1-1-2002

An Efficient Multiway Hypergraph Partitioning Algorithm for VLSI Layout

Lixin Tao
Pace University

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT

Number 184, December 2002

An Efficient Multiway Hypergraph Partitioning Algorithm for VLSI Layout

Lixin Tao

PACE UNIVERSITY

Lixin Tao is Professor of Computer Science at Pace University.

Dr. Tao holds the Ph.D. in Computer Science from the University of Pennsylvania. He has conducted extensive research in parallel and distributed computing, Internet computing, distributed component technologies, software engineering, and operations research. His contribution in operations research focuses on graph embedding, combinatorial optimization and their applications.

An Efficient Multiway Hypergraph Partitioning Algorithm for VLSI Layout

Lixin Tao

Abstract

In this paper, we propose an effective multiway hypergraph partitioning algorithm. We introduce the concept of net gain and embed it in the selection of cell moves. Unlike traditional FM-based iterative improvement algorithms in which the selection of the next cell to move is only based on its cell gain, our algorithm selects a cell based on both its cell gain and the sum of all net gains for those nets incident to the cell. To escape from local optima and to search broader solution space, we propose a new perturbation mechanism. These two strategies significantly enhance the solution quality produced by our algorithm. Based on our experimental justification, we smoothly decrease the number of iterations from pass to pass to reduce the computational effort so that our algorithm can partition large benchmark circuits with reasonable run time. Compared with the recent multiway partitioning algorithms proposed by Dasdan and Aykanat [5], our algorithm significantly outperforms theirs in term of solution quality (cutsize) and run time: the average improvements in terms of average cutsize over their PLM3 and PFM3 are 47.64% and 36.76% with only 37.17% and 9.66% of their run time respectively.

Index Terms: Multiway circuit partitioning, hypergraph partitioning, iterative improvement, VLSI design

1. Introduction

The problem of hypergraph partitioning has been around for at least a quarter of a century. It focuses on dividing a given hypergraph into a collection of smaller blocks subject to balance constraints while having the number of connections among these blocks minimized. Early applications of the problem were centered on VLSI circuit design [1], and it is still a major research direction. In recent years, with the rapid development in the field of database and its applications, such as very large databases, web database and large decision support systems, the application of multiway hypergraph partitioning has been extended into other areas including data mining [14], data classifications and efficient storage of very large databases on disks [19].

Usually, circuits are represented as hypergraphs [2] while the cells and the nets in circuits are represented by vertices and hyperedges respectively. The hypergraph partitioning problem is an NP-hard problem [8]. (In [8], graph partitioning was proved to be an NP-complete problem, which is a special case of hypergraph partitioning). The only way to solve this problem is to use heuristic approaches for obtaining suboptimal solutions.

In this paper, we propose a new effective algorithm for multiway circuit partitioning and compare the results obtained by our algorithm with recently developed algorithms [5] by Dasdan and Aykanat on a set of widely used benchmark circuits. The experimental results show that our algorithm significantly outperforms theirs in both solution quality and execution time. The remainder of this paper is organized as follows. In Section 2, we present some definitions and notations for partitioning and describe the formulation of the multiway partitioning problem. Section 3 briefly surveys the related work. We give the motivation for our new algorithm in Section 4. Section 5 formally presents our algorithm. Section 6 presents the experimental justification for our algorithm. Section 7 gives the data structure and the complexity analysis. In Section 8, we report the experimental studies. We conclude in Section 9.

2. Problem Description and Notations

We use a hypergraph $H(C,N)$ to represent a circuit, where $C=\{c_i \mid i=1,2,.....,M_c\}$ (M_c is the number of cells) is the vertex (cell) set; $N=\{n_j \mid j=1,2,.....,M_n\}$ (M_n is the number of nets) is the hyperedge (net) set with each n_j being a subset of C with cardinality $|n_j| \geq 2$.

Definition 1 A *k-way partition* ($k \geq 2$) $\pi=\{B_i \mid i=1,2,.....,k\}$ of a circuit divides the cell set C into k disjoint blocks $B_1, B_2, \ldots, B_k$ such that:

$$B_i \cap B_j = \varnothing \ (i \neq j) \text{ and } \bigcup_{i=1}^{k} B_i = C.$$

Definition 2 A net n_j is said to be *incident* to a cell c_i if $c_i \in n_j$. If a net n_j is incident to a cell c_i, then we say that c_i is *on* n_j. The set of nets incident to c_i is denoted by $N(c_i)=\{n_j \in N \mid c_i \in n_j\}$.

Definition 3 A net n_j is said to be *incident* to a block B_i, if $n_j \cap B_i \neq \varnothing$.

Definition 4 Cells *u* and *v* are neighbors if and only if $N(u) \cap N(v) \neq \varnothing$.

Definition 5 The *degree* $d(c_i)$ of the cell c_i is defined as the number of nets incident to it, $d(c_i)=|N(c_i)|$. The *maximum cell degree* max *d(c)* is defined as $\max_{c_i}[d(c_i)]$. The *average cell degree* ave *d*(*c*) is defined as $\sum_{i=1}^{M_c} \frac{d(c_i)}{M_c}$. The *weight of the cell* c_i is represented by $w(c_i)$.

Definition 6 The *degree* $d(n_j)$ of the net n_j is defined as the number of cells on it, $d(n_j)=|n_j|$. The *maximum net degree* max *d*(*n*) is defined as $\max_{n_j}[d(n_j)]$. The *average net degree* ave *d*(*n*) is defined as $\sum_{j=1}^{M_j} \frac{d(n_j)}{M_j}$. The *weight of the net* n_j is represented by $w(n_j)$.

Definition 7 The *pin* is a connection point of a cell and a net.

The total number M_p of pins for a given circuit is $M_p = \sum_{i=1}^{M_c} d(c_i) = \sum_{j=1}^{M_n} d(n_j)$.

Definition 8 The *density D* of a circuit is defined as:

$$D = \frac{\sum_{j=1}^{M_n} (d(n_j) \cdot [d(n_j)-1])}{M_c (M_c-1)}.$$

Definition 9 If a net n_j is incident to a block B_i and $0<|n_j \cap B_i|<|n_j|$, then the net n_j is a *cut*, i.e., a net n_j is a cut if it is incident to more than one block. A *cutset* is the set of all cut nets. If a net n_j is incident to a block B_i and $|n_j \cap B_i|=|n_j|$, then the net n_j is not a cut.

Definition 10 If a net n_j is incident to h ($h \geq 2$) blocks, there are three cost metrics depending on the cost assigned to the cut net n_j:

It is called "*cost 1" metric* if we assign a cost of 1 to the cut net n_j;

It is called "*cost h - 1" metric* if we assign a cost of h - 1 to the cut net n_j;

It is called "*cost h (h – 1)/2*" metric if we assign a cost of h (h - 1)/2 to the cut net n_j;

Like most hypergraph multiway partitioning algorithms, we will concern ourselves with the "*cost 1*" metric in this paper.

Definition 11 A set $E(B_f)$ of *external nets* of a block B_f is defined as

$$E(B_f) = \{n_j \in N \mid 0<|n_j \cap B_f|<|n_j|\}.$$

Definition 12 A set $I(B_f)$ of *internal nets* of a block B_f is defined as

$$I(B_f) = \{n_j \in N \mid |n_j \cap B_f|=|n_j|\}.$$

Definition 13 $n_j(l)$ is defined as the *number of cells on net* n_j that are in block B_l, i.e., $n_j(l)=|n_j \cap B_l|$.

Definition 14 Given two blocks B_s and B_t, $s \neq t$, for each cell $c_i \in B_s$, its *external cost* $C_i(s, t)$ is defined as $C_i(s, t) = \sum_{n_j \in E_i(s,t)} w(n_j)$,

where $E_i(s, t) = \{n_j \in E(B_s) \mid c_i \in n_j \wedge n_j(t)=|n_j|-1\}$ is the subset of $E(B_s)$ that would be deleted from the cutset if c_i is moved from B_s to B_t. Each cell has (k - 1) external costs, each of which corresponds to a move direction (target block).

Definition 15 The cost $C_i(s, s)$ of a cell c_i in block B_s is called its *internal cost* and is defined as

$$C_i(s, s) = \sum_{n_j \in I_i(s)} w(n_j),$$

where $I_i(s) = \{n_j \in I(B_s) \mid c_i \in n_j\}$ is the subset of $I(B_s)$ that would be added to the cutset if c_i is moved from B_s to any other block. Each cell only has one internal cost.

Definition 16 The *cell gain* $g_i(s,t)$ of the move of cell c_i from its source block B_s to its target block B_t is defined as $g_i(s,t) = C_i(s,t) - C_i(s,s)$.

Definition 17 The *cutsize* of a k-way partition $\pi = \{B_i \mid i = 1, 2, ..., k\}$ is equal to the sum of the weights of all cuts.

$$\text{cutsize}(\pi) = \sum_{j=1}^{M_n} w(n_j) - \sum_{q=1}^{k} \sum_{n_j \in I(B_q)} w(n_j).$$

Definition 18 The *weight* $w(B_i)$ *of a block* B_i is defined as $w(B_i) = \sum[w(c_i) \mid c_i \in B_i]$.

The *weight* $w(H)$ *of a circuit* H is defined as $w(H) = \sum_{i=1}^{M_c} w(c_i)$.

Definition 19 Given a k-way partition $\pi = \{B_i \mid i = 1, 2, ..., k\}$, π is said to be *balanced* if for each B_i $(i = 1, 2, ..., k)$, the following balancing constraint is satisfied:

$$\left\lceil \frac{w(H)}{k} \cdot (1-t) \right\rceil \le w(B_i) \le \left\lfloor \frac{w(H)}{k} \cdot (1+t) \right\rfloor, \quad 0 < t < 1,$$

t is called the *balance tolerance*.

Definition 20 The *k-way partitioning* problem is to find a balanced partition $\pi = \{B_i \mid i = 1, 2, ..., k\}$ such that the cutsize(π) is minimized. The k-way partitioning is called *bipartitioning* for $k = 2$, and *multiway partitioning* for $k > 2$.

3. Related Work

In 1970, Kernighan and Lin (KL) [12] proposed a well-known heuristic for the two-way graph partitioning algorithm which has become the basis for most of the subsequent partitioning algorithms in this area. The algorithm is called an iterative improvement algorithm because it is based on the cell moves to improve the solution iteratively until a local minimum is obtained. The KL starts with a balanced two-way partition, and it performs a number of passes until a local minimum of cutsize is found. A pass consists of a number of pairwise cell swappings between the two blocks. Schweikert and Kernighan proposed a more practical model, namely the the hypergraph model, for the circuit partitioning problem [17].

Fiduccia and Mattheyses (FM) [7] presented a modified version of algorithm KL to speed up the search. They introduced a new data structure (bucket list of cell gains) to achieve the linear run time per pass. They also proposed a cell move instead of swapping a pair of cells in one move. This allows for more flexibility in selecting candidate cell moves.

Krishnamurthy (KR) [13] suggested that the lack of an "intelligent" tie-breaking scheme among many possible cell moves with the same gain could cause the algorithm FM to make "bad" choices. He enhanced the algorithm FM with a look ahead scheme that looks ahead up to r^{th} level of cell gains to choose a cell move.

Sanchis (SA) [18] extended the algorithm FM with Krishnamurthy's look ahead scheme to multiway partitioning. Sanchis's algorithm is the first hypergraph multiway partitioning algorithm since all previous algorithms are for two-way partitioning. SA is extensively used as a benchmark in performance comparison for different multiway hypergraph partitioning algorithms. All the FM-based partitioning algorithms, such as KL, FM, KR and SA, are iterative improvement algorithms. They dominate both VLSI CAD research community and industry practice for several reasons. They are generally intuitive (the obvious way to improve a given solution is to repeatedly make it better via cell moves), flexible in adapting to different optimization objectives, easy to implement, and relatively fast.

Park and Park [15] pointed out that the cell move operation is largely influenced by the balancing constraint, and they proposed a cost function that comprises both the cutsize and the balance degree (that is the sum of all size differences for every pair of different blocks) with a positive weighting factor. They proved that a minimum cost multiway partitioning obtained by their algorithm corresponds to a balanced minimum cutsize as defined in SA if the weighting factor is larger than the number of cells in a circuit. The SA is then used to solve the multiway partition problem under this objective function.

Dutt and Deng [6] pointed out that the FM-based iterative improvement algorithms could only remove small clusters from the cutset while it likely locks bigger clusters in the cutset. They divided the cell gain into initial gain calculated before a cell movement and the update gain generated from the cell movement afterwards. By focusing on the update gain when choosing cells to move, they reported very successful results for bipartitioning experiments.

Cong et al. [3] proposed the concept of loose nets and stable nets. They focused on the status of nets instead of cells as often emphasized in the traditional algorithms. Their algorithm was developed for bipartitioning and the experimental results are also encouraging.

Karypis et al. [11] proposed a partitioning algorithm (hMETIS) based on the multilevel paradigm. A sequence of successively coarser hypergraphs is constructed. A bipartioning of the smallest hypergraph is performed and it is used to generate a bipartitioning of the original hypergraph by successively projecting and refining the bipartitioning to the next level of finer hypergraphs.

Ouyang et al. [16] introduced a cooperative multilevel hypergraph partitioning algorithm. The experimental results show that their algorithm outperforms hMETIS (a reduction of 3% to 15% in terms of the cutsize for 4-way and 8-way partitioning).

Recently, Dasdan and Aykanat [5] (DA) developed two multiway partitioning algorithms using a relaxed locking mechanism. The first one (PLM) uses the locking mechanism in a relaxed manner. It allows multiple moves for each cell in a pass by introducing the phase concept so that each pass may contain more than one phase, and each cell has a chance to be moved only once in each phase. The second algorithm (PFM) does not use the locking mechanism at all. A cell can be moved as many times per pass as possible based on its mobility value. The performance of the proposed algorithms was experimentally evaluated in comparison with the Simulated Annealing algorithm (SAA) and SA on some common benchmark circuits. Their results outperform SA significantly on multiway partitioning and their performance is comparable to that of SAA with much less run time.

Cong and Lim [4] proposed a multiway partitioning algorithm with pairwise cell movements. It starts with an initial multiway partition, and then applies the bipartitioning heuristic (FM) to pairs of blocks concurrently to improve the quality of the overall multiway partitioning solution.

Yang and Wong [20] presented a network flow based partitioning algorithm to solve bipartitioning problem and they claimed that multiway partitioning can be accomplished by recursively applying the network flow based algorithm.

Yeh et al. [21] proposed a primal-dual multiway partitioning algorithm that alternates "primal" passes of cell moves with "dual" passes of net moves. Hauck and Borriello [10] concluded that dual passes "are not worthwhile".

The existing multiway partitioning can be classified into two primary approaches: recursive and direct. The recursive approach [12] applies bipartitioning recursively until the desired number of partitions is obtained; the direct approach partitions the circuit directly. Among all the previous work mentioned above, the DA, the primal–dual, and the pairwise movement algorithms are direct multiway partitioning algorithms. The recursive approach is computationally simple and fast. However, it suffers from the following three major limitations. First, if the number k of partitions is not a power of 2, we cannot obtain the desired multiway k partitioning by using the bipartitioning recursively. Second, it becomes harder and harder to reduce the cutsize since cut nets in early stages cannot be removed when the bipartitioning performs on finer graphs. For instance, a highly optimized cutset at one stage may cause the following stage to work on dense blocks. Those dense blocks cause negative effects on the solution while applying further bipartitioning on them. Third, the recursive partitioning actually aims at minimization of cost $(k - 1)$ metric, not cost 1 metric that is often the objective to be minimized.

Tie-breaking strategies play an important role in circuit partitioning. Even when gain vectors are used, ties may still occur among the cell gains. Hagen et al. [9] observed that Sanchis [18], and most likely Krishnamurthy [13], used random selection schemes. They [9] found that the LIFO (Last-In-First-Out) bucket organization is distinctly superior to FIFO (First-In-First-Out) and random bucket organizations. They attribute the success of LIFO to its enforcement of "locality" in the choice of cells to move, i.e., cells that are naturally clustered together will tend to move sequentially.

All FM-based iterative improvement algorithms are started with a random initial solution. Each cell has $(k–1)$ cell gains. A cell with maximum move gain in a particular moving direction is chosen from all possible movements that will not violate the balance constraint. The selected cell is then moved to the target block and is locked. The cell gains of all the affected neighbors are updated accordingly. The next cell is chosen in the same way from all the remaining free (unlocked) cells and is moved to its target block. The cell move process is repeated until all the cells are locked or there are no legal moves available due to the balance constraint. Assume that there are q $(q \le M_c)$ cell moves all together. Then all the partial gain sums $S_p = \sum_{x=1}^{p} g_x(a_x, b_x)$, $p = 1, 2, ..., q$, where $g_x(a_x, b_x)$ is the gain of moving the x^{th} cell from block a_x to block b_x given that the first $(x - 1)$ cell moves have already been made, are calculated and the maximum partial gain sum S_β is chosen. This corresponds to a point of minimum cutsize in the entire moving process. All the cells moved after the β^{th} cell move is reversed to the original blocks so that the actually moved cells are the sequence of moving first cell from block a_1 to block b_1, second cell from a_2 to b_2 ,..., the β^{th} cell from a_β to b_β , where $a_1, a_2, ..., a_\beta, b_1, b_2, ..., b_\beta \in \{1, 2, ..., k\}$, k is the number of blocks. The whole process is called a pass. A number of passes, which is called a run, is performed until the maximum partial gain sum is no longer positive. Then we say that the local optimum with respect to the initial solution is obtained.

4. Motivation

For several reasons, the solution quality produced by FM-based iterative improvement algorithms is often poor, especially for larger circuits. First, it has been criticized for its well-known shortsightedness as its way of choosing a cell to move. It is only based on local information (cell gain) of the immediate decrease

in cutsize. For example, it may be better to move a cell with smaller gain, as it may allow many good moves later. Thus it tends to be trapped in local optimum, which strongly depends on the initial solution. Second, many cells have the same cell gain, especially for larger circuits. The FM-based iterative improvement approach has no insightful scheme to choose which of these cells to move. Only critical net information is used for the cell gain calculation. We call a net a critical net if its cutstate (that indicates whether the net is cut or not) will be changed immediately after a move of that related cell. After taking a close look at a particular net, we find that if one cell on the net is locked in a block, the only way to remove the net from the cutset is to pull all other cells to the block where the locked cell has already been moved in. If two or more cells of a net are locked in two or more blocks, then there is no way to remove the net from the cutset in the current pass. Due to the missing of dynamic net information, bigger clusters are very likely to be locked in the cutset. Third, the cell gain ranges from $-\max_{c_i}[d(c_i)]$ to $\max_{c_i}[d(c_i)]$. The bucket structure consists of an array of $\{2 \cdot \max_{c_i}[d(c_i)] + 1\}$ entries to which cells with the same cell gain are linked. Since the entire $M_c(k-1)$ cell gains are distributed in this short range, many cells may have the same cell gain; a better tie-breaking scheme like the look-ahead capability proposed by Krishnamurthy [13] is needed. Fourth, even though the uphill moves, which are always the best one from among all legal moves, are accepted in each pass, the possibility for exploring broader solution space and finding better solution is limited as the result of lacking a better strategy to escape from a local optimum.

To enhance the solution quality for the iterative improvement algorithms, we must try to overcome the weak points mentioned above. This is the motive for developing our new algorithm.

5. Proposed Algorithm

Basically, the structure of our algorithm is similar to that of other FM-based iterative algorithms. The whole algorithm consists of a user specified number of runs. Each run comprises a number of passes that cannot be known in advance. We use the conventional locking mechanism in our algorithm. That is, each cell can only move at most once in each pass. Each pass has at most M_c iterations (cell moves). A local optimum is obtained at the end of each run. The algorithm outputs the best one from all the local optima at the end.

Before getting into the details of our algorithm, we need some definitions for the convenience of description. A net is called a *free net* if all cells incident to it are unlocked. If all the locked cells incident to a net are distributed in a single block, the net is called a *loose net*. In this case, the block in which the locked cells are located is called the *locked block* for that net, and all the other blocks in which the free cells are located are called the *free blocks* for that net. If the locked cells are distributed in two or more blocks, the net is called a *locked net*.

In an attempt to remove big clusters from the cutset, more dynamic net information is needed. Here we introduce the concept of *net gain* for each loose net. If a cell is moved to a block and locked there, we use net gain values to encourage its neighboring cells to be moved subsequently to the locked block where the moved cell is just locked in. The net that straddles the cut line is thus removed. Unlike other FM-based iterative improvement algorithms in which the selection of the next cell to move is only based on its cell gain, our algorithm selects a cell based on both its cell gain and the sum of all net gains for those loose nets incident to the cell.

For each net, there is a net gain array (called net_gain) of size k associated with it. Initially, all nets are free and all the k elements are set to zero for each array. After a cell c is moved to a block B_L, it is then locked during the current pass. For each loose net, there is exactly one locked block and the number of

free blocks is between 1 and $(k-1)$. The elements of a net_gain array are only defined for loose nets since it makes no sense to have it for either locked nets or free nets.

As mentioned earlier, we use the values in the net_gain array to encourage the free cells in free blocks of a loose net to move to the locked block of the net. The *net gain* of a loose net n_α for one of its free blocks B_F is defined as follows:

$$\text{net\_gain}[n_\alpha][B_F] = \left\lceil \frac{\max_{n_j}[d(n_j)]}{d(n_\alpha)} \cdot \frac{\sum_{c \in S_L}[1 + d(c) - cut(c)]}{\sum_{c \in S_F}[1 + d(c) - cut(c)]} \right\rceil ,$$

where

$\max_{n_j}[d(n_j)]$ is the maximum net degree in the given circuit;

$d(n_\alpha)$ is the degree of the net n_α;

S_L is the set of locked cells of net n_α in its locked block B_L;

S_F is the set of free cells of net n_α in its free block B_F;

$d(c)$ is the number of nets incident to cell c;

cut (c) is the number of nets incident to cell c that are in cutset.

Basically, each element of a net gain array is a product of two fractional expressions:

$$\frac{\max_{n_j}[d(n_j)]}{d(n_\alpha)} \quad (1), \quad \text{and} \quad \frac{\sum_{c \in S_L}[1 + d(c) - cut(c)]}{\sum_{c \in S_F}[1 + d(c) - cut(c)]} \quad (2).$$

The purpose of expression (1) is to give smaller cut nets higher chance to move. The smaller the net degree of n_α is, the bigger the value of expression (1) is. Expression (2) indicates that the more locked cells the net n_α has in its locked block B_L, or the fewer cells the net n_α has in its free block B_F, the higher value net_gain[n_α][B_F] has. It also indicates that for locked cells with more internal nets (which are not in the cutset) and for free cells with fewer internal nets, the net_gain[n_α][B_F] gets higher value. This approach is more appropriate than that used by [3] for the bipartitioning problem since [3] does not distinguish the internal nets from the external nets, and it does not have a mechanism to remove this gain once the loose net becomes locked.

All the free cells of the loose net n_α in the free block B_F have the same net_gain[n_α][B_F]. The net_gain[n_α][B_F] encourages all the free cells currently in the free block B_F to move to the locked block B_L for ultimately removing the loose net n_α from the cutset. The selection of cell movement in our algorithm is based on the move gains of each cell. For each cell, there is a move gain array of size k (called move_gain) associated with it. Also, we use an array *net_cell_gain* for each cell that has the same structure as move_gain array. The net_cell_gain[c][B_L] has the value

$$\sum_{n_\alpha \in S_c} \text{net\_gain}[n_\alpha][B_F]$$

where S_c is the set of all loose nets that are incident to free cell c and share the same locked block. For each free cell c of the loose net n_α in the free block B_F, the *move gain* of cell c to the locked block B_L is defined as:

$$\text{move\_gain}[c][B_L] = \text{cell\_gain}[c][B_L] + \text{net\_cell\_gain}[c][B_L]$$

where $\text{cell\_gain}[c][B_L]$ is calculated as in the conventional FM-based algorithms (defined in Definition 16).

Unlike the cell_gain, each element of net_cell_gain is always positive. Originally, the range for elements in cell_gain array is from $-\max_{c_i}[d(c_i)]$ to $+\max_{c_i}[d(c_i)]$. Due to the introduction of net_cell_gain, the extended range for elements in a move_gain array reduces the opportunity of many cells having the same gain value and makes the tie-breaking strategy such as look ahead unnecessary.

It should be mentioned that in the process of calculating cell gain values, only the information of critical nets is used. We use the net gain concept to dynamically check the status of each net and to make the selection of the next cell to move more effective. The status of each net in the cutset is changed as follows: free → loose → locked, or free → loose → disappear. Once a cell is moved, the status of the nets incident to it should be updated accordingly. If a net becomes locked, all elements of its net_gain array are set to zero immediately to avoid making wrong decision for later cell selection.

In order to reduce the computational effort without significant degradation of the solution quality, we smoothly decrease the number of iterations from pass to pass by a fractional factor r. The following experimental justification shows that most of the maximum partial gain sums are at the first half of the array of partial gain sums of each pass; and, with the evolution from one pass to the next pass, the maximum partial gain sum is gradually moved to the start part of the array.

Additionally, to escape from being trapped in a local optimal solution, and to try to explore broader solution space, we perturb the current solution by the following scheme. Once a run is terminated, we find the cut nets that are included in both the initial solution and the final solution of a run. These cut nets may be the obstacles for the solution to escape from local optimum. We randomly force a certain percentage amount of these cut nets to be removed from the current cutset by moving them into the current smallest block if the balance constraint can be satisfied. The current solution becomes the start point for the next run to explore the new solution space. We will present the experimental justification to show the advantages of using the perturbation mechanism in improving the solution quality.

We call our algorithm NGSP (Net Gain Solution Perturbation).

NGSP Algorithm

Input :	*NumOfRun*:	positive integer, used to define the number of runs;
	r :	floating-point number, $0 < r \leq 1$, used to decrease the number of moves from pass to pass;
	p:	floating-point number, $0 < p \leq 1$, used to choose a percentage of nets in the perturbation function;

function **main_function()**:

Generate a random initial *k*-way partitioning as the initial solution;
finalCutsize = *localCutsize* = *currentCutsize*;
count = 0;
ratio=1;
start /* for each pass */
Compute cell_gain for each cell and initialize all cells as unlocked;
Initialize net_cell_gain and move_gain;
Build bucket list;
repeat
Choose a legal cell *c* with maximum move gain;
Make the cell move tentatively and lock it;
Update the cell_gain arrays and move_gain arrays of all affected cells;
Compute partial gain sum;
update_net_gain(*c*);
until the body has repeated $M_c \cdot ratio$ times or there are no legal moves available
Find the maximum partial gain sum S_β;
if ($S_\beta > 0$)
Make the first β cell moves permanent;
currentCutsize = *currentCutsize* - S_β;
localCutsize = *currentCutsize* ;
$ratio = ratio \cdot r$;
go to **start;**
else
if (*finalCutsize* > *localCutsize*)
finalCutsize = *localCutsize*;
count = *count* + 1;
if *count* > *NumOfRun*
output the *finalCutsize* and corresponding solution, **stop;**
else
new_explore(*p*);
go to **start;**

---

function **update_net_gain(** cell *c*)
/* updating net_gain arrays, net_cell_gain arrays and move_gain arrays*/
for each net n_α incident to the moved cell *c*
net_gain[n_α][1] = net_gain[n_α][2] = ……= net_gain[n_α][*k*] = 0
if net n_α is a loose net
for each free block B_F of n_α containing some free cells of n_α
Calculate net_gain[n_α][B_F];
for each free cell *f* of net n_α in block B_F
net_cell_gain[*f*][B_L] = net_cell_gain[*f*][B_L] + net_gain[n_α][B_F];

move_gain[f][B_L] = move_gain[f][B_L] + net_cell_gain[f][B_L];
Update bucket lists;

---

function **new_explore**(float p)
/* It is used to perturb current local optimum solution for exploring new solution space. The perturbed solution is used as an initial solution for next run. */
commonNets = (cut nets in initial solution for the run) $\cap$ (cut nets in current solution);
h = size of *commonNets*;
if $h = 0$
Output the *finalCutsize* and corresponding solution;
Stop.
else for ($j = 0$; $j \le p \cdot h$; j++) /* $0 < p \le 1$. */
Randomly take a net from *commonNets* and move all cells incident to it to the smallest block if the balance constraint is satisfied;

---

6. Experimental Justification for the Proposed Algorithm

We have conducted the following experiments on many benchmark circuits for different values of partition number k to provide experimental justification for our algorithm.

6.1 Decreasing the number of iterations from pass to pass to reduce the run time

As stated in the previous section, we decrease the number of iterations from pass to pass by a fractional factor r to reduce the overall run time needed by our algorithm. Following experiments find where (at which moves) the maximum partial gain sum occurs for each pass. In these experiments, we take $r = 1$ (i.e., the maximum number of moves in each pass is equal to the number of cells) to see the exact number of moves before the maximum partial gain sums occur. The "move no." in Tables 1 through 3 is bound from above by the number of cells.

The experimental results for different values of k on three benchmark circuits (defined later) are as follows. We use MPGS to denote the maximum partial gain sum.

Table 1
test06 (number of cells = 1752) for 10-way partitioning

pass no.	1	2	3	4	5	6	7	8	9	10	11	12	13
MPGS	773	267	123	66	2	6	15	4	33	3	10	7	1
move no.	1294	1215	1116	1086	1	39	14	914	157	55	21	20	1
pass no.	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
MPGS	1	1	1	3	6	1	4	10	1	1	2	2	1
move no.	1	1	1	60	41	2	26	71	10	2	21	10	4

Table 2
primary2 (number of cells = 3014) for 7-way partitioning

pass no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
MPGS	1608	393	50	36	6	19	6	27	9	1	1	1	1	1	1	1	1
move no.	2412	2293	1859	125	24	1665	14	1351	27	1	1	1	1	1	1	1	1
pass no.	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**
MPGS	3	5	24	1	4	13	4	9	1	12	1	2	1	1	1	1	1
move no.	15	44	81	5	240	97	60	8	6	94	4	2	1	1	1	17	3

Table 3
avq_large* (number of cells = 25178) for 5-way partitioning

pass no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
MPGS	16861	1042	1365	186	18	21	73	236	11	42	4	9	5	2	33	4	6
move no.	20104	19109	17221	15693	82	50	14850	15708	60	244	37	23	5	1	137	4	5
pass no.	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**
MPGS	6	2	5	10	2	2	4	1	2	7	3	5	3	1	1	37	4
move no.	7	18	2	7	3	1	2	1	2	13	4	13	2	1	1	5673	3
pass no.	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**
MPGS	15	12	2	7	3	3	2	11	6	9	4	4	13	1	8	7	2
move no.	61	41	2	4	3	27	17	12	20	43	2	9	64	1	79	25	27

* The total number of passes is 203. For the remainder 152 passes, the move number is no more than 104 and the maximum partial gain sum is less than 22. For saving space, we omit the details.

The experimental results show that for 81.8% (test06), 89.3% (primary2), 97.1% (avq_large), 98.7% (golem3, table not included due to the space limitation) of the passes in a run, the maximum partial gain sums occur in the first half of cell moves. With the increase of the number of cells in the circuits, the chance is significantly increased for the maximum partial gain sum to occur in the first half of the cell moves. With the evolution from pass to pass, the location of the maximum partial gain sum tends to be smaller and closer to the initial part of the cell moves. This is the reason why we introduce a parameter r to reduce the number of iterations from pass to pass. This strategy makes the algorithm very effective for solving large circuit partitioning problems.

6.2 Improving the solution quality with the perturbation mechanism

To show the evidence of improvement in the solution quality by using the perturbation mechanism, we execute our algorithm s times (s initial solutions) with the perturbation mechanism, with p runs for each execution; then we execute the algorithm $s \cdot p$ times ($s \cdot p$ initial solutions) without the perturbation mechanism. We take the average cutsize over s solutions for the first case, and the average cutsize over $s \cdot p$ solutions for the second case. The

experimental results on benchmark circuits struct (k = 10), primary2 (k = 10), biomed (k = 7), and avq_large (k = 5) are shown in Tables 4 through 7 respectively.

Table 4

struct for 10-way partitioning

EXPERIMENTAL CASE	(1) With perturbation , each execution includes 10 runs	(2) Without perturbation
EXECUTION TIMES	10	100
AVERAGE CUTSIZE	156.8	183.5
TOTAL RUN TIME (sec.)	86	137
IMPROVEMENT of (1) over (2) in cutsize (%)	14.6	

Table 5

primary2 for 10-way partitioning

EXPERIMENTAL CASE	(1) With perturbation , each execution includes 10 runs	(2) Without perturbation
EXECUTION TIMES	10	100
AVERAGE CUTSIZE	575.4	621.7
TOTAL RUN TIME (sec.)	112	141
IMPROVEMENT of (1) over (2) in cutsize (%)	7.4	

Table 6

biomed for 7-way partitioning

EXPERIMENTAL CASE	(1) With perturbation , each execution includes 8 runs	(2) Without perturbation
EXECUTION TIMES	10	80
AVERAGE CUTSIZE	335.7	363.9
TOTAL RUN TIME (SEC.)	401	467
IMPROVEMENT of (1) over (2) in cutsize (%)	7.7	

Table 7

avq_large for 5-way partitioning

EXPERIMENTAL CASE	(1) With perturbation , each execution includes 5 runs	(2) Without perturbation
EXECUTION TIMES	10	50
AVERAGE CUTSIZE	889.8	999.9
TOTAL RUN TIME (sec.)	4386	4633
IMPROVEMENT of (1) over (2) in cutsize (%)	11	

We conclude from Tables 4 through 7 that the improvement of average cutsize for the algorithm with perturbation over that without perturbation is from 7.4% to 14.6% with almost the same run time. Therefore, the perturbation mechanism is necessary for improving the performance of our algorithm.

6.3 A convincing example

The following simple example shows how the concept of net gain is applied, and the advantage of embedding it in the selection of cell moves in our algorithm. The simple circuit comprises 6 nets with 15 cells. We assume that all the cell weights and all the net weights have the value of 1. We solve the k-way partition problem with $k = 3$ here. The number of cells in each block is limited to the range of 4 to 6 due to the balance constraint.

Figure 1 shows the initial solution with the cutsize of 5. At the beginning, all the values for net_gain arrays and net_cell_gain arrays are set to 0, and each move_gain value is set to the same as its corresponding cell_gain value. As defined previously, we select a candidate cell for moving based on its move_gain value. The 2-dimentional move_gain array can be represented by the following matrix (3), where each row corresponds to a cell and each column corresponds to a block:

$$\text{move\_gain}[15][3] = \begin{array}{c|ccc} 1 & - & 0 & 0 \\ 2 & - & 0 & 0 \\ 3 & - & 0 & 0 \\ 4 & - & 0 & 0 \\ 5 & - & 1 & 0 \\ 6 & 0 & - & -1 \\ 7 & 1 & - & 0 \\ 8 & 0 & - & 1 \\ 9 & -1 & - & -1 \\ 10 & 0 & - & 0 \\ 11 & 0 & 0 & - \\ 12 & 0 & 0 & - \\ 13 & 0 & 0 & - \\ 14 & 0 & 0 & - \\ 15 & 0 & 0 & - \end{array} \quad (3)$$

$$\text{move\_gain}[15][3] = \begin{array}{c|ccc} 1 & - & 0 & 0 \\ 2 & - & 0 & 0 \\ 3 & - & 0 & 0 \\ 4 & - & 0 & 0 \\ 5 & - & - & - \\ 6 & -2 & - & -2 \\ 7 & 1 & - & 0 \\ 8 & 0 & - & 1 \\ 9 & -1 & - & -1 \\ 10 & 0 & - & 0 \\ 11 & 0 & 1 & - \\ 12 & 0 & 1 & - \\ 13 & 0 & 0 & - \\ 14 & 0 & 0 & - \\ 15 & 0 & 0 & - \end{array} \quad (4)$$

In this case, cell 5, which has the highest move_gain value, is moved from block 1 to block 2. Then, it is locked in block 2 for the current pass. This movement removes net 2 from the cutset, and hence reduces the cutsize from 5 to 4. Net 3 is the only other net incident to the moved cell 5 and becomes a loose net. Based on the definition of the loose net, block 2 is the locked block for net 3, and block 3 is the free block for net 3. The net_gain value of net 3 for free block 3 is thus updated as follows:

$$\text{net\_gain}[3][3] = \left\lceil \frac{4}{4} \cdot \frac{(1+2-1)}{(1+1-1)+(1+2-2)} \right\rceil = 1.$$

Moreover, all the net_cell_gain values and move_gain values corresponding to the free cells (cell 11 and cell 12) on net 3 for free block 3 are updated according to the algorithm. The updated move_gain matrix is shown in (4).

At this stage, neither cell 11 nor cell 12 can be moved due to the balance constraint. The algorithm moves cell 7 from block 2 to block 1 and locks cell 7 in block 1. The cutsize is then reduced to 3. Now, net 1 becomes a loose net. Since all free cells of net 1 are in locked block 1, no further updating for this movement is needed. The next step is to move cell 8 to block 3. The cutsize is further reduced to 2.

Without embedding the concept of net gain, as in the conventional SA algorithm, the cell_gain values for all the free cells (1, 2, 10, 11, 12, 13) have the same value of 0, and they are all legal moves (balance constraints are satisfied). Using the conventional approach, a cell is randomly selected for this situation. In the case of moving cell 10 from block 2 to block 1, the cutsize cannot be reduced further more.

Having the net_gain values as part of a cell's move_gain values, we obtain the move_gain matrix shown in (5). Both cells 11 and 12 have the highest move_gain value of 1. We choose cell 11 to move to block 2 and lock it. (It is easy to see later that either choice will lead to the same final solution.) Net 3 is still a loose net. Currently, there are two locked cells 5 and 11 in locked block 2 and one free cell 12 in free block 3 for this loose net. Now, the net_gain[3][3] is $\left\lceil \frac{4}{4} \cdot \frac{(1+2-1)+(1+1-1)}{(1+2-2)} \right\rceil = 3$. The new move_gain matrix is shown in (6).

$$\text{move\_gain}[15][3] = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{array} \begin{bmatrix} - & 0 & 0 \\ - & 0 & 0 \\ - & -1 & -1 \\ - & -1 & -1 \\ - & - & - \\ -2 & - & -2 \\ - & - & - \\ - & - & - \\ -1 & - & -1 \\ 0 & - & 0 \\ 0 & 1 & - \\ 0 & 1 & - \\ 0 & 0 & - \\ -1 & -1 & - \\ -1 & -1 & - \end{bmatrix} \quad (5)$$

$$\text{move\_gain}[15][3] = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{array} \begin{bmatrix} - & 0 & 0 \\ - & 0 & 0 \\ - & -1 & -1 \\ - & -1 & -1 \\ - & - & - \\ -2 & - & -2 \\ - & - & - \\ - & - & - \\ -1 & - & -1 \\ 0 & - & 0 \\ - & - & - \\ 0 & 4 & - \\ 0 & 0 & - \\ -1 & -1 & - \\ -1 & -1 & - \end{bmatrix} \quad (6)$$

After moving cell 12 from block 3 to block 2, the cutsize is reduced to 1. The optimal solution is obtained as shown in Figure 2.

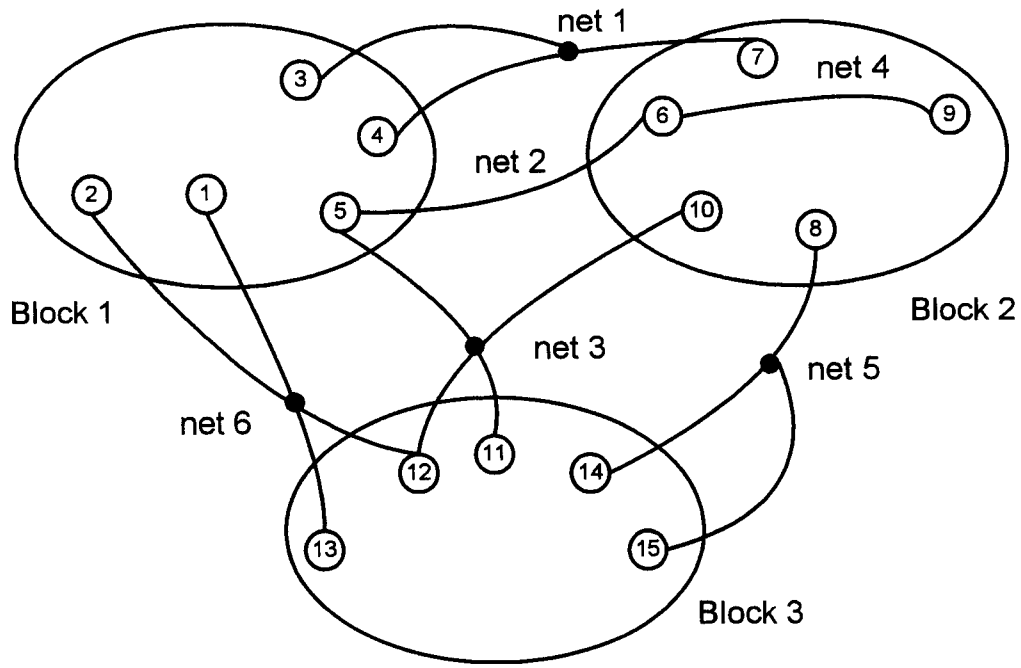

Figure 1

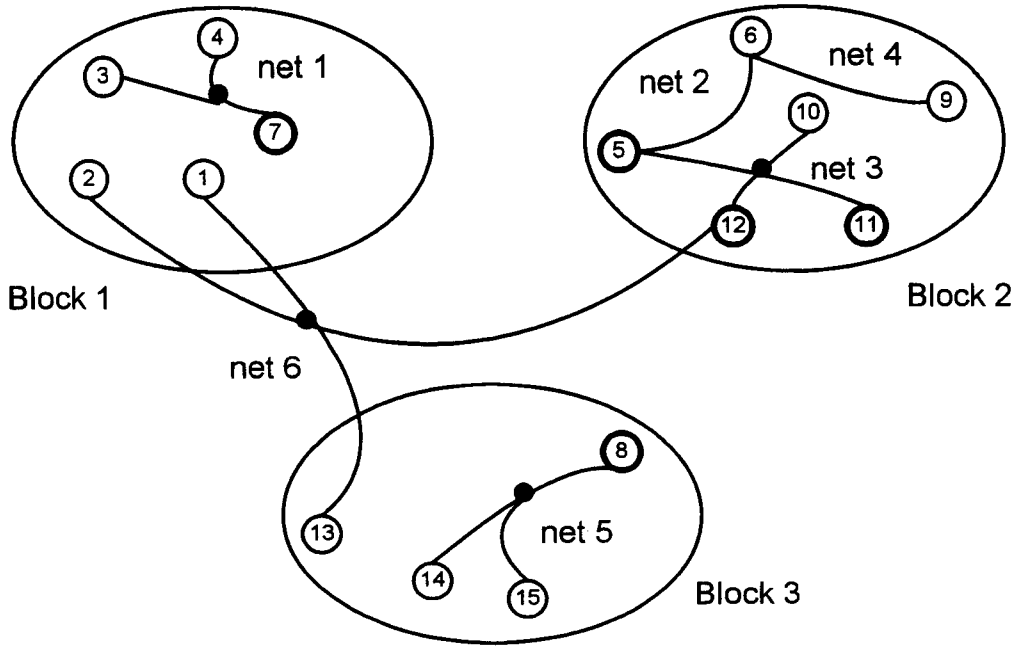

Figure 2

○ Unlocked Cell ○ Locked Cell

7. Data Structure and Complexity Analysis

The bucket data structure [5] is used in our algorithm. Basically, a bucket data structure has an array of pointers and each of these pointers points to a doubly-linked list of nodes. An upper bound of the bucket array size is set to a user-defined parameter u ($u \geq \max_{c_i} [d(c_i)]$) to avoid having extremely huge bucket size. The size of a bucket array is therefore ($\max_{c_i} [d(c_i)] + u + 1$). There are $k(k\text{-}1)$ such bucket arrays defined and each corresponds to a moving direction (from a source block to a target block). Based on the experimental results from [9], we also employ the LIFO structure for the doubly-linked list. Single insertion or deletion can be done in constant time on the bucket list while finding a node among $k(k\text{-}1)$ bucket lists need $O(k^2)$ time.

As other FM-based approaches, we need to compute the cell gains and initialize the move gains for each initial solution. This takes $O(M_p k)$ time. Building the bucket list can be done in $\Theta(k^2 u)$ time. One insertion can be done in constant time on the bucket array lists. Therefore, inserting all cell nodes into the bucket arrays of lists takes $O(M_c k)$ time. Since there are $k(k\text{-}1)$ different possible moves, selecting a legal candidate needs $\Theta(k^2)$ time. By adapting the locking mechanism, the move gain updating procedure after each cell move needs $O(\frac{M_p k u}{M_c})$. The update_net_gain function can be done in $O(k \cdot \max_{c_i}[d(c_i)] \cdot \max_{n_j}[d(n_j)])$.

Therefore, the repeat loop takes $O(M_c k^2 + M_p k u + M_c k \max_{c_i} [d(c_i)] \cdot \max_{n_j} [d(n_j)])$. Since a program structure is employed inside the repeat loop tracking the subsequence of the maximum partial gain sum, finding the maximum partial gain sum can be done in constant time. Overall, our NGSP algorithm has the complexity of

$$O(k^2 u + M_c k^2 + M_p k u + M_c k \max_{c_i} [d(c_i)] \cdot \max_{n_j} [d(n_j)]) \text{ per pass.}$$

The total number of passes for one run cannot be known in advance though the maximum number of passes recorded in our experiments is around 300.

8. Experimental Studies

This section presents the details of the experimental framework and lists the experimental results. We conduct the experimental studies for our NGSP algorithm in comparison with SA, DA (including six versions: PLM 1, 2, 3 and PFM 1, 2, 3) algorithms. Like [5], the level parameter of SA is set to one. We chose PLM3 and PFM3 among DA algorithms for comparison since they produced better results than other versions of PLM and PFM respectively in [5]. The performance comparisons have been done on seven widely used ACM/SIGDA benchmark circuits. All algorithms are implemented using the C++ programming language and all experiments are done on a 433MHz Pentium Celeron based Windows NT 4 workstation with 128M physical memory. To make a fair comparison, all algorithms are performed with a random initial solution and the same balance criterion among all blocks (balance tolerance of 0.1). All weights for cells and nets are set to 1.

8. 1 Benchmark circuits

Table 8 shows seven benchmark circuits that are used for performance comparisons. Among these circuits, the number of cells ranges from 1752 to 103048 and the density of circuit ranges from 0.00449 to 0.07983.

Table 8

The characteristics of the benchmark circuits used for performance comparison

Benchmark Circuit	M_c	M_n	M_p	ave.$d(c)$	ave.$d(n)$	max.$d(c)$	max.$d(n)$	D
test06	1752	1641	6638	3.79	4.05	6	388	0.079830
struct	1888	1888	5375	2.85	2.85	4	16	0.004490
primary2	3014	3029	11219	3.72	3.70	9	37	0.008204
biomed	6417	5711	20912	3.26	3.66	6	860	0.062038
industry2	12142	12949	47193	3.89	3.64	12	584	0.011770
avg_large	25678	25384	82751	3.29	3.26	7	4042	N/A
golem3	103048	144949	338419	3.28	2.33	22	39	N/A

8. 2 Performance comparisons

The results obtained by SA, PLM3, PFM3, and NGSP on seven benchmark circuits for four partitions (k = 2, 5, 7, 10) are shown in Table 9 (for the average cutsize and the standard deviation) and Table 10 (for the best cutsize and the worst cutsize) respectively. The bold values in each row are the best ones. For benchmark circuits test06, struct, primary2, and biomed, the average is over 50 runs; for industry2 and avg_large, the average is over 20 runs; for large circuit golem3, the average is over 10 runs.

Table 9

The average cutsize (the standard deviation) for four algorithms

Benchmark Circuits	k	SA	PLM3	PFM3	NGSP
test06	2	82.7 (10.6)	79.2 **(6.2)**	89.1 (9.4)	**67.7** (7.0)
	5	298.2 (24.8)	207.2 (19.6)	158.4 (20.0)	**133.0, (8.7)**
	7	336.2 (20.7)	240.2 (22.2)	183.8 (25.7)	**139.74 (9.01)**
	10	375.5 (16.9)	264.4 (19.0)	210.7 (28.2)	**177.06 (12.59)**
struct	2	51.3 (4.3)	49.9 (3.7)	50.6 (11.2)	**34.9 (2.9)**
	5	311.3 (27.3)	209.2 (23.9)	151.1 (27.7)	**91.2 (5.7)**
	7	400.4 (30.1)	309.0 (31.7)	235.1 (33.4)	**116.4 (7.0)**
	10	503.4 (30.0)	421.7 (26.0)	339.7 (33.6)	**155.3 (11.1)**
primary2	2	272.2 (40.0)	257.7 (44.6)	240.8 (28.5)	**159.0 (22.4)**
	5	874.4 (27.0)	738.7 (30.5)	512.7 (24.0)	**430.4 (18.4)**
	7	952 (24.2)	829.5 (26.3)	615.9 (29.3)	**500.74 (17.22)**
	10	1028.9 (21.3)	876.4 (23.9)	746.5 (23.3)	**584.2 (13.7)**
biomed	2	128.8 (47.0)	174.6 (17.6)	192.8 (25.3)	**87.7 (4.6)**
	5	714.4 (56.5)	573.4 (36.1)	487.5 (19.0)	**263.6 (19.7)**
	7	845.9 (44.6)	686.3 (33.1)	588.4 (23.2)	**324.2 (15.7)**
	10	940.5 (29.7)	817.7 (29.6)	729.3 (23.6)	**378.9 (12.6)**
industry2	2	633.9 (154.4)	624.4 (173.1)	690 (88.1)	**232.9 (41.23)**
	5	2750.4 (109.5)	2002.9 (113.9)	1368.4 (104.1)	**907.3 (79.5)**
	7	2995.5 (78.9)	2156.3 (105.8)	1656.7 (106.7)	**1206.35 (113.76)**
	10	3091.3 (77.8)	2306.2 **(57.1)**	1750.2 (104.7)	**1487.55** (78.09)
avg_large	2	803.4 (158.6)	880.2 (77.9)	611.3 **(49.9)**	**412.9** (96.92)
	5	3992.5 (114.4)	2215.6 (79.2)	1478.4 (69.1)	**835.3 (64.3)**
	7	4608.4 (78.8)	2816.6 (66.1)	1975.6 (62.9)	**948.3 (42)**
	10	5081.3 (88.39)	3661.1 (101.6)	2758.9 (131.1)	**1200.45 (53.36)**
golem3	2	3299.8 (289.7)	4012 (301.0)	3208.3 (198.7)	**1607.1 (170.74)**
	5	23492 (416.9)	9846.8 (545.3)	5931*	**3878.9 (231.01)**

	7	27177 (453.9)	12320 (411)	N/A	**4552.9 (241.38)**
	10	29010 (328.8)	N/A	N/A	**5264.4 (155.77)**

* We only execute one time with run time = 28223 seconds.

Table 10

The best cutsize (the worst cutsize) for four algorithms

Benchmark Circuits	k	SA	PLM3	PFM3	NGSP
test06	2	**60** (109)	68 (89)	69 (109)	**60 (82)**
	5	238 (347)	170 (249)	113 (213)	**110 (149)**
	7	289 (372)	191 (287)	133 (240)	**121 (160)**
	10	335 (408)	217 (302)	151 (2622)	**150 (202)**
struct	2	43 (63)	43 (55)	**33** (75)	**33 (45)**
	5	226 (364)	137 (252)	90 (233)	**82 (105)**
	7	334 (492)	228 (378)	154 (312)	**99 (139)**
	10	439 (595)	367 (485)	267 (409)	**134 (176)**
primary2	2	180 (373)	173 (353)	171 (311)	**139 (215)**
	5	809 (933)	665 (804)	463 (570)	**381 (463)**
	7	879 (1024)	772 (882)	536 (683)	**459 (533)**
	10	977 (1083)	810 (918)	703 (797)	**549 (611)**
biomed	2	83 (279)	130 (215)	140 (231.3)	**83 (104)**
	5	574 (841)	497 (637)	428 (531)	**222 (307)**
	7	682 (935)	612 (744)	534 (645)	**292 (366)**
	10	784 (997)	797 (894)	683 (790)	**352 (404)**
industry2	2	282 (1067)	386 (941)	454 (845)	**190 (354)**
	5	2485 (2963)	1710 (2176)	1175 (1583)	**714 (1054)**
	7	2809 (3154)	1958 (2361)	1481 (1835)	**952 (1354)**
	10	2847 (3276)	2205 (2408)	1582 (1921)	**1309 (1650)**
avg_large	2	359 (1112)	768 (1019)	519 (696)	**185 (476)**
	5	3593 (4205)	2026 (2318)	1394 (1580)	**712 (973)**
	7	4438 (4816)	2745 (2933)	1887 (2073)	**887 (1059)**
	10	4831 (5272)	3501 (3811)	2615 (2895)	**1103 (1289)**
golem3	2	2624 (3886)	3491 (4542)	2972 (3607)	**1417 (1927)**
	5	22606 (24388)	8999 (10931)	5931*	**3396 (4166)**
	7	26409 (28290)	11727 (12944)	N/A	**4125 (4815)**
	10	28577 (29580)	N/A	N/A	**5022 (5478)**

* We only execute once with run time = 28223 seconds.

For the large benchmark circuit golem3, the cutsize for PLM3 with $k = 10$, the cutsize for PFM3 with $k = 7$ and $k = 10$, are not available due to the unacceptable execution time which is much more than 28800 seconds (8 hours) for one execution.

We can conclude from Table 9 and Table 10 that for all the 28 instances (7 benchmark circuits multiplied by 4 different partitions), NGSP always significantly outperforms SA, PLM3, and PFM3 in terms of solution quality (minimum cutsize, average cutsize, and maximum cutsize). For the standard deviation, NGSP also gets the minimum values for most of the instances. It implies that NGSP is the most stable one among all the four algorithms since its final solutions do not heavily depend on the initial solution.

Based on Table 9, the average improvements of NGSP over SA, PLM3, and PFM3 in terms of the average cutsize for $k = 2, 5, 7, 10$ are shown in Table 11. From Table 11, we observed that the average cutsizes of both PLM3 and PFM3 are greater than that of SA for bipartitioning. This observation is consistent with that in [5]. NGSP beats SA in the average cutsize significantly for bipartitioning.

Table 11

Average improvements (%) of the average cutsize of NGSP over SA, PLM3 and PFM3

k	SA	PLM3	PFM3
2	40.91	44.03	41.79
5	67.51	52.67	33.84
7	65.87	49.63	36.2
10	62.12	44.23	35.22

Based on Table 10, the average improvements of NGSP over SA, PLM3, and PFM3 in terms of the best cutsize for $k = 2, 5, 7, 10$ are shown in Table 12. Table 12 shows that in terms of the best cutsize SA also beats PLM3 and PFM3 for bipartitioning. But for multiway partitioning the rank for the best cutsize among these three algorithms is PFM3, PLM3, and SA.

Table 12

Average improvements (%) of the best cutsize of NGSP over SA, PLM3 and PFM3

k	SA	PLM3	PFM3
2	25.3	39.55	35.31
5	66.53	50.97	29.15
7	65.94	52.48	31
10	62.45	64.31	32.6

Table 13 shows the average run time for different k values on seven benchmark circuits. Table 14 shows the total amount of run time required by SA (28 instances), PLM3 (27 instances), PFM3 (26 instances), and NGSP (28 instances). Table 15 presents the ratio of total amount of run time required for SA, PLM3 and PFM3 with respect to NGSP. It can be seen that SA takes the smallest run time (around 1/11.2 of NGSP), PFM3 takes far more time than NGSP (10.35 times of NGSP), and PLM3 takes run time 2.69 times of that of NGSP. It should be pointed out that the run time for large benchmark circuits is also affected by the limitation of physical memory in the test machine since hard disk swapping was needed.

Table 13

Average run times (seconds) for different algorithms on seven benchmark circuits

Benchmark Circuits	k	SA	PLM3	PFM3	NGSP
test06	2	0.35	0.80	1.48	9.98
	5	0.47	11.08	27.68	11.96
	7	0.56	28.70	67.52	18.9
	10	0.70	76.36	165.32	22.76
struct	2	0.35	0.62	1.56	2.70
	5	0.50	10.90	24.44	4.28
	7	0.57	27.32	59.28	5.48
	10	0.76	59.18	158.10	7.08
primary2	2	1.02	2.60	4.58	5.80
	5	1.03	25.08	72.68	28.74
	7	1.28	66.04	159.70	35.96
	10	1.52	182	309.24	20.60
biomed	2	2.02	3.30	13.78	14.30
	5	2.43	48.48	98.68	30.56

	7	2.75	131	211.42	38.74
	10	3.58	373	631.24	53.40
industry2	2	5.12	12.18	43.15	72.6
	5	6.34	122.50	514.90	148.15
	7	8.71	327.54	1396.75	126.35
	10	11.05	818.21	4507.95	160.95
avg_large	2	10.91	18.40	99.95	256.7
	5	14.79	195.80	989.20	356.55
	7	18.42	532	2419.60	509.45
	10	20.85	3073	5508.25	649.15
golem3	2	82.60	147	1701	848.9
	5	145.05	2715	28223	1139
	7	160.03	7200	N/A	1455
	10	228.5	N/A	N/A	2149.9

Table 14

Total amount (seconds) of time required for SA, PLM3, PFM3, and NGSP

SA	PLM3	PFM3	NGSP
732.76 (28)	16208.08 (27)	47410.45 (26)	4579.04 (26), 6034.04 (27), 8183.94 (28)

Table 15

The ratio of total amount of time required for SA, PLM3, and PFM3 with respect to NGSP

SA	PLM3	PFM3	NGSP
0.089	2.69	10.35	1

The previous experimental results show that our NGSP significantly outperforms DA (both PLM and PFM) in solution quality with much less run time. Although SA is a very fast algorithm, its very poor solution quality is not acceptable. The large run time for PLM3 and PFM3 makes them unsuitable to solve multiway partitioning for large circuits (for example, golem3).

9. Conclusion

In this paper, we proposed a new effective multiway partitioning algorithm called NGSP. The new algorithm incorporates the concept of net gain into the selection of cell moves and uses a new perturbation mechanism to extend solution space in enhancing the solution quality. NSGP uses a mechanism to smoothly decrease the number of iterations from pass to pass to reduce the computational effort. It makes our algorithm capable of dealing with large size benchmark circuits. According to our experimental studies, NGSP significantly outperforms, in term of solution quality and run time, the recent multiway partitioning algorithms proposed by Dasdan and Aykanat [5].

Our ongoing research will seek an adaptive scheme to further reduce the number of moves in a pass and try to find a refined perturbation mechanism to guide the searching of solution space more effectively.

References

[1] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning", Integration, *the VLSI Journal*, vol. 19, pp. 1-81, 1995.

[2] C. Berge, Graphs and Hypergraphs, , *New York, American Elsevier*, 1976.

[3] J. Cong, P. Li, S. Lim, T. Shibuya and D. Xu, "Large scale circuit partitioning with loose/stable net removal and signal flow based clustering", *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, Nov. pp. 441-446, 1997.

[4] J. Cong and S. Lim, "Myltiway partitioning with pairwise movement", *Proc. IEEE/ACM int'l Conf. on Computer-Aided Design*, pp. 512-516, 1998.

[5] A. Dasdan and C. Aykanat, "Two novel multiway circuit partitioning algorithms using relaxed locking", *IEEE Trans. on Computer- Aided Design*, vol.16, no.2, pp. 169-178, 1997.

[6] S. Dutt and W. Deng, "VLSI circuit partitioning by cluster-removal using iterative improvement techniques", *Proc. IEEE/ACM Int'l Conf. On Computer-Aided Design*, Nov. pp.92-99, 1996.

[7] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions", *Proc. ACM/IEEE Design Automation Conf.*, pp.175-181, 1982.

[8] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the theory of NP-completeness, *W. H. Freeman, San Francisco*, CA, 1979.

[9] L. W. Hagen, D. J. Huang and A. B. Kahng, "On implementation choices for iterative improvement partitioning algorithms", *Proc. European Design Automation Conference*, pp. 144-149, 1995.

[10] S. Hauck and G. Borriello, "An evaluation of bipartitioning techniques", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 849-866, 1997.

[11] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, "Multilevel hypergraph partitioning: Applications in VLSI domain", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, March, pp.69 –79, 1999.

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

[13] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks", *IEEE Trans. on Computers*, vol. 33, no. 5, pp.438-446, 1984.

[14] B. Mobasher, N. Jain, E. H. Han and J. Srivastava, "Web mining: Pattern discovery from world wide web transactions", *Technical Report TR 96-50, Department of Computer Science, University of Minnesota, Minneapolis*, 1996.

[15] C. I. Park and Y. B. Park, "An Efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no.11, pp.1686-1694, 1993.

[16] M. Ouyang, M. Toulouse, K. Thulasiraman, F. Glover and J. Deogun, "Multilevel Cooperative Search for the Circuit/Hypergraph Partitioning Problem", to appeare in *IEEE Trans. on CAD.*

[17] D. G. Schweikert and B. W. Kernighan, "A proper model for the partitioning of electrical circuits", *Proc. 9th Design Automation Workshop*, pp.57-62, 1972.

[18] L. A. Sanchis, "Multiple-way network partitioning", *IEEE Trans. on Computers*, vol. 38, no. 1, pp.62-81, 1989.

[19] S. Shekhar and D. R. Liu, "Partitioning similarity graphs: A framework for declustering problems", *Information Systems*, vol. 21, no. 4, pp. 475-496, 1996.

[20] H. H. Yang and D. F. Wong, "Efficient network flow based min-cut balanced partitioning", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol.15, no.12, pp.1533-1539, 1996.

[21] C. W. Yeh, C. K. Cheng and T. Y. Lin, "A general purpose, multiway partitioning algorithm", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no.12, pp. 1480-1488, 1994.

School of Computer Science and Information Systems
Pace University
Technical Report Series

EDITORIAL BOARD

The School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. *Technical Reports* is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from the pages you have submitted. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
School of CS & IS - Suite 412 Graduate Center
Pace University
1 Martine Avenue
White Plains, NY 10606-1932