitness* function. It is worth noting that nowhere except in the evaluation function is there any information (in the Genetic Algorithm) about the problem to be solved. For the circuit par-

titioning problem, the evaluation function measures the worth (number of cuts) of any chromosome (partition) for the circuit to be solved.

Reproduction Module

This module is perhaps the most significant component in the Genetic Algorithm. Operators in the reproduction module, mimic the biological evolution process, by using unary (mutation type) and higher order (crossover type) transformation to create new individuals. *Mutation* is simply the introduction of a random element, that creates new individuals by a small change in a single individual. When mutation is applied to a bit string, it sweeps down the list of bits, replacing each by a randomly selected bit, if a probability test is passed. On the other hand, *crossover* recombines the genetic material in two parent chromosomes to make two children. It is the structured yet random way that information from a pair of strings is combined to form an offspring.

4.3.3 Population Module

This module contains techniques for population initialization, generation replacement, and parent selection techniques. The initialization techniques generally used are based on pseudo-random methods. The algorithm will create its starting population by filling it with pseudo-randomly generated bit strings. Strings are selected for mating based on their fitness, those with greater fitness are awarded more offspring than those with lesser fitness. Parent selection techniques that are used, vary from stochastic to deterministic methods. The probability that a string i is selected for mating is p_i, "the ratio of the fitness of string i to the sum of all string fitness values", $p_i = \frac{fitness_i}{\sum_j fitness_j}$. The ratio of individual fitness to the fitness sum denotes a ranking of that string in the population. The Roulette Wheel Selection method is conceptually the simplest stochastic selection technique used. The ratio p_i is used to construct a weighted roulette wheel, with each string occupying an area on the wheel in proportion to this ratio. The wheel is then employed to determine the string that participates in the reproduction. A random number generator is invoked to determine the location of the spin on the roulette wheel.

4.3.4 GA Implementation

Figure 6 shows a simple Genetic Algorithm. The algorithm begins with an encoding and initialization phase during which each string in the population is assigned a uniformly distributed random point in the solution space. Each iteration of the genetic algorithm begins by evaluating the fitness of the current generation of strings. A new generation of offspring is created by applying crossover and mutation to pairs of parents who have been selected

based on their fitness. The algorithm terminates after some fixed number of iterations.

```
                GENETIC ALGORITHM
1. Encode Solution Space
2.(a) set pop_size, max_gen, gen=0;
   (b) set cross_rate, mutate_rate;
3. Initialize Population.
4. While  max_gen ≥ gen
            Evaluate Fitness
            For (i=1 to pop_size)
               Select (mate1,mate2)
               if (rnd(0,1) ≤ cross_rate)
                  child = Crossover(mate1,mate2);
               if (rnd(0,1) ≤ mutate_rate)
                  child = Mutation();
               Repair child if necessary
            End For
            Add offsprings to New Generation.
            gen = gen + 1
     End While
5. Return best chromosomes.
```

Figure 6: A generic Genetic Algorithm

4.4 Tabu Search

The Tabu Search technique was originally proposed in 1990 by Glover [21] as an optimization tool to solve nonlinear covering problems. Tabu Search has recently been applied to problems such as integer programming, scheduling [31], circuit partitioning [10] and graph coloring [28]. In general terms, Tabu Search is an iterative improvement procedure that starts from some initial feasible solution (i.e., assignment of cells to blocks for the partitioning problem) and attempts to determine a better solution in the manner of a steepest descent algorithm. However, Tabu Search is characterized by an ability to escape local optima which usually cause simple descent algorithms to terminate by using a short term memory of recent solutions. Moves that return a previously generated partition are considered *tabu* and are not considered. Moreover, Tabu Search permits back-tracking to previous solutions, which may ultimately lead, via a different direction, to partitions with fewer cut–nets.

4.4.1 Tabu Move

There are different attributes that can be used in creating the short term memory of Tabu lists for the circuit partitioning problem. One possibility is to identify attributes of a move based on the module value to be swapped from one block to another. Another way of identifying attributes of a move is to introduce additional information, referring not only to the modules to be moved but to positions (blocks) occupied by these modules. The recorded move attributes are used in Tabu Search to impose the constraints, called Tabu restrictions, that prevent moves from being chosen.

4.4.2 Tabu List

Tabu list management concerns updating the Tabu list; i.e., deciding on how many and which moves have to be made Tabu within any iteration of the search. The size of the Tabu list can noticeably affect the final results; a long list may give a lower overall minimum cost, but is usually obtained in a long time. Further, a shorter list may trap the routine in a cyclic search pattern. Empirical results show that Tabu list sizes that provide good results, often grow with the size of the problem. An appropriate list size depends on the strength of the Tabu restrictions employed.

4.4.3 Aspiration Criteria

Aspiration is a heuristic rule based on cost values that can temporarily release a solution from its Tabu status. The purpose of Aspiration is to increase the flexibility of the algorithm while preserving the basic features that allow the algorithm to escape from local optima and avoid cyclic behavioral. At all times during the search procedure, each cost c has an Aspiration value that is the current lowest cost of all solutions arrived at from solutions with value c. The actual Aspiration rule is that, if the cost associated with a Tabu solution is less than the Aspiration value associated with the cost of the current solution, then the Tabu status of the Tabu solution is *temporarily* ignored.

4.4.4 A Tabu Search Implementation for Circuit Partitioning

A Tabu Search implementation for circuit partitioning requires an initial feasible solution (partition), an associated cost —in this case, the net cut, a list of Tabu solutions and a maximum number of moves, (`max_num_iter`). The stopping criteria for the current implementation is described in detail in Section 4.4.5. The Tabu list and Aspiration levels corresponds to short term memory and the core search routine corresponds to intermediate and long term memory. Intermediate and long term memory functions are employed to achieve *regional intensification* and *global diversification* of the search [21, 22] of the search landscape

4.4.5 List Management & Stopping Criteria

Our Tabu list management techniques are based on static and dynamic approaches. In the static approach, the size of the Tabu lists remains fixed for the entire search period. The size of the Tabu list is chosen to be a function of the number of nodes within the circuit to be partitioned. Our experimentation with the Tabu Search algorithm indicates that choosing a $tabu\_list\_size = \alpha \times nodes$ (where α ranges from 0.1 to 0.2) yields good results in most cases. The dynamic implementation allows the size of the Tabu list to oscillate between two ranges. The first range is determined when

```
                                TABU SEARCH
Input:
      The netlist or the Graph G= (V,E)
      K = number of partitions; | T | = size of TabuList
      max_num_iter = maximum number of iterations
Initialization:
      Initial Partition = Generate a random solution
      s = (V1, V2.., Vk)
      num_iter = 0; bestpart = s; bestcut = f(s);
Main Loop:
      While num_iter < max_num_iter
            Pick module mi at random and compute
          ΔC for the interchange with mj
            Pick best module associated with best gain
            If (move not in TabuList) then
                  accept the move
                  Update TabuList;
                  Update the Aspiration Level;
            End If
            If (move in TabuList) then
                  If (Cost(s) < Aspiration(s0) then
                        Override the TabuList Status and accept the move
                        Update TabuList;
                        Update the Aspiration Level;
      End While
OutPut:
      Best Partition = bestpart; Best Cut = bestcut
```

Figure 7: A Tabu Search Algorithm With Aspiration

cycling occurs (Tabu lists are too short). The second range is determined when deterioration in solution quality occurs, which is caused by forbidding too many moves (Tabu lists are too large). Best sizes lie in an intermediate range between these extremes.

The stopping conditions used in this implementation are based on the following: (i) The search will terminate when `num_iter` is greater than the maximum number of iterations allowed `max_num_iter`, (ii) The search will terminate if no improvement on the best solution found so far can be made for a given number `max_num_iter` of consecutive iterations. The maximum number of iterations after which the routine terminates, depends on whether the routine starts from a random starting point or a good initial starting point. Experiments performed show the following: For random starting points, the algorithm requires more iterations to converge to good final solutions, so the maximum number of allowable iterations is set to "$max\_num\_iter = 100 \times nodes$", whereas for good starting points "$max\_num\_iter = 20 \times nodes$". The final solution gives the overall best partition and best cut after the specified maximum number of moves.

5 Multilevel Methods

The gist of multilevel partitioning is to construct a sequence of successively coarser graphs, to partition the coarsest graph (subject to a balance constraint) and to project the partitioning onto the next level finer graph while performing FM–type iterative improvement to further improve the partition The power of multilevel partitioning is evidenced during the iterative improvement phase, where moving one vertex across the partition boundary

corresponds to moving an entire group of vertices.

5.1 Clustering Phase

The purpose of clustering is to create a small network such that a good partition of the small network is not significantly worse than a partition of the flattened network. Ideally, large nets are contracted to nets with a few vertices so that the iterative improvement algorithm, which is very good at dealing with small nets (and not so good at dealing with large nets) can be used to its full potential. Several authors [26, 34, 3, 35] suggest clustering based on vertex matchings.

Definition 7 *A* **matching** *of $G = (V, E)$ is a subset of the edges with the property that no two edges share the same vertex. A* **heavy edge** *matching means edges with the heaviest weights are selected first. A* **maximum matching** *means every vertex has been matched.*

Candidates for clustering are selected based on the results of vertex matchings [3, 33]; matched vertices are then merged into "super–vertices". Matchings may be selected randomly or by decreasing net–weight (i.e. using heavy edge matchings).

Vertex matchings are used extensively because they tend to locate independent logical groupings of vertices. Independence is important in that it prevents the accumulation of many vertices in one super–vertex. Karypis and Kumar [33] use the following clustering schemes:

❶ select pairs of vertices that are present in same hyperedges by finding maximal matching of vertices based on a clique–graph representation (**edge coarsening**)

❷ find heavy–edge matching of vertices by non–increasing hyperedge size; after all hyperedges have been visited, collapse matched vertices (**hyperedge coarsening**)

❸ after hyperedges have been selected for matching, for each hyperedge that has not been contracted, its (unmatched) vertices are contracted together (**modified hyperedge coarsening**)

❹ for each vertex $v \in V$, consider all vertices that belong to hyperedges incident on v, whether they are matched or not

In [33], Karypis points out that there is no consistently better coarsening scheme for all netlists. Examples can be constructed for any of the above coarsening methods which fail to determine the correct partitioning. Karypis also suggests that a good stopping point for coarsening is when there are $30K$ vertices where K indicates the desired number of partitions.

After the coarsening phase, an initial bisection which satisfies the balance constraint is performed. It is not necessary at this point to produce an

"optimal" bisection since that is ultimately the purpose of the refinement phase.

5.2 Refinement Phase

The refinement process consists of successively applying an iterative improvement phase to successively finer hypergraphs, while unclustering after each pass of the interchange heuristic. Due to the space complexity of Sanchis' K–way FM algorithm and because vertices are clustered into logical super–vertices, Karypis *et al.* [36] use a downhill–only variant of FM which does not require the use of a bucket list. Their refinement method visits vertices in random order and moves them if they result in a positive gain (and preserve the balance criterion). If a vertex v is internal to the block being considered, then it is not moved; if v is a boundary vertex, it can be moved to a block which houses v's neighbors. The move which generates the highest gain is effectuated. In experiments, Karypis' *et al.* refinement method converges in only a few passes.

6 Mathematical Methods

Mathematical methods use enumerative or numerical techniques to determine optimal block assignments. Enumerative techniques such as branch–and–cut solve partitioning as an integer linear programming problem to optimality, but take a long time to do so. Numerical techniques construct vertex orderings using eigenvectors and split them.

6.1 An INP Formulation of Hypergraph Partitioning

The K–way partitioning problem can be formulated as a 0–1 Integer Non–linear Program (INP) [7]. Assume there are $j = 1 \ldots K$ blocks, $i = 1 \ldots |V|$ vertices, $k = 1 \ldots |E|$ hyperedges and $i' = 1 \ldots p_k$ vertices per hyperedge k; define $x_{ij} = 1$ if vertex i is in block j and 0 otherwise. Thus, if a specific net consists of vertices 1 through 4, then it will be uncut if

$$x_{i1}x_{i2}x_{i3}x_{i4} = 1$$

The non–linear programming problem consists of maximizing the sum of uncut nets (hence minimizing the sum of cut–nets)

$$\max \quad \sum_{j=1}^{K} \prod_{i=1}^{|E|} x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{j=1}^{K} x_{ij} = 1 \quad \forall\, i \tag{2}$$

$$l_j \leq \sum_{i=1}^{|V|} x_{ij} \leq u_j \quad \forall\, j \tag{3}$$

$$x_{ij} = \{0, 1\} \tag{4}$$

$$y_{kj} = \{0, 1\} \tag{5}$$

The objective function maximizes the sum of uncut nets where $k(i')$ finds the coordinate of i'th vertex of edge k in V. Constraints (2) stipulate that each vertex be assigned to exactly one block. Constraints (3) impose block size limits. Finally, the last two sets of constraints specify all variables in a feasible solution must take on values 0 or 1.

6.2 An ILP Formulation of Hypergraph Partitioning

We can rewrite the INP as an Integer Linear Program (ILP) by introducing indicator variables $y_{kj} = 1$ if net k is entirely in block j and 0 otherwise. The linear programming problem is nearly identical to the non–linear programming formulation except that objective is

$$\max \quad \sum_{j=1}^{K} \sum_{k=1}^{|E|} y_{kj}$$

and the net constraints are replaced by $y_{kj} \leq x_{k(i')j} \quad \forall\, i',\, j,\, k$. These constraints indicate whether vertex $k(i')$ is incident on hyperedge j; they effectively remove the nonlinearity in the objective function. They make sense because, for example if a net consists of vertices 1 and 2, and if $x_{1j}, x_{2j} \in \{0, 1\}$ then $x_{1j}x_{2j} \leq x_{1j}$ and $x_{1j}x_{2j} \leq x_{2j}$. The remaining constraints are identical to their counterparts in the INP model.

In the case where there are two blocks, the difficulty is found in the block size constraint. Without this constraint, the constraints matrix is unimodular and if the constraints matrix is unimodular, the restriction $x_{ij} = \{0, 1\}$ can be relaxed into $x_{ij} = [0, 1]$. The resulting linear program can be solved in polynomial time using interior point methods. However, block size constraints are useful in order to avoid situations where a disproportionate number of vertices are assigned to one block. In the following subsections, we derive a well–studied eigenvector technique from the ILP formulation.

6.3 An Eigenvector Method

If we examine the exact 0–1 formulation of graph partitioning, and set $K = 2$, $l_1 = u_1 = l_2 = u_2 = |V|/2$, we obtain the following problem:

$$\begin{aligned}
\max \quad & \sum_{k=1}^{|E|} y_{k1} + \sum_{k=1}^{|E|} y_{k2} && (6)\\
\text{s.t.} \quad & y_{k1} \leq x_{k(i')1} && (7)\\
& y_{k2} \leq x_{k(i')2} && (8)\\
& x_{i1} + x_{i2} = 1 \quad \forall i && (9)\\
& \sum_{i=1}^{|V|} x_{i1} = \frac{|V|}{2} && (10)\\
& \sum_{i=1}^{|V|} x_{i2} = \frac{|V|}{2} && (11)\\
& x_{ij} = \{0,1\} \text{ and } y_{kj} = \{0,1\} && (12)
\end{aligned}$$

If the variables in Equations (9) and (12) are rescaled so that $x_{i1} + x_{i2} = 0$ i.e.: $x_{i1} = -x_{i2}$ or $x_i = \{-1, 1\}$ then constraints (9) through (12) can be rewritten as:

$$\begin{aligned}
& \sum_{i=1}^{|V|} x_i = 0\\
& x_i = \pm 1 \quad i = 1 \ldots |V|
\end{aligned}$$

Equations (6) through (8), on the other hand need first to be combined into a quadratic cost function in the following manner: since the x_i are all binary variables

$$\begin{aligned}
y_{k1} \leq x_{k(i')1} \quad \text{for } i' = 1, 2 \quad & \text{is equivalent to}\\
x_{k(1)1} x_{k(2)1} \leq x_{k(i')1} \quad & \text{for } i' = 1, 2
\end{aligned}$$

With some algebra, the cost function can be identified as $(-)$ the off–diagonal terms of the product of $\mathbf{x}^T\mathbf{L}\mathbf{x}$ where $\mathbf{L}$ is the Laplacian matrix of the graph induced by the ILP. Recall, the Laplacian is defined as

$$\ell_{ij} = \begin{cases} \sum a_{ij} & \text{if } i = j\\ -a_{ij} & \text{if } i, j \text{ connected, but } i \neq j\\ 0 & \text{otherwise} \end{cases}$$

The diagonal terms of $\mathbf{L}$ are always greater than zero so the minimization problem is equivalent to optimizing the off–diagonal elements. This establishes the equivalence between Equations (6)-(11) and the standard graph

bisection formulation [25, 12]:

$$\min \quad \mathbf{x}^T\mathbf{L}\mathbf{x} \tag{13}$$

$$\text{s.t.} \quad x_i = \pm 1 \tag{14}$$

$$\sum_{i}^{|V|} x_i = 0 \tag{15}$$

Due to its discrete nature, this problem is very difficult to solve exactly except for very small instances. A reasonable approximation is to relax the first constraint to

$$\mathbf{x}^T\mathbf{x} = 1 \tag{16}$$

which essentially replaces the solution space hypercube with a sphere and rounds the solution of the continuous problem to the nearest feasible discrete point. The continuous problem is normally solved by minimizing the corresponding Lagrangian:

$$L(\mathbf{x};\lambda) = \mathbf{x}^T\mathbf{L}\mathbf{x} - \lambda(\mathbf{x}^T\mathbf{x} - 1)$$

The solution of which is

$$\lambda\mathbf{x} = \mathbf{L}\mathbf{x}$$

i.e.: λ is an eigenvalue of the $\mathbf{L}$. The maximum is given by

$$\mathbf{x}^T\mathbf{L}\mathbf{x} = \lambda\mathbf{x}^T\mathbf{x} = \lambda$$

by constraint (16) The minimum is given by λ_2 because $\lambda_1 = 0$ gives $\mathbf{x} = \mathbf{1}$ which violates constraint (15).

The components of the second eigenvector which are < 0 represent the coordinates of vertices in the first block; components of the second eigenvector which are ≥ 0 represent the coordinates of vertices in the second block. The effect is that the eigenvector components of vertices which are strongly (weakly) connected are close (far away), thus strongly connected vertices are more likely to be assigned to the same block.

7 Testing and Results

In this section, we compare a variety of partitioning techniques including branch–and–cut on the ILP formulation of partitioning, multi–level interchange via hMetis, Tabu Search, Simulated Annealing, Genetic Algorithm and the eigenvector technique. Test problem statistics are provided in Table 1. Using the test problems outlined in Table 1, we make assumptions that the vertices are partitioned into two blocks, vertices are weighted according to the number of incident nets and each block is allowed to have between as close to 50% of total vertex weight as possible.

Table 1: Test Problem Data from [40]

Circuit	Cells	Nets	Pins	Pads	Cell Degree			Net Size		
					MAX	$\overline{x}$	σ	MAX	$\overline{x}$	σ
Fract	125	147	462	24	7	3.1	1.6	17	3.1	2.2
Prim1	833	902	2908	81	9	3.4	1.2	18	3.2	2.5
Struct	1888	1920	5471	64	4	2.8	0.6	17	2.8	1.9
Ind1	2271	2192	7743	814	10	3.4	1.1	318	3.5	9.0
Prim2	2907	3029	18407	107	9	3.7	1.5	37	3.7	3.8
Bio	6417	5742	21040	97	6	3.2	1.0	861	3.6	20.8
Ind2	12142	13419	48158	495	12	3.8	1.8	585	3.5	10.9
Ind3	15057	21808	65416	374	12	4.3	1.4	325	2.9	3.2
Avq.s	21854	22124	76231	64	7	3.4	1.4	4042	3.4	53.3
Avq.l	25144	25384	82751	64	7	3.2	1.2	4042	3.2	49.8

7.1 Using hMetis

hMetis is a hypergraph partitioning package based on the multilevel interchange paradigm [1]. To obtain our results, we used the following parameter settings

hMetis **Parameter Settings**

`Nparts`	2	partition into 2 blocks
`UBFactor`	1	balance factor of 49:51
`Nruns`	20	20 runs
`CType`	1	coarsen type: hybrid hypergraph first–choice
`RType`	1	refine type: Fiduccia–Mattheyses
`VCycle`	1	V–cycle refinement on final solution of each bisection
`Reconst`	0	no
`dbglvl`	0	don't print intermediate output

7.2 Using Advanced Search Heuristics

This section summarizes some of the parameters selected for the different advanced search techniques and a comparison in terms of solution quality and CPU time.

7.2.1 Tabu Search Parameter Setting

As explained in Section 4.4 there are a number of parameters that need to be tuned for the Tabu Search implementation. This includes the tabu criteria, tabu size, aspiration criteria and stopping criteria. The following are some of the parameters used [11]:

Tabu Search **Parameter Settings**

[1] It is available free of charge at `http://www-users.cs.umn.edu/~karypis/metis/hmetis/`

`Partitions`	2	partition into 2 blocks
`TabuCriteria`	1	Move Tabu if module is registered
`TabuLength`	1	Oscillates between 2 values
`Aspiration`	1	override tabu status if better than best
`Stopping Criteria`	10	nbiter = 10 × nmods

7.2.2 Genetic Algorithms Parameter Setting

The performance of any genetic algorithm implementation depends on several parameters including the population size, generation size, crossover rate, mutation rate, selection and replacement methods. The following are some parameters used in our current implementation[6]:

Genetic Algorithms **Parameter Settings**

`Partitions`	2	partition into 2 blocks
`init-tech`	1	Grasp Populates 25% of initial Population
`crssrate`	99%	Cross Over Rate
`mutrate`	0.1%	Mutation Rate
`Selection`	3	Ranking
`Replacemenent`	3	Replace Most Inferior Parent with Child
`Repair Alg`	2	Simple Repair Heuristic
`popsize`	100	Population Size
`gensize`	100	Generation Size (Stopping Criteria)

7.2.3 Simulated Annealing Parameter Setting

According to Section 4.2 we have adapted an annealing schedule similar to that proposed by White[54]. The most important parameters deal with the selection of the initial temperature, final temperature, the amount of decrease of temperature and most importantly the number of iterations performed within each temperature. The following summarizes the parameter settings used for the current implementation [5].

Simulated Annealing **Parameter Settings**

`Partitions`	2	partition into 2 blocks
`InitTemp`	T_0	Temp doubled until $\chi \geq 80\%$
`FinalTemp`	T_{fin}	0.01
`IterPerTemp`	L_k	5 × NumOfMods
`TempDecrease`	α	0.99

7.2.4 Comparison

Table 2 shows the results obtained for different benchmarks. The best, average and standard deviation are listed for Simulated Annealing, Tabu Search and Genetic Algorithms. The results for Simulated Annealing and Tabu

Circuit	Simulated Annealing			Genetic Algorithms			Tabu Search		
	Best	$\overline{x}$	σ	Best	$\overline{x}$	σ	Best	$\overline{x}$	σ
Fract	11	12.2	2.4	24	26.8	1.5	11	11	0
Prim1	63	72.2	6.1	61	67.7	3.4	54	59	3.5
Struct	47	54	7.5	47	61.9	6.0	47	64	12.7
Ind1	51	60.2	7.9	61	78.1	9.0	49	54	5.5
Prim2	225	240	11.2	178	206	10.8	191	207	18.4
Bio	126	136	10.1	88	122	16.8	119	134	12.3
Ind2	371	398	19.1	279	381	46.4	353	398	34.7
Ind3	522	623	34.5	533	718	55.1	299	401	119.6
Avq.s	384	507	71.8	484	622	56.4	516	613	111.2
Avq.l	536	656	79.5	483	593	79.8	329	681	178.2

Table 2: Advanced Search Techniques Results, Best, Averge and Standard Deviation

Search are based on five runs whereas results for the Genetic Algorithm are based on 100 generations. GRASP has been used to construct the initial solution for all heuristic techniques. The Genetic Algorithm mainly uses 25% of the solutions obtained by GRASP and the remaining 75% of the population is based on random initial solutions. The table indicates that on average the Tabu Search and Genetic Algorithm produce better results than the Simulated Annealing technique. However the Genetic Algorithms consumes on average 10 times slower than the Tabu Search implementation. For some of the large circuits the Simulated Annealing technique achieves better results than Genetic Algorithms and Tabu Search.

7.3 Solving the ILP Formulation

CPLEX with the following custom parameter settings is used to solve the ILP:

CPLEX Parameter Settings

`node select parameter`	best estimate strategy
`variable select`	pseudo–reduced costs
`start algorithm`	primal simplex
`objective lower bound`	$\lvert E\rvert -$ (hMetis objective value)

The rationale behind selecting the objective lower bound to be equal to $|E| - hM$ is because the result obtained by hMetis, which is a heuristic, should be a underestimation of the optimal number of uncut nets. Giving the CPLEX such a tight lower bound should reduce the size of the search tree considerably.

7.4 Using the Eigenvector Technique

We constructed the Laplacian of the graph representation of the various netlists with net weights equal to $w_i = \frac{1}{\left(\sum_{i \neq j} a_{ij}\right) - 1}$. We then found the second smallest eigenvalue and Fiedler vector, $\mathbf{v}$ of $\mathbf{L}$ and sorted it in ascending

circuit	ILP	time	hM	time	SA	time	GA	time	TS	time	eig	time
Fract	11	4	11	0.1	11	17	24	22	11	0.3	12	0.1
Prim1	53	2d	53	0.33	63	125	61	147	54	3	91	4.8
Struct	34	3d	34	0.29	47	281	47	307	47	4	46	14.5
Ind1	-	-	19	0.56	51	312	61	335	49	8	30	21
Prim2	-	-	158	2.10	225	462	178	478	191	13	284	14.5
Bio	-	-	83	2.02	126	935	88	991	119	27	150	60
Ind2	-	-	191	3.7	371	2413	279	2809	353	81	525	119
Ind3	-	-	282	5.5	522	4322	533	4993	299	144	475	148
Avq.s	-	-	144	5.5	384	5086	484	4555	516	251	478	1191
Avq.l	-	-	143	4.8	536	6930	483	5274	329	416	687	1419

Table 3: 2–Way Partitioning results: (Objective: Number of Cuts, Time: Seconds)

order of components to yield $\mathbf{v}^*$. We then reordered the vertices of the netlist according to the coordinates of $\mathbf{v}^*$ with respect to $\mathbf{v}$. Finally, the number of cuts was computed by finding the cut position between rows $\lfloor \frac{|V|}{2} \rfloor$ and $\lfloor \frac{|V|}{2} \rfloor + 1$ of $\mathbf{A}$.

7.5 Comparison

Table 3 introduces the results obtained using (a) ILP formulation introduced in Section 6.2 (b) hMetis (c) Simulated Annealing (d) Genetic Algorithms (e) Tabu Search and (f) Eigenvector Technique. The table is divided into three parts, small, medium and large benchmarks. The hMetis technique produces optimal results for small circuits. Exact partitions for medium and large circuits was not possible due to the large CPU time required. It is evident from Table 3 that hMetis outperforms other heuristic search techniques in terms of quality of solutions and CPU time. A comparison between the advanced search techniques clearly indicates that Tabu Search produces better results than those obtained by Simulated Annealing and produces results comparable to those obtained by the Genetic Algorithm. Tabu Search is on average 10 to 15 faster than Simulated Annealing and Genetic Algorithms. The Eigenvector approach produces good results for small and medium circuits, but the solution quality deteriorates as the size of the benchmarks increase in size.

8 Summary

Hypergraph partitioning is a useful tool in a variety of scientific disciplines, although the examples we presented were from the VLSI CAD research area. The goal of this paper was to expose the reader to a variety of partitioning techniques ranging from multilevel methods to advanced search heuristics to mathematical programming methods. In the results section, we compared

five different methods using run–time and number of cuts as points of comparison.

9 Acknowledgements

This research is partially supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) operating grant (OGP 0043417).

References

[1] C. J. Alpert, A. E. Caldwell, A. B. Kahng, and I. L. Markov, *Hypergraph Partitioning with Fixed Vertices*, IEEE Transactions on Computer–Aided Design of Circuits and Systems, 19 (2000), pp. 267–272.

[2] C. J. Alpert and A. B. Kahng, *Recent Directions in Netlist Partitioning: A Survey*, Integration: The VLSI Journal, (1995), pp. 1–93.

[3] ———, *Multilevel Circuit Partitioning*, IEEE Transactions on Computer–Aided Design of Circuits and Systems, 17 (1998).

[4] S. Areibi, *The Effect of Clustering and Local Search on Genetic Algorithms*, in Recent Advances in Soft Computing, Leicester, UK, July 1999.

[5] ———, *Simple Yet Effective Techniques to Improve Flat Multiway Circuit Partitioning*, in 2000 Canadian Conference on Electrical and Computer Engineering, Nova Scotia, Canada, May 2000, IEEE.

[6] ———, *Memetic Algorithms for VLSI Physical Design Automation: Implementation Issues*, in Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, California, July 2001, Morgan Kaufman, pp. 140–145.

[7] S. Areibi and A. Vannelli, *Advanced Search Techniques for Circuit Partitioning*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1993, pp. 77–97.

[8] S. Areibi and A. Vannelli, *Advanced Search Techniques for Circuit Partitioning*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 16 (1994), pp. 77–98.

[9] ———, *A GRASP Clustering Technique for Circuit Partitioning*, 35 (1997), pp. 711–724.

[10] ———, *Tabu Search: A Meta Heuristic for Netlist Partitioning*, VLSI Journal, 11 (2000), pp. 259–283.

[11] ———, *Tabu Search: Implementation & Complexity Analysis for Netlist Partitioning*, Int'l Journal of Computers and Their Applications, (2002). In Print.

[12] T. F. Chan, J. R. Gilbert, and S.-H. Teng, *Geometric Spectral Partitioning*, tech. report, Xerox PARC, 1995.

[13] J. Cong and S. K. Lim, *Multiway Partitioning with Pairwise Movement*, in Proceedings of the 1998 International Conference on Computer–Aided Design, 1998.

[14] A. E. Dunlop and B. W. Kernighan, *A Procedure for Placement of Standard–Cell VLSI Circuits*, IEEE Transactions on Computer–Aided Design, CAD–4 (1985), pp. 291–307.

[15] S. Dutt and W. Deng, *VLSI circuit partitioning by cluster–removal using iterative improvement techniques*, in Proceedings of the International Conference on Computer–Aided Design, 1996.

[16] H. Etawil, S. Areibi, and A. Vannelli, *ARP: A Convex Optimization Based Method for Global Placement*, in IEEE Transactions on Computer–Aided Design, 2001.

[17] T. Feo and M. Resende, *A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem*, Operation Research Letters, 8 (1989), pp. 67,71.

[18] T. Feo, M. Resende, and S. Smith, *A Greedy Randomized Adaptive Search Procedure for The Maximum Ind ependent Set*, Operations Research, (1994). Journal of Operations Research.

[19] C. M. Fiduccia and R. M. Mattheyses, *A Linear–Time Heuristic for Improving Network Partitions*, in Proceedings of the 19th IEEE/ACM Design Automation Conference, 1982.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP–Completeness*, Freeman, San Francisco, CA, 1979.

[21] F. Glover, *Tabu Search Part I*, ORSA Journal on Computing, 1 (1990), pp. 190–206.

[22] ———, *Tabu Search Part II*, ORSA Journal on Computing, 2 (1990), pp. 4–32.

[23] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Inc, Reading, Massachusetts, 1989.

[24] L. Hagen, D. J.-H. Huang, and A. B. Kahng, *On Implementation Choices for Iterative Improvement Partitioning Algorithms*, IEEE Transactions on Computer-Aided Design of Circuits and Systems, 16 (1997), pp. 1199–1205.

[25] K. M. Hall, *An r–Dimensional Quadratic Placement Algorithm*, Management Science, 17 (1970), pp. 219–229.

[26] B. Hendrickson and R. Leland, *A Multilevel Algorithm for Partitioning Graphs*, in Proceedings of the 1995 Supercomputing Conference, 1995.

[27] ———, *The Chaco User's Guide, Version 2.0*, Sandia National Laboratories, July 1995.

[28] A. Hertz and D. Werra, *Using Tabu Search Technique for Graph Coloring*, Computing, (1987), pp. 345–351.

[29] J. Holland, *Adaption in Natural and Artificial Systems*, University of Michigan, Press, Ann Arbor, 1975.

[30] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, *An Efficient General Cooling Schedule for Simulated Annealing*, in IEEE International Conference on CAD, November 1986, pp. 381–384.

[31] J. Hurink and S. Knust, *A Tabu Search Algorithm for Scheduling A Single Robot in a Job-Shop Environment*, Journal of Discrete Applied Mathematics, (2002), pp. 181–203.

[32] E. Ihler, D. Wagner, and F. Wagner, *Modelling hypergraphs by graphs with the same mincut properties*, Information Processing Letters, 45 (1993), pp. 171–175.

[33] G. Karypis, *Multilevel Hypergraph Partitioning*, in Multilevel Optimization in VLSICAD, Kluwer Academic Publishers, 2003.

[34] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *Multilevel Hypergraph Partitioning: Application in VLSI Domain*, in Proceedings of the 34th ACM/IEEE Design Automation Conference, 1997.

[35] G. Karypis and V. Kumar, *hMETIS: A Hypergraph Partitioning Package, Version 1.5.3*, University of Minnesota Department of Computer Science/Army HPC Research Center, 1998.

[36] ———, *Multilevel k–way Hypergraph Partitioning*, in Proceedings of the 36th ACM/IEEE Design Automation Conference, 1999, pp. 343–348.

[37] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, (1970), pp. 291–307.

[38] S. Kirkpatrick, C. Gelatt, and M. Vecchi, *Optimization BY Simulated Annealing*, Science, 220 (May 1983), pp. 671–680.

[39] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, *GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization*, IEEE Transactions on Computer–Aided Design, 10 (1991), pp. 356–365.

[40] K. Kozminski, *Benchmarks for Layout Synthesis–Evolution and Current Status*, in Proceedings of the 28th IEEE/ACM Design Automation Conference, 1991, pp. 265–270.

[41] B. Krishnamurty, *An improved min–cut algorithm for partitioning VLSI networks*, IEEE Transactions on Computers, C-33 (1984), pp. 438–446.

[42] P. V. Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*, Kluwer Academic Publishers, Boston, 1988.

[43] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, 1990.

[44] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlog, Berlin, Heidelberg, 1992.

[45] S. Nahar, S. Sahni, and E. Shragowitz, *Experiments with Simulated Annealing*, in Proceedings of The 22*nd* DAC, Las Vegas, Nevada, 1985, IEEE/ACM, pp. 748–752.

[46] A. Pothen, H. D. Simon, and K. P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 430–452.

[47] C. Reeves, *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc, New York, 1993.

[48] B. M. Riess, K. Doll, and F. M. Johannes, *Partitioning very large circuits using analytical placement techniques*, in Proceedings of the 31st IEEE/ACM Design Automation Conference, 1994, pp. 646–651.

[49] L. A. Sanchis, *Multi–way Network partitioning*, IEEE Transactions on Computers, 38 (1989), pp. 62–81.

[50] D. G. Schweikert and B. W. Kernighan, *A Proper Model for the Partitioning of Electrical Circuits*, in Proceedings of the 9th IEEE/ACM Design Automation Conference, 1972, pp. 57–62.

[51] M. E. Sezer and D. D. Siljak, *On Structural Decomposition and Stabilization of Large–Scale Control Systems*, IEEE Transactions on Automatic Control, AC–26 (1981), pp. 439–444.

[52] D. D. Siljak and M. B. Vukcevic, *Decentralization, Stabilization, and Estimation of Large–Scale Linear Systems*, IEEE Transactions on Automatic Control, (1976), pp. 363–366.

[53] V. Venkatasubramanian and I. Androulakis, *A Genetic Algorithm Framework for Process Design and Optimization*, Computers Chemical Engineering, 15 (1991), pp. 217–228.

[54] S. White, *Concepts of Scale in Simulated Annealing*, in IEEE Int Conf on Computer Design, IEEE, November 1984, pp. 646–651.