

**GRAPH AND HYPERGRAPH
PARTITIONING**

A THESIS

**SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE**

**By
Ali Dağdan
September, 1993**

GRAPH AND HYPERGRAPH PARTITIONING

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Ali Daşdan

tarafından bağışlanmıştır.

By


Ali Daşdan

September, 1993

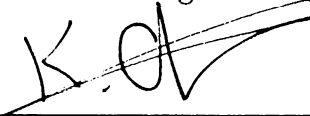
B 1234

TK
7876
D37
1992

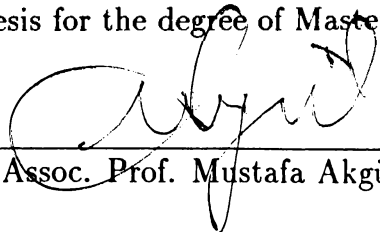
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Ceydet Aykanat (Advisor)

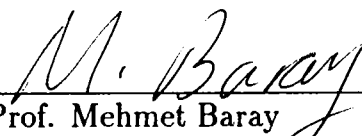
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Kemal Oflazer

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Assoc. Prof. Mustafa Akgül

Approved for the Institute of Engineering and Science:


Prof. Mehmet Baray
Director of the Institute

ABSTRACT

GRAPH AND HYPERGRAPH PARTITIONING

Ali Daşdan

M.S. in Computer Engineering and Information Science

Advisor: Asst. Prof. Cevdet Aykanat

September, 1993

Graph and hypergraph partitioning have many important applications in various areas such as VLSI layout, mapping, and graph theory. For graph and hypergraph partitioning, there are very successful heuristics mainly based on Kernighan-Lin's minimization technique. We propose two novel approaches for multiple-way graph and hypergraph partitioning. The proposed algorithms drastically outperform the best multiple-way partitioning algorithm both on randomly generated graph instances and on benchmark circuits. The proposed algorithms convey all the advantages of the algorithms based on Kernighan-Lin's minimization technique such as their robustness. However, they do not convey many disadvantages of those algorithms such as their poor performance on sparse test cases. The proposed algorithms introduce very interesting ideas that are also applicable to the existing algorithms without very much effort.

Keywords: Graph Partitioning, Hypergraph Partitioning, Circuit Partitioning, Local Search Heuristics, Partitioning Algorithms

ÖZET

ÇİZGE VE HİPERÇİZGE PARÇALAMA

Ali Daşdan

Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans

Danışman: Yrd. Doç. Dr. Cevdet Aykanat

Eylül, 1993

Çizge ve hiperçizge parçalama, çok büyük ölçekli tümleşik devre tasarımı, paralel bilgisayarlarda hesaplama yükünün işlemcilerle dağıtımı, çizge kuramı gibi bir çok alanda önemli uygulamaları olan işlemlerdir. Çizge ve hiperçizge parçalama işlemleri için, Kernighan-Lin'in tekniğine dayanan çok başarılı buluşsal algoritmalar vardır. Biz bu çalışmamızda, çok yollu çizge ve hiperçizge parçalamak için iki tane yeni yaklaşım önerdik. Önerilen algoritmalar, rastgele üretilmiş çizge örneklerinde ve algoritmaları karşılaştırmak için kullanılan standart devrelerde şu anda çok yollu çizge ve hiperçizge parçalamak için kullanılan en iyi algoritmadan çok daha iyi sonuçlar verdi. Önerilen algoritmalar, eski algoritmaların çizge ve hiperçizge problemlerindeki yeni ve değişik gereklere kolayca uyarlanabilme gibi iyi özelliklerini taşımalarına rağmen, eski algoritmaların yoğunluğu çok seyrek olan çizge ve hiperçizge problemleri üzerinde kötü sonuçlar vermesi gibi kötü özelliklerini taşımamaktadırlar. Önerilen yaklaşımların getirdiği çok ilginç fikirler, eski algoritmalara da çok büyük bir çaba gerektirmeden uygulanabilir.

Anahtar Sözcükler: Çizge Parçalama, Hiperçizge Parçalama, Devre Parçalama, Buluşsal Algoritmalar, Parçalama Algoritmaları

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my supervisor Dr. Cevdet Aykanat for his guidance, suggestions, and invaluable encouragement throughout the development of this thesis. I would like to thank Dr. Kemal Oflazer for his encouragement as well as for reading and commenting on the thesis. I would also like to thank Dr. Mustafa Akgül for reading and commenting on the thesis. I owe special thanks to Dr. Mehmet Baray for providing a pleasant environment for study. I am grateful to my family and my friends for their infinite moral support and help.

Bu alıřmamı,
herřeyimi borlu olduėum *anneme ve babama*,
ve
ailemizin en kk yesi
Ceren'e
adıyorum.

Contents

1	INTRODUCTION	1
1.1	Combinatorial Optimization Problems	1
1.2	Graph and Hypergraph Partitioning Problems	2
1.3	Previous Approaches	3
1.4	Motivation	5
1.5	Experiments and Results	7
1.6	Outline	7
2	GRAPH PARTITIONING	8
2.1	Introduction	8
2.2	Basic Concepts	9
2.3	Graph Partitioning Problem	11
2.4	Multiple-way Graph Partitioning	13
2.4.1	Gain Concept	13
2.4.2	Effects of a Vertex Move	14
2.4.3	Balance Conditions	16
3	HYPERGRAPH PARTITIONING	18

3.1	Introduction	18
3.2	Basic Concepts	18
3.3	Hypergraph Partitioning Problem	22
3.4	Multiple-way Hypergraph Partitioning	23
3.4.1	Gain Concept	23
3.4.2	Effects of a Vertex Move	24
3.4.3	Balance Conditions	29
4	PARTITIONING ALGORITHMS	31
4.1	Local Search	31
4.2	Neighborhood Structure	35
4.3	Previous Approaches	36
4.4	Bipartitioning versus Multiple-way Partitioning	38
4.5	Data Structures	40
4.6	Reading Hypergraphs and Graphs	42
4.7	Initial Partitions	43
4.8	Cutsizes Calculation	45
4.9	Prefix Sum Calculation	46
4.10	Main Claim	47
4.11	Partitioning by Locked Moves	49
4.12	Partitioning by Free Moves	53
4.13	Complexity Analysis	59
4.13.1	Time Complexity Analysis	59

4.13.2	Space Complexity Analysis	61
5	EXPERIMENTS AND RESULTS	64
5.1	Implementation of Algorithms	64
5.2	Balance Condition	64
5.3	K-PFM Algorithms	65
5.4	K-PLM Algorithms	65
5.5	Comments on Neighborhood Structure of Algorithms	67
5.6	Notation	68
5.7	Test Graphs	69
5.7.1	Random Graphs	69
5.7.2	Geometric Graphs	69
5.7.3	Grid Graphs	70
5.7.4	Ladder Graphs	71
5.7.5	Tree Graphs	71
5.8	Test Hypergraphs	73
5.9	General Comments on Experiments	73
5.10	General Comments for Experiments on Graphs	74
5.11	Performance of K-PFM Algorithms on Graphs	74
5.11.1	Different Freedom Value Functions	77
5.11.2	Determining Scale Factor	78
5.12	Performance of K-PLM Algorithms on Graphs	81
5.13	General Comments for Experiments on Hypergraphs	81

<i>CONTENTS</i>	x
5.14 Performance of K-PFM Algorithms on Hypergraphs	82
5.15 Performance of K-PLM Algorithms on Hypergraphs	83
5.16 Behaviour of Freedom Value Function	84
5.17 Convergence of Algorithms	84
5.18 Distribution of Cutsizes	85
5.19 Distribution of Move Gains	86
6 CONCLUSIONS	87
7 APPENDICES	90
A FILE FORMATS	91
B PLOTS FOR EXPERIMENTS	93
C TABLES FOR EXPERIMENTS	101

List of Figures

2.1	An algorithm for initial cost computation in a graph	14
2.2	An algorithm for gain updates in a graph	16
3.1	An algorithm for initial cost computation in a hypergraph	25
3.2	An algorithm for gain updates in a hypergraph	30
4.1	A general local search algorithm	34
4.2	Bucket data structure for a part in a given partition	42
4.3	An initial partitioning algorithm	44
4.4	A cutsize calculation algorithm for graphs	45
4.5	A cutsize calculation algorithm for hypergraphs	46
4.6	Change of gains of selected moves in Sanchis' Algorithm (one pass contains 250 moves)	47
4.7	The generic direct multiple-way partitioning-by-locked-moves algorithm	50
4.8	The generic direct multiple-way partitioning-by-free-moves algorithm	56
5.1	Random graph generation algorithm	70
5.2	Geometric graph generation algorithm	71

5.3	Grid generation algorithm	72
5.4	Tree generation algorithm	72
B.1	Freedom Value for move gains at different move counts n , for $G_{max} = 100$, $N = 1000$, and $K = 8$	94
B.2	Convergence of K-PLM1 Algorithm, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 2, 4$, and 8	94
B.3	Convergence of K-PFM1 Algorithm, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 2, 4$, and 8	95
B.4	Convergence of K-PLM1 and PLM12 Algorithms, a plot of cut- size versus number of moves performed until local minimum is found, for $K = 4$	95
B.5	Convergence of K-PLM1 and PLM11 Algorithms, a plot of cut- size versus number of moves performed until local minimum is found, for $K = 4$	96
B.6	Convergence of K-PLM1 and PLM10 Algorithms, a plot of cut- size versus number of moves performed until local minimum is found, for $K = 4$	96
B.7	Convergence of K-PLM1 and PLM9 Algorithms, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 4$	97
B.8	Distribution of cutsizes for K-PLM1 and K-PFM1 Algorithms, a cutsizes on x-axis has been found the corresponding value on y-axis times by the algorithms	97
B.9	Change of gains of selected moves in K-PLM1 Algorithm	98
B.10	Change of cutsizes at each move in K-PLM1 Algorithm (final cutsizes is 18)	98
B.11	Change of gains of selected moves in K-PFM1 Algorithm	99

B.12 Change of cutsize at each move in K-PFM1 Algorithm (final cutsize is 9) 99

B.13 Change of gains of selected moves in K-PLM3 Algorithm 100

B.14 Change of cutsize at each move in K-PLM3 Algorithm (final cutsize is 0) 100

List of Tables

C.1	Properties of Random Test Graphs	102
C.2	Properties of Geometric Test Graphs	102
C.3	Properties of Grid Test Graphs	102
C.4	Properties of Ladder Test Graphs	103
C.5	Properties of Tree Test Graphs	103
C.6	Properties of Benchmark Circuits (multiply w_T by 1000, $c_{max} =$ 1 and $c_{min} = 1$ for all circuits)	103
C.7	Execution time averages (and standard deviations) for random graphs	104
C.8	Cutsizes averages (and standard deviations) for random graphs .	105
C.9	Execution time averages (and standard deviations) for geometric graphs	106
C.10	Cutsizes averages (and standard deviations) for geometric graphs	107
C.11	Execution time averages (and standard deviations) for grid graphs	108
C.12	Cutsizes averages (and standard deviations) for grid graphs . . .	109
C.13	Execution time averages (and standard deviations) for ladder graphs	109
C.14	Cutsizes averages (and standard deviations) for ladder graphs . .	110
C.15	Execution time averages (and standard deviations) for tree graphs	110

C.16 Cutsizes averages (and standard deviations) for tree graphs . . .	110
C.17 Cutsizes averages for random graphs (with different freedom value functions for K-PFM1)	111
C.18 Cutsizes averages for random graphs (with different freedom value functions for K-PFM2)	111
C.19 Cutsizes averages for random graphs (with different freedom value functions for K-PFM3)	112
C.20 Cutsizes averages for random graphs (optimizing S for K-PFM1)	112
C.21 Cutsizes averages for random graphs (optimizing S for K-PFM2)	113
C.22 Cutsizes averages for random graphs (optimizing S for K-PFM3)	114
C.23 Cutsizes averages for random graphs (for K-PLM-like algorithms)	115
C.24 Cutsizes averages for geometric graphs (for K-PLM-like algorithms)	115
C.25 Execution time averages (and standard deviations) for bench- mark circuits	116
C.26 Cutsizes averages (and standard deviations) for benchmark circuits	117
C.27 Minimum Cutsizes for benchmark circuits	118
C.28 Cutsizes averages for some benchmark circuits	118
C.29 Minimum Cutsizes for some benchmark circuits	119

List of Symbols

\mathcal{N}	the set of natural numbers
$\mathcal{N}(1, N)$	every natural number between 1 and N inclusive
\mathcal{O}	the big O-notation
G	a graph
H	a hypergraph
V	set of vertices
E	set of edges (or nets)
v	a vertex
e	an edge (or a net)
d_i	degree of vertex v_i
$d(v_i)$	degree of vertex v_i
$ V $	number of vertices in V
N	number of vertices in V
$ E $	number of edges (or nets) in E
M	number of edges (or nets) in E
p	total number of terminals of nets in E
Π	a partition
K	number of parts in a partition
P_k	k th part in a partition
$w(v)$	weight of vertex v
$c(e)$	weight of edge (or net) e
w_T	total vertex weight
c_T	total edge weight
D_{exp}	expected average vertex degree before generation of graph
D_{act}	actual average vertex degree after generation of graph
$D_{v,max}$	maximum vertex degree (also $D_{v,x}$)
D_v	average vertex degree
$D_{e,max}$	maximum net degree (also $D_{e,x}$)
D_e	average net degree
w_{max}	maximum vertex weight (also w_x)
w_{min}	minimum vertex weight (also w_n)
c_{max}	maximum edge weight
c_{min}	minimum edge weight

E_k	set of external edges of part P_k
I_k	set of internal edges of part P_k
$\chi(\Pi)$	cutsizes of partition Π
$E_m(f, t)$	external edges of vertex $v_m \in P_f$ with respect to P_t
$I_m(f, f)$	external edges of vertex $v_m \in P_f$
$C_m(f, t)$	cost of vertex $v_m \in P_f$ with respect to P_t
$G_m(f, t)$	move gain of vertex $v_m \in P_f$ with respect to P_t
$\delta_i(k)$	number of terminals of net e_i in part P_k
γ_q	reduction in cutsizes at the q th move in a pass
Q	maximum number of moves in a pass
σ_q	(prefix) sum of the first q reductions in cutsizes
<i>gainsum</i>	maximum prefix sum
$\Phi_m(f, t)$	freedom value of vertex $v_m \in P_f$ with respect to P_t
R	constant used to make freedom value be in $(0, 1)$
S	scale factor
G_{max}	maximum move gain
n_m	move count of vertex v_m
s	a solution
$N(s)$	neighborhood of solution s
$\chi(s)$	cost of solution s
$K - PLM$	multiple-way partitioning by locked moves
$K - PFM$	multiple-way partitioning by free moves
$B(k)$	upper bound on size of part P_k
$b(k)$	lower bound on size of part P_k
α	tolerance constant in balance condition
ϵ	a very small value greater than zero
$s(P_k)$	size of part P_k

Please see some additions and corrections on page 124 !

Chapter 1

INTRODUCTION

1.1 Combinatorial Optimization Problems

Many problems that arise in practical situations are combinatorial optimization problems which involve a finite set of configurations from which solutions satisfying a number of rigid requirements are selected. The goal is to find a solution of the minimum or maximum cost (or the optimum cost) provided that a cost can be assigned to each solution.

Many combinatorial optimization problems are *hard* in the sense that they are NP-hard or harder [13]. There are no known deterministic polynomial time algorithms to find the optimal solution to any of those hard problems. The algorithms employing the complete enumeration techniques are not reasonable to use because the complexity of these techniques is usually exponential in the size of the problem and hence, they require a great amount of time to find the optimal solution for even very small problem instances. As a result, *heuristic* algorithms (or heuristics) that run in a low-order polynomial time have been employed to obtain good solutions to these hard problems, where, by a good solution, we mean a solution that is hopefully close to the optimal solution to the problem.

The methods used for designing heuristic algorithms tend to be rather problem specific. *Local search* is one of the few general approaches to solving hard combinatorial optimization problems. Local search is based on trial and error method, which is probably the oldest optimization method.

Before deriving a local search algorithm for a problem, a neighborhood structure for any solution must be chosen. For each potential solution to the problem, this structure specifies a neighborhood which consists of a set of solutions that are in some sense close to that solution. A rule as to how a neighbor solution can be generated by modifying a given solution is associated with the neighborhood structure.

Starting from some given initial solution, a local search algorithm tries to find a better solution which is a neighbor of the first. If a better neighbor is found, a search starts for a better neighbor of that one, and so on. Since the set of solutions is finite, this search must halt, that is, the local search algorithm must end at a *locally optimum* solution, which does not have a better neighbor solution. Local search algorithms are also called *iterative improvement algorithms* because they iteratively improve an initial solution so as to find a locally optimal solution.

Suppose that the problem is a minimization problem and so the smaller the cost of the solution found, the better the solution. The modification of a given solution to obtain a neighbor in the neighborhood of the given solution is called a *move*. If the move results in a neighbor with a better cost, the move is a downhill move. On the other hand, if the move results in a neighbor with a worse cost, the move is an uphill move. The basic local search algorithm employs only downhill moves.

1.2 Graph and Hypergraph Partitioning Problems

Graph partitioning problem is an example of the problems to which the local search method has been successfully applied. Given a graph, graph partitioning problem is concerned with finding a partition of the graph into a predetermined number of nonempty, pairwise disjoint parts such that the sizes of the parts are bounded and the total size, cutsize, of the edges in the cut, those edges that connect different parts, is minimized. Graph partitioning problem is an NP-hard combinatorial optimization (minimization) problem [13].

The importance of the graph partitioning problem is mostly due to its connection to the problems whose solutions depend on the divide-and-conquer paradigm [26]. A partitioning algorithm partitions a problem into semi-independent subproblems, and tries to reduce the interaction between these

subproblems. This division of a problem into simpler subproblems results in a substantial reduction in the search space [34].

Graph partitioning has many important applications in various areas such as VLSI layout [2, 21, 22, 33, 40], mapping of computation to processors in a parallel computer environment [7, 8, 30], sparse matrix calculations [14, 15, 24], and so on. There are also some theoretical justifications for the usage of graph partitioning in VLSI layout. For example, it is shown that a provable good graph partitioning algorithm can be tailored into a provable good layout algorithm [2].

A hypergraph is a generalization of a graph such that an edge, called a net in a hypergraph, of a hypergraph can connect more than two vertices. *Hypergraph partitioning problem* is exactly the same as graph partitioning problem except that the structure to be partitioned here is a hypergraph. Since an edge in a graph can only connect two vertices, edges do not properly represent electrical interconnections. As a result, a hypergraph is better suited to electrical circuits in which some of the nets have three or more connected devices [32]. Hence, not surprisingly, hypergraph partitioning has important applications in VLSI layout [4, 9, 11, 12, 28, 35, 36]. The hypergraph partitioning problem is also NP-hard [13].

If a graph (hypergraph) is to be partitioned into more than two parts, then the problem is referred to as multiple-way graph (hypergraph) partitioning problem. When there is only two parts in the partition, the problem is called graph (hypergraph) bipartitioning problem.

1.3 Previous Approaches

Since both graph and hypergraph partitioning problems are unfortunately hard problems, we should resort to heuristics to obtain at least a near-optimal solution. The most successful heuristic algorithm proposed for graph partitioning problem is due to Kernighan-Lin [19]. Kernighan-Lin (KL) algorithm is a very sophisticated improvement on the basic local search procedure, involving an iterated backtracking procedure that typically finds significantly better solutions [17].

KL algorithm was adopted to hypergraph partitioning problem by

Schweikert-Kernighan [32]. KL algorithm uses a *swap-neighborhood structure* in which a neighbor of a given solution is obtained by interchanging a pair of vertices between two distinct parts in the solution. Given a graph (hypergraph) KL algorithm associates each vertex with a property called the *gain* of the vertex which is exactly the reduction in the cutsize when the vertex is moved in the partition. The swap-neighborhood structure happens to increase the running time of KL algorithm since a pair of vertices must be found to interchange. Fiduccia-Mattheyses [12] introduces the *move-neighborhood structure* in which a neighbor of a given solution is obtained by moving a vertex from one part to another in the solution. They also devise a very sophisticated data structure called the bucket list data structure which reduces the time complexity of KL algorithm to linear in the size of the hypergraph by keeping the vertices in sorted order with respect to their gains and by making the insertion and deletion operations cheaper. Krishnamurthy [20] adds the level gain concept which helps to break ties better in selecting a vertex to move. The first level gain in Krishnamurthy's (KR) algorithm is exactly the same as the gain in Fiduccia-Mattheyses' (FM) algorithm. Sanchis [31] generalizes KR algorithm to a multiple-way partitioning algorithm. Note that all the previous approaches before Sanchis' (SN) algorithm are originally bipartitioning algorithms. SN algorithm is a direct multiple-way partitioning algorithm in which, at any time during iterative partitioning, a vertex can be moved into any of the parts in the partition. However, the move should be legal, that is, it should not violate the balance condition which imposes certain bounds on the sizes of the parts in the partition. SN algorithm exploits the local minimization technique of KL algorithm, the move-neighborhood structure, balance condition, and bucket list data structure of FM algorithm, and the level gain approach of KR algorithm.

Now, since the minimization technique of KL algorithm is the basis of the many partitioning algorithms that have followed it, we explain this technique as it is used in Fiduccia-Mattheyses' algorithm, that is, in terms of vertex moves. First, an initial partition is generated. The gains of the vertices are also determined. The first move includes the legal move of the vertex with the maximum gain. This vertex is then tentatively moved and locked. A locked vertex is set aside and not considered again until all the vertices are moved and locked once, which corresponds to a pass of the algorithm. After the first move, the next legal move with the maximum gain is moved and locked. This process goes in the same manner until the end of the pass. Note that there is a recorded sequence of the moves and their respective gains at the end of the first pass. At the end of the pass, a subsequence of moves from the recorded

sequence that yields the maximum reduction in the cutsize is selected and realized permanently but starting with the first move in the recorded sequence. This operation is called the prefix sum calculation. Using the current partition obtained at that pass, another pass goes on in exactly the same manner. These passes are performed until there is not any improvement in the cutsize, which corresponds a locally minimum partition. Since a pass involves the move of each vertex once, there may exist uphill moves during the pass. The permission of uphill moves in a pass makes this minimization technique better.

All the previous partitioning algorithms use the minimization technique above. Vijayan [40] extends this technique so that a vertex is not locked as soon as it is moved. The vertex is allowed to reside in each part once before it is locked.

Henceforth when we say that a move is selected, we mean that the move is performed or the vertex associated with the move is actually moved. In other words, selecting a move has the same meaning as performing a move.

1.4 Motivation

When we examine the Kernighan-Lin's minimization technique, it reveals that moves with positive gains, those that decrease the cutsize, become more useful during the early stages of the sequence of the moves performed during a pass and that moves with negative gains, those that increase the cutsize, become more useful towards the end of the sequence of the moves performed during a pass. Hence, we should perform as more moves with positive gains as we can during a pass as long as this process does not lead us to become stuck in a poor local minimum. After some experimentation, we can observe that moves with positive gains, especially those performed in the first pass, occur actually during the early stages of the move sequence. However, we can also observe that, after some point in a pass, the moves that are selected to be moved mostly consist of those with negative gains. Experiments indicate that a move performed at an earlier stage in a pass can have positive gain again in a later stage such that its move gain is larger than those of the moves remaining but it cannot be performed because it is locked. The reason why this move is not performed has been to prevent the cell-moving process from thrashing or going into an infinite loop [12, 20, 40]. We think that this reason is not plausible

because we can find some other means to avoid thrashing or infinite number of moves during partitioning. Therefore, we make the following claim, on which all our work is based. Our claim states that *given a hypergraph with N vertices, allowing each vertex to be moved (possibly) more than once in a pass with the requirement that the occurrence of infinite number of moves having no profit be prevented improves the cutsizes more than allowing each vertex to be moved exactly once in a pass.*

We bring the *move-and-lock phase* concept for the sake of simplicity of the discussion of this claim. A move-and-lock phase contains a sequence of temporary moves and their respective locks. A pass may consist of one or more move-and-lock phases. If a move-and-lock phase is not the last one in a pass, then all the vertices that are temporarily moved during this phase are unlocked and reinserted into the appropriate bucket lists, according to their recomputed gains, for the succeeding move-and-lock phases in that pass. On the other hand, if a move-and-lock phase constitutes the last such phase in a pass, the prefix subsequence of moves which maximizes the prefix sum of move gains in that pass is realized permanently. We now propose three novel approaches exploiting the basic claim:

1. During a pass, we can make more than one move-and-lock phase such that each move-and-lock phase consists of N moves.
2. During a pass, we can make more than one move-and-lock phase such that each move-and-lock phase consists of less than N moves.
3. During a pass, we can make more than N moves but we do not employ the locking mechanism at all. Yet, there should still be some means to restrict the repeated selections of moves.

We considered all of these ways for partitioning. The items (1) and (2) establish the basis of multiple-way partitioning-by-locked-moves method, which also subsumes SN algorithm, (in Section 4.11) and the item (3) establishes the basis of multiple-way partitioning-by-free-moves method (in Section 4.12). Both of these methods are proposed and implemented in this work for graph partitioning as well as hypergraph partitioning. We expect that these methods explore the search space of the problem better.

1.5 Experiments and Results

We evaluated the graph partitioning algorithms on the graph instances that were randomly generated using the algorithms in the literature. The types of graph instances included random, geometric, grid, ladder, and tree graphs. The random and geometric graphs are standard test beds for graph partitioning algorithms [17, 3]. The other types of graphs were used to evaluate the partitioning algorithms because the KL algorithm is observed to fail badly on these types of graphs [6, 15]. We evaluated the hypergraph partitioning algorithms on the real VLSI circuits which had been taken from *ACM/SIGDA Design Automation Benchmarks*. We also did experiments to determine the best setting of the parameters in the proposed algorithms.

The proposed partitioning algorithms performed drastically better than SN algorithm, which is the best KL-like multiple-way partitioning algorithm at the moment, on both the graph and hypergraph instances. The results on the benchmark circuits correlate favorably with those in the existing partitioning literature.

1.6 Outline

We present some preliminaries from graph theory, a formal definition of the graph partitioning problem, and basic concepts related to graph partitioning and graph partitioning algorithms in Section 2. The analogous issues for hypergraphs are given in Section 3. An explanation concerned with the local search technique which constitutes the basis for the algorithms we considered, the previous approaches to the partitioning problem, a detailed investigation of the proposed algorithms and their analysis are all presented in Section 4. The following section, Section 5, includes the algorithms which were used to generate the graph instances, the details of each group of experiments that we conducted, and the results and general observations obtained from the results. Finally, the main conclusions are in Section 6. Since we still have a large number of tables and plots giving the results of the experiments although we skipped most of them, these tables are all given in appendices for the sake of clarity while presenting the text.

Chapter 2

GRAPH PARTITIONING

This chapter establishes the basic concepts on Graph Partitioning. It includes some preliminary concepts from graph theory, the definition of the graph partitioning problem, and the concepts related to the partitioning algorithms, which are examined in Chapter 4. We utilized the references [23, 38] for the definitions and notations.

2.1 Introduction

The importance of the graph partitioning problem is mostly due to its connection to the problems whose solutions depend on the divide-and-conquer paradigm [26]. A partitioning algorithm partitions a problem into semi-independent subproblems, and tries to reduce the interaction between these subproblems. This division of a problem into simpler subproblems results in a substantial reduction in the search space [34]. Graph partitioning is the basis of hypergraph partitioning, which is more general and more difficult. Graph partitioning has a number of important applications. An exhaustive list of these applications combined with the relevant references is given below.

- VLSI placement [2, 21, 22, 33].
- VLSI routing [40].
- VLSI circuit simulation [1, 10].
- memory segmentation to minimize paging [19].

- mapping of computation to processors and load balancing [7, 8, 30].
- efficient sparse Gaussian elimination [14, 15, 24].
- solving various graph problems [25].
- laying out of machines in advanced manufacturing systems [39].
- computer vision [16].

Some researchers have also utilized the graph partitioning problem as a test bed to evaluate the search and optimization algorithms they proposed.

2.2 Basic Concepts

A graph $G = (V, E)$ consists of a finite set V of *vertices* (or *nodes*) and a finite set E of *edges*. Each edge is identified with a pair of vertices. We use the symbols u, v, v_1, v_2, \dots to represent the vertices and the symbols e, e_1, e_2, \dots to represent the edges of a graph unless otherwise specified. The term graph here denotes undirected graphs, i.e., the edge $e_i = \{u, v\}$ and the edge $e_j = \{v, u\}$ represent the same edge.

Given an edge $e = \{u, v\}$, we say that the edge e is *incident to* its end vertices u and v , and that the vertices u and v are *adjacent* or *neighbors*. If two edges have a common end vertex, then those edges are said to be *adjacent*.

The number of edges incident to a vertex v_i is called the *degree* of the vertex and is denoted by $d(v_i)$ or simply d_i . A vertex of degree 0 is called an *isolated vertex*.

A graph $G = (V, E)$ has $|V| = N$ vertices and $|E| = M$ edges. Each vertex v in V has a positive integer weight $w(v)$ (w for weight) and each edge e in E has a positive integer weight $c(e)$, (c for capacity).

Given a graph $G = (V, E)$, we say that $\Pi = (P_1, \dots, P_K)$ is a *K-way partition* of G if each *part* P_k is a nonempty subset of the vertex set V , all the parts are pairwise disjoint, and the union of the K parts is equal to V . Formally, $\Pi = (P_1, \dots, P_K)$ is a *K-way partition* of $G = (V, E)$ if

1. $P_k \subset V, P_k \neq \emptyset$ for each $k \in \{1, \dots, K\}$,

2. $P_k \cap P_l = \emptyset$ for each $k, l \in \{1, \dots, K\}$ and $(k \neq l)$,
3. $\bigcup_{k=1}^K P_k = V$.

Note that the number K of parts in a partition of G is bounded above by the number of vertices in G .

For simplicity, we say that $i \in \mathcal{N}(N_1, N_2)$ if $N_1 \leq i \leq N_2$ and $i, N_1, N_2 \in \mathcal{N}$ where \mathcal{N} is the set of natural numbers. Then, when we say that $i \in \mathcal{N}(1, N)$ for a vertex v_i in the vertex set V with N vertices, we mean that v_i is any vertex in V . Similarly, when we say that $k \in \mathcal{N}(1, K)$ for a part P_k in the K -way partition Π , we mean that P_k is any part in Π .

Consider a K -way partition $\Pi = (P_1, \dots, P_K)$ of a graph $G = (V, E)$ with N vertices and M edges. Then,

- $s(P_k)$ denotes the *size* of the part P_k for $k \in \mathcal{N}(1, K)$. The size of the part P_k equals the sum of the weights of the vertices in P_k . That is,

$$s(P_k) = \sum_{v \in P_k} w(v). \quad (2.1)$$

- The total vertex weight w_T is the sum of the weights of all the vertices in the vertex set V . That is,

$$w_T = \sum_{v \in V} w(v) = \sum_{k=1}^K s(P_k). \quad (2.2)$$

- The total edge weight c_T is the sum of the weights of all the edges in the edge set E . That is,

$$c_T = \sum_{e \in E} c(e). \quad (2.3)$$

- $E_k = \{e \in E \mid e = \{u, v\} \wedge u, v \in V \wedge u \in P_k \wedge v \notin P_k\}$ is the set of *external edges* of the part P_k for all $k \in \mathcal{N}(1, K)$. The set of external edges of a part P_k consists of those edges whose one end vertex lies in the part P_k and the other end vertex lies in another part in the partition Π .
- $I_k = \{e \in E \mid e = \{u, v\} \wedge u, v \in V \wedge u, v \in P_k\}$ is the set of *internal edges* of the part P_k for all $k \in \mathcal{N}(1, K)$. The set of internal edges of a part P_k consists of those edges whose both end vertices lie in P_k .

- The edges that connect different parts in the partition Π , that is, the external edges, are said to *contribute to the cut* or *cross the cut*.
- The cost $\chi(\Pi)$ of the partition is also called the *cutsizes*. The cutsizes is the sum of the weights of all the edges contributing to the cut. That is,

$$\chi(\Pi) = \frac{1}{2} \sum_{k=1}^K \sum_{e \in E_k} c(e), \quad (2.4)$$

or

$$\chi(\Pi) = c_T - \sum_{k=1}^K \sum_{e \in I_k} c(e). \quad (2.5)$$

- A K -way partition is also a *multiple-way partition*, and the partitioning operation is called *K -way partitioning* or *multiple-way partitioning*. If there are only two parts, i.e., $K = 2$, then Π is called also a *bipartition* or a *2-way partition*.
- A partition is *balanced* if the parts have about the same size. A partition is *perfectly balanced* if the parts have exactly the same size. A perfectly balanced partition is highly unlikely in a multiple-way partitioning if the vertex weights are not equal.
- The average (vertex) degree D_v of the graph G can be found by the equation

$$D_v = \frac{2M}{N} \quad (2.6)$$

where $2M$ is equal to the sum of the degrees of all the vertices in G .

- The maximum (minimum) vertex degree of the graph G is the maximum (minimum) of the set of the degrees of the vertices in G and is denoted by $D_{v,max}$ ($D_{v,min}$).
- The maximum (minimum) vertex weight is the maximum (minimum) of the set of the weights of the vertices in G and is denoted by w_{max} (w_{min}). The maximum (minimum) edge weight is the maximum (minimum) of the set of the weights of the edges in G and is denoted by c_{max} (c_{min}).

2.3 Graph Partitioning Problem

A formal definition of the Graph Partitioning Minimization Problem (GPP) is given below. In this definition, an instance is obtained by specifying particular values for all the problem parameters.

Problem: The Graph Partitioning Minimization Problem.

Instance: A graph $G = (V, E)$, a vertex weight function $w : V \rightarrow \mathcal{N}$, an edge weight function $c : E \rightarrow \mathcal{N}$, a number $K \geq 2$, $K \in \mathcal{N}$, maximum and minimum part sizes $B(k) \in \mathcal{N}$ and $b(k) \in \mathcal{N}$, respectively, for $k \in \mathcal{N}(1, K)$.

Configurations: All K -way partitions $\Pi = (P_1, \dots, P_K)$.

Solutions: All feasible configurations, i.e., all K -way partitions $\Pi = (P_1, \dots, P_K)$ such that

$$b(k) \leq s(P_k) \leq B(k) \text{ for all } k \in \mathcal{N}(1, K)$$

Question: Find a solution such that the cutsizes

$$\chi(\Pi) = \frac{1}{2} \sum_{k=1}^K \sum_{e \in E_k} c(e)$$

is minimum over all the solutions.

Intuitively, we are given a graph $G = (V, E)$. Each vertex and each edge have a positive weight. Each K -way partition $\Pi = (P_1, \dots, P_K)$ of the vertex set V into nonempty, pairwise disjoint parts P_k (for $k \in \mathcal{N}(1, K)$), is a configuration. Given an upper bound $B(k)$ and a lower bound $b(k)$ on the size of each part P_k , we regard as solutions those partitions (or feasible configurations) in which the size of each part P_k is in the range between $b(k)$ and $B(k)$. We are then asked to find the partition (or partitions) that has the minimum cutsizes over all the solutions.

The graph partitioning minimization problem is NP-hard [13]. In order to see how large the search space of GPP is, let us simplify the problem. Suppose that $G = (V, E)$ is a graph with N vertices each of which has unit weight, and that the number N of the vertices is a perfect multiple of the number K of partitions and so let $N/K = s$, i.e., each part has a part size of s . Then, there are $\binom{N}{s}$ ways of choosing the first part, $\binom{N-s}{s}$ ways of choosing the second part, and so on. Since the ordering of the parts is immaterial, the number of feasible partitions is

$$\frac{1}{K!} \binom{N}{s} \binom{N-s}{s} \cdots \binom{2s}{s} \binom{s}{s} = \frac{N!}{K!(s!)^K} \quad (2.7)$$

For $N = 100$ and $K = 2$, the number of feasible partitions is greater than 10^{27} , and for $N = 100$ and $K = 4$, it is greater than 10^{54} . Today, there are graph partitioning instances with $N = 50000$. Hence, it is clear that the number of feasible partitions is too large to search exhaustively.

2.4 Multiple-way Graph Partitioning

2.4.1 Gain Concept

Let $G = (V, E)$ be a graph with N vertices and $\Pi = (P_1, \dots, P_K)$ a K -way partition of G . Let $f, t \in \mathcal{N}(1, K)$ be two numbers (f represents the part from which a vertex is moved, and t represents the part to which the vertex is moved.) The *cost* $C_m(f, t)$ of a vertex v_m in P_f with respect to a part P_t (m for *moved* vertex) is defined as

$$C_m(f, t) = \begin{cases} \sum_{e \in E_m(f, t)} c(e) & \text{if } f \neq t \\ \sum_{e \in I_m(f, f)} c(e) & \text{otherwise} \end{cases} \quad (2.8)$$

where

$$E_m(f, t) = \{e \in E_f \mid e = \{v_m, u\} \wedge u \in P_t\} \quad (2.9)$$

is the subset of the set of the external edges of the part P_f whose one end vertex is v_m and the other end vertex lies in the part P_t , and

$$I_m(f, f) = \{e \in I_f \mid e = \{v_m, u\} \wedge u \in P_f\} \quad (2.10)$$

is the subset of the set of the internal edges of the part P_f whose one end vertex is v_m and the other end vertex lies in the part P_f . The edges in the sets $E_m(f, t)$ and $I_m(f, f)$ are called the *external edges* and *internal edges* of the vertex v_m with respect to the part P_t , respectively.

The *move gain* $G_m(f, t)$ of the vertex v_m in the part P_f with respect to the part P_t is given by the equation

$$G_m(f, t) = C_m(f, t) - C_m(f, f). \quad (2.11)$$

That is, the gain $G_m(f, t)$ of a vertex v_m in the part P_f with respect to the part P_t is the difference between the sum of the weights of the external edges of v_m whose the other end vertex is in P_t and the sum of the weights of the internal edges of v_m . The gain of a vertex represents the decrease that results in the cutsize when the vertex is moved. The gain of a vertex with respect to the part where the vertex is present is zero.

In the K -way partition, each vertex has K costs. These costs constitute the *cost vector* of the vertex. For each vertex v_m in P_f , the entry $C_m(f, f)$ is the *internal cost* of v_m and the other $(K - 1)$ entries are the *external costs* of v_m with respect to each part other than P_f in Π .

Algorithm: Initial Cost Computation Algorithm

Input: a graph $G = (V, E)$ with N vertices, a K -way partition $\Pi = (P_1, \dots, P_K)$ of G

Output: vertices in V with all cost vectors computed

1. **for** each vertex v_i , where $v_i \in P_f$ and $i \in \mathcal{N}(1, N)$, **do**
 - 1.1. **for** each part number t , where $t \in \mathcal{N}(1, K)$ **do**
 - 1.1.1. let $C_m(f, t) \leftarrow 0$ /* initialize the cost */
 - 1.2. **endfor**
 - 1.3. **for** each edge $e = \{v_i, u\}$ **do**
 - 1.3.1. find the part P_t such that $u \in P_t$
 - 1.3.2. let $C_m(f, t) \leftarrow C_m(f, t) + c(e)$
 - 1.4. **endfor**
 2. **endfor**
-

Figure 2.1. An algorithm for initial cost computation in a graph

Figure 2.1 illustrates the pseudocode for the algorithm which computes the initial cost vectors of the vertices in a graph assuming an initial feasible partition. Note that the computation of the initial gains of the vertices can be done easily by using Equation 2.11 provided that the initial costs are given.

2.4.2 Effects of a Vertex Move

Let $G = (V, E)$ be a graph and $\Pi = (P_1, \dots, P_K)$ a K -way partition of G . Let $f, t \in \mathcal{N}(1, K)$ be two numbers. Consider the move of the vertex v_m in the part P_f to the part P_t , where $f \neq t$. We now give the effects of this move.

1. **Effect on Cutsizes :** The cutsize should be updated by the equation

$$\chi(\Pi) \leftarrow \chi(\Pi) - G_m(f, t) \quad (2.12)$$

where $G_m(f, t)$ is the gain of the vertex v_m before the move. Hence, the decrease in the cutsize is equal to $G_m(f, t)$, which is expected by the definition of the gain concept. Note that a negative gain value (i.e., $G_m(f, t) < 0$) increases the cutsize.

2. **Effect on Part Sizes :** The part size of the part P_f decreases and the part size of the part P_t increases by the move of v_m . Hence, The following changes

in the parts sizes should be done.

$$s(P_f) \leftarrow s(P_f) - w(v_m) \quad (2.13)$$

$$s(P_t) \leftarrow s(P_t) + w(v_m) \quad (2.14)$$

3. Effect on Vertex Moved : There is no change in the entries of the cost vector of the vertex v_m , which is moved. The only change in the cost vector is in the interpretation of some entries. The entry $C_m(f, f)$ was the internal cost of v_m before the move and the entry $C_m(f, t)$ was the external cost of v_m to the part P_t before the move. After the move, the entry $C_m(t, t)$ becomes the internal cost of v_m and the entry $C_m(t, f)$ becomes the external cost of v_m to the part P_f where $C_m(f, f)$ before the move is equal to $C_m(t, f)$ after the move and $C_m(f, t)$ before the move is equal to $C_m(t, t)$ after the move. However, since the internal cost of v_m is changed, the gains of v_m to every part (other than P_t) in Π must be recomputed using Equation 2.11.

4. Effect on Neighbor Vertices : The algorithm in Figure 2.2 calculates the changes in the costs and gains of the neighbor vertices that result from the vertex moved. The move of the vertex v_m from the part P_f to the part P_t affects only the costs $C_r(k, f)$ and $C_r(k, t)$ of a vertex $v_r \in P_k$ adjacent to v_m . If $k \neq f$ and $k \neq t$, this means that there is no change in the internal cost $C_r(k, k)$ and hence, only two gain values $G_r(k, f)$ and $G_r(k, t)$ should be updated using Equation 2.11. However, if either $k = f$ or $k = t$, this means that there is a change in the internal cost of v_r and hence, all the gain values for all the moves of v_r from P_k to all the other parts in the partition should be recomputed using Equation 2.11.

It should be noted that partitioning algorithms existing in the literature lock the vertex moved, thus preventing the further moves of such vertices. In such algorithms, the gain updates mentioned in the item (3) should not be considered at all. Similarly, the cost and gain updates mentioned in the item (4) should be considered only for the unlocked vertices adjacent to the vertex moved. However, one of the proposed algorithms (to be discussed later) does not lock a vertex after it is moved, and associates an attribute, referred here as the *freedom value*, with each vertex. This freedom value is a function of the current gain of a vertex. Thus, an update in the gain of a vertex results in an update in the freedom value of that vertex. The gain updates in the item (3) and the cost and gain updates in the item (4) should be carried out for the vertex moved and all its neighbor vertices in the proposed algorithm.

Algorithm: Gain Update Algorithm

Input: a graph $G = (V, E)$ with N vertices, a K -way partition $\Pi = (P_1, \dots, P_K)$ of G ,
move of $v_m \in P_f$ to P_t

Output: updated costs and gains of neighbors of v_m

1. **for** each edge e incident to v_m **do**
 - 1.1. find $v_r \in P_k$ such that $e = \{v_m, v_r\}$ and $P_k \in \Pi$
 - 1.2. let $C_r(k, f) \leftarrow C_r(k, f) - c(e)$
 - 1.3. let $C_r(k, t) \leftarrow C_r(k, t) + c(e)$
 - 1.4. **if** $(f \neq k \wedge t \neq k)$ **then**
 - 1.4.1. update only $G_r(k, f)$
 - 1.4.2. update only $G_r(k, t)$
 - 1.5. **else** /* there is change in internal cost of neighbor vertex */
 - 1.5.1. update all $(K - 1)$ gains of v_r , i.e., all gains other than $G_r(k, k)$
 - 1.6. **endif**
 2. **endfor**
-

Figure 2.2. An algorithm for gain updates in a graph

2.4.3 Balance Conditions

It is possible that the total weight w_T of all the vertices is not a perfect multiple of the number of parts. Even if there is a partition where the part sizes are the same, the balance on the part sizes is broken with the first move. In addition, if the vertices do not have the same weight, then it is also a hard problem to divide these vertices into parts such that the sum of the pairwise differences between the part sizes is minimized. Therefore, some changes in the part sizes should be tolerable. This tolerance is established by means of imposing lower and upper bounds on the part sizes. These bounds constitutes the *balance condition*.

The main idea behind any balance condition should be that, during the course of the graph partitioning algorithm, there always exists at least one vertex to move without violating the balance condition and that the move is not exactly the opposite of the previous move [20]. This idea is good but it may be difficult to guarantee it.

Now, we define our balance condition: Let $G = (V, E)$ be a graph and $\Pi = (P_1, \dots, P_K)$ a K -way partition of G . Then, we have $b(k) \leq s(P_k) \leq B(k)$ for each P_k in Π by the definition of the graph partition problem. What remains

is to specify the values of these lower and upper bounds. We define

$$b(k) = \lfloor \frac{w_T}{K}(1 - \alpha) \rfloor \quad (2.15)$$

and

$$B(k) = \lceil \frac{w_T}{K}(1 + \alpha) \rceil \quad (2.16)$$

where α , ($0 < \alpha < 1$), is a constant. Thus, we allow a part size to be $100\alpha\%$ more or $100\alpha\%$ less than its value in a perfectly balanced partition. Moreover, during initial partitioning, we can increase α to relax the balance condition. We call a move *legal* if it does not violate the balance condition [31].

Chapter 3

HYPERGRAPH PARTITIONING

This chapter establishes the underlying concepts for Hypergraph Partitioning. It includes some preliminary concepts from hypergraph theory, the definition of the hypergraph partitioning problem, and the concepts related to the partitioning algorithms, which are examined in Chapter 4. We utilized the references [23, 38] for the definitions and notations.

3.1 Introduction

The applications of the Hypergraph Partitioning Problem can be listed exhaustively as follows:

- VLSI placement [4, 9, 11, 12, 28, 35, 36].
- VLSI routing [4, 28].
- VLSI circuit simulation [27, 37, 41].

3.2 Basic Concepts

A hypergraph $H = (V, E)$ consists of a finite set V of *vertices* (or *cells*) and a finite set $E \subseteq 2^V$ of *hyperedges* (or *nets*), where 2^V is the power set of the vertex set V . Each net e in E is a subset of V . The elements of a net e in E are called its *terminals*. We use the symbols u, v, v_1, v_2, \dots to represent the vertices and

the symbols e, e_1, e_2, \dots to represent the nets of a hypergraph unless otherwise specified.

Given a net e in E , we say that the net e is *incident to* the vertex v if $v \in e$, and that the terminals of the net e are *adjacent* or *neighbors*. If a net e is incident to a vertex v then we say that the net e is *on* the vertex v and the vertex v is *on* the net e . A net with two-terminals is called a *two-terminal net* and a net with more than two terminals is a *multi-terminal net*. Terminals are also called *pins*.

The *degree* of a vertex v_i in V is equal to the number of nets incident to v_i and is denoted by $d(v_i)$ or simply d_i . A vertex of degree 0 is called an *isolated vertex*. The degree of a net e in E is equal to the number of its terminals and is denoted by $|e|$. We assume that, for any net e in E , the degree $|e| \geq 2$.

A hypergraph $H = (V, E)$ has $|V| = N$ vertices and $|E| = M$ nets. Each vertex v in V has a positive integer weight $w(v)$ and each net e in E has a positive integer weight $c(e)$. The total number of terminals in H is denoted by p which can be calculated by the equation

$$p = \sum_{e \in E} |e|, \quad (3.1)$$

or

$$p = \sum_{v \in V} d(v). \quad (3.2)$$

Note that $M = \mathcal{O}(p)$ since every net is at least a two-terminal net. If we further assume that every vertex is contained in at least one net, namely, if the degree of each vertex is at least 1, then we have $N = \mathcal{O}(p)$. The latter assumption is not imposed unless otherwise specified.

A graph $G = (V, E)$ is also a hypergraph $H = (V, E)$ with the property that every net in H is a two-terminal net. That is, hypergraphs are generalization of graphs. If H is a graph, the total number p of terminals in H becomes equal to $2M$, where M is the number of nets in H .

Given a hypergraph $H = (V, E)$, we say that $\Pi = (P_1, \dots, P_K)$ is a *K-way partition* of H if each *part* P_k is a nonempty subset of the vertex set V , all the parts are pairwise disjoint, and the union of the K parts is equal to V . Formally, $\Pi = (P_1, \dots, P_K)$ is a *K-way partition* of $H = (V, E)$ if

1. $P_k \subset V, P_k \neq \emptyset$ for each $k \in \mathcal{N}(1, K)$,

2. $P_k \cap P_l = \emptyset$ for each $k, l \in \mathcal{N}(1, K)$ and $(k \neq l)$,
3. $\bigcup_{k=1}^K P_k = V$.

Note that the number K of parts in a partition of H is bounded above by the number of vertices in H .

Consider a K -way partition $\Pi = (P_1, \dots, P_K)$ of a hypergraph $H = (V, E)$ with N vertices, M nets, and p terminals. For the sake of completeness, we repeat some definitions from Chapter 2. Then,

- $s(P_k)$ denotes the *size* of the part P_k for $k \in \mathcal{N}(1, K)$. The size of the part P_k equals the sum of the weights of the vertices in P_k . That is,

$$s(P_k) = \sum_{v \in P_k} w(v). \quad (3.3)$$

- The total vertex weight w_T is defined as

$$w_T = \sum_{v \in V} w(v) = \sum_{k=1}^K s(P_k). \quad (3.4)$$

- The total net weight c_T is defined as

$$c_T = \sum_{e \in E} c(e). \quad (3.5)$$

- $E_k = \{e \in E \mid e \cap P_k \neq \emptyset \wedge e - P_k \neq \emptyset\}$ is the set of *external nets* of the part P_k for all $k \in \mathcal{N}(1, K)$. The set of external nets of a part P_k consists of those nets that have at least one terminal in P_k and at least one terminal in another part in the partition Π .
- $I_k = \{e \in E \mid e \cap P_k \neq \emptyset \wedge e - P_k = \emptyset\}$ is the set of *internal nets* of the part P_k for all $k \in \mathcal{N}(1, K)$. The set of internal nets of a part P_k consists of those nets that have all its terminals in P_k .
- $\delta_i(k) = |\{v \in e_i \mid v \in P_k\}|$ is the number of terminals of the net e_i that are present in the part P_k .
- If there are k parts such that a net e has at least one terminal in each of these parts, the net e is said to connect k parts in the partition.
- The nets that connect different parts in the partition Π , that is, the external nets, are said to *contribute to the cut* or *cross the cut*.

- The cost $c(\Pi)$ of the partition is also called the *cutsizes*. The cutsizes is the sum of the weights of all the nets contributing to the cut. That is,

$$\chi(\Pi) = c_T - \sum_{k=1}^K \sum_{e \in I_k} c(e). \quad (3.6)$$

Each net e crossing the cut contributes an amount of $c(e)$ to the cutsizes regardless of the number of parts that e connects. However, this is not the only possible definition of the cutsizes for hypergraphs. For example, if the net e connects k parts then e can contribute an amount of $(k-1)c(e)$ to the cutsizes. Note that Equation 3.6 reduces to Equation 2.4 when H is a graph.

- A K -way partition is also a *multiple-way partition*, and the partitioning operation is called *K -way partitioning* or *multiple-way partitioning*. If there are only two parts, i.e., $K = 2$, then Π is called also a *bipartition* or a *2-way partition*.
- We say that a partition is *balanced* if the parts have about the same size. A partition is *perfectly balanced* if the parts have exactly the same size. A perfectly balanced partition is highly unlikely in a multiple-way partitioning if the vertex weights are not equal.
- The average vertex degree D_v of the hypergraph H can be found by the equation

$$D_v = \frac{P}{N}. \quad (3.7)$$

The average net degree D_e of the hypergraph H can be found by the equation

$$D_e = \frac{P}{M}. \quad (3.8)$$

Hence, the following equation holds:

$$D_e M = D_v N \quad (3.9)$$

- The maximum (minimum) vertex degree of the hypergraph H is the maximum (minimum) of the set of the degrees of the vertices in H and is denoted by $D_{v,max}$ ($D_{v,min}$). The maximum (minimum) net degree of H is the maximum (minimum) of the set of the degrees of the nets in H and is denoted by $D_{e,max}$ ($D_{e,min}$).
- The maximum (minimum) vertex weight is the maximum (minimum) of the set of the weights of the vertices in H and is denoted by w_{max} (w_{min}). The maximum (minimum) net weight is the maximum (minimum) of the set of the weights of the nets in H and is denoted by c_{max} (c_{min}).

3.3 Hypergraph Partitioning Problem

A formal definition of the Hypergraph Partitioning Minimization Problem (HPP) is given below.

Problem: The Hypergraph Partitioning Minimization Problem.

Instance: A hypergraph $H = (V, E)$, a vertex weight function $w : V \rightarrow \mathcal{N}$, a net weight function $c : E \rightarrow \mathcal{N}$, a number $K \geq 2, K \in \mathcal{N}$, maximum and minimum part sizes $B(k) \in \mathcal{N}$ and $b(k) \in \mathcal{N}$, respectively, for $k \in \mathcal{N}(1, K)$.

Configurations: All K -way partitions $\Pi = (P_1, \dots, P_K)$.

Solutions: All feasible configurations, i.e., all K -way partitions $\Pi = (P_1, \dots, P_K)$ such that

$$b(k) \leq s(P_k) \leq B(k) \text{ for all } k \in \mathcal{N}(1, K)$$

Question: Find a solution such that the cutsizes

$$\chi(\Pi) = c_T - \sum_{k=1}^K \sum_{e \in I_k} c(e)$$

is minimum over all the solutions.

Intuitively, we are given a hypergraph $H = (V, E)$. Each vertex and each net have a positive weight. Each K -way partition $\Pi = (P_1, \dots, P_K)$ of the vertex set V into nonempty, pairwise disjoint parts P_k , (for $k \in \mathcal{N}(1, K)$), is a configuration. Given an upper bound $B(k)$ and a lower bound $b(k)$ on the size of each part P_k , we regard as solutions those partitions in which the size of each part P_k is in the range between $b(k)$ and $B(k)$. We are then asked to find the partition (or partitions) that has the minimum cutsizes over all the solutions.

The hypergraph partitioning minimization problem is NP-hard [13]. Since graphs are special versions of hypergraphs, GPP is a special version or a restricted version of HPP. Any partitioning algorithm that can produce a solution to HPP can produce a solution to GPP without any modifications in the algorithm. However, an algorithm for GPP may not be used for HPP. Some parts of the algorithm need to be altered.

Additional constraints [29] that can be imposed in HPP are itemized below.

- The number of parts in a partition is minimized provided that there are bounds on the part sizes.

- The total number of external nets of each part is bounded.
- A certain set of nets must contribute to the cut.
- A certain set of nets must not contribute to the cut.

The algorithms we investigated can be modified to handle these constraints without too much additional effort. We did not consider to meet these constraints, however.

3.4 Multiple-way Hypergraph Partitioning

3.4.1 Gain Concept

Let $H = (V, E)$ be a hypergraph with N vertices and $\Pi = (P_1, \dots, P_K)$ a K -way partition of H . Let $f, t \in \mathcal{N}(1, K)$ be two numbers (f represents the part *from* which a vertex is moved, and t represents the part *to* which the vertex is moved.) The *cost* $C_m(f, t)$ of a vertex v_m in P_f with respect to a part P_t (m for *moved* vertex) is defined as

$$C_m(f, t) = \begin{cases} \sum_{e \in E_m(f, t)} c(e) & \text{if } f \neq t \\ \sum_{e \in I_m(f, f)} c(e) & \text{otherwise} \end{cases} \quad (3.10)$$

where

$$E_m(f, t) = \{e_i \in E_f \mid v_m \in e_i \wedge \delta_i(t) = |e_i| - 1\} \quad (3.11)$$

is the subset of the set of external nets of the part P_f whose one terminal is v_m and all the other terminals lie in the part P_t , and

$$I_m(f, f) = \{e_i \in I_f \mid v_m \in e_i \wedge \delta_i(f) = |e_i|\} \quad (3.12)$$

is the subset of the set of internal nets of the part P_f whose one terminal is v_m and all the other terminals lie in the part P_f .

The *move gain* $G_m(f, t)$ of the vertex v_m in the part P_f with respect to the part P_t is given by the equation

$$G_m(f, t) = C_m(f, t) - C_m(f, f). \quad (3.13)$$

That is, the gain $G_m(f, t)$ of a vertex v_m in the part P_f with respect to the part P_t is the difference between the sum of the weights of the nets whose the

only terminal in P_f is v_m and all the other terminals are in P_t , and the sum of the weights of the nets that include v_m and that have all their terminals in P_f . The gain of a vertex represents the decrease that results in the cutsize when the vertex is moved. The gain of a vertex with respect to the part where the vertex is present is zero.

In the K -way partition, each vertex has K costs. These costs constitute the *cost vector* of the vertex. For each vertex v_m in P_f , the entry $C_m(f, f)$ is the *internal cost* of v_m and the other $(K - 1)$ entries are the *external costs* of v_m with respect to each part other than P_f in Π .

The *cutstate* of a net indicates whether the net contributes to the cut or not. A net is *critical* if there exists a vertex on it such that the vertex would change the cutstate of the net if it is moved. Specifically, a net e_i is critical if and only if either there exists a part P_k in the partition such that $\delta_i(k) = |e_i|$ or there exist two different parts P_k and P_l in the partition such that $\delta_i(k) = 1$ and $\delta_i(l) = |e_i| - 1$. The gain of a vertex in a hypergraph depends only on the critical nets incident to the vertex. Figure 3.1 illustrates the pseudocode for the algorithm which computes the initial cost vectors of the vertices in a hypergraph assuming an initial feasible partition. Note that the computation of the initial gains of the vertices can be done easily by using Equation 3.13 provided that the initial costs are given.

3.4.2 Effects of a Vertex Move

Let $H = (V, E)$ be a hypergraph and $\Pi = (P_1, \dots, P_K)$ a K -way partition of H . Let $f, t \in \mathcal{N}(1, K)$ be two numbers. Consider the move of the vertex v_m in the part P_f to the part P_t , where $f \neq t$. We now give the effects of this move.

1. Effect on Cutsizes : The cutsizes should be updated by the equation

$$\chi(\Pi) \leftarrow \chi(\Pi) - G_m(f, t) \quad (3.14)$$

where $G_m(f, t)$ is the gain of the vertex v_m before the move. Hence, the decrease in the cutsizes is equal to $G_m(f, t)$, which is expected by the definition of the gain concept.

2. Effect on Part Sizes : The part size of the part P_f decreases and the part size of the part P_t increases by the move of v_m . Hence, The following changes

Algorithm: Initial Cost Computation Algorithm

Input: a hypergraph $H = (V, E)$ with N vertices, a K -way partition $\Pi = (P_1, \dots, P_K)$ of H

Output: vertices in V with all costs computed

1. **for** each vertex v_i , where $v_i \in P_f$ and $i \in \mathcal{N}(1, N)$, **do**
 - 1.1. **for** each part number t , where $t \in \mathcal{N}(1, K)$, **do**
 - 1.1.1. let $C_m(f, t) \leftarrow 0$ /* initialize the cost */
 - 1.2. **endfor**
 - 1.3. **for** each net e_j incident to v_i **do**
 - 1.3.1. **if** $(\delta_j(f) = 1)$ **then**
 - 1.3.1.1. look for the part P_t such that $\delta_j(t) = |e_j| - 1$
 - 1.3.1.2. **if** P_t is found **then**
 - 1.3.1.2.1. let $C_i(f, t) \leftarrow C_i(f, t) + c(e_j)$
 - 1.3.1.3. **endif**
 - 1.3.2. **else if** $(\delta_j(f) = |e_j|)$ **then**
 - 1.3.2.1. let $C_i(f, f) \leftarrow C_i(f, f) + c(e_j)$
 - 1.3.3. **endif**
 - 1.4. **endfor**
 2. **endfor**
-

Figure 3.1. An algorithm for initial cost computation in a hypergraph

in the parts sizes should be done.

$$s(P_f) \leftarrow s(P_f) - w(v_m) \quad (3.15)$$

$$s(P_t) \leftarrow s(P_t) + w(v_m) \quad (3.16)$$

3. Effect on Vertex Moved : There is no change in the entries of the cost vector of the vertex v_m , which is moved. The only change in the cost vector is in the interpretation of some entries. The entry $C_m(f, f)$ was the internal cost of v_m before the move and the entry $C_m(f, t)$ was the external cost of v_m with respect to the part P_t before the move. After the move, the entry $C_m(t, t)$ becomes the internal cost of v_m and the entry $C_m(t, f)$ becomes the external cost of v_m to the part P_f where $C_m(f, f)$ before the move is equal to $C_m(t, f)$ after the move and $C_m(f, t)$ before the move is equal to $C_m(t, t)$ after the move. However, since the internal cost of v_m is changed, the gains of v_m to every part (other than P_t) in Π must be recomputed using Equation 3.13.

4. Effect on Neighbor Vertices : Consider the move of the vertex v_m from the part P_f to the part P_t . The cutstate of a net that is not incident to v_m

cannot change by the move. The cutstate of a net that is incident to v_m can change by the move if the net is critical. Moreover, a net which is not critical either before or after the move cannot affect the gains of any of the vertices to which the net is incident. We now derive the updates to be done both before and after the move.

1. Using the definition of the criticality of a net, a net e_i incident to v_m is critical before the move of v_m if and only if one (or more) of the following cases holds:
 - 1.a. $\delta_i(f) = 0$ and there exists a part P_k such that $\delta_i(k) = |e_i|$,
 - 1.b. $\delta_i(f) = 1$ and there exists a part P_k such that $\delta_i(k) = |e_i| - 1$,
 - 1.c. $\delta_i(f) = |e_i| - 1$ and there exists a part P_k such that $\delta_i(k) = 1$,
 - 1.d. $\delta_i(f) = |e_i|$.

Before the move of $v_m \in P_f$, it must be valid that $\delta_i(f) \geq 1$. Thus, the case (1.a) is not possible at all, and can be eliminated.

2. Using the definition of the criticality of a net, a net e_i incident to v_m is critical after the move of v_m if and only if one (or more) of the following cases holds:
 - 2.a. $\delta_i(t) = 0$ and there exists a part P_k such that $\delta_i(k) = |e_i|$,
 - 2.b. $\delta_i(t) = 1$ and there exists a part P_k such that $\delta_i(k) = |e_i| - 1$,
 - 2.c. $\delta_i(t) = |e_i| - 1$ and there exists a part P_k such that $\delta_i(k) = 1$,
 - 2.d. $\delta_i(t) = |e_i|$.

After the move of $v_m \in P_f$, it must be valid that $\delta_i(t) \geq 1$. Thus, the case (2.a) is not possible at all, and can be eliminated.

Considering the cases above for the net e_i , it reveals that before the move of v_m in the item (1), the part P_k is either identical to the part P_t , to which v_m is moved, or not. Hence, if P_k is identical to P_t ($t = k$), then the following table summarizes the resulting cases with the move:

Before the move of v_m			After the move of v_m		
Case	$\delta_i(f)$	$\delta_i(t)$	$\delta_i(f)$	$\delta_i(t)$	Case
1.b	1	$ e_i - 1$	0	$ e_i $	2.d
1.c	$ e_i - 1$	1	$ e_i - 2$	2	-
1.d	$ e_i $	0	$ e_i - 1$	1	2.b

Hence, the case (1.b) is equivalent to the case (2.d), and the case (1.d) is equivalent to the case (2.b). Only the cases (1.c), (1.d), (2.c), and (2.d) should then be considered during the cost and gain updates. But, if P_k is not identical to P_t ($t \neq k$), then the following table summarizes the resulting cases with the move:

Before the move of v_m				After the move of v_m			
Case	$\delta_i(f)$	$\delta_i(t)$	$\delta_i(k)$	$\delta_i(f)$	$\delta_i(t)$	$\delta_i(k)$	Case
1.b	1	0	$ e_i - 1$	0	1	$ e_i - 1$	2.b
1.c	$ e_i - 1$	0	1	$ e_i - 2$	1	1	-
1.d	$ e_i $	0	0	$ e_i - 1$	1	0	2.b

Hence, the case (1.b) is equivalent to the case (2.b), and the case (1.d) is equivalent to the case (2.b). Only the cases (1.c), (1.d), (2.c), and (2.d) should then be considered during the cost and gain updates.

Considering the cases above, it reveals that after the move of v_m in the item (2), the part P_k is either identical to the part P_f , from which v_m is moved, or not. Hence, if P_k is identical to P_f ($f = k$), then the following table summarizes the resulting cases with the move:

Before the move of v_m			After the move of v_m		
Case	$\delta_i(f)$	$\delta_i(t)$	$\delta_i(f)$	$\delta_i(t)$	Case
1.d	$ e_i $	0	$ e_i - 1$	1	2.b
-	2	$ e_i - 2$	1	$ e_i - 1$	2.c
1.b	1	$ e_i - 1$	0	$ e_i $	2.d

Hence, the case (2.b) is equivalent to the case (1.d), and the case (2.d) is equivalent to the case (1.b). Only the cases (1.c), (1.d), (2.c), and (2.d) should

then be considered during the cost and gain updates. But, if P_k is not identical to P_f ($f \neq k$), then the following table summarizes the resulting cases with the move:

Before the move of v_m				After the move of v_m			
Case	$\delta_i(f)$	$\delta_i(t)$	$\delta_i(k)$	$\delta_i(f)$	$\delta_i(t)$	$\delta_i(k)$	Case
1.b	1	0	$ e_i - 1$	0	1	$ e_i - 1$	2.b
-	1	$ e_i - 2$	1	0	$ e_i - 1$	1	2.c
1.b	1	$ e_i - 1$	0	0	$ e_i $	0	2.d

Hence, the case (2.b) is equivalent to the case (1.b), and the case (2.d) is equivalent to the case (1.b). Only the cases (1.c), (1.d), (2.c), and (2.d) should then be considered during the cost and gain updates. Therefore, after examining the data tabulated in the last four tables, we recognize that the move of v_m from P_f to P_t affects the vertices on the nets satisfying any number of the four cases (1.c), (1.d), (2.c), and (2.d).

In the case (1.d), before the move of v_m , the net e_i is an internal net of the part P_f . The move makes e_i cross the cut. Hence, the contribution of e_i to the internal cost of each terminal of e_i should be eliminated before the move. Note that if locking is used, this case can occur at most once for e_i in a pass involving the move of each vertex in a hypergraph at most once.

In the case (1.c), before the move of v_m , the net e_i contributes to the external cost of a vertex $v_r \in e_i$ which is not present in the part P_f . Since the move causes e_i to make no contribution to the external cost of v_r any more, the contribution should be eliminated before the move. Note that if locking is used, this case can occur at most three times for e_i in a pass involving the move of each vertex in a hypergraph at most once.

In the case (2.d), after the move of v_m , the net e_i becomes an internal net of the part P_t . The move removes e_i from the cut. Hence, e_i should be made to contribute to the internal costs of all its terminals. Note that if locking is used, this case can occur at most once for e_i in a pass involving the move of each vertex in a hypergraph at most once.

In the case (2.c), after the move of v_m , the net e_i becomes critical and so contributes to the external cost of a vertex $v_r \in e_i$ which is not present in the

part P_i . Note that if locking is used, this case can occur at most three times for e_i in a pass involving the move of each vertex in a hypergraph at most once.

The algorithm in Figure 3.2 calculates the changes in the costs and gains of the neighbor vertices that result from the vertex moved in the light of the above discussion. As mentioned in Section 2, any update in the costs and gains of a vertex should be forwarded to an update in the freedom value of the vertex if the algorithm does not use locking.

3.4.3 Balance Conditions

The balance conditions are the same as given in Section 2.4.3. However, the value of the constant α can be different for a hypergraph. We specify the exact values of this constant when we present the experiments in Section 5.

Algorithm: Gain Update Algorithm

Input: a hypergraph $H = (V, E)$ with N vertices, a K -way partition $\Pi = (P_1, \dots, P_K)$ of H , move of $v_m \in P_f$ to P_t

Output: updated costs and gains neighbors of v_m

1. **for** each net e_j incident to v_m **do**
 - 1.1. **if** $(\delta_j(f) = |e_j|)$ **then** /* before the move (the case 1.d) */
 - 1.1.1. **for** each vertex $v_r \in e_j$, where $v_r \in P_f$ and $m \neq r$ **do**
 - 1.1.1.1. let $C_r(f, f) \leftarrow C_r(f, f) - c(e_j)$
 - 1.1.1.2. update all $(K - 1)$ gains of v_r , i.e., all gains other than $G_r(f, f)$
 - 1.1.2. **endfor**
 - 1.2. **else if** $(\delta_j(f) = |e_j| - 1)$ **then** /* before the move (the case 1.c) */
 - 1.2.1. find the vertex $v_r \in e_j$, $v_r \in P_k$ such that $m \neq r$ and $f \neq k$
 - 1.2.2. let $C_r(k, f) \leftarrow C_r(k, f) - c(e_j)$
 - 1.2.3. update only $G_r(k, f)$
 - 1.3. **endif**
 - 1.4. let $\delta_j(f) \leftarrow \delta_j(f) - 1$ /* updates to indicate the move */
 - 1.5. let $\delta_j(t) \leftarrow \delta_j(t) + 1$
 - 1.6. **if** $(\delta_j(t) = |e_j|)$ **then** /* after the move (the case 2.d) */
 - 1.6.1. **for** each vertex $v_r \in e_j$, where $v_r \in P_t$ and $m \neq r$ **do**
 - 1.6.1.1. let $C_r(t, t) \leftarrow C_r(t, t) + c(e_j)$
 - 1.6.1.2. update all $(K - 1)$ gains of v_r , i.e., all gains other than $G_r(t, t)$
 - 1.6.2. **endfor**
 - 1.7. **else if** $(\delta_j(t) = |e_j| - 1)$ **then** /* after the move (the case 2.c) */
 - 1.7.1. find the vertex $v_r \in e_j$, $v_r \in P_k$ such that $m \neq r$ and $t \neq k$
 - 1.7.2. let $C_r(k, t) \leftarrow C_r(k, t) + c(e_j)$
 - 1.7.3. update only $G_r(k, t)$ and anything depending on $G_r(k, t)$
 - 1.8. **endif**
 2. **endfor**
-

Figure 3.2. An algorithm for gain updates in a hypergraph

Chapter 4

PARTITIONING ALGORITHMS

This chapter first gives the basic ideas of the local search technique since we restricted our attention to only local search partitioning algorithms. It also contains the previous graph and hypergraph partitioning algorithms. After this background information, the proposed algorithms along with the data structures come. The time and space complexity analysis of the algorithms is also discussed.

4.1 Local Search

A combinatorial optimization problem is either a minimization problem or a maximization problem and consists of the following three features:

- a set of instances, (an instance is obtained by specifying particular values for all the problem parameters),
- for each instance, a finite set of feasible configurations,
- a function that assigns a cost to each instance and each solution (or feasible configuration).

The goal is to find a solution of minimum cost or maximum cost, that is, the optimal solution [13].

Many combinatorial optimization problems are *hard* in the sense that they are NP-hard or harder. There are no deterministic known polynomial time

algorithms to find the optimal solution to any of those hard problems. The algorithms employing the complete enumeration techniques are not reasonable to use the complexity of these techniques is usually exponential in the size of the problem and hence, they require a great amount of time to find the optimal solution for even very small problem instances. As a result, *heuristic* algorithms (or heuristics) that run in a low-order polynomial time have been employed to obtain good solutions to these hard problems, where by a good solution, we mean a solution that is hopefully close to the optimal solution to the problem.

The methods used for designing heuristic algorithms tend to be rather problem specific. *Local search* is one of the few general approaches to solving hard combinatorial optimization problems. Local search is usually based on trial and error. All the algorithms that we consider in this study are local search algorithms.

The first choice that must be made in order to derive a local search algorithm for a combinatorial optimization problem is the choice of a *neighborhood structure*. This structure specifies a neighborhood for each solution, that is, a set of solutions that are in some sense close to that solution. For example, our algorithms use a move-neighborhood structure as explained in Section 4.2.

The second choice is the choice of devising an algorithm to generate an initial solution to the problem. The algorithm must be a polynomial time algorithm and the initial solution must be a feasible configuration although it can be generated randomly and can have a very poor cost. However, there are algorithms that allow infeasible configurations to occur but they penalize their occurrence by utilizing certain measures.

Starting from some given initial solution, a local search algorithm tries to find a better solution which is a neighbor of the first. If a better neighbor is found, a search starts for a better neighbor of that one, and so on. Since the set of solutions is finite, this search must halt, that is, the local search algorithm must end at a *locally optimum* solution, which does not have a better neighbor solution. Local search algorithms are also called *iterative improvement algorithms* because they iteratively improve an initial solution in search of a locally optimal solution. In fact, we use these algorithms to find the global optimum but this goal seems to be impossible to reach because of the hardness of the problem.

During the search for a locally optimal solution, we use two more algorithms: one polynomial-time algorithm is needed to modify the current solution so as to generate a new solution in the neighborhood of the current solution. The other polynomial-time algorithm is needed to find the cost of a given solution. The number of iterative steps to arrive at a locally optimal solution is not known. For some local search algorithms associated with certain problem instances, the number of steps can be exponentially dependent on the size of the problem on these instances [18].

Assume that s denotes a solution to a certain combinatorial optimization problem and that $N(s)$ denotes the neighborhood of s . A neighbor solution in $N(s)$ can usually be found in three different ways [8]:

1. in the *first descent* method, the neighbor solution is the first solution in $N(s)$ that has a better cost than that of s ,
2. in the *steepest descent* method, all the solutions in $N(s)$ are examined and the neighbor solution is chosen to be the one with the best cost in $N(s)$,
3. in the random descent method, the neighbor is randomly chosen among the solutions in $N(s)$.

Neighbor selection in the steepest descent method takes more time than the first descent method. The random descent method comes in between on the average. In our algorithms, we use the steepest descent method. However, the search time to find a neighbor solution is significantly decreased by using appropriate data structures. A neighbor solution is found without examining all the solutions at each iteration step.

Modification of a solution s to obtain another solution s' in $N(s)$ is called a *move*. Suppose that the problem is a minimization problem and that $\chi(s)$ denotes the cost of s . In a move, if $\chi(s') < \chi(s)$, then we obtained a solution with a better cost. This is a *downhill* move. On the other hand, if $\chi(s') > \chi(s)$, then we have a solution with a worse cost. This is a *uphill* move. Allowing uphill moves is an attempt to escape from being trapped in a poor locally optimal solution.

We now give a general local search algorithm (in Figure 4.1) that subsumes the algorithms investigated in this work. Note that our partitioning problems

Algorithm: A Local Search Algorithm

Input: a combinatorial optimization problem

Output: a locally optimum solution to the problem

1. generate initial solution s
 2. find cost $\chi(s)$ of initial solution s
 3. **repeat**
 - 3.1. **for** K_1 iterations (vertex moves) **do**
 - 3.1.1. select the best neighbor s' in $N(s)$
/* The solutions in $N(s)$ are obtained by vertex moves */
/* prevent selection of previously selected solutions as much as possible */
 - 3.1.2. let $s \leftarrow s'$
 - 3.2. **endfor**
 - 3.3. find the subsequence of vertex moves from the sequence with K_1 vertex moves such that those vertex moves in the subsequence enable us to arrive at the best solution in this pass
 - 3.4. **if** the subsequence is not empty **then**
 - 3.4.1. execute the vertex moves in the subsequence
/* if it is empty then local optimum is found */
 - 3.4. **endif**
 4. **until** a locally optimum solution has been found
-

Figure 4.1. A general local search algorithm

are minimization problems and we partition graphs and hypergraphs. This is why we use the term vertex move in the local search algorithm. The constant K_1 depends on the partitioning algorithm. For example, it is equal to the number of vertices of the input graph in Kernighan-Lin algorithm. In the algorithm in Figure 4.1, we move from a solution to a neighboring solution by a sequence of (at most K_1) vertex moves. When we regard each move in the algorithm in Figure 4.1 as a sequence of (at most K_1) vertex moves, i.e., a move is not equivalent to a vertex move but to a sequence of vertex moves, this algorithm employs only downhill moves. The algorithm arrives at a locally optimal solution with respect to the neighborhood structure that uses *this* move definition to obtain neighboring solutions. Since we do not know how many vertex moves we should perform to get the best improvement in a pass, (a pass is a single iteration of the repeat loop in the algorithm in Figure 4.1), we execute a sequence of K_1 vertex moves and then determine the subsequence that contains the vertex moves yielding the best improvement in the pass. That is, we determine the set of the vertex moves which constitutes one move. Note

that the number of the vertex moves in the subsequence computed in the step 3.3 in the algorithm in Figure 4.1 is usually different for each pass.

However, when we regard each move in the algorithm in Figure 4.1 as a vertex move, this algorithm employs downhill moves as well as uphill moves since, in step 3.1.1, there may be no neighbors that have a lower cost than that of s and hence, the neighbor with the smallest cost increase is selected, which corresponds to an uphill move. After performing a sequence of K_1 vertex moves in a pass, we find the subsequence of vertex moves whose execution yields the best improvement in the pass. If the subsequence is empty, then a locally optimum solution is found. On the other hand, if there is a cost improvement, we find the new solution and proceed to another pass on this solution. The first move in the subsequence is also the first move among the K_1 moves in the recorded sequence and the moves in the subsequence must be executed in the same order as in the record.

The algorithms we considered in this work have the same structure as the local search algorithm in Figure 4.1. They all perform a number of passes until a locally optimum solution is found. You should notice that a solution is a locally optimum solution in these algorithms with respect to the neighborhood structure that employs the moves each of which contains all the vertex moves performed in a pass. That is, each pass corresponds to exactly one move in this neighborhood structure. However, all the moves performed in a pass are vertex moves. Henceforth by a move, we mean a vertex move, and, by a move-neighborhood structure, we mean the neighborhood structure with respect to vertex moves instead of the moves containing a sequence of vertex moves.

4.2 Neighborhood Structure

The algorithms that we investigate in this study are all based on the *move-neighborhood* structure. A partition Π has a neighbor partition Π' if Π' can be obtained from Π by moving a vertex from one part to another in Π . Formally, let $H = (V, E)$ be a hypergraph with N vertices, and Π and Π' be two K -way partitions of H . Then, the partition $\Pi' = (P_1, \dots, P_k - \{v\}, \dots, P_l \cup \{v\}, \dots, P_K)$ is a neighbor of the partition $\Pi = (P_1, \dots, P_k, \dots, P_l, \dots, P_K)$ for some $k, l \in \mathcal{N}(1, K)$ and $(k \neq l)$, and for some vertex v in $P_k \in \Pi$. The partition Π has at most $K(K-1)(N/K) = N(K-1)$ neighbors if each part

has N/K vertices.

4.3 Previous Approaches

We now review the local search partitioning algorithms existing in the literature. These heuristics and the proposed heuristics carry the basic minimization feature of the Kernighan-Lin algorithm and hence, they are called Kernighan-Lin style algorithms. We do not consider other types of algorithms that have been used for the HPP or the GPP.

Kernighan-Lin's Approach :

This algorithm (or heuristic) [19] was originally proposed for the graph bipartitioning problem. Kernighan-Lin (KL) Algorithm is also a local search algorithm and has become the basis of many graph and hypergraph partitioning algorithms. Our algorithms are also partially based on KL algorithm.

KL algorithm uses a swap-neighborhood structure. In this neighborhood structure, two partitions are neighbors if one partition can be obtained from another by swapping two vertices between different parts in one of the partitions. Formally, let $G = (V, E)$ be a N -vertex graph and Π, Π' two K -way partitions of G . Then, the partition $\Pi = (P_1, \dots, P_k, \dots, P_l, \dots, P_K)$ and the partition $\Pi' = (P_1, \dots, (P_k - \{v\}) \cup \{u\}, \dots, (P_l - \{u\}) \cup \{v\}, \dots, P_K)$ are neighbors for some $k, l \in \mathcal{N}(1, K)$ and for some vertices $v \in P_k, u \in P_l$. The partition Π has $(K(K-1)/2)(N/K)^2$ neighbors if each part has N/K vertices. A solution has more neighbors in a swap-neighborhood structure than those in a move-neighborhood structure.

This algorithm assumes that every vertex has the same weight. It works as follows: first, an initial partition is generated. We then determine the vertex pair whose swap results in the largest swap gain, i.e., the largest decrease in the cutsizes or the smallest increase (if no decrease is possible). This pair is tentatively interchanged and locked. The locking prohibits them from taking part in any further swaps. After that, we look for a second pair of vertices whose interchange improves the cutsizes the most, and do the same for this pair also. We continue in this way, but we keep a record of all tentative swaps and their gains. We finish when all the vertices are locked. At this time, we have interchanged both parts and are back to the original (initial) cutsizes.

Starting with the first swap in the record, we perform the subsequence of swaps which result in the smallest cutsize. The following pass begins with unlocking all vertices and proceeds in the same manner. These passes are repeated until there is no improvement in the cutsize which corresponds to a locally minimum partition.

This algorithm allows uphill moves to reduce the danger of being trapped in a poor local minimum. This feature of the algorithm enables the algorithm produce better partitions than the algorithms that employ only downhill moves. Also, this algorithm is quite robust. We can accommodate additional constraints such as partitioning into unequal-sized parts, required parts for certain vertices. However, it has some disadvantages. The algorithm handles only identical vertex weights. This restriction is not suitable for real applications. The algorithm has a complexity of $\mathcal{O}(N^2 \log N)$ per pass for a graph with N vertices. It has been observed that the algorithm performs poorly on sparse graphs and on some special types of graphs such as ladder graphs [6]. Furthermore, the quality of the solution generated by this algorithm strongly depends on the initial partition. However, this feature is common to all the local search partitioning algorithms.

Schweikert-Kernighan's Approach :

This approach [32] is an enhancement to KL algorithm in order to handle hypergraphs easily and correctly. Before this study, KL algorithm were applied to hypergraph problem instances by first representing the hypergraph in terms of a graph.

Feduccia-Mattheyses' Approach :

Feduccia-Mattheyses (FM) Algorithm [12] was originally proposed for the hypergraph partitioning problem but it can be applied to the graph partitioning problem equally well. This algorithm introduces the move-neighborhood structure instead of the swap-neighborhood structure. In addition, an efficient data structure called the bucket list data structure is proposed. This data structure helps to sort the vertices with respect to their move gains in time linear in the number of the vertices and keep the vertices in a sorted order according to their move gains during the partitioning iterations. Moreover, it also reduces the time complexity of the KL algorithm to linear in the number of the vertices and the edges (or the size of the hypergraph). Because of these features of FM algorithm, many following algorithms are based on this

algorithm.

Krishnamurthy's Approach :

KL and FM algorithms choose arbitrarily between vertices that have equal gain and equal weight. Krishnamurthy (KR) algorithm [20] introduces more look-ahead into the gain computation so that we can distinguish between such vertices with respect to the gains they make possible in later moves [23]. KR algorithm is a bipartitioning algorithm and generalizes the gain concept in KL and FM algorithms. In KR algorithm, each vertex has more than one gain, called level gains. The first level gain of a vertex is the same as its gain in KL and FM algorithms.

Sanchis' Approach :

Sanchis (SN) algorithm [31] is the generalization of KR algorithm for direct multiple-way hypergraph partitioning. Since graphs are special cases of hypergraphs, SN algorithm can also be used for graph partitioning. SN algorithm exploits the local minimization technique of KL algorithm, the move-neighborhood structure, balance condition, and bucket list data structure of FM algorithm, and the level gain approach of KR algorithm.

Vijayan's Approach :

Vijayan (VI) algorithm [40] is a direct multiple-way hypergraph partitioning algorithm similar to SN algorithm with the following minor exception. SN algorithm locks a vertex as soon as it moves but VI algorithm allows a vertex to reside in each part once and then it locks the vertex.

4.4 Bipartitioning versus Multiple-way Partitioning

When designing a multiple-way partitioning algorithm, we either start from scratch or adapt a 2-way partitioning algorithm to a multiple-way algorithm. Adapting a 2-way partitioning algorithm to a multiple-way algorithm is originally proposed in Kernighan-Lin [19]. This adaptation can be done in two ways. Both of these ways involve the repeated uses of a 2-way partitioning algorithm.

In Partitioning by Recursive Bipartition (PRB) [19], we first create a 2-way

partition of the given graph. Then, we perform 2-way partitioning on each of these two parts. By repeatedly bipartitioning a part obtained in the previous partitioning step, we can obtain as many parts as required. But, if the number of parts required is not a power of 2, then the partitioning becomes difficult because at each bipartitioning step, the part sizes must be bounded in such a way that the final part sizes satisfy a particular balance condition. This algorithm also suffers from a serious drawback. Partitioning at a particular level of hierarchy ignores connections to the vertices in the other parts. A partition at an earlier step biases a partition at a later step. Besides, the first partitioning tries to minimize the cutsize and hence, tends to maximize the internal connections of the parts. However, this makes further bipartitioning of these parts more difficult. PRM algorithm can be used to improve the initial partitions that are generated randomly.

In Partitioning by Pairwise Min-cut (PPM) [19], we first create a direct multiple-way initial partitioning of the given graph. Then, for each pair of parts, we apply a bipartitioning algorithm to reduce the cutsize between these pairs. Passes are performed until there is no improvement in the cutsize between the parts in each pair. This method produce better partitions than PRM does. PPM algorithm can be used to improve the partitioning results of a multiple-way partitioning algorithm. The disadvantage of this method is that this method still needs an initial multiple-way partition which must be generated by other means.

In PRB and PPM, any of KL algorithm, FM algorithm or KR algorithm can be used as the bipartitioning algorithm. We can generate an initial bipartition randomly, next convert this initial bipartition to multiple-way partition with PRB, and then improve this multiple-way partition with PPM.

In direct multiple-way partitioning [31], we start from scratch, i.e., we do not use a bipartitioning algorithm. At each step in a pass, a vertex in a part can move into any of the other parts in the partition. Note that only SN and VI algorithms are direct multiple-way partitioning algorithms. All other heuristics were originally proposed for bipartitioning and can be used as bipartitioning algorithms in either PRM or PPM approaches for multiple-way partitioning. Direct multiple-way partitioning algorithms are capable of handling partitions involving an arbitrary number of parts. In this work, we propose novel direct multiple-way partitioning algorithms for both graphs and hypergraphs.

4.5 Data Structures

We now provide the data structures that are common to all of the algorithms proposed and implemented in this work. Instead of giving the details of the data structures, we enumerate the type of information that is inserted into or extracted from the data structures during partitioning. This way of presenting the data structures is taken from Krishnamurthy [20]. We do this enumeration by listing two sets of operations such that the operations in one of the sets are primitive, i.e., we should be capable of performing them in constant time, and the operations in the other set are not primitive.

Given a graph $G = (V, E)$ (hypergraph $H = (V, E)$) with N vertices and M edges, and $\Pi = (P_1, \dots, P_K)$ a K -way partition of G (H). Suppose that $i \in \mathcal{N}(1, N)$, $j \in \mathcal{N}(1, M)$, and $k \in \mathcal{N}(1, K)$. The primitive operations are itemized below:

- given a vertex v_i , return its degree d_i and its weight $w(v_i)$,
- given an edge e_j , return its weight $c(e_j)$,
- return the total vertex weight w_T ,
- given a part P_k , return its upper bound $B(k)$, lower bound $b(k)$, size $w(P_k)$, and number of vertices in P_k ,
- given a vertex v_i , return the part P_k such that $v_i \in P_k$,
- given a vertex v_i , determine whether v_i is locked or not,
- given a vertex v_i , return the number n_i of moves that v_i has done up to a given point during partitioning,
- given a vertex v_i , return its cost with respect to a given part,
- given a vertex v_i , determine whether its move to a part is legal or not,
- given a bucket array, return a vertex with the maximum gain, a vertex with the minimum gain, and the number of non-empty buckets,
- given a vertex v_i , insert it into a bucket list according to its move gain,
- given an iteration step in a pass, obtain the vertex that is moved at this step and such information about it as its move gain, its source and destination parts,

The non-primitive operations are itemized below:

- given a vertex v_i , find the edges (nets) incident to v_i (the complexity of this step is $\mathcal{O}(d_i)$),
- given a edge (net) e_j , find its end vertices (terminals) (the complexity of this step is $\mathcal{O}(|e_j|)$, which is constant for graphs),
- given a vertex v_i , delete it from a bucket list (the complexity of deleting a vertex from a bucket array depends on the size of the array),

A bucket list for a move direction is depicted in Figure 4.2. For K -way partitioning, there are $K(K - 1)$ bucket arrays each of which corresponds to a move direction. Each bucket array has such a size that the move with the maximum possible gain (or freedom value) and the move with the minimum possible gain (or freedom value) can be inserted into the bucket. Each bucket points to a doubly linked list (bucket list) which contains the moves (or vertices) having the same gain proportional to the index of the bucket. Each bucket array has a maximum index pointer (`max_inx` in Figure 4.2) pointing to the bucket list that contains the moves with the maximum move gain in the move direction of the array. The search time for a move with the maximum move gain in the move direction of the bucket array is made constant by this index pointer. A move can be inserted into a bucket list in constant time because the insertion is made to the head of the bucket list connected to the bucket with the index calculated according to the gain of the move. After an insertion to a bucket list in a bucket array, the maximum index pointer of the array should be updated if the gain of the inserted move creates a bucket index larger than the current maximum index pointer. This update operation of the maximum index pointer requires constant time since it only involves an assignment operation. The deletion of a move from a bucket list in a bucket array also takes constant time but the update of the maximum index pointer of the array causes the deletion operation to have a worst-case time complexity proportional to the size of the array since the bucket list, from which a deletion is performed, may be the bucket list pointed to by the maximum index pointer, and it may become empty after the deletion and hence, the maximum index pointer is required to be updated to point to the next non-empty bucket list in the array. In the worst-case, this update operation involves a scan down from one bucket at the top of the array to another at the bottom. The minimum

index pointer (`min_inx` in Figure 4.2) is only helpful during the update of the maximum index pointer. It is not strongly required.

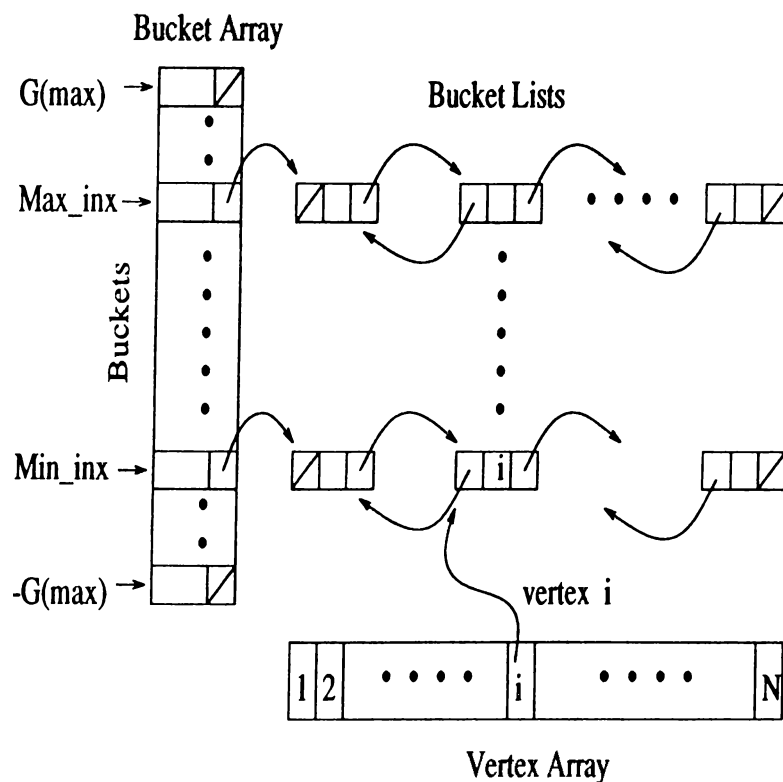


Figure 4.2. Bucket data structure for a part in a given partition

4.6 Reading Hypergraphs and Graphs

Suppose that $H = (G, E)$ is a hypergraph with N vertices, M nets, and p terminals to be partitioning into K parts. We store a hypergraph in the format given in Section A. Then, the hypergraph H can be read in $\mathcal{O}(N + MK + p)$ time. This time also includes the time to initialize the data structure keeping the number of terminals of each net in the parts. Also, since $M = \mathcal{O}(p)$, the reading time becomes $\mathcal{O}(N + pK)$.

Suppose that $G = (V, E)$ is a graph with N vertices and M edges to be partitioned into K parts. We store a graph in the format given in Section A. Then, the graph G can be read in $\mathcal{O}(M + N)$ time.

When reading the input hypergraph (or graph), we determine many properties of the hypergraph (or graph) such as the number of vertices, the number of nets (or edges), the maximum vertex degree, the maximum vertex weight, the maximum net (or edge) weight, and so on. These properties are used later in the partitioning algorithms.

4.7 Initial Partitions

Our partitioning algorithms like all the other local search partitioning algorithms require an initial solution. Usually, in partitioning problems, initial partitions are generated randomly. That is, vertices are assigned to parts randomly. The only constraint is to produce a feasible initial partition, the one that does not violate the balance condition. In general, the quality of the final partition of a partitioning algorithm depends strongly on the quality of the initial partition.

We now give an algorithm in Figure 4.3 to generate an initial partition for a K -way partitioning algorithm. Our algorithms use this initial partitioning algorithm. This algorithm can be used for both hypergraph partitioning algorithms and graph partitioning algorithms with very minor modifications which are shown in the algorithm. We assume that the weights of vertices do not differ considerably from one another, and that the partition becomes feasible when $\alpha < 1$. This restriction is due to the requirement that each part in the partition be nonempty. If there is very large differences among the weights of vertices, then α values may be defined to be different for each part. In our case, the maximum vertex weight should be less than the upper bound on the size of a part. The time complexity of the initial partitioning algorithm depends on the type of partitioning. For graph partitioning, we do not consider Step 4.6 in the algorithm. Then, the time complexity of the algorithm becomes $\mathcal{O}(K + NK + K\alpha) = \mathcal{O}(NK)$ since the value of α is at most 1. For hypergraph partitioning, we do consider Step 4.6 in the algorithm. Then, the time complexity of the algorithm becomes $\mathcal{O}(K + NK + p + K\alpha) = \mathcal{O}(NK + p)$.

Because of the balance condition, all the parts can not be at their upper bound or at their lower bound at the same time. To enable every vertex to be present in any part, we must have, for each part P_k ,

$$B(k) - b(k) \geq w_{max} \quad (4.1)$$

Algorithm: Initial Partitioning Algorithm

Input: a hypergraph H with N vertices, K , w_T

Output: a feasible initial partition $\Pi = (P_1, \dots, P_K)$ of H

1. **for** each part P_k , $k \in \mathcal{N}(1, K)$ **do**
 - 1.1. let $s(P_k) \leftarrow 0$ /* initialize part sizes */
 2. **endfor**
 3. let $\Pi \leftarrow (P_1, \dots, P_K)$ /* initialize partition */
 4. **for** each vertex v_i , $i \in \mathcal{N}(1, N)$ **do**

/* assign v_i to a randomly selected part with minimum size */

 - 4.1. find the part P_k with minimum weight
 - 4.2. construct the set $S_{min} = \{P \mid P \in \Pi, s(P) = s(P_k)\}$
 - 4.3. select P_{min} from the set S_{min} randomly
 - 4.4. assign v_i to P_{min}
 - 4.5. let $s(P_{min}) \leftarrow s(P_{min}) + w(v_i)$
 - 4.6. **for** each net e_j incident to v_i **do** /* this is necessary only for hypergraphs */
 - 4.6.1. let $\delta_j(min) \leftarrow \delta_j(min) + 1$
 - 4.7. **endfor**
 5. **endfor**
 6. let $\alpha \leftarrow 0.1$
 7. **repeat**
 - 7.1. **for** each part P_k , $k \in \mathcal{N}(1, N)$ **do**
 - 7.1.1. let $b(k) \leftarrow \lfloor \frac{w_T}{K}(1 - \alpha) \rfloor$
 - 7.1.2. let $B(k) \leftarrow \lceil \frac{w_T}{K}(1 + \alpha) \rceil$
 - 7.2. **endfor**
 - 7.3. **if** $b(k) \leq w(P_k) \leq B(k)$ for each $k \in \mathcal{N}(1, N)$ **then**
 - 7.3.1. Π is feasible
 - 7.4. **else**
 - 7.4.1. let $\alpha \leftarrow \alpha + 0.05$ /* Π is not feasible */
 - 7.5. **endif**
 - 7.6. **if** $\alpha \geq 1.0$ **then** exit /* assumption is violated */
 8. **until** Π is feasible
-

Figure 4.3. An initial partitioning algorithm

Algorithm: Cutsizes Calculation Algorithm
Input: a graph G with M edges, a K -way partition Π , c_T
Output: the cutsize $\chi(\Pi)$

1. let $incost \leftarrow 0$ /* the sum of weights of internal edges of parts */
2. for each edge $e_j, j \in \mathcal{N}(1, M)$ do
 - 2.1. if there exists a part $P_k, k \in \mathcal{N}(1, K)$, such that e_j is in I_k then
 - /* if both end vertices of e_j are in P_k */
 - 2.1.1. let $incost \leftarrow incost + c(e_j)$
 - 2.2. endif
3. endfor
4. let $\chi(\Pi) \leftarrow c_T - incost$

Figure 4.4. A cutsizes calculation algorithm for graphs

where w_{max} is the maximum vertex weight. Moreover, during partitioning, we should have at least one part P_k such that

$$s(P_k) - w_{max} \geq b(k) \quad (4.2)$$

and at least one part P_l such that

$$s(P_l) + w_{max} \leq B(l) \quad (4.3)$$

Otherwise, we may not make any more moves.

4.8 Cutsizes Calculation

The cutsize of a given hypergraph (or a graph) can be calculated while the gains of the vertices are computed. However, we now give two algorithms, one for hypergraphs and one for graphs, to calculate the cutsize. These algorithms can be used either to calculate the initial cutsize or to verify the cutsize after each pass of the partitioning algorithm. Thus, we can be sure that the partitioning algorithm does its job correctly. In fact, we used these algorithms to verify the cutsize after each pass of our partitioning algorithms. The algorithm in Figure 4.4 is for graphs and the algorithm in Figure 4.5 is for hypergraphs. The time complexity of the algorithm in Figure 4.4 is $\mathcal{O}(M)$ and the time complexity of the algorithm in Figure 4.5 is $\mathcal{O}(MK)$.

Algorithm: Cutsizes Calculation Algorithm

Input: a hypergraph H with M nets, a K -way partition Π , c_T

Output: the cutsizes $\chi(\Pi)$

1. let $incost \leftarrow 0$ /* the sum of weights of internal nets of parts */
 2. for each net e_j , $j \in \mathcal{N}(1, M)$ do
 - 2.1. if there exists a part P_k , $k \in \mathcal{N}(1, K)$, such that $N_i(k) = |e_j|$ then
 - /* if all terminals of e_j are in P_k */
 - 2.1.1. let $incost \leftarrow incost + c(e_j)$
 - 2.2. endif
 3. endfor
 4. let $\chi(\Pi) \leftarrow c_T - incost$
-

Figure 4.5. A cutsizes calculation algorithm for hypergraphs

4.9 Prefix Sum Calculation

The maximum prefix sum is the sum of the total improvement of the moves selected to be performed permanently from the recorded sequence of moves during a pass. Assume that Q moves are attempted in a pass and let γ_q denote the move gain of the q^{th} temporary move, for $q \in \mathcal{N}(1, Q)$, in this move sequence. Then, the sum of the move gains of all prefix move sequences are computed as

$$\sigma_q = \sum_{i=1}^q \gamma_i \text{ for all } q \in \mathcal{N}(1, Q). \quad (4.4)$$

Note that, σ_q denotes the overall decrease (or increase if $\sigma_q < 0$) in the cutsizes resulting from the first q moves. Then, the maximum prefix sum, referred hereafter as *gainsum*, is computed as

$$gainsum = \max_{1 \leq q \leq Q} \{\sigma_q\}. \quad (4.5)$$

Here, let q_{max} denote the value of q that maximizes the prefix sum. The gainsum can be positive, zero, or negative. Zero or negative gainsum values terminate the algorithm indicating that the initial solution given to the current pass is a local minimum. If gainsum is positive, the algorithm proceeds to the next pass after making the first q_{max} moves permanent. In the previous partitioning algorithms, the constant Q equals the number of the vertices in the input hypergraph (or graph). In the proposed algorithms, Q can be greater than the number of vertices.

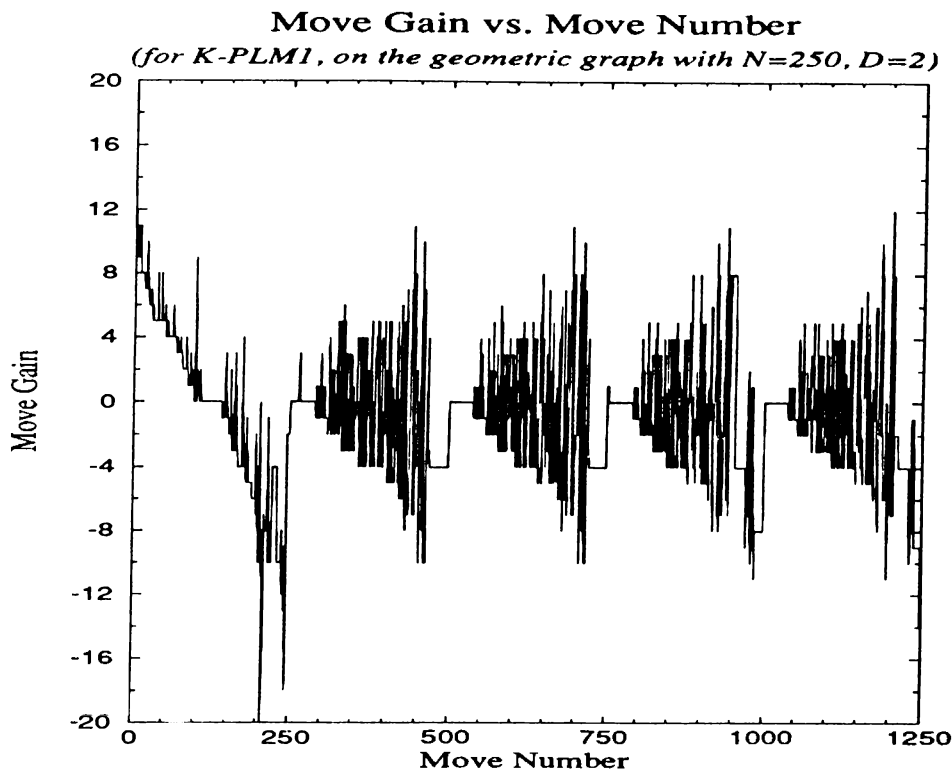


Figure 4.6. Change of gains of selected moves in Sanchis' Algorithm (one pass contains 250 moves)

4.10 Main Claim

When we examine the Kernighan-Lin's minimization technique, it reveals that moves with positive gains, those that decrease the cutsize, become more useful during the early stages of the sequence of the moves performed during a pass and that moves with negative gains, those that increase the cutsize, become more useful towards the end of the sequence of the moves performed during a pass. Hence, we should perform as more moves with positive gains as we can during a pass as long as this process does not lead us to become stuck in a poor local minimum. After some experimentation, we can observe that moves with positive gains, especially those performed in the first pass, occur actually during the early stages of the move sequence. However, we can also observe that, after some point in a pass, the moves that are selected mostly consist of those with negative gains. (Recall that selecting a move has the same meaning as performing a move.) These observations are illustrated in Figure 4.6. Experiments indicate that a move performed at an earlier stage in a pass can have positive gain again in a later stage such that its move gain is larger than those of the moves remaining but it cannot be reselected because it is locked. The reason why this move is not reselected has been to prevent the cell-moving process from thrashing or going into an infinite loop [12, 20, 40].

We think that this reason is not plausible because we can find some other means to avoid thrashing or infinite number of moves during partitioning. Therefore, we make the following claim, on which all our work is based.

Main Claim: *Given a hypergraph with N vertices, allowing each vertex to be moved (possibly) more than once in a pass with the requirement that the occurrence of infinite number of moves having no profit be prevented improves the cutsize more than allowing each vertex to be moved exactly once in a pass.*

We bring the *move-and-lock phase* concept for the sake of simplicity of the discussion of this claim. A move-and-lock phase contains a sequence of temporary moves and their respective locks. A pass may consist of one or more move-and-lock phases. If a move-and-lock phase is not the last one in a pass, then all the vertices that are temporarily moved during this phase are unlocked and reinserted into the appropriate bucket lists, according to their recomputed gains, for the succeeding move-and-lock phases in that pass. On the other hand, if a move-and-lock phase constitutes the last such phase in a pass, the prefix subsequence of moves which maximizes the prefix sum of move gains in that pass is realized permanently. We now propose three novel approaches exploiting the main claim:

1. During a pass, we can make more than one move-and-lock phase such that each move-and-lock phase consists of N moves.
2. During a pass, we can make more than one move-and-lock phase such that each move-and-lock phase consists of less than N moves.
3. During a pass, we can make more than N moves but we do not employ the locking mechanism at all. Yet, there should still be some means to restrict the repeated selections of moves.

We considered all of these ways for partitioning. The items (1) and (2) establish the basis of multiple-way partitioning-by-locked-moves method (in Section 4.11) and the item (3) establish the basis of multiple-way partitioning-by-free-moves method (in Section 4.12). Both of these methods are proposed and implemented in this work. We expect that these methods explore the search space of the problem better. Experimental results support the expectation. These methods are explained in detail in the following sections.

4.11 Partitioning by Locked Moves

Partitioning by Locked Moves (PLM) algorithm is a direct multiple-way partitioning algorithm. Hence, a vertex can move into any of $(K - 1)$ parts in a K -way partition but this move should obviously not violate the balance condition, i.e., it should be a legal move. At any time, there are at most $K(K - 1)$ move directions to select from. A vertex is prevented from being reselected by locking it. Notice that a vertex is removed from the bucket lists as soon as it is locked.

Hereafter when we refer to SN algorithm, we mean the SN algorithm with only first level gains allowed. The SN algorithm does a number of passes until a locally minimum partition is obtained. In each pass, all the vertices in the given hypergraph (or graph) are locked as soon as they are moved. Assuming that we have N vertices, we move N vertices in a pass and then stop the pass and start another pass after calculating the move gain and updating the cutsizes. In our terms, SN algorithm makes one move-and-lock phase consisting of N moves in a pass.

We propose the direct multiple-way Partitioning-by-Locked-Moves (K-PLM) algorithm to realize the first two approaches to our main claim. The generic K-PLM algorithm is given in Figure 4.7. The algorithm employ two constants K_1 and K_2 . The constant K_1 determines the number of move-and-lock phases in a pass, and the constant K_2 determines the number of vertices moved in a single move-and-lock phase. In each pass of the K-PLM algorithm, we thus do $K_1 K_2$ vertex moves. After a move-and-lock phase, we unlock all the K_2 vertices moved, and start another move-and-lock phase until we do $K_1 K_2$ vertex moves. For $K_1 = 1$ and $K_2 = N$, the K-PLM algorithm reduces to SN algorithm. Usually, the exact values of these constants are dependent on the input hypergraph (or graph). However, we let K_1 depend on K and K_2 on N . These constants may depend on the other properties of the input hypergraph (or graph), for example, we can have that K_2 is proportional to N or to the average (vertex or net) degree of the input hypergraph. The main idea in determining the values of these constants is letting $K_1 K_2 \geq N$ and $K_2 \leq N$. The exact values of these constants are given in Section 5.

We now explain the steps of the algorithm in Figure 4.7 in detail and give the time complexity of each step. We explain the steps in terms of hypergraphs but give the necessary modifications for graphs also. Suppose that $H = (V, E)$

Algorithm: Multiple-way Partitioning-by-Locked-Moves Algorithm

Input: an initial K -way partition of the hypergraph H with N vertices, M nets, and p pins

Output: a locally minimum partition $\Pi = (P_1, \dots, P_K)$ of H

1. initialize buckets
 2. **repeat**
 - 2.1. obtain temporary copy of some data structures to work on
 - 2.2. **for** K_1 iterations **do**
 - 2.2.1. compute gains and initialize vertices as unlocked
 - 2.2.2. insert vertices into buckets on basis of their gains
 - 2.2.3. **for** K_2 iterations **do**
 - 2.2.3.1. select a vertex to move
 - 2.2.3.2. delete the vertex from bucket lists and lock it
 - 2.2.3.3. **if** the move does not violate the balance condition **then**
 - 2.2.3.3.1. make a tentative move of the vertex and record the move
 - 2.2.3.3.2. update costs and gains of neighbor vertices, and the bucket lists
 - 2.2.3.4. **endif**
 - 2.2.4. **endfor**
 - 2.2.5. **if** $K_2 < N$ **then** free the buckets list nodes
 - 2.3. **endfor**
 - 2.4. find the maximum prefix sum *gainsum* of move gains of K_2 moves
 - 2.5. **if** *gainsum* > 0 **then** /* there is an improvement in cutsize */
 - 2.5.1. permanently move vertices yielding *gainsum*
 - 2.5.2. decrease cutsize by *gainsum*
 - 2.6. **endfor**
 3. **until** *gainsum* ≤ 0 /* Π is locally minimum */
-

Figure 4.7. The generic direct multiple-way partitioning-by-locked-moves algorithm

is a hypergraph with N vertices, M nets, and p terminals to be partitioned into K parts. Similarly, suppose that $G = (V, E)$ is a graph with N vertices and M edges to be partitioned into K parts.

- (step 1) We initialize the buckets by initializing the indices for each bucket array and by initializing the bucket list pointers in the bucket arrays. Since there are $K(K - 1)$ bucket arrays each of which has $2G_{max} + 1$ buckets, we can initialize all the arrays in $\mathcal{O}(K^2G_{max})$ time.
- (step 2.1) Since we make tentative moves in a pass, we should not destroy the information contained in some data structures at the beginning of the pass. These data structures are the array holding the partition information, the array holding the number of terminals of each net in each partition (this array does not exist for graphs), and the array holding the information indicating the parts of each vertex. Thus, we should obtain temporary copies of these data structures. If $K_2 < N$, this step requires $\mathcal{O}(K + MK + N) = \mathcal{O}(MK + N)$ time for the hypergraph H , and $\mathcal{O}(K + N)$ time for graphs. If $K_2 = N$, i.e., the case in the SN algorithm, this step requires $\mathcal{O}(K + MK) = \mathcal{O}(MK)$ time for the hypergraph H , and $\mathcal{O}(K)$ time for graphs.
- (step 2.2.1) The initial gains of vertices in the hypergraph H are computed using the algorithm in Figure 3.1, and those of vertices in the graph G are calculated using the algorithm in Figure 2.1. Although these algorithms compute the cost vectors of the vertices, the move gains of the vertices can easily be obtained from the cost vector by the definition of the move gain. Initial gain computations take $\mathcal{O}(NK + pK)$ and $\mathcal{O}(NK + M)$ time for hypergraphs and graphs, respectively.
- (step 2.2.2) We insert the vertices into bucket arrays by first calculating the indices of the buckets that correspond to the gains of the vertices. We then place each vertex to the head of $(K - 1)$ bucket lists each of which is connected to a bucket in the bucket arrays. We pass over each vertex once and insert each vertex into $(K - 1)$ bucket lists each of which requires constant time. Thus, this step requires $\mathcal{O}(NK)$ time.
- (step 2.2.3.1) There are $K(K - 1)$ move directions for the K -way partitioning. We can make a move in each of these move directions or not. We search these move directions and select the first move that has the maximum gain and that does not violate the balance condition. Since each

bucket array has a index pointer indicating the bucket list corresponding to the moves with the maximum gain, the search only involves these index pointers and we only examine the nodes at the head of the bucket lists, i.e., there is not a search along the bucket lists. Moreover, if there is no move which does not violate the balance condition, we then select the move with the maximum gain anyway but set a variable indicating that the move is not possible. Considering the number of move directions and the constant time to reach the vertices in a move direction, we can select a move in $\mathcal{O}(K^2)$ time. This selection time can be reduced by utilizing a heap but we did not do so [31].

- (step 2.2.3.2) After selecting a move, we delete from the bucket lists all the moves associated with the vertex of the selected move, and lock the vertex. Thus, the deletion of a vertex from $(K - 1)$ bucket lists needs $\mathcal{O}(KG_{max})$ time.
- (step 2.2.3.3.1) If there is a possible move that does not violate the balance condition, then we make this move. We record the properties of this move such as the move gain. We also update the part sizes in the partition structure and change the part where the vertex is present. This step can be done in constant time.
- (step 2.2.3.3.2) If there is a possible move, we update the costs and move gains of the neighbor vertices of the vertex moved and also update the bucket arrays and lists so as to correct the information in them. We update the move gains of the neighbor vertices in the hypergraph H using the algorithm in Figure 3.2, and those of the neighbor vertices in the graph G using the algorithm in Figure 2.2. Note that this update process is applied to only the unlocked neighbors of the vertex moved. The update process results in a number of insertions into and deletions from the bucket lists. The algorithm in Figure 3.2 runs in $\mathcal{O}(pKG_{max})$ for N vertex moves since each net can have a constant number of update operations during N vertex moves. The algorithm in Figure 2.2 requires $\mathcal{O}(MKG_{max})$ time for N vertex moves.
- (step 2.2.5) If $K_2 < N$ then there remain vertices in the bucket lists at the end of a move-and-lock phase. Since we recompute the vertex gains and insert the moves associated with the vertices into the bucket lists again, we should remove the moves associated with the vertices that are present in the bucket lists at this point in the algorithm. The nodes in

all the bucket lists can be deallocated in $\mathcal{O}(K^2G_{max} + KN)$ time.

- (step 2.4 and step 2.5) We now have $K_1K_2 = \mathcal{O}(K_1N)$ successive moves recorded. We should select from the beginning of the record those moves whose execution produces the maximum decrease in the cutsize. That is, we calculate the prefix sum, *gainsum*, of the move gains of the moves in the recorded sequence, and perform the subsequence of moves yielding the maximum prefix sum. The step 2.4 and the step 2.5 both require $\mathcal{O}(K_1N)$ time for the graph G , but the step 2.4 requires $\mathcal{O}(K_1N)$ time and the step 2.5 requires $\mathcal{O}(K_1p)$ time for the hypergraph H .

4.12 Partitioning by Free Moves

The proposed direct multiple-way Partitioning by Free Moves (K-PFM), brings in different concept from all the other iterative improvement partitioning algorithms that employ locking. In the K-PFM algorithm, locking is not used at all. Each vertex can move as freely as possible in a pass. Each vertex can make different number of moves. The move capability of a vertex is only restricted by the number of moves that the vertex has performed. This is done to prevent the vertices from doing a lot of moves without any significant decrease in the cutsize. The move capability of a vertex is dependent on a new concept called freedom value of the vertex as previously mentioned in Section 2.4.2. We also call the number of moves that a vertex has performed as the *move count* of the vertex. The freedom value of a vertex depends on its move count and its current move gain. The larger the move count of a vertex is, the lower the chance the vertex is selected to move again in a pass (and thus, the smaller the freedom value is), and the higher the gain of the vertex is, the higher the chance the vertex is selected to move again in a pass (and thus, the larger the freedom value is.)

In the generic K-PFM algorithm given in Figure 4.8 the vertices are not inserted into the bucket lists on the basis of their gains but on the basis of their freedom values. Any update in a cost of the vertex propagates to the corresponding gain and freedom value of the vertex also. We do K_1 vertex moves in a pass of the K-PFM algorithm. The constant K_1 is usually dependent on N and K but larger than N . The main idea, which is similar to the one in the K-PLM algorithm, is letting $K_1 \geq N$. As in the generic K-PLM algorithm, move-and-lock phases can also be employed but we did not try it.

We now define the freedom value concept for hypergraphs. The definition of the freedom value for graphs is the same as its definition for hypergraphs. Let $H = (V, E)$ be a hypergraph and $\Pi = (P_1, \dots, P_K)$ a K -way partition of H . The cost and gain concepts are defined as in Section 3.4.1. The *freedom value* $\Phi_m(f, t)$ of a vertex v_m in the part P_f with respect to the part P_t is defined as

$$\Phi_m(f, t) = \frac{1}{1 + e^{(-G_m(f, t)R)}} \text{ if } n_m = 0 \quad (4.6)$$

and

$$\Phi_m(f, t) = \frac{1}{1 + \sqrt{n_m}e^{(-G_m(f, t)R)}} \text{ if } n_m \neq 0 \quad (4.7)$$

where n_m is the move count of v_m , $G_m(f, t)$ is the gain of v_m in P_f with respect to P_t , and R is a constant to ensure that the value of $\Phi_m(f, t)$ is in the range $(0, 1)$. The move count of v_m is incremented by 1 but we can change the way the move count is incremented. Other freedom value functions with similar properties are examined in Section 5.

The value of the constant R can be calculated as follows: Let $\Phi_m(f, t)$ be in the interval $[\epsilon, 1 - \epsilon]$ for a very small positive constant ϵ (e.g., $\epsilon = 0.01$). Then,

$$R = \left(\frac{1}{G_{max}}\right) \ln\left(\frac{1 - \epsilon}{\epsilon}\right) \quad (4.8)$$

where G_{max} is the maximum gain that a vertex can have. It is equal to the product of the maximum net weight c_{max} and the maximum vertex degree $D_{v, max}$ in the hypergraph H , i.e., $G_{max} = D_{v, max}c_{max}$.

We cannot map the Φ values of the vertices as in Equation 4.7 into the buckets because these values fall in the range between 0 and 1. We multiply the Φ value of a vertex with a *scale factor* S and then map the vertex into the bucket list connected to the bucket with the index equal to this new scaled value that is also floored to be converted into an integer. That is, we map the vertex v_m with the freedom value $\Phi_m(f, t)$ into the bucket list of the bucket with the index equal to $\lfloor S\Phi_m(f, t) \rfloor$. We later present experiments to determine the values of the scale factor S in Section 5.

Let the vertex v_i in the part P_f and v_j in the part P_f have the move counts n_i and n_j respectively. Consider the gains and freedom values with respect to the part P_t in the partition Π . Then, if $n_i = n_j$ then

$$G_i(f, t) \leq G_j(f, t) \Leftrightarrow \Phi_i(f, t) \leq \Phi_j(f, t). \quad (4.9)$$

However, since the freedom value of a vertex is also floored to make it integer, the above order between the move gains and the freedom values is not preserved. This brings some randomization into the K-PFM algorithm.

Since the scale factor S restricts the size of the bucket arrays, it helps control the space requirement of the algorithms. It also affects the running time of the algorithms.

Instead of calculating the values corresponding to the exponential function in Equation 4.7 each time a freedom value is required, we used a table lookup technique to speed up the calculation of the freedom value function since exponentiation is an expensive operation. The table is implemented with an array, called *eval* (Exponential VALues array), containing the values of the exponential function in the range from the minimum possible gain to the maximum possible gain. Note that the values in the array are not approximations to the exponential values but the true values. However, they would be approximate values if the gains were not integer numbers. The usage of the array is as follows: For a gain value G in the range $[0, 2G_{max}]$, we have

$$eval[G] = e^{(-R(G_{max}-G))} \quad (4.10)$$

Then, for a gain value G in the range $[-G_{max}, G_{max}]$,

$$e^{(-GR)} = eval[G_{max} - G] \quad (4.11)$$

and so the freedom value $\Phi_m(f, t)$ of the vertex v_m in the part P_f with respect to the part P_t becomes

$$\Phi_m(f, t) = \frac{1}{1 + eval[G_{max} - G_m(f, t)]} \quad \text{if } n_m = 0 \quad (4.12)$$

$$\Phi_m(f, t) = \frac{1}{1 + \sqrt{n_m} eval[G_{max} - G_m(f, t)]} \quad \text{if } n_m \neq 0 \quad (4.13)$$

We now explain the steps of the algorithm in Figure 4.8 in detail and give the time complexity of each step. We explain the steps in terms of hypergraphs but give the necessary modifications for graphs also. Suppose that $H = (V, E)$ is a hypergraph with N vertices, M nets, and p terminals to be partitioned into K parts. Similarly, suppose that $G = (V, E)$ is a graph with N vertices and M edges to be partitioned into K parts.

- (step 1) We initialize the buckets by initializing the indices for each bucket array and by initializing the bucket list pointers in the bucket arrays.

Algorithm: Multiple-way Partitioning-by-Free-Moves Algorithm

Input: an initial K -way partition of the hypergraph H with N vertices, M nets, and p terminals, the array *eval* filled

Output: a locally minimum partition $\Pi = (P_1, \dots, P_K)$ of H

1. initialize buckets
 2. **repeat**
 - 2.1. obtain temporary copy of the partition data structure to work on
 - 2.2. compute gains and initialize move counts of vertices
 - 2.3. insert vertices into buckets on basis of their freedom values
 - 2.4. **for** K_1 iterations **do**
 - 2.4.1. select a vertex to move
 - 2.4.2. **if** the move violate the balance condition **then** exit this loop
 - 2.4.3. make a tentative move of the vertex, record the move and increment the move count
 - 2.4.4. insert previously moved vertex (if one exists) into bucket lists
 - 2.4.5. update costs, gains and freedom values of the vertex moved and neighbor vertices, and the bucket lists
 - 2.4.6. delete currently moved vertex from bucket lists
 - 2.4.7. make currently moved vertex as previously moved vertex
 - 2.5. **endfor**
 - 2.6. find the maximum prefix sum *gainsum* of move gains of K_2 moves
 - 2.7. **if** *gainsum* > 0 **then** /* there is an improvement in cutsize */
 - 2.7.1. permanently move vertices yielding *gainsum*
 - 2.7.2. decrease cutsize by *gainsum*
 - 2.8. **endfor**
 3. **until** *gainsum* ≤ 0 /* Π is locally minimum */
-

Figure 4.8. The generic direct multiple-way partitioning-by-free-moves algorithm

Since there are $K(K - 1)$ bucket arrays each of which has S buckets, we can initialize all the bucket arrays in $\mathcal{O}(K^2S)$ time. (Notice that since the freedom value of a vertex can be at most $S - 1$ and at least 0, a bucket array has S buckets.)

- (step 2.1) Since we make tentative moves in a pass, we should not destroy the information contained in some data structures at the beginning of the pass. These data structures are the array holding the partition information, the array holding the number of terminals of each net in each partition (this array does not exist for graphs), and the array holding the information indicating the parts of each vertex. Thus, we should obtain temporary copies of these data structures. If $K_2 < N$, this step requires $\mathcal{O}(K + MK + N) = \mathcal{O}(MK + N)$ time for the hypergraph H , and $\mathcal{O}(K + N)$ time for graphs. If $K_2 = N$, i.e., the case in the SN algorithm, this step requires $\mathcal{O}(K + MK) = \mathcal{O}(MK)$ time for the hypergraph H , and $\mathcal{O}(K)$ time for graphs.
- (step 2.2) The initial gains of vertices in the hypergraph H are computed using the algorithm in Figure 3.1, and those of vertices in the graph G are calculated using the algorithm in Figure 2.1. Although these algorithms compute the cost vectors of the vertices, the move gains of the vertices can easily be obtained from the cost vector by the definition of the move gain. Initial gain computations take $\mathcal{O}(NK + pK)$ and $\mathcal{O}(NK + M)$ time for hypergraphs and graphs, respectively.
- (step 2.3) We insert the vertices into bucket arrays by first calculating the indices corresponding to the freedom values of the vertices. We then place each vertex to the head of $(K - 1)$ bucket lists each of which is connected to a bucket in the bucket arrays. We pass over each vertex once and insert each vertex into $(K - 1)$ bucket lists each of which requires constant time. Thus, this step requires $\mathcal{O}(NK)$ time.
- (step 2.4.1) There are $K(K - 1)$ move directions for the K -way partitioning. We can make a move in each of these move directions or not. We search these move directions and select the first move that has the maximum gain and that does not violate the balance condition. Since each bucket array has a index pointer indicating the bucket list corresponding to the moves with the maximum gain, the search only involves these index pointers and we only examine the nodes at the head of the bucket lists, i.e., there is not a search along the bucket lists. Moreover, if there is

no move which does not violate the balance condition, we then select the move with the maximum gain anyway but set a variable indicating that the move is not possible. Considering the number of move directions and the constant time to reach the vertices in a move direction, we can select a move in $\mathcal{O}(K^2)$ time. This selection time can be reduced by utilizing a heap but we did not do so [31].

- (step 2.4.2) When we cannot find a move after searching all the $K(K - 1)$ move directions, we exit the inner loop. Instead of exiting, we can temporarily delete a vertex at the head of the bucket lists and search for a new move. This deletion and search step can be continued by inserting the previously deleted vertex into the bucket lists and deleting another vertex. However, we prefer exiting. This step takes constant time.
- (step 2.4.3) We move the selected vertex. We record the properties of this move such as the move gain. We also update the part sizes in the partition structure, change the part where the vertex is present, and increment the move count of the vertex. This step can be done in constant time.
- (step 2.4.4, step 2.4.6 and step 2.4.7) If a vertex is selected to move, we then prevent this vertex to be reselected in the very next selection step by deleting the vertex from the bucket lists after it is moved and the necessary updates are performed on the cost, gain and freedom values of the vertex moved and its neighbor vertices. This action seems to improve the partitioning results. Thus, the currently moved vertex is not present in the bucket lists at the very next selection step. After a new vertex is selected, the previously selected vertex must be inserted into the bucket lists to perform correct updates. Of these three steps, step 2.4.4. requires $\mathcal{O}(K)$ time, step 2.4.6. requires $\mathcal{O}(KS)$ time, and step 2.4.7. requires constant time.
- (step 2.4.5) If there is a possible move, we update the costs, the move gains, and the freedom values of the neighbor vertices of the vertex moved and also update the bucket arrays and lists so as to correct the information in them. We update the gains of the neighbor vertices in the hypergraph H using the algorithm in Figure 3.2, and those of the neighbor vertices in the graph G using the algorithm in Figure 2.2. The update operation for the vertex moved is done by the steps 2.4.4, 2.4.6 and 2.4.7. The update process results in a number of insertions into and deletions from the bucket lists. The algorithm in Figure 3.2 runs in

$\mathcal{O}(D_{v,max}D_{e,max}KS)$ time where $D_{v,max}$ is the maximum vertex degree and $D_{e,max}$ is the maximum net degree. The algorithm in Figure 2.2 requires $\mathcal{O}(D_{v,max}KS)$ time where $D_{v,max}$ is the maximum vertex degree. In the K-PLM algorithm, we are sure that all the N vertices are moved once and locked, and we know the sum of their degrees. In K-PFM algorithm, we do not know the move count of each vertex. The move counts are different for each vertex. We can obtain only a worst-case time complexity of the gain update algorithms. Moreover, since we do not know the probability distribution of the move counts of the vertices, it seems very difficult to obtain an average-case time complexity of the gain update algorithms.

- (step 2.6. and step 2.7) We now have K_1 successive moves recorded. We should select from the beginning of the record those moves whose execution produces the maximum decrease in the cutsizes. That is, we calculate the prefix sum, *gainsum*, of the move gains of the moves in the recorded sequence, and perform the subsequence of moves yielding the maximum prefix sum. The step 2.6 and the step 2.7 both require $\mathcal{O}(K_1)$ time for the graph G , but the step 2.6 requires $\mathcal{O}(K_1)$ time and the step 2.7 requires $\mathcal{O}(K_1D_{v,max})$ time for the hypergraph H .

4.13 Complexity Analysis

4.13.1 Time Complexity Analysis

We now present a time complexity analysis of the algorithms we investigate. We give the time complexity analysis of each algorithm both for hypergraphs and graphs. Moreover, the time complexities of reading the input and creating an initial partition are given in this section for the sake of completeness. These time complexities are not included in the time complexity of any partitioning algorithm examined below.

Suppose that $G = (V, E)$ is a graph with N vertices and M edges to be partitioned into K parts.

1. The graph G can be read in $\mathcal{O}(M + N)$ time.

2. An initial K -way partitioning for G can be obtained using the algorithm in Figure 4.3 in $\mathcal{O}(NK)$.
3. If ($K_2 < N$) then the K-PLM algorithm has a time complexity of $\mathcal{O}(K_1K(NK + G_{max}(N + M + K)))$. For $K = 2$, this time complexity reduces to $\mathcal{O}(K_1G_{max}(N + M))$.
4. If ($K_2 = N$) then the K-PLM algorithm has a time complexity of $\mathcal{O}(K^2G_{max} + K_1K(NK + G_{max}(N + M)))$. For $K = 2$, this time complexity reduces to $\mathcal{O}(K_1G_{max}(N + M))$.
5. The time complexity of the SN algorithm is $\mathcal{O}(K^2G_{max} + K(NK + G_{max}(N + M))) = \mathcal{O}(K(NK + G_{max}(N + M + K)))$. This time complexity is obtained by letting $K_1 = 1$ in the time complexity of the K-PLM algorithm for $K_2 = N$. The time complexity of the SN algorithm for $K = 2$ reduces to $\mathcal{O}(G_{max}(N + M))$.
6. The time complexity of the K-PFM algorithm without the time of filling the array *eval* becomes $\mathcal{O}(K^2S + M + NK + K_1K(K + SD_{v,max}))$ where $D_{v,max}$ is the maximum vertex degree in G . This time complexity reduces to $\mathcal{O}(M + N + K_1SD_{v,max})$ for $K = 2$.

Suppose that $H = (G, E)$ is a hypergraph with N vertices, M nets, and p terminals to be partitioned into K parts.

1. The hypergraph H can be read in $\mathcal{O}(N + MK + p) = \mathcal{O}(N + pK)$ time. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(pK)$.
2. An initial K -way partitioning for H can be obtained using the algorithm in Figure 4.3 in $\mathcal{O}(NK + p)$ time. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(pK)$.
3. If ($K_2 < N$) then the K-PLM algorithm has a time complexity of $\mathcal{O}(MK + K_1K(NK + G_{max}(N + p + K)))$. With $M = \mathcal{O}(p)$, we have the time complexity $\mathcal{O}(K_1K(NK + G_{max}(N + p + K)))$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(K_1K(pK + G_{max}(p + K)))$. For $K = 2$, this time complexity reduces to $\mathcal{O}(M + K_1G_{max}(N + p)) = \mathcal{O}(K_1G_{max}(N + p))$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(K_1G_{max}p)$.

4. If $(K_2 = N)$ then the K-PLM algorithm has a time complexity of $\mathcal{O}(K^2G_{max} + MK + K_1K(NK + G_{max}(N + p)))$. With $M = \mathcal{O}(p)$, we have the time complexity $\mathcal{O}(K^2G_{max} + K_1K(NK + G_{max}(N + p)))$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(K^2G_{max} + K_1Kp(K + G_{max}))$. For $K = 2$, this time complexity reduces to $\mathcal{O}(M + K_1G_{max}(N + p)) = \mathcal{O}(K_1G_{max}(N + p))$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(K_1G_{max}p)$.
5. The time complexity of the SN algorithm is $\mathcal{O}(K(M + NK + G_{max}(N + p + K))) = \mathcal{O}(K(NK + G_{max}(N + p + K)))$. This time complexity is obtained by letting $K_1 = 1$ in the time complexity of the K-PLM algorithm for $K_2 = N$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(K(pK + G_{max}(p + K)))$. The time complexity of the SN algorithm for $K = 2$ reduces to $\mathcal{O}(M + G_{max}(N + p)) = \mathcal{O}(G_{max}(N + p))$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(G_{max}p)$, which is the same as the time complexity of FM algorithm, which assumes that both $M = \mathcal{O}(p)$ and $N = \mathcal{O}(p)$ hold.
6. The time complexity of the K-PFM algorithm is $\mathcal{O}(K(M + p + N + KS + K_1(K + D_{v,max}D_{e,max}S)))$ where $D_{v,max}$ is the maximum vertex degree and $D_{e,max}$ is the maximum net degree in H . With $M = \mathcal{O}(p)$, we have the time complexity $\mathcal{O}(K(p + N + KS + K_1(K + D_{v,max}D_{e,max}S)))$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(K(p + KS + K_1(K + D_{v,max}D_{e,max}S)))$. For $K = 2$, this time complexity reduces to $\mathcal{O}(M + p + N + K_1D_{v,max}D_{e,max}S) = \mathcal{O}(p + N + K_1D_{v,max}D_{e,max}S)$. If we further assume that $N = \mathcal{O}(p)$, then the time complexity becomes $\mathcal{O}(p + K_1D_{v,max}D_{e,max}S)$.

4.13.2 Space Complexity Analysis

Suppose that $G = (V, E)$ is a graph with N vertices and M edges to be partitioned into K parts. The graph N can be stored in $\mathcal{O}(N + M)$ space. We can hold the information about the K parts in the partition in $\mathcal{O}(K)$ space. The array mapping each vertex to a part in the partition requires $\mathcal{O}(N)$ space. We also maintain an array storing the cost vector of each vertex, and the pointers to the bucket lists for each vertex. This array has $\mathcal{O}(NK)$ space. Each vertex is inserted into $(K - 1)$ bucket lists, so the bucket lists have $\mathcal{O}(NK)$ list nodes. We have $K(K - 1)$ bucket arrays. For the K-PLM algorithm, the

bucket arrays need $\mathcal{O}(K^2G_{max})$ space, and the array recording the moves in a pass needs $\mathcal{O}(K_1N)$ space. For the K-PFM algorithm, the bucket arrays need $\mathcal{O}(K^2S)$ space, the array recording the moves in a pass needs $\mathcal{O}(K_1)$ space, and the array *eval* needs $\mathcal{O}(G_{max})$ space. Therefore, we have the following:

1. the K-PLM algorithm for graphs requires $\mathcal{O}(M + K^2G_{max} + NK + K_1N)$ space,
2. the SN algorithm for graphs requires $\mathcal{O}(M + K^2G_{max} + NK)$ space,
3. the K-PFM algorithm for graphs requires $\mathcal{O}(M + K^2S + NK + K_1 + G_{max})$ space.

Suppose that $H = (G, E)$ is a hypergraph with N vertices, M nets, and p terminals to be partitioned into K parts. The hypergraph H along with the information holding the number of terminals in each part of each net can be stored in $\mathcal{O}(N + MK + p)$ space. We can hold the information about the K parts in the partition in $\mathcal{O}(K)$ space. The array mapping each vertex to a part in the partition requires $\mathcal{O}(N)$ space. We also maintain an array storing the cost vector of each vertex, and the pointers to the bucket lists for each vertex. This array has $\mathcal{O}(NK)$ space. Each vertex is inserted into $(K - 1)$ bucket lists, so the bucket lists have $\mathcal{O}(NK)$ list nodes. We have $K(K - 1)$ bucket arrays. For the K-PLM algorithm, the bucket arrays need $\mathcal{O}(K^2G_{max})$ space, and the array recording the moves in a pass needs $\mathcal{O}(K_1N)$ space. For the K-PFM algorithm, the bucket arrays need $\mathcal{O}(K^2S)$ space, the array recording the moves in a pass needs $\mathcal{O}(K_1)$ space, and the array *eval* needs $\mathcal{O}(G_{max})$ space. Therefore, we have the following:

1. the K-PLM algorithm for hypergraphs requires $\mathcal{O}(MK + p + K^2G_{max} + NK + K_1N)$ space. With $M = \mathcal{O}(p)$, this complexity reduces to $\mathcal{O}(pK + K^2G_{max} + NK + K_1N)$ space. If we further assume that $N = \mathcal{O}(p)$ then the complexity becomes $\mathcal{O}(K^2G_{max} + p(K_1 + K))$.
2. the SN algorithm for hypergraphs requires $\mathcal{O}(MK + p + K^2G_{max} + NK)$ space. With $M = \mathcal{O}(p)$, this complexity reduces to $\mathcal{O}(pK + K^2G_{max} + NK)$. If we further assume that $N = \mathcal{O}(p)$ then the complexity becomes $\mathcal{O}(pK + K^2G_{max})$.
3. the K-PFM algorithm for hypergraphs requires $\mathcal{O}(MK + p + K^2S + NK + K_1 + G_{max})$ space. With $M = \mathcal{O}(p)$, this complexity reduces to

$\mathcal{O}(pK + K^2S + NK + K_1 + G_{max})$. If we further assume that $N = \mathcal{O}(p)$ then the complexity becomes $\mathcal{O}(pK + K^2S + K_1 + G_{max})$.

Chapter 5

EXPERIMENTS AND RESULTS

This chapter includes all test problems, the details of experiments done and the results obtained.

5.1 Implementation of Algorithms

The SN algorithm, the generic K-PFM algorithm, the generic K-PLM algorithm, and the other utility programs were all implemented in the C programming language. Most of the functions used in the programs are common to all the programs. All the experiments were carried out on a SUN SPARC station ELC¹ under SunOS Release 4.1.3² operating system.

5.2 Balance Condition

In all our experiments on both graph and hypergraph instances, we initially set $\alpha = 0.10$ in the balance condition, given in Section 2.4.3, of the partitioning algorithms. Surprisingly, the initial partitions of all the graph and hypergraph instances became feasible at this value of α . Hence, we allowed a part size to be 10% more or 10% less than its value in a perfectly balance partition. Also, all the final partitions obtained by the algorithms we evaluated were also feasible and satisfied the balance condition with this value of α since only legal moves

¹SUN Workstation is a registered trademark of Sun Microsystems, Inc.

²SunOS is an unregistered trademark of Sun Microsystems, Inc.

are allowed during the iterative partitioning process. We expect that setting larger values to α in the balance condition yields partitions with lower cutsizes. We performed too few experiments to check this expectation, however.

5.3 K-PFM Algorithms

Recall that the generic K-PFM algorithm in Figure 4.8 has the parameter K_1 . We obtained different K-PFM-like algorithms by setting various values to this parameter. The algorithms obtained were also renamed so that we could refer to them more easily. The settings and the algorithms obtained are tabulated below:

K_1	Resulting Algorithm
N	K-PFM1
NK	K-PFM2
NK^2	K-PFM3

In the above table, N is the number of the vertices in the input hypergraph (or graph), and K is the number of parts in the required partition.

By setting other values to K_1 , any other K-PFM-like algorithms are also possible. Note that the parameter K_1 denotes the number of moves performed in a pass of the generic K-PFM algorithm. In every pass, the value of this parameter was held constant during our experimentation. We can employ an adaptive scheme such that the value of this parameter varies from one pass to another. The time and space complexities of these K-PFM-like algorithms can easily be obtained by putting the value of K_1 in the time and space complexity expressions of the generic K-PFM algorithm in Section 4.13. Notice that the rank order of these algorithms with respect to their time complexities from the smallest to the largest is K-PFM1, K-PFM2, and K-PFM3.

5.4 K-PLM Algorithms

Recall that the generic K-PLM algorithm in Figure 4.7 uses two parameters K_1 and K_2 . We obtained different K-PLM-like algorithms by setting various

values to these parameters. The algorithms obtained were also renamed so that we could refer to them more easily. The settings and the algorithms obtained are tabulated below:

K_1K_2	K_2	Resulting Algorithm
N	$4N/4$	K-PLM1
N	$3N/4$	K-PLM2
N	$2N/4$	K-PLM3
N	$1N/4$	K-PLM4
NK	$4N/4$	K-PLM5
NK	$3N/4$	K-PLM6
NK	$2N/4$	K-PLM7
NK	$1N/4$	K-PLM8
NK^2	$4N/4$	K-PLM9
NK^2	$3N/4$	K-PLM10
NK^2	$2N/4$	K-PLM11
NK^2	$1N/4$	K-PLM12

In the above table, N is the number of the vertices in the input hypergraph (or graph), and K is the number of parts in the required partition. Note that K-PLM1 algorithm is identical to SN algorithm and henceforth we use the label K-PLM1 instead of SN when we refer to SN algorithm.

By setting other values to these parameters, any other K-PFM-like algorithms are also possible. Note that the parameter K_1 denotes the number of move-and-lock phases in a pass of the generic K-PLM algorithm and that the parameter K_2 denotes the number of moves in a single move-and-lock phase. As in experiments with the K-PFM-like algorithms, the values of these parameters were held constant during our experimentation. We can employ an adaptive scheme such that the value of these parameters vary from one pass to another. The time and space complexities of these K-PLM-like algorithms can easily be obtained by putting the value of K_1 in the time and space complexity expressions of the generic K-PLM algorithm in Section 4.13. Note that $K_2 = \mathcal{O}(N)$, which is already incorporated into the time complexity of the generic K-PLM algorithm.

5.5 Comments on Neighborhood Structure of Algorithms

Recall that all the algorithms we investigated in this study use the move-neighborhood structure. If the input hypergraph (or graph) has N vertices and it is to be partitioned into K parts, then the number of solutions in a neighborhood is bounded above by $N(K - 1)$. Although these algorithms use the same neighborhood structure, they need not perform the same on a problem instance. The function that is employed to select one solution over another also counts. Besides, the total number of solutions in all the neighborhoods throughout the operation of the algorithm affects the performance.

Notice that all the K-PLM-like algorithms use the move gain function to move one solution to another whereas the K-PFM-like algorithms use the freedom value function. Now, let us examine the the number of solutions inspected during a pass of the generic K-PLM and K-PFM algorithms.

The generic K-PFM algorithm has the same number of solutions, namely, at most $N(K - 1)$, in all the neighborhoods explored during a pass. Since a pass consists of K_1 moves, the total number of solutions is bounded above by $K_1 N(K - 1)$. Particularly, the K-PFM1 algorithm examines at most $N^2(K - 1)$ solutions.

The generic K-PLM algorithm does not examine the same number of solutions at each move in a pass. The neighborhood shrinks at each move. Thus, it examines at most $N(K - 1)$ solutions at the first move, at most $(N - 1)(K - 1)$ solutions in the second move, and so on. At the end of a move-and-lock neighborhood, that is, at the K_2 th move, it examines $(N - (K_2 - 1))(K - 1)$ solutions. Hence, during a pass involving $K_1 K_2$ moves, the generic K-PLM algorithm examines at most $K_1 K_2 (K - 1)(2N - K_2 + 1)/2$ solutions. Particularly, the K-PLM1 algorithm examines at most $(K - 1)N(N + 1)/2$ solutions.

From these calculations, the following observations reveal.

- The respective bounds for K-PFM1 and K-PLM1 are equal asymptotically but the bound for K-PFM1 is twice that for K-PLM1 when N is large.
- The bounds for the K-PFM-like algorithms are larger by a constant factor

than those for the K-PLM algorithms if the algorithms perform the same number of moves in a pass.

- The bounds for the K-PFM-like algorithms with $K_1 \geq NK$ are larger by a factor depending on K than those for K-PLM1 algorithm.
- The algorithms with more than one move-and-lock phase examine more solutions than K-PLM1 algorithm even if the number of moves in a pass is held to be the same as that of K-PLM1 algorithm. For example, the bound for K-PLM3 algorithm is larger by a factor of $3/2$ than that for K-PLM1 algorithm. This may be one of the reasons why the results obtained by these algorithms seem to be better than those by K-PLM1 algorithm.

5.6 Notation

We now explain in the following table the meanings of the column headings used in the tables giving the data about the test graphs and hypergraphs because most of these headings are common to all tables.

Heading	Meaning
N	number of vertices
M	number of edges
p	total number of terminals
w_T	total vertex weight
c_T	total edge weight
D_{exp}	expected average vertex degree before generation of graph
D_{act}	actual average vertex degree after generation of graph
$D_{v,max}$	maximum vertex degree (also $D_{v,x}$)
D_v	average vertex degree
$D_{e,max}$	maximum net degree (also $D_{e,x}$)
D_e	average net degree
w_{max}	maximum vertex weight (also w_x)
w_{min}	minimum vertex weight (also w_n)
c_{max}	maximum edge weight
c_{min}	minimum edge weight

5.7 Test Graphs

We have used as our test beds 5 different types of graphs: random, geometric, grid, ladder, tree. The random and geometric graphs are standard test beds for graph partitioning algorithms [17, 3]. The other types of graphs were used to evaluate the partitioning algorithms because the KL algorithm was observed to fail badly on these types of graphs [6, 15]. The vertices and edges in all the graphs are weighted. The vertex weights in the test graphs are uniformly distributed in the range from 1 to 4. The edge weights in the test graphs are uniformly distributed in the range from 1 to 5. When the vertex weights are selected from a pool of uniformly distributed random numbers, we ensure that the variance between the weights remain small and thus a more balanced initial partition is easily generated. We now present the definitions of the test graphs and the algorithms to generate them. Moreover, the properties of these test graphs are given in tables as mentioned in the following sections.

5.7.1 Random Graphs

A *random graph* $G_{N,p}$ [17] is a graph with N vertices, where each pair of vertices constitutes an edge with probability p . Since $G_{N,p}$ can have at most $p \binom{N}{2}$ edges, the sum of the degrees of the vertices of $G_{N,p}$ is equal to $2p \binom{N}{2}$ by Equation 2.6. Then, the expected average vertex degree in the random graph $G_{N,p}$ is $2p \binom{N}{2} / N = p(N-1)$. We generated 15 random graphs whose properties are depicted in Table C.1.

The algorithm in Figure 5.1 generates a random graph $G_{N,p}$. It flips a coin with probability p for all $N(N-1)/2$ potential edges. The time complexity of the algorithm is $\mathcal{O}(N^2)$.

5.7.2 Geometric Graphs

A *geometric graph* U_{N,D_v} [17] is a graph with N vertices and with an average vertex degree D_v , and generated as follows: first, pick $2N$ independent numbers

Algorithm: Random Graph Generator
Input: number N of vertices, expected average vertex degree D_v
Output: a random graph $G_{N,p}$ with $p = D_v/(N - 1)$

1. generate N vertices but 0 edges
2. let $p \leftarrow D_v/(N - 1)$
3. **for** each vertex $v_i, i \in \mathcal{N}(1, N)$ **do**
 - 3.1. **for** each vertex $v_j, j \in \mathcal{N}(i, N)$ **do**
 - 3.1.1. add the edge $\{i, j\}$ to $G_{N,p}$ with probability p if this edge is not present in $G_{N,p}$
 - 3.2. **endfor**
4. **endfor**

Figure 5.1. Random graph generation algorithm

uniformly from the interval $(0, 1)$, and view these as the coordinates of N points in the unit square. That is, group the $2N$ numbers pairwise and treat each pair as a coordinate. These points represent the vertices. We place an edge between two vertices if and only if their Euclidean distance is r or less, where $r = \sqrt{D_v/(N\pi)}$, i.e., both points lie in a circle of radius r . This expression for the radius can be obtained using the reasoning below: Since the vertices of a geometric graph are distributed uniformly in the unit square, we have N vertices in an area of 1. A vertex, not too close to the boundaries, is connected to every vertex in an area of πr^2 . Thus, the expected average degree D_v is $N\pi r^2$. We generated 15 geometric graphs whose properties are depicted in Table C.2.

The algorithm in Figure 5.2 generates a geometric graph U_{N,D_v} . The time complexity of the algorithm is $\mathcal{O}(N^2)$.

5.7.3 Grid Graphs

A random *grid* graph $G_{N,p}$ is a random planar graph with N vertices and an edge probability p . Each vertex in $G_{N,p}$ can have at most 4 adjacent vertices. The grids we use have a height of h and a width of w such that $N = hw$. That is, the grids have a rectangular shape. The expected average vertex degree D_v is $p(2(2N - h - w))/N$. We generated 9 grid graphs whose properties are depicted in Table C.3. The width of all the test graphs were set to 10. The

Algorithm: Geometric Graph Generator

Input: number N of vertices, expected average vertex degree D_v

Output: a geometric graph U_{N,D_v}

1. generate N vertices but 0 edges
 2. let $r \leftarrow \sqrt{D_v/(N\pi)}$
 3. **for** each vertex $v_i, i \in \mathcal{N}(1, N)$ **do**
 - 3.1. **for** each vertex $v_j, j \in \mathcal{N}(i, N)$ **do**
 - 3.1.1. add the edge $\{i, j\}$ to U_{N,D_v} if the distance between v_i and v_j is $\leq r$ and if this edge is not present in U_{N,D_v} .
 - 3.2. **endfor**
 4. **endfor**
-

Figure 5.2. Geometric graph generation algorithm

algorithm in Figure 5.3 generates a random grid. Its time complexity is $\mathcal{O}(N)$.

5.7.4 Ladder Graphs

A *ladder* graph is actually a grid graph but it can have a width of 2 or 3. We generated 6 random graphs whose properties are depicted in Table C.4. The widths of all the test graphs were set to 2. The grid generation algorithm can be used to generate ladder graphs also.

5.7.5 Tree Graphs

Trees [8] are best candidates for experiments on very sparse graphs. The average vertex degree in a tree is close to 2. We generated 3 tree graphs whose properties are depicted in Table C.5. An N -vertex tree T_N is generated by the algorithm in Figure 5.4. The time complexity of the algorithm is $\mathcal{O}(N)$.

Algorithm: Grid Generator

Input: number N of vertices, expected average vertex degree D_v , height h , and width w

Output: a grid $G_{N,p}$ with height h and width w

1. generate N vertices but 0 edges
 2. let $p \leftarrow D_v N / (2(2N - h - w))$
 3. **for** each vertex $v_i, i \in \mathcal{N}(1, N)$ **do**
 - 3.1. let $\text{down_neighbor} \leftarrow i + w$
 - 3.2. let $\text{right_neighbor} \leftarrow i + 1$
 - 3.3. **if** ($\text{down_neighbor} < N$) **then**
 - 3.3.1. add the edge $\{i, \text{down_neighbor}\}$ to $G_{N,p}$ with probability p
 - 3.4. **endif**
 - 3.5. **if** ($i \bmod w \neq (N - 1)$) /* if v_i is not at the very right */ **then**
 - 3.5.1. add the edge $\{i, \text{right_neighbor}\}$ to $G_{N,p}$ with probability p
 - 3.6. **endif**
 4. **endfor**
-

Figure 5.3. Grid generation algorithm

Algorithm: Tree Generator

Input: number N of vertices

Output: a tree T_N

1. generate N vertices but 0 edges
 2. let $S \leftarrow \{v_i\}$ where the vertex v_i is chosen randomly /* $i \in \mathcal{N}(1, N)$ */
 3. let T be the vertex set of T_N
 4. **repeat**
 - 4.1. randomly choose v_j from S /* $j \in \mathcal{N}(1, N)$ */
 - 4.2. randomly choose v_k from T /* $k \in \mathcal{N}(1, N)$ */
 - 4.3. add the edge $\{j, k\}$ to T_N if this edge is not present in T_N
 - 4.4. let $S \leftarrow S \cup \{v_k\}$
 5. **until** $S = T$
-

Figure 5.4. Tree generation algorithm

5.8 Test Hypergraphs

We did not use hypergraph instances that were randomly generated. Instead, we used real VLSI benchmark circuits as hypergraph instances. These circuits are a subset of the standard-cell circuits from **The International Workshop on Layout Synthesis'92 (LayoutSynth92)** which are maintained in and distributed by **Microelectronics Center of North Carolina (MCNC)** with the support of **ACM/SIGDA**. They are currently called the **ACM/SIGDA Design Automation Benchmarks**. In order to make these circuits to be used in the partitioning algorithms, we, like other researchers, deleted certain nonessential features of these circuits, for example, all the nets with only one terminal were deleted, the nets including a vertex (or cell in VLSI terminology) more than once were enforced to include the vertex only once. All the net weights were taken to be 1 whereas the vertex weights were calculated to be proportional to the area of the vertex. Since we usually did not determine the vertex weights by approximating the area of the vertex, the vertex weights happened to be large; yet, for some certain applications, they can be decreased by dividing the areas of all the vertices with a certain number depending on the application. The circuits are depicted in Table C.6. Henceforth we use the word circuits when we refer to the test hypergraphs.

5.9 General Comments on Experiments

For an experiment performed on a graph or hypergraph instance, we present at most two tables: one table for the average cutsizes obtained and another table for the average running times of the partitioning algorithms on these instances. This restriction is due to the large number of tables. The time of an partitioning algorithm on a problem instance is in *seconds*, and is equal to the sum of the time to read in the input problem instance, the time to create an initial partition of that instance, the time of all the passes performed until a locally minimum partition is found, the time to output the result, and finally the time to verify the cutsize at each pass during partitioning by the algorithms in Figure 4.4 and in Figure 4.5. Most of the time, the standard deviations are also given. The time of an partitioning algorithm on a problem instance was measured by the SunOS command `time`, and the time of the algorithm was obtained by adding the `user` time and `system` time (please refer to manpages

of time command for further information.)

5.10 General Comments for Experiments on Graphs

In the tables, there are two columns with the headings *RATIO* and *IMP*. The other column headings are self-explanatory. The *RATIO* value in a row was found by the equation

$$RATIO = \frac{(\text{Running time of } K\text{-PLM3})}{(\text{Running time of } K\text{-PLM1})} \quad (5.1)$$

where the running times were taken from the same row. Thus, *RATIO* value gives the ratio of the running time of K-PFM3 algorithm with respect to that of K-PLM1 algorithm. The *IMP* value in a row was found by the equation

$$IMP = 100.0 \times \frac{(\text{Cutsizes by } K\text{-PFM3}) - (\text{Cutsizes by } K\text{-PLM1})}{(\text{Cutsizes by } K\text{-PLM1})} \quad (5.2)$$

where the cutsizes values were taken from the same row. Thus, *IMP* value gives the percentage improvement done by K-PFM3 algorithm in the cutsizes with respect to that of K-PLM1 algorithm. The values between parantheses in the rows represent the respective standard deviations.

Recall that the definition of the freedom value function involves a parameter called the scale factor *S*. During experimenting with K-PFM-like algorithms on the test graphs, the setting of *S* was as follows: *S* = 400 when *N* = 250, *S* = 600 when *N* = 500, and *S* = 800 when *N* = 1000.

5.11 Performance of K-PFM Algorithms on Graphs

The K-PLM1 algorithm were run 100 times on a test graph whereas any of the K-PFM-like algorithms were run 10 times on the same test graph. This is because the running time of K-PLM1 algorithm is smaller compared to those of the K-PFM-like algorithms. The results of experiments on random graphs are given in Table C.8 and Table C.7, those on geometric graphs in Table C.10 and Table C.9, those on grid graphs in Table C.12 and Table C.11, those on ladder graphs in Table C.14 and Table C.13, those on tree graphs in Table C.16 and Table C.15,

We now present the general observations obtained from the experiments presented in the tables. Note that there can be some anomalies violating these general observations.

- The running time of any partitioning algorithm tend to correlate directly with the number of vertices, the average vertex degree, and the number of parts as expected from the time complexities of these algorithms.
- The partitioning algorithms can be ordered with respect to their running times, from the one with the largest running time to the one with the smallest running time, as K-PFM3, K-PFM2, K-PFM1, K-PLM1. Although K-PFM1 and K-PLM1 algorithms perform the same number of moves in pass, the running time of K-PFM1 happens to be larger than that of K-PLM1 because K-PFM1 algorithm uses such time consuming functions as the square root and exponential functions, performs more passes, and performs more update operations due to the fact that there are always N potential moves at any time in a pass where N is the number of vertices of the input graph.
- The cutsize obtained tend to increase as the number of vertices, the average vertex degree, and the number of parts increase.
- The improvement made by the K-PFM-like algorithms with respect to K-PLM1 algorithm happens to decline as the average vertex degree of the graph increases. This result was also observed by other researchers [5, 6] for bipartitioning and was claimed to be due to the presence of very few locally optimal partitions in such dense graphs. This claim has not been proved yet. Our observation indicates that this result also holds for multiple-way partitioning. Bui et al. [5, 6] propose a heuristic algorithm to improve the performance of KL bipartitioning algorithm based on this result. The heuristic first uses a maximum random matching algorithm to coalesce vertices into pairs, thus forming a smaller graph of higher average vertex degree, and then runs KL algorithm on this graph to obtain a bipartition. Vertex pairs in this bipartition are then separated to create for the original graph a bipartition that is then used as an initial partition for KL algorithm.
- The partitioning algorithms can be ordered with respect to the quality of the cutsize they found, from the best to the worst, as K-PFM3, K-PFM2,

K-PFM1, K-PLM1. The K-PFM-like algorithms outperform K-PLM1 algorithm drastically on almost all test graphs. The only anomaly occurred in bipartitioning the geometric graphs whose average vertex degrees were all equal to 16. This anomaly reveals that K-PLM1 algorithm is better in bipartitioning and in partitioning very dense graphs.

- The improvement achieved by the K-PFM-like algorithms becomes better than that by K-PLM1 algorithm as the test graph becomes more sparse. Since the real applications are usually very sparse, this feature is very promising.
- The K-PFM-like algorithms dominate K-PLM1 algorithm more in the reduction achieved in the cutsizes of the test graphs with some special structure such as the geometric graphs, the grids. This feature is very good because these graphs are closer to real applications [17, 15].
- The partitioning algorithms can be ordered with respect to their number of passes, from the one with the largest number of passes to the one with the smallest, as K-PFM1, K-PLM1, K-PFM2, K-PFM3. This is because K-PFM2 and K-PFM3 algorithms perform more moves in a pass than other algorithms, and the potential number of moves at any time in a pass of K-PFM1 algorithm is larger than that of K-PLM1 algorithm. The fact that K-PFM1 algorithm performs more number of passes than K-PLM1 algorithm although they perform the same number of moves in a pass may provide a support to the claim that K-PFM1 algorithm explores the search space better than K-PLM1 algorithm.
- The number of passes done by any of the partitioning algorithms seems to be proportional directly to the number of vertices, the average vertex degree, and the number of parts.
- The maximum average number of passes on random graphs is 13. That number is 9 on geometric graphs, 16 on grids, 11 on ladders, and 10 on trees although this maximum is far from the average of the average number of passes.
- The standard deviations in the cutsizes obtained by the K-PFM-like algorithms tend to be smaller than those by K-PLM1 algorithm. However, the ratios of the standard deviations to the respective average cutsizes happen to be larger for the K-PFM-like algorithms.

- The running times of the partitioning algorithms on the geometric graphs seem to be the smallest. This may be due to the fact that these graphs have built-in clusters which can be more easily identifiable.
- The running times of the partitioning algorithms on the random graphs seem to be the largest than those on the other types of graphs. This may be due to the smooth structure of this type of graphs.
- The running times of the partitioning algorithms on the grids, on the ladders and on the trees seem to be larger than expected. This may be because these graphs have so regular structure that the algorithms encounter more ties when they want to select a move.
- It seems that K-PLM1 algorithm is better at bipartitioning. K-PLM1 algorithm does bipartitioning better than it does K -way partitioning for $K > 2$. This may be due to the small number of solutions examined in bipartitioning.

5.11.1 Different Freedom Value Functions

The main criterion for the freedom value function for a vertex seems to be the one that is proportional directly to the move gain of the vertex, and indirectly to the move count of the vertex. Since the expression in Equation 4.7 is not the single freedom value function, we experimented with other freedom value functions all of which we devised. They are tabulated in the following table.

Label	Freedom Value Expression
R1	$\Phi_1 = \Phi_m(f, t) = 1/(1 + n_m^{1/2} e^{(-G_m(f,t)R)})$
R2	$\Phi_2 = \Phi_m(f, t) = 1/(1 + n_m^1 e^{(-G_m(f,t)R)})$
R3	$\Phi_3 = \Phi_m(f, t) = 1/(1 + n_m^2 e^{(-G_m(f,t)R)})$
R4	$\Phi_4 = \Phi_m(f, t) = 1/(1 + n_m^{1/3} e^{(-G_m(f,t)R)})$
R5	$\Phi_5 = \Phi_m(f, t) = 1/(1 + n_m^{1/4} e^{(-G_m(f,t)R)})$
R6	$\Phi_6 = \Phi_m(f, t) = 1/(1 + n_m^{1/5} e^{(-G_m(f,t)R)})$
R7	$\Phi_7 = \Phi_m(f, t) = 1/(n_m^{1/2}(1 + e^{(-G_m(f,t)R)}))$
R8	$\Phi_8 = \Phi_m(f, t) = 1/(n_m^{1/2} + e^{(-G_m(f,t)R)})$
R9	$\Phi_9 = \Phi_m(f, t) = (G_m(f, t) + G_{max})/(2n_m^{1/2}G_{max})$

Note that Φ_1 is identical to the one in Equation 4.7 and that the heading *label* refers to the column label of the tables presenting the results of the experiments with these freedom value functions. The results obtained with these freedom value functions are presented in Table C.17 for K-PFM1 algorithm, Table C.18 for K-PFM2 algorithm, and Table C.19 for K-PFM3 algorithm. Each cutsize average is the average of the cutsize values from 12 runs with each algorithm. The entries in the tables do not represent the actual average cutsizes obtained but the ratio of the average cutsize with respect to the one by Φ_1 . Also, the entries in a pair of parentheses in the first column of a table represent the average cutsize values obtained by Φ_1 so that the values for the other freedom value functions can be found by multiplying this cutsize value by respective ratio values in the other columns. During experimentation with these freedom value functions, the same scale factor S was used for all of them and the experiments were performed only on the random graphs. Based on the results presented in these tables, we can list our general observations as follows:

- For K-PFM1 algorithm, the functions Φ_1 and Φ_8 are better as the graph becomes more sparse. The functions Φ_5 and Φ_6 are better as the graphs becomes denser.
- For K-PFM2 algorithm, the function Φ_4 gives the best results.
- For K-PFM3 algorithm, the functions Φ_4 and Φ_7 give the best results.
- For all the algorithms, the function Φ_9 gives the worst results.
- As the graph becomes denser, the results by all these functions get closer.
- The results by these functions except for the function Φ_9 are very close to each other.
- Since the overall winner seems to be the function Φ_4 , it reveals that the effect of the move count of a vertex on the freedom of the vertex should be reduced.

5.11.2 Determining Scale Factor

Note that the scale factor S is used in the freedom value function and is a parameter of the generic K-PFM algorithm. It controls the mapping density of the vertices into buckets. Recall that $H = (V, E)$ being a hypergraph and

$\Pi = (P_1, \dots, P_K)$ a K -way partition of H , the freedom value $\Phi_m(f, t)$ of a vertex v_m in the part P_f with respect to the part P_t is defined as

$$\Phi_m(f, t) = \frac{1}{1 + \sqrt{n_m} e^{(-G_m(f, t)R)}} \text{ if } n_m \neq 0 \quad (5.3)$$

where

$$R = \left(\frac{1}{G_{max}}\right) \ln\left(\frac{1-\epsilon}{\epsilon}\right) \text{ with } \epsilon = 0.01 \quad (5.4)$$

When the moves associated with v_m are mapped into the buckets, we use the form $\lfloor S\Phi_m(f, t) \rfloor$ instead of $\Phi_m(f, t)$ (henceforth referred to as the floored freedom value function.) Note that the freedom value function in Equation 4.7 preserves the order of the move gains to which it is proportional as shown by the inequality 4.9. However, the floored form of this function does not preserve the order so that the gains of the moves in the same bucket list are not identical. This feature of the floored freedom value function randomizes the move selection process. Now, let Φ_1 and Φ_2 be two freedom value functions belonging to two different vertices but in the same move direction. Also, assume that n represents the same move counts of these vertices, and G_1 and G_2 the move gains of these vertices in the same move direction. Thus,

$$\Phi_1 = \frac{1}{1 + \sqrt{n} e^{(-G_1 R)}} \text{ if } n \neq 0, \quad (5.5)$$

and

$$\Phi_2 = \frac{1}{1 + \sqrt{n} e^{(-G_2 R)}} \text{ if } n \neq 0. \quad (5.6)$$

Let $G_2 = G_1 + \Delta G$ and $G_1 = G$ where $\Delta G > 0$. Now, we want to derive a lower bound to the scale factor S based on the requirement that the moves with the gains G_2 and G_1 be mapped into different buckets. If we require to satisfy the inequality $\lfloor S\Phi_2 \rfloor \geq \lfloor S\Phi_1 \rfloor + 2$, we obtain the inequality $S\Phi_2 \geq S\Phi_1 + 1$ after some algebraic manipulations involving the properties of the floor operation. After a long list of algebraic manipulations, we obtain the following inequality.

$$S \geq \frac{\sqrt{n}}{\rho^{G/G_{max}}(\rho^{\Delta G/G_{max}} - 1)} + \frac{\rho^{G/G_{max}} \rho^{\Delta G/G_{max}}}{\sqrt{n}(\rho^{\Delta G/G_{max}} - 1)} + \frac{\rho^{\Delta G/G_{max}} + 1}{\rho^{\Delta G/G_{max}} - 1} \quad (5.7)$$

where $\rho = (1 - \epsilon)/\epsilon$ and $\rho = 99$. For $\Delta G = 1$, $n = K^2$ (because the average move count of a vertex in an algorithm with NK^2 moves in a pass is K), this inequality reduces to

$$S \geq \frac{K}{\rho^{G/G_{max}}(\rho^{1/G_{max}} - 1)} + \frac{\rho^{G/G_{max}} \rho^{1/G_{max}}}{K(\rho^{1/G_{max}} - 1)} + \frac{\rho^{1/G_{max}} + 1}{\rho^{1/G_{max}} - 1}. \quad (5.8)$$

Notice that the maximum vertex degree G_{max} is equal to 140 for the test graphs and equal to 9 for the benchmark circuits. Also notice that the gain G in the preceding inequality can be at least $-G_{max}$ and at most G_{max} . The following table lists the different values of S used in our experiments.

Label	Value of S	Label	Value of S
R1	10	R6	600
R2	50	R7	800
R3	100	R8	1000
R4	200	R9	2000
R5	400	R10	3000

The results obtained with these scale factor values are presented in Table C.20 for K-PFM1 algorithm, Table C.21 for K-PFM2 algorithm, and Table C.22 for K-PFM3 algorithm. Each cutsize average is the average of the cutsize values from 12 runs with each algorithm. The entries in the tables do not represent the actual average cutsizes obtained but the ratio of the average cutsize with respect to the one by $S = 10$. Also, the entries in a pair of parentheses in the first column of a table represent the average cutsize values obtained by $S = 10$ so that the values for the scale values can be found by multiplying this cutsize value by respective ratio values in the other columns. Also note that the heading *label* in the above table refers to the column label of the tables presenting the results of the experiments. Based on the results presented in these tables, we can list our general observations as follows:

- As the graph becomes denser, the results get closer.
- Larger values of S seem to have a better effect on the cutsize but, for example, when we increase S from 50 to 3000, i.e., by a factor of 60, we get a 1% improvement in the cutsize on the random graph with $N = 500$, $D = 2$, and $K = 2$.
- The results obtained with the different values of S seem to be very close to one another provided that $S \geq 50$. Thus, the rule is that we should choose a scale factor so that it is not too small as well as not too large. The lack of a strong correlation between the cutsize and the scale factor is very beneficial since the scale factor is involved in both the space requirement and the running time of the K-PFM-like algorithms.

- There is at most a three fold increase in the running times of the K-PFM-like algorithms when the scale factor is increased up to the value of 3000.

5.12 Performance of K-PLM Algorithms on Graphs

The results of experiments on random graphs are given in Table C.23, those on geometric graphs in Table C.24. For each algorithm, 10 runs were performed.

We now present the general observations obtained from the experiments presented in the tables. Note that there can be some anomalies violating these general observations.

- Almost always let $K_1K_2 = NK^2$. That is, perform large number of moves in a pass.
- As the graph gets more sparse, the value of K_2 should be increased. Here, the algorithm with $K_2 = 3N/4$ performs better. As the graph gets denser, the value of K_2 should be decreased. Here, the algorithm with $K_2 = N/4$ performs better. As the average vertex degree of the graph goes in between, the value of K_2 should be in between. Here, the algorithm with $K_2 = 2N/4$ performs better. Note that $K_1K_2 = NK^2$ in each case.
- Even when the algorithms has $K_1K_2 = N$, namely, they perform the same number of moves in a pass as that by K-PFM1, the results get better as more than one move-and-lock phase is carried out in a pass.

5.13 General Comments for Experiments on Hypergraphs

In the tables, there are two columns with the headings *RATIO* and *IMP*. The other column headings are self-explanatory. The *RATIO* value in a row was found by the equation

$$RATIO = \frac{(\text{Running time of } K-PLM3)}{(\text{Running time of } K-PLM1)} \quad (5.9)$$

where the running times were taken from the same row. Thus, *RATIO* value gives the ratio of the running time of K-PFM3 algorithm with respect to that of K-PLM1 algorithm. The *IMP* value in a row was found by the equation

$$IMP = 100.0 \times \frac{(Cutsize\ by\ K-PFM3) - (Cutsize\ by\ K-PLM1)}{(Cutsize\ by\ K-PLM1)} \quad (5.10)$$

where the cutsize values were taken from the same row. Thus, *IMP* value gives the percentage improvement done by K-PFM3 algorithm in the cutsize with respect to that of K-PLM1 algorithm. The values between parantheses in the rows represent the respective standard deviations.

Recall that the definition of the freedom value function involves a parameter called the scale factor *S*. During experimenting with K-PFM-like algorithms on the circuits, the setting of *S* was as follows: $S = 200$ when $0 \leq N \leq 200$, $S = 800$ when $200 \leq N \leq 1000$, and $S = 2000$ when $1000 \leq N \leq 3100$.

5.14 Performance of K-PFM Algorithms on Hypergraphs

The K-PLM1 algorithm were run 20 times on a circuit whereas any of the K-PFM-like algorithms were run 10 times on the same circuit. This is because the running time of K-PLM1 algorithm is smaller compared to those of the K-PFM-like algorithms. The results of experiments on the circuits are given in Table C.26, Table C.27, and Table C.25.

We now present the general observations obtained from the experiments presented in the tables.

- The running time of any partitioning algorithm tends to correlate directly with the number of vertices, and the number of parts as expected from the time complexities of these algorithms.
- The partitioning algorithms can be ordered with respect to their running times, from the one with the largest running time to the one with the smallest running time, as K-PFM3, K-PFM2, K-PFM1, K-PLM1. The explanation is the same as the one for graphs.
- The cutsize obtained tend to increase as the number of vertices, and the number of parts increase.

- The partitioning algorithms can be ordered with respect to the the quality of the cutsizes they found, from the best to the worst, as K-PFM3, K-PFM2, K-PFM1, K-PLM1. The K-PFM-like algorithms outperform K-PLM1 algorithm drastically on almost all circuits except that K-PLM1 algorithm is better than K-PFM1 algorithm when $K = 2$.
- The partitioning algorithms can be ordered with respect to the number of times the algorithm found the minimum cutsizes for a certain circuit, from the one that found the minimum cutsizes the most to the one with the least, as K-PFM3, K-PFM2, K-PLM1, K-PFM1. This order is due to the poor performance of K-PFM1 algorithm for bipartitioning. However, they can be ordered with respect to the quality of the minimum cutsizes they found, from the best one to the worst, as K-PFM3, K-PFM2, K-PFM1, K-PLM1.
- The improvement made by the K-PFM-like algorithms seems to correlate directly with the average net degree but there is an anomaly for the circuit primary1.
- The partitioning algorithms can be ordered with respect to their number of passes, from the one with the largest number of passes to the one with the smallest, as K-PFM1, K-PLM1, K-PFM2, K-PFM3.
- The maximum average number of parts on the circuits is 16 although this maximum is far from the average of the average number of passes.
- K-PLM1 algorithm performs better in bipartitioning.

5.15 Performance of K-PLM Algorithms on Hypergraphs

We only ran K-PLM11 algorithm 10 times on the circuits on which the K-PFM-like algorithms performed the best and the worst. The results by K-PLM1 algorithm and the K-PFM-like algorithms were taken from the tables mentioned in the preceding section. The results of experiments on the circuits are given in Table C.28, and Table C.29.

From the tables, it reveals that the performance of K-PLM11 algorithm is better than those of K-PLM1 and K-PFM1 algorithms. However, the other

K-PFM-like algorithms outperformed K-PLM11 algorithm. Interestingly, K-PLM11 algorithm like K-PLM1 algorithm performed better when the circuit was bipartitioned.

5.16 Behaviour of Freedom Value Function

A plot of the freedom value function with respect to the move gain at different move counts is given in Figure B.1. The move counts represent the average and the maximum move counts that can occur in the K-PFM-like algorithms. We now present some general observations on these curves.

- If the move counts are not too large, the shape of the curve does not change. However, when the move counts are sufficiently small then the freedom value function maps more moves into the bucket with the largest index, and more moves into the bucket with the smallest index but it better differentiates the moves with the gains in between. When the move counts are sufficiently large then the freedom value function maps more moves into the bucket with the smallest index but it better differentiates the moves with the gains not too small. Hence, the freedom value function does not concern small and large gains much at the earlier partitioning steps but does concern medium and large gains much at the later partitioning steps.
- The floored freedom value function incorporates some randomization into the partitioning process. Thus, a move selected at one partitioning step does not necessarily correspond to the move with the largest gain at that step.

5.17 Convergence of Algorithms

Plots of the convergence curves of the algorithms are given in Figure B.2, Figure B.3, Figure B.4, Figure B.5, Figure B.6, and Figure B.7. These curves represent the general trends for each algorithm. We now present some general observations on these curves.

- The curves first slope sharply downward and then smooth out. Since the initial solution is randomly generated, improving it happens to be very easy. But, later partitioning steps involve a more powerful exploration capability and so the improvement done at the later steps reduces.
- The convergences of K-PLM1 algorithm is very rapid with respect to those of the other algorithms.
- For K-PFM1 algorithm, it is apparent that the number of passes increases with the increasing number of parts.
- For K-PLM-like algorithms, the curves become more spiky as the number of moves in a move-and-lock phase goes up.

5.18 Distribution of Cutsizes

Note that the optimum cutsizes of the geometric graph with $N = 500$ and $D_{exp} = 2$ when the graph is bipartitioned is zero. We conducted an experiment in which we did 15000 runs of both K-PFM1 and K-PLM1 algorithms to bipartition this graph. The histograms (though it is a line graph for the sake of clarity) are given in Figure B.8. In the figure, the x-axis represents the cutsizes in the range from 0 to the maximum one encountered. The y-axis represents the number of times each cutsize has been found by these algorithms. The optimum cutsize was found by K-PFM1 algorithms 3340 times but by K-PLM1 algorithm only 9 times. In addition, K-PFM1 algorithm found the small cutsize values more than K-PLM1 did. In other words, the probability that K-PFM1 algorithm found the optimum on this graph is 0.22 and the probability that K-PLM1 algorithm found the optimum on this graph is 0.0006, which is less than that of K-PFM1 algorithm by a factor of 371. Also, the average cutsize of the cutsizes by K-PFM1 algorithm is 4.99 and that by K-PLM1 algorithm is 21.36. Notice that the average cutsize found by K-PLM1 algorithm in 15000 runs is very close to the one in 100 runs. This fact seem to provide a support that the results of K-PLM1 algorithm cannot be improved substantially by performing large number of runs of it.

5.19 Distribution of Move Gains

The move gains of the moves selected during the partitioning process of an algorithm may provide another picture of the algorithm. We ran K-PLM1, K-PFM1, and K-PLM3 algorithms on the geometric graph with $N = 250$ and $D_v = 2$. The graph was partitioned into 4 parts. The same initial partitions were used for three of the algorithms. For each algorithm, we present two plots: one indicates the change of the gain of the selected move at each move, and the other the change of the cutsize at again each move. The change of the cutsize is drawn so that the x-axes of the plots match exactly. The cutsize curve does not represent the one after the prefix sum operation, which is the case in the above convergence curves. In order to provide clear plots, we selected a very small problem instance. Each group of 250 moves corresponds to a pass. Based on these plots, we can make the following general observations:

- For K-PLM1 algorithm, in the first pass, larger gains occur at the first half and the second half of the pass mainly includes only moves with negative gains. In later passes, the curves become more spiky and again the second half of each pass includes moves with negative gains. This fact is exactly the one we mentioned before stating our main claim in Section 4.10. It seems that K-PLM1 algorithm wastes the half of each pass.
- Curves corresponding to K-PFM1 and K-PLM3 algorithms are similar. In the first passes, the first halves are similar to that of K-PLM1 algorithm but the second halves do include more moves with nonnegative gains. This is an effect of allowing a vertex to be reselected. Note that the curves corresponding to later passes are spiky but less spiky than that of K-PLM1 algorithm.

You should refer to Section 4.10 to compare what is claimed there with the data presented in these plots. Note that these plots represent the general trends of the algorithms. They are not the special cases of the algorithms.

Chapter 6

CONCLUSIONS

In this work, we reformulated the multiple-way graph and hypergraph partitioning concepts in a general way. We can use this formulation in algorithms that do not use the locking mechanism at all. Rewriting the initial gain computation and gain update algorithms in terms of the cost concept resulted in simplifications in these algorithms, which constitute a very important part of any partitioning algorithms.

After realizing that allowing a vertex to move only once in a pass tended to degrade the performance of the partitioning algorithms, we proposed two novel approaches for multiple-way graph and hypergraph partitioning. Each approach includes a generic algorithm which can be used to generate many partitioning algorithm by changing the parameters in these generic algorithms. Usually, these parameters can be set in such a way that a better performance is obtained by spending more time. The proposed algorithms are expected to explore the search space of the problems better because of two reasons: one of the reasons is that they examine more solutions during performing the same number of moves as does Sanchis' algorithm, which is the most sophisticated multiple-way partitioning algorithm based on Kernighan-Lin's minimization technique, and the other reason is that they allow a move to be reselected as long as its selection is profitable and so do not restrict the partitioning process. One of the proposed algorithm does not use locking at all by introducing a new metric, called freedom value. This metric has many interesting features, one of which is that it allows a more randomized partitioning process.

We did many experiments to evaluate the performance of the Sanchis' algorithm and the proposed algorithm on both randomly generated graph instances

and benchmark circuits, which correspond to hypergraph instances. The proposed algorithms outperformed Sanchis' algorithm drastically on the graph instances. We observed that the performance of Sanchis' algorithm got better as the average degree of the test graph increased in multiple-way partitioning also. This observation extends the one noted for the bipartitioning case. The better performance of the proposed algorithm on graphs that are closer to the real applications is very promising. The proposed algorithms also yielded very good results on the benchmark circuits. During our experimentation, we also noted that Sanchis' algorithm tended to perform better for bipartitioning. This observation reveals that Sanchis' algorithm produces better results when the search space is smaller, which is the case both in bipartitioning and in partitioning of large average degree graphs. Note that Sanchis' algorithm represents all the previous partitioning algorithm based on Kernighan-Lin's minimization technique, that is, any observation on this algorithm is also applies to those employing the same technique.

One of the proposed algorithms also includes Sanchis' algorithm as a special case. The proposed algorithm convey all the advantages of the algorithms based on Kernighan-Lin's minimization technique such as their robustness. However, they do not convey many disadvantages of those algorithms such as their poor performance on sparse test cases.

The proposed K-PFM-like algorithms seem to perform better under tight balance conditions than Sanchis' algorithm since the number of move directions always stays the same during iterative partitioning. Also, the freedom value function can simulate the locking mechanism if the move counts of the vertices after their moves are increased by a very large number so that the moves associated with these vertices go into the bucket list with index zero.

If one has a program implementing Sanchis' algorithm, the modification of this program to implement the generic K-PLM algorithm seems to be very easy. Moreover, a K-PLM-like algorithm can be used to generate an initial solution to a K-PFM-like algorithm but the opposite is also possible. This is because a solution generated by one of the proposed algorithms is usually not a local minimum of the other algorithm due to the different mechanisms employed during partitioning.

The proposed algorithms do not rule out all the previous partitioning algorithms. The ideas introduced by these algorithms can also be applied to the

previous algorithms. For example, performing many move-and-lock phases in a pass can easily be utilized even in Kernighan-Lin algorithm with the swap-neighborhood structure. We expect that the proposed algorithms will be powerful competitors to the existing algorithms.

As a future study, we can try to optimize the parameters of the proposed generic algorithms although the performance of these algorithms did not correlate strongly with some of these parameters such as the scale factor. In addition, we can try to reduce the time complexities of the proposed algorithms. At each step during iterative partitioning in the proposed algorithms (also in the previous algorithms), only one move is selected. We can employ new heuristics such that a group of moves can be selected at each step. We can also redefine the move counts of vertices such that the vertices have a number of move counts each of which is with respect to a different part in the partition. Application of the ideas introduced in this work to the areas where the partitioning is a very useful tool is an open arena. For example, the proposed algorithms can also be used for mapping and VLSI placement without too much modification effort. We expect similar good performance of these algorithms in these areas.

Chapter 7

APPENDICES

Appendix A

FILE FORMATS

Let $H = (V, E)$ be a hypergraph with N vertices and M edges. The file format for H is as follows:

```
N
M
<net 1 weight> <net 1 degree> <terminals of net 1 here>
<net 2 weight> <net 2 degree> <terminals of net 2 here>
.....

<net M weight> <net M degree> <terminals of net M here>
<weight of vertex 1>
<weight of vertex 2>
.....

<weight of vertex N>
```

Let $G = (V, E)$ be a graph with N vertices and M edges. The file format for G is as follows:

```
N
M
<edge 1 weight> <edge 1 degree> <end vertex 1> <end vertex 2>
<edge 2 weight> <edge 2 degree> <end vertex 1> <end vertex 2>
.....
```

```
<edge M weight> <edge M degree> <end vertex 1> <end vertex 2>  
<weight of vertex 1>  
<weight of vertex 2>  
.....  
  
<weight of vertex N>
```

Appendix B

PLOTS FOR EXPERIMENTS

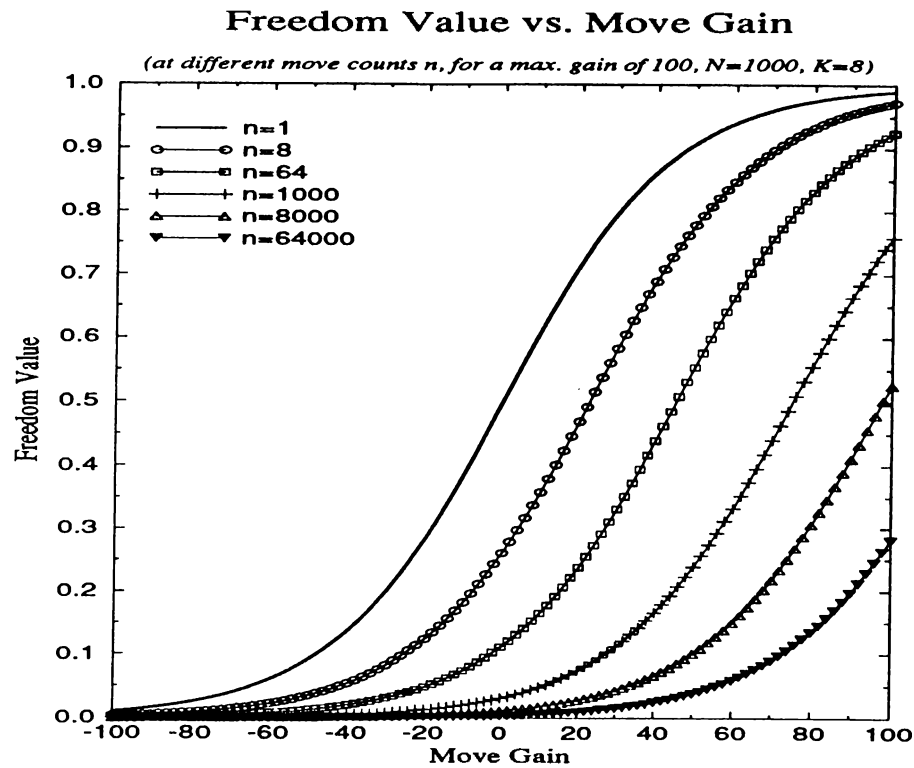


Figure B.1. Freedom Value for move gains at different move counts n , for $G_{max} = 100$, $N = 1000$, and $K = 8$

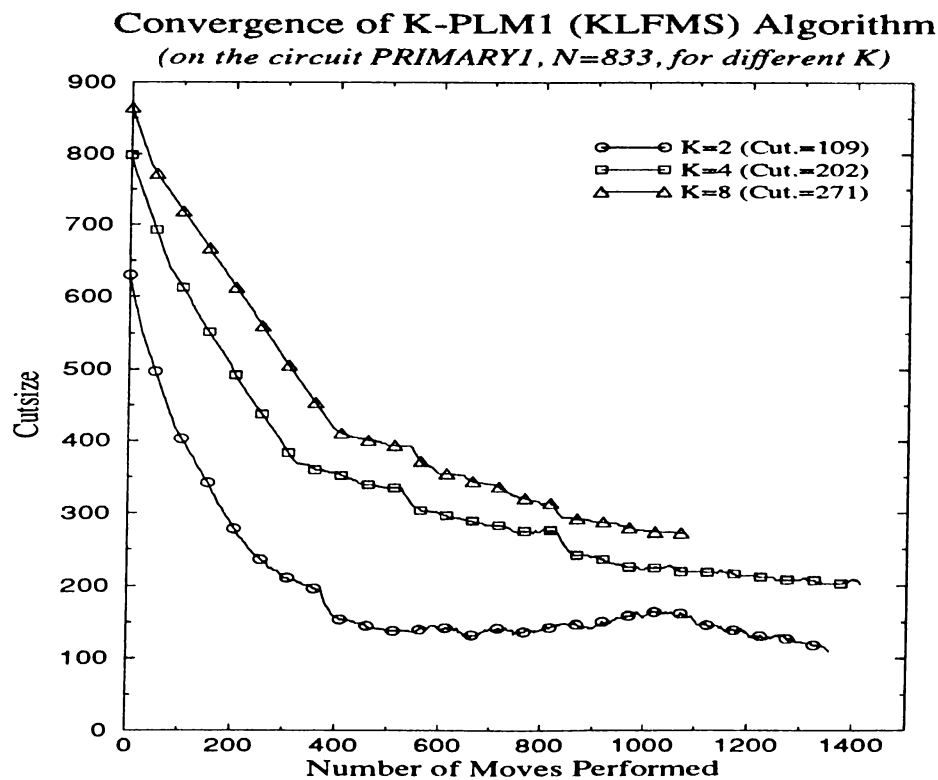


Figure B.2. Convergence of K-PLM1 Algorithm, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 2, 4$, and 8

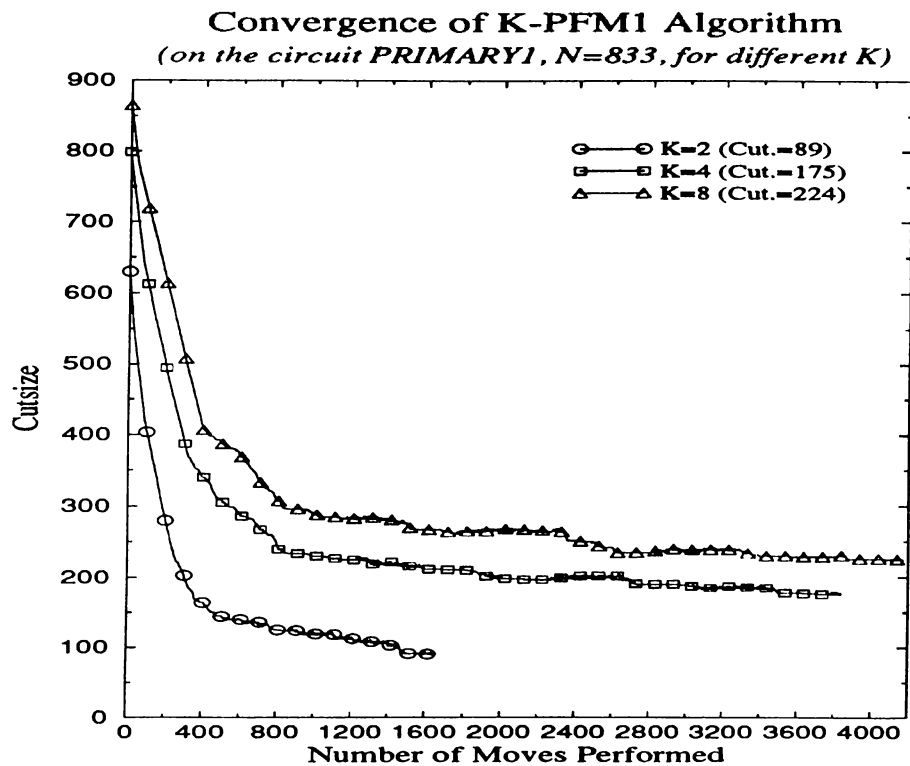


Figure B.3. Convergence of K-PFM1 Algorithm, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 2, 4$, and 8

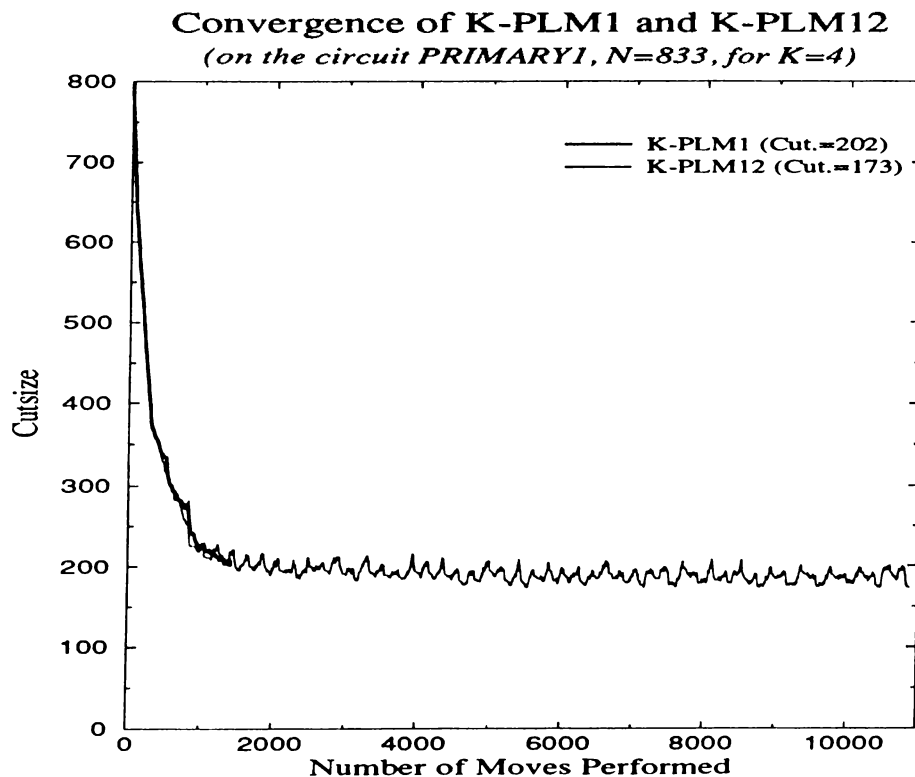


Figure B.4. Convergence of K-PLM1 and PLM12 Algorithms, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 4$

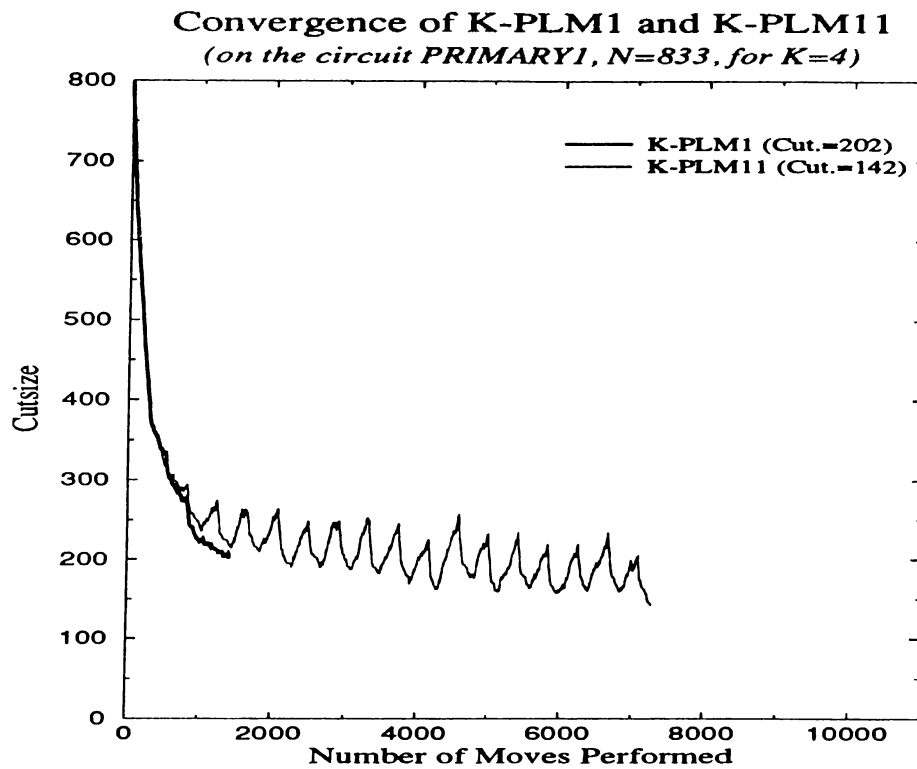


Figure B.5. Convergence of K-PLM1 and PLM11 Algorithms, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 4$

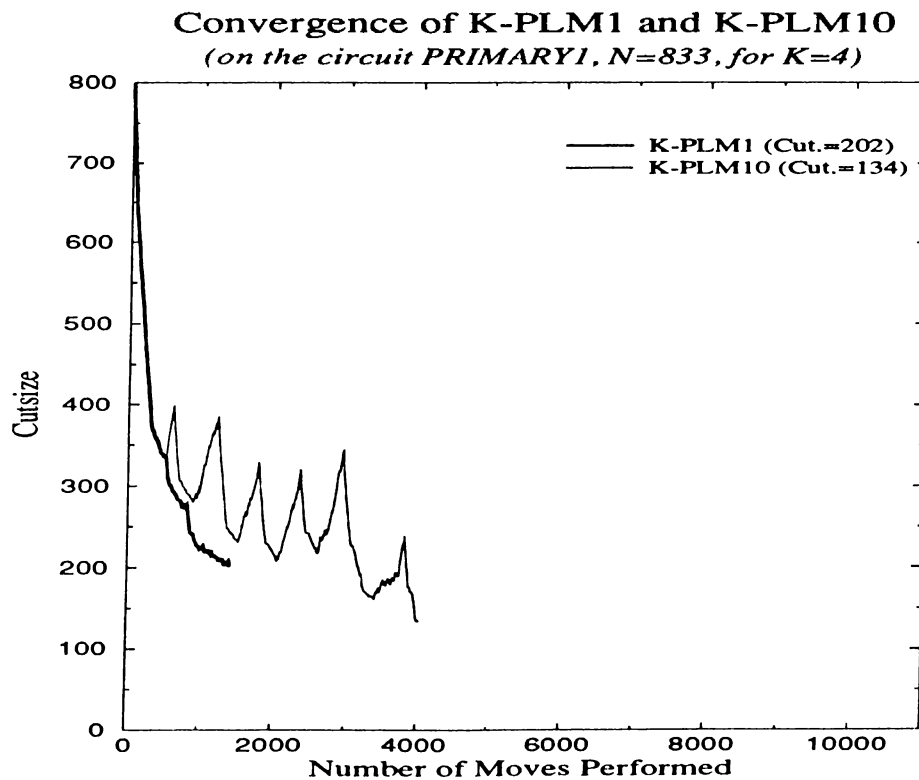


Figure B.6. Convergence of K-PLM1 and PLM10 Algorithms, a plot of cutsizes versus number of moves performed until local minimum is found, for $K = 4$

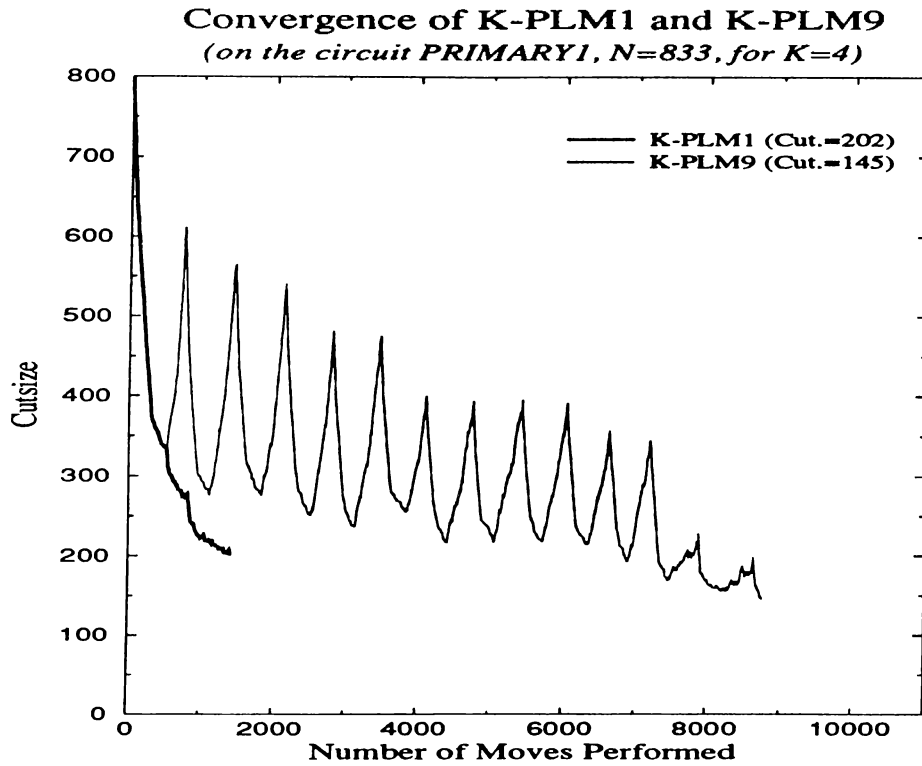


Figure B.7. Convergence of K-PLM1 and PLM9 Algorithms, a plot of cutsize versus number of moves performed until local minimum is found, for $K = 4$

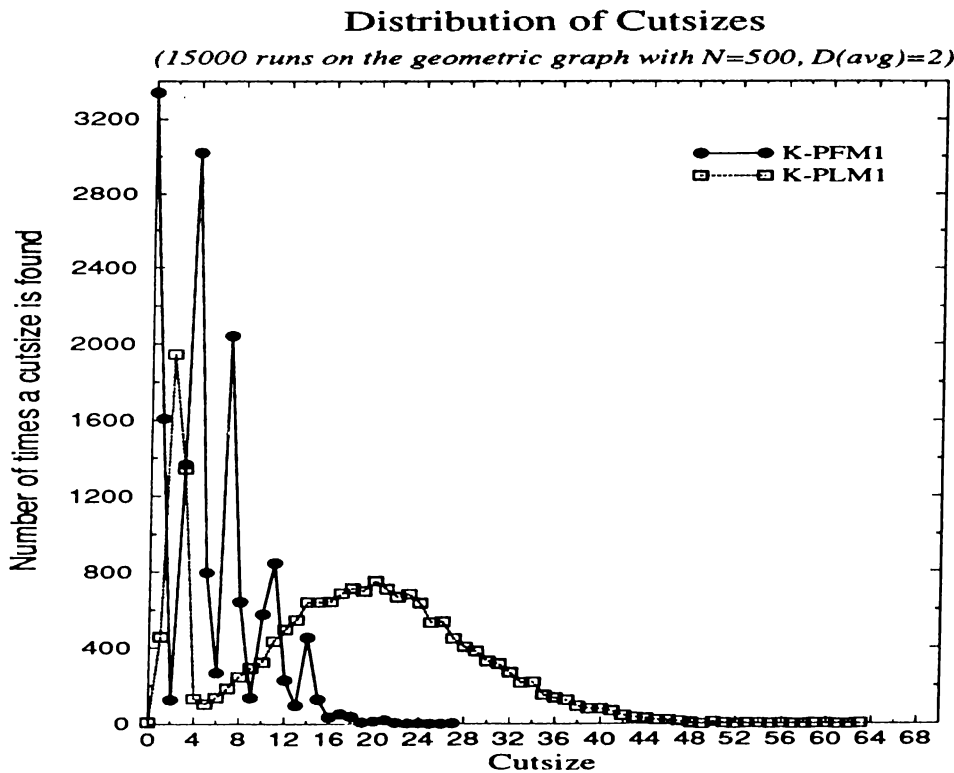


Figure B.8. Distribution of cutsizes for K-PLM1 and K-PFM1 Algorithms, a cutsize on x-axis has been found the corresponding value on y-axis times by the algorithms

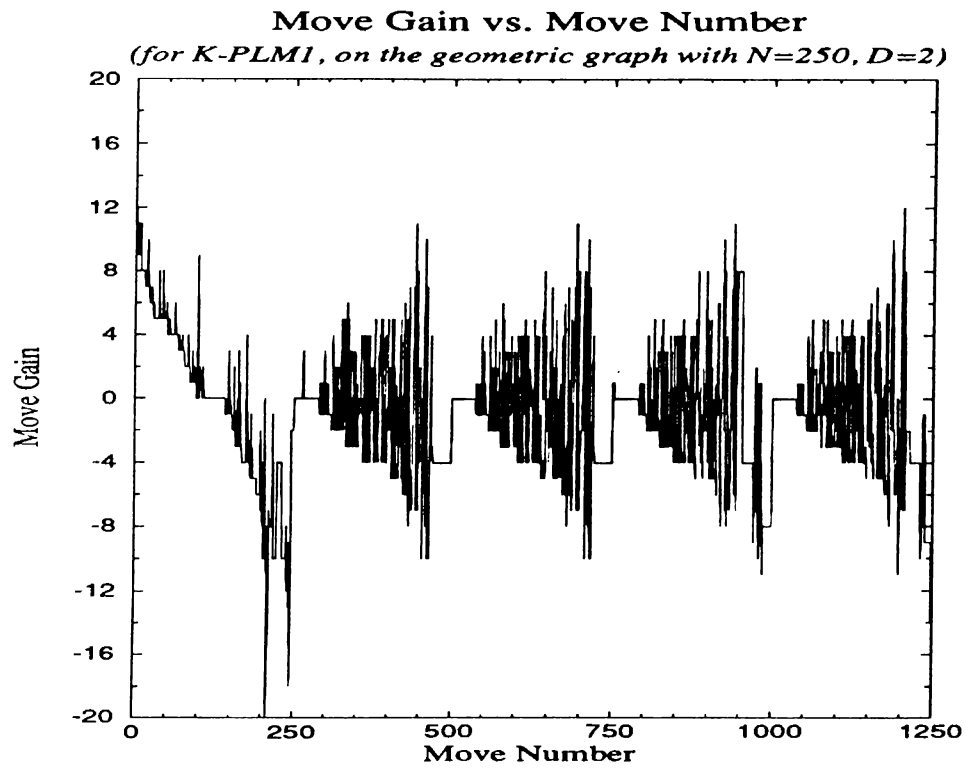


Figure B.9. Change of gains of selected moves in K-PLM1 Algorithm

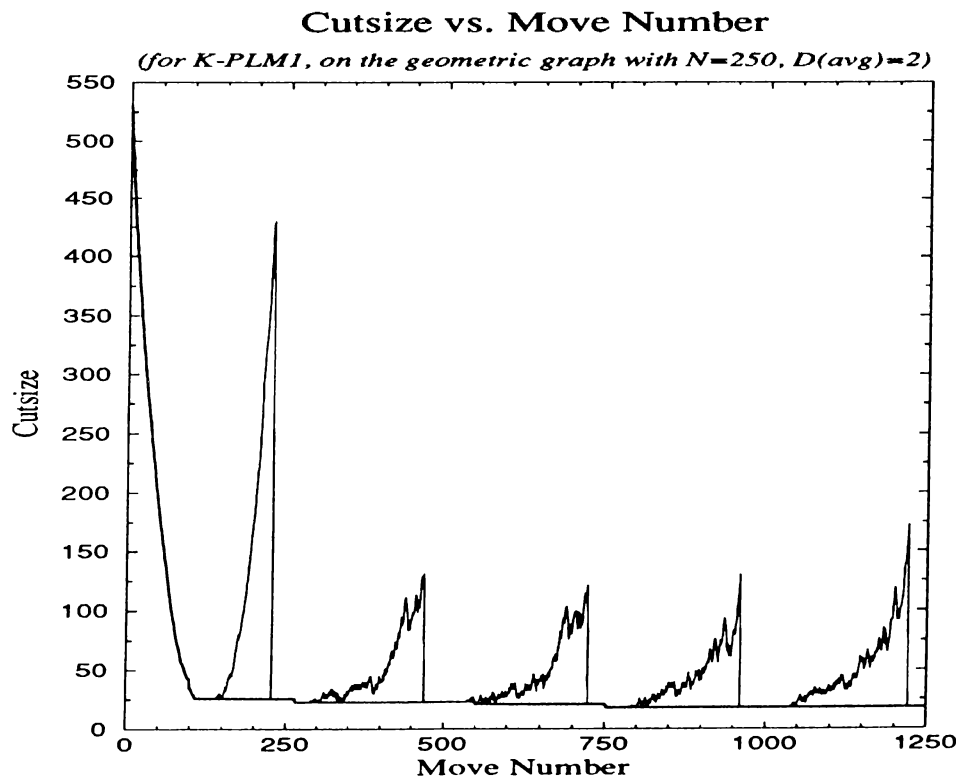


Figure B.10. Change of cutsize at each move in K-PLM1 Algorithm (final cutsize is 18)

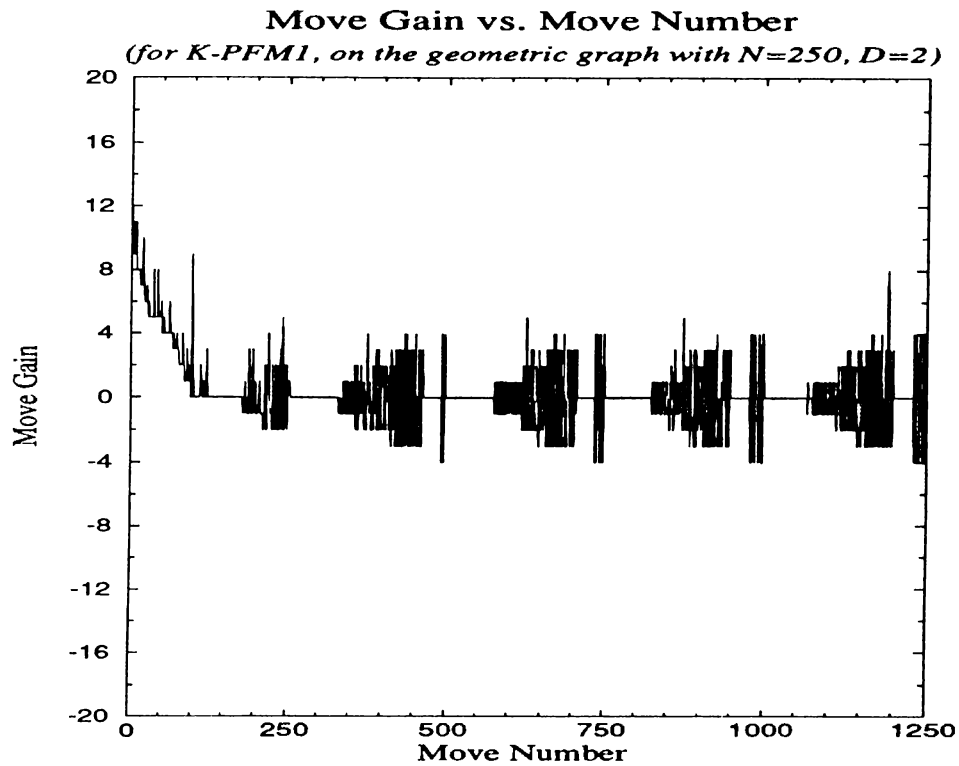


Figure B.11. Change of gains of selected moves in K-PFM1 Algorithm

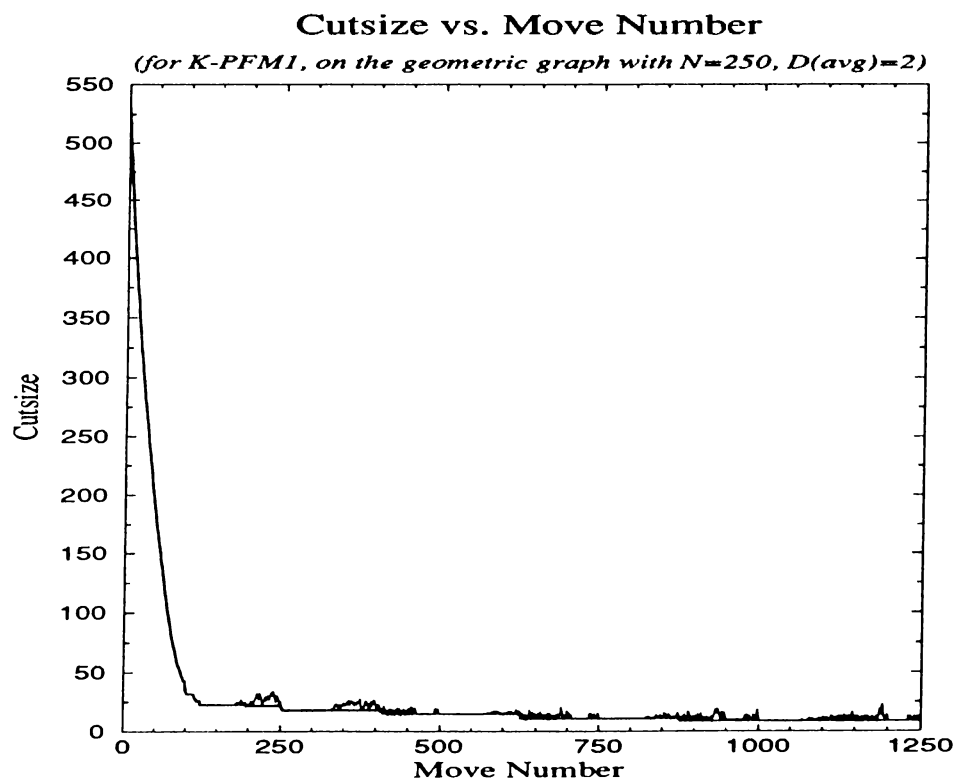


Figure B.12. Change of cutsizes at each move in K-PFM1 Algorithm (final cutsizes is 9)

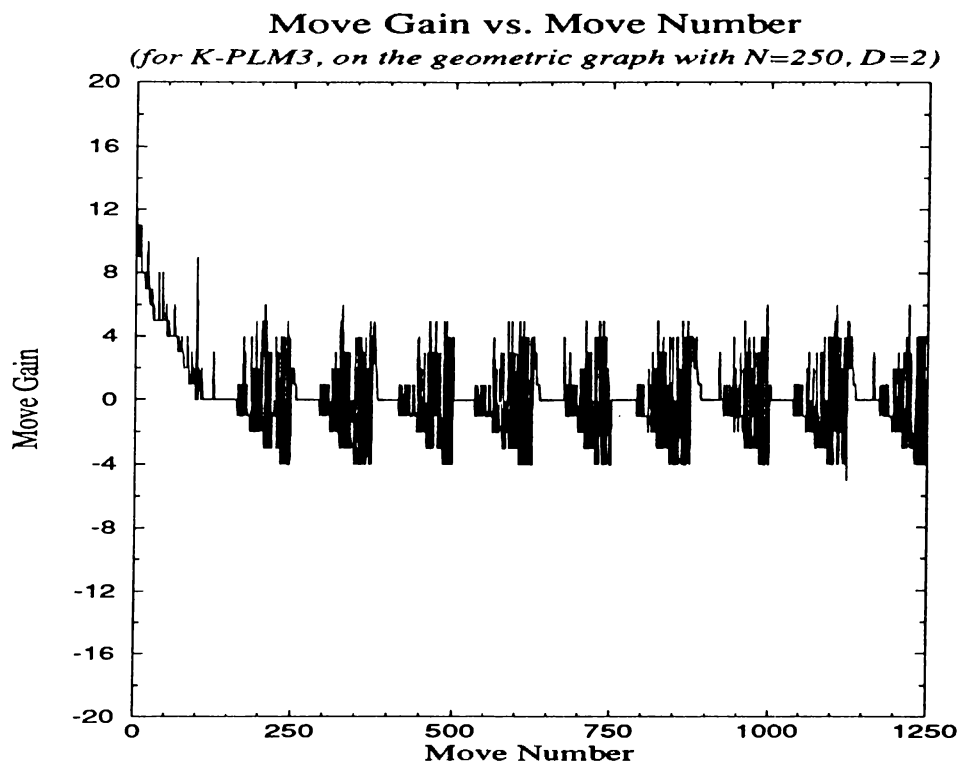


Figure B.13. Change of gains of selected moves in K-PLM3 Algorithm

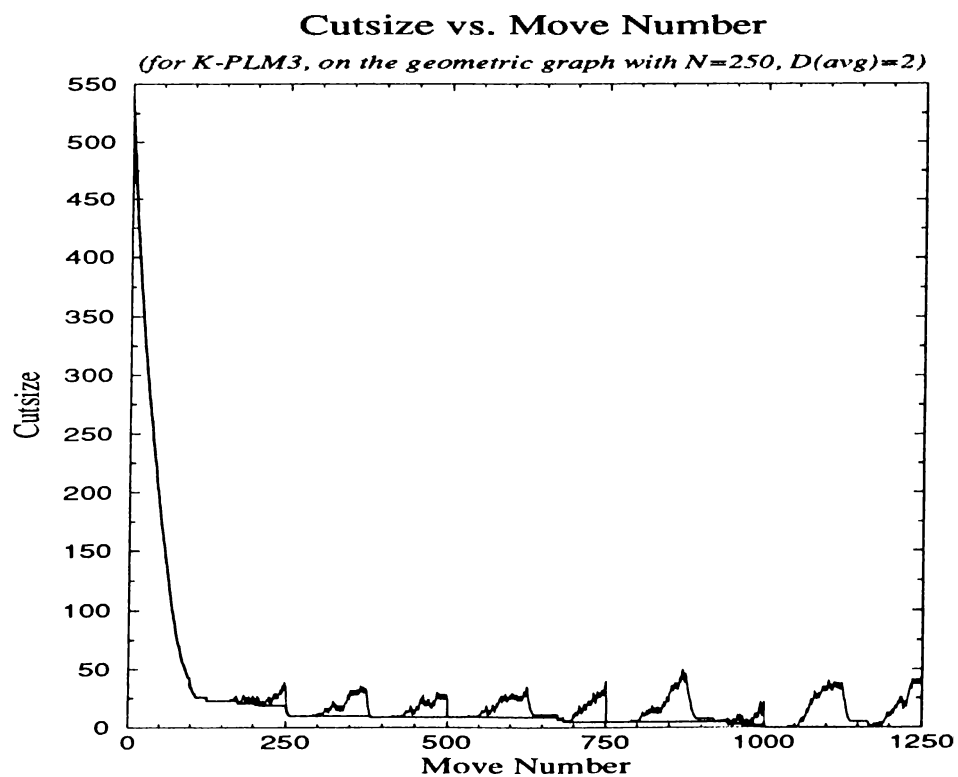


Figure B.14. Change of cutsizes at each move in K-PLM3 Algorithm (final cutsizes is 0)

Appendix C

TABLES FOR EXPERIMENTS

Table C.1. Properties of Random Test Graphs

RANDOM GRAPHS										
N	M	w_T	c_T	D_{exp}	D_{act}	$D_{v,max}$	w_{max}	w_{min}	c_{max}	c_{min}
250	242	622	717	2	1.94	6	4	1	5	1
250	359	639	1106	3	2.87	9	4	1	5	1
250	552	612	1684	4	4.42	11	4	1	5	1
250	1002	614	3001	8	8.02	15	4	1	5	1
250	2023	622	6013	16	16.18	27	4	1	5	1
500	488	1248	1487	2	1.95	9	4	1	5	1
500	700	1258	2051	3	2.80	8	4	1	5	1
500	984	1265	2928	4	3.94	11	4	1	5	1
500	1989	1253	5879	8	7.96	16	4	1	5	1
500	3980	1273	11960	16	15.92	27	4	1	5	1
1000	992	2529	3017	2	1.98	8	4	1	5	1
1000	1485	2513	4485	3	2.97	9	4	1	5	1
1000	1928	2499	5740	4	3.86	12	4	1	5	1
1000	4005	2498	12051	8	8.01	18	4	1	5	1
1000	7973	2494	23956	16	15.95	28	4	1	5	1

Table C.2. Properties of Geometric Test Graphs

GEOMETRIC GRAPHS										
N	M	w_T	c_T	D_{exp}	D_{act}	$D_{v,max}$	w_{max}	w_{min}	c_{max}	c_{min}
250	233	621	702	2	1.86	9	4	1	5	1
250	340	627	1027	3	2.72	7	4	1	5	1
250	464	600	1389	4	3.71	11	4	1	5	1
250	889	621	2654	8	7.11	13	4	1	5	1
250	1823	615	5536	16	14.58	28	4	1	5	1
500	488	1262	1417	2	1.95	7	4	1	5	1
500	744	1250	2257	3	2.98	9	4	1	5	1
500	977	1252	2920	4	3.91	10	4	1	5	1
500	1908	1192	5620	8	7.63	17	4	1	5	1
500	3755	1194	11311	16	15.02	27	4	1	5	1
1000	994	2499	2975	2	1.99	7	4	1	5	1
1000	1508	2473	4541	3	3.02	11	4	1	5	1
1000	2007	2539	6006	4	4.01	13	4	1	5	1
1000	3891	2533	11897	8	7.78	18	4	1	5	1
1000	7615	2445	22766	16	15.23	28	4	1	5	1

Table C.3. Properties of Grid Test Graphs

GRID GRAPHS										
N	M	w_T	c_T	D_{exp}	D_{act}	$D_{v,max}$	w_{max}	w_{min}	c_{max}	c_{min}
250	129	610	371	1	1.03	4	4	1	5	1
250	238	632	699	2	1.90	4	4	1	5	1
250	379	624	1106	3	3.03	4	4	1	5	1
500	276	1272	851	1	1.10	4	4	1	5	1
500	494	1237	1482	2	1.98	4	4	1	5	1
500	746	1333	2221	3	2.98	4	4	1	5	1
1000	481	2523	1427	1	0.96	4	4	1	5	1
1000	992	2505	2962	2	1.98	4	4	1	5	1
1000	1509	2488	4531	3	3.02	4	4	1	5	1

Table C.4. Properties of Ladder Test Graphs

LADDER GRAPHS										
N	M	w_T	c_T	D_{exp}	D_{act}	$D_{v,max}$	w_{max}	w_{min}	c_{max}	c_{min}
250	126	636	389	1	1.01	3	4	1	5	1
250	253	620	747	2	2.02	3	4	1	5	1
500	253	1215	722	1	1.01	3	4	1	5	1
500	498	1272	1531	2	1.99	3	4	1	5	1
1000	475	2454	1429	1	0.95	3	4	1	5	1
1000	960	2505	2873	2	1.92	3	4	1	5	1

Table C.5. Properties of Tree Test Graphs

TREE GRAPHS										
N	M	w_T	c_T	D_{exp}	D_{act}	$D_{v,max}$	w_{max}	w_{min}	c_{max}	c_{min}
250	249	614	759	2	1.99	10	4	1	5	1
500	499	1253	1487	2	2.00	14	4	1	5	1
1000	999	2475	2976	2	2.00	20	4	1	5	1

Table C.6. Properties of Benchmark Circuits (multiply w_T by 1000, $c_{max} = 1$ and $c_{min} = 1$ for all circuits)

BENCHMARK CIRCUITS											
Name	N	M	p	w_T	c_T	$D_{v,x}$	D_v	$D_{e,x}$	D_e	w_x	w_n
sioo	602	383	1771	1924	383	4	2.94	128	4.62	3587	797
balu	701	702	2493	1377	702	9	3.56	117	3.55	4783	398
primary1	833	902	2908	266	902	9	3.49	18	3.22	1800	45
struct	1888	1888	5375	2850	1888	4	2.85	16	2.85	2320	928
industry1	2271	2186	7731	4403	2186	9	3.40	318	3.54	5712	552
primary2	3014	3029	11219	534	3029	9	3.72	37	3.70	1800	45

Table C.7. Execution time averages (and standard deviations) for random graphs

PROBLEM			EXECUTION TIME AVERAGES (in seconds)				RATIO
N	D	K	PLM ₁	PFM ₁	PFM ₂	PFM ₃	
250	2	2	0.31 (0.10)	0.44 (0.08)	0.78 (0.24)	1.61 (0.51)	5.26
		4	0.54 (0.13)	1.35 (0.30)	3.50 (0.81)	9.08 (1.69)	16.94
		8	1.43 (0.23)	4.14 (0.85)	15.82 (3.71)	104.81 (31.28)	73.19
	3	2	0.45 (0.12)	0.69 (0.20)	1.07 (0.37)	1.78 (0.52)	3.97
		4	0.78 (0.15)	1.80 (0.28)	5.89 (2.00)	15.97 (5.98)	20.42
		8	1.86 (0.32)	5.25 (1.59)	19.51 (7.35)	116.17 (43.57)	62.36
	4	2	0.58 (0.16)	1.02 (0.20)	1.55 (0.42)	2.47 (0.81)	4.22
		4	1.13 (0.22)	2.67 (0.48)	6.74 (2.29)	18.55 (6.54)	16.47
		8	2.40 (0.44)	7.21 (1.82)	20.15 (5.93)	148.67 (76.20)	61.84
	8	2	0.99 (0.28)	1.71 (0.55)	2.32 (0.76)	3.69 (1.19)	3.73
		4	1.87 (0.46)	3.65 (0.68)	11.81 (3.69)	45.14 (17.54)	24.09
		8	3.55 (0.90)	7.03 (3.53)	37.90 (11.96)	232.65 (76.34)	65.63
16	2	1.77 (0.46)	3.27 (0.90)	5.16 (1.62)	7.14 (2.00)	4.03	
	4	3.46 (0.84)	6.29 (2.00)	26.18 (10.25)	43.33 (14.69)	12.53	
	8	5.54 (1.34)	14.42 (5.38)	62.19 (17.19)	333.82 (124.15)	60.27	
500	2	2	0.80 (0.18)	1.38 (0.37)	2.22 (0.51)	4.35 (1.25)	5.41
		4	1.34 (0.27)	3.72 (1.03)	9.89 (2.24)	23.50 (9.72)	17.60
		8	3.08 (0.48)	8.60 (2.55)	43.08 (19.67)	230.08 (83.96)	74.75
	3	2	1.10 (0.26)	2.02 (0.53)	3.43 (1.10)	5.50 (2.29)	5.02
		4	1.72 (0.29)	6.17 (1.46)	13.75 (5.04)	35.29 (13.26)	20.49
		8	3.88 (0.66)	12.89 (3.21)	54.04 (11.58)	179.29 (30.26)	46.20
	4	2	1.34 (0.35)	2.33 (0.51)	3.71 (1.07)	7.60 (2.05)	5.67
		4	2.16 (0.41)	8.46 (2.70)	20.29 (5.26)	35.81 (10.02)	16.58
		8	4.50 (0.90)	16.78 (4.18)	50.59 (18.20)	308.33 (124.09)	68.53
	8	2	2.19 (0.55)	3.91 (1.24)	7.54 (2.26)	9.91 (3.42)	4.52
		4	3.81 (0.88)	10.15 (2.15)	28.89 (10.36)	86.62 (30.48)	22.72
		8	7.16 (1.31)	22.58 (9.31)	86.86 (32.93)	557.78 (113.71)	77.86
16	2	3.95 (0.98)	8.09 (2.80)	16.60 (5.22)	22.62 (8.62)	5.72	
	4	8.04 (1.89)	17.17 (5.04)	50.39 (17.18)	153.30 (48.35)	19.07	
	8	12.99 (3.01)	35.37 (12.30)	153.45 (66.49)	917.45 (353.11)	70.62	
1000	2	2	1.93 (0.40)	3.28 (0.47)	6.52 (1.75)	12.71 (4.21)	6.57
		4	2.78 (0.41)	8.64 (2.56)	25.72 (5.81)	57.07 (39.19)	20.54
		8	6.29 (0.78)	22.19 (4.61)	90.56 (24.19)	484.54 (167.23)	77.02
	3	2	2.48 (0.65)	6.08 (1.50)	9.56 (1.29)	16.84 (5.59)	6.78
		4	3.64 (0.54)	12.62 (3.25)	34.20 (11.85)	89.56 (25.27)	24.62
		8	8.03 (1.27)	32.16 (8.36)	124.93 (57.04)	693.39 (197.14)	86.33
	4	2	3.02 (0.79)	6.98 (2.31)	12.48 (3.06)	21.20 (8.42)	7.03
		4	4.39 (0.63)	17.16 (3.49)	48.79 (12.14)	131.83 (40.04)	30.02
		8	9.54 (1.57)	36.58 (7.42)	139.58 (46.18)	600.81 (188.19)	62.95
	8	2	4.82 (1.12)	12.09 (4.43)	20.64 (6.06)	39.49 (10.80)	8.18
		4	8.58 (1.72)	21.52 (4.67)	74.83 (33.52)	191.18 (62.12)	22.28
		8	15.55 (3.17)	55.61 (23.25)	206.13 (84.97)	1230.27 (294.46)	79.12
16	2	9.08 (2.09)	22.00 (4.20)	38.87 (10.64)	61.79 (20.93)	6.80	
	4	18.17 (4.39)	35.92 (7.68)	130.37 (32.11)	472.99 (201.56)	26.03	
	8	29.57 (6.32)	70.15 (14.95)	450.57 (101.38)	2663.37 (941.41)	90.08	

Table C.8. Cutsizes averages (and standard deviations) for random graphs

PROBLEM			CUTSIZE AVERAGES				IMP.
N	D	K	PLM ₁	PFM ₁	PFM ₂	PFM ₃	(%)
250	2	2	44.64 (9.51)	37.40 (7.35)	25.10 (5.41)	20.10 (3.27)	54.97
		4	95.26 (11.03)	69.90 (10.77)	42.00 (3.58)	39.30 (2.49)	58.74
		8	123.03 (12.10)	88.70 (8.04)	57.30 (5.46)	52.20 (2.27)	57.57
	3	2	118.30 (10.23)	104.40 (6.90)	97.40 (6.89)	92.30 (5.04)	21.98
		4	226.30 (14.95)	192.30 (11.68)	166.30 (5.39)	157.80 (7.32)	30.27
		8	288.63 (15.22)	239.50 (11.38)	213.70 (4.88)	209.70 (4.63)	27.35
	4	2	276.62 (16.06)	266.90 (10.67)	249.40 (13.40)	243.20 (7.98)	12.08
		4	486.91 (18.66)	449.60 (9.25)	418.60 (3.88)	408.00 (8.87)	16.21
		8	624.38 (20.71)	580.30 (14.24)	536.00 (4.12)	528.30 (4.69)	15.39
	8	2	717.19 (18.75)	702.50 (14.15)	686.50 (9.81)	690.80 (14.40)	3.68
		4	1207.50 (27.24)	1172.50 (16.60)	1128.80 (10.04)	1116.30 (12.20)	7.55
		8	1524.57 (23.42)	1490.80 (18.99)	1421.90 (12.10)	1411.80 (7.28)	7.40
	16	2	1883.64 (23.65)	1881.90 (25.73)	1856.30 (20.21)	1858.40 (21.14)	1.34
		4	3045.23 (33.83)	3054.20 (16.35)	2953.30 (18.32)	2953.30 (14.21)	3.02
		8	3776.29 (30.37)	3744.00 (27.37)	3660.70 (10.41)	3641.60 (8.63)	3.57
	500	2	2	99.02 (13.74)	75.30 (10.83)	65.30 (8.33)	55.30 (6.31)
4			197.42 (18.24)	137.90 (6.91)	106.20 (7.08)	95.10 (5.09)	51.83
8			250.19 (20.19)	193.00 (13.10)	125.50 (7.70)	116.20 (4.28)	53.56
3		2	200.54 (18.19)	174.00 (10.24)	158.40 (9.21)	141.60 (11.42)	29.39
		4	384.47 (26.66)	303.60 (13.21)	271.30 (10.05)	252.40 (11.77)	34.35
		8	489.21 (20.89)	411.20 (17.06)	337.40 (10.80)	327.80 (4.77)	32.99
4		2	438.48 (19.70)	403.20 (20.40)	379.00 (14.01)	367.90 (8.83)	16.10
		4	771.61 (29.94)	676.70 (20.47)	620.20 (15.20)	613.20 (13.24)	20.53
		8	983.20 (27.32)	883.10 (16.31)	794.60 (12.12)	772.30 (9.26)	21.45
8		2	1371.95 (35.38)	1344.80 (32.75)	1305.20 (22.17)	1296.60 (19.12)	5.49
		4	2292.97 (39.59)	2218.50 (41.85)	2130.40 (22.47)	2103.10 (23.49)	8.28
		8	2879.47 (36.67)	2800.50 (41.43)	2648.90 (16.46)	2623.10 (15.38)	8.90
16		2	3659.59 (43.67)	3648.20 (27.77)	3580.80 (40.84)	3586.40 (22.74)	2.00
		4	5920.67 (47.87)	5940.10 (53.34)	5764.60 (44.78)	5707.10 (29.62)	3.61
		8	7322.51 (53.63)	7283.60 (53.21)	7058.70 (28.04)	7025.20 (12.80)	4.06
1000		2	2	217.81 (22.16)	172.80 (11.47)	141.70 (12.28)	117.50 (9.12)
	4		399.91 (33.37)	288.30 (17.60)	220.70 (9.40)	198.00 (9.33)	50.49
	8		510.79 (36.42)	386.50 (24.00)	259.70 (7.43)	246.90 (6.33)	51.66
	3	2	499.60 (33.95)	418.80 (12.59)	377.80 (14.02)	362.00 (21.84)	27.54
		4	903.25 (42.99)	717.20 (19.98)	637.70 (17.57)	598.20 (11.69)	33.77
		8	1131.02 (43.14)	931.30 (24.51)	795.30 (21.47)	756.50 (9.56)	33.11
	4	2	833.87 (37.28)	739.00 (22.16)	707.50 (12.15)	677.70 (10.98)	18.73
		4	1472.54 (42.05)	1257.20 (24.62)	1170.20 (27.96)	1120.60 (14.87)	23.90
		8	1859.41 (48.25)	1658.30 (29.55)	1481.60 (18.83)	1428.30 (9.42)	23.19
	8	2	2836.86 (58.16)	2761.20 (52.38)	2700.60 (38.68)	2640.60 (37.99)	6.92
		4	4707.62 (61.52)	4581.30 (29.89)	4355.40 (29.33)	4319.60 (31.07)	8.24
		8	5902.53 (59.77)	5706.40 (72.32)	5429.20 (30.49)	5349.90 (33.11)	9.36
	16	2	7324.39 (76.78)	7325.60 (57.33)	7209.60 (96.31)	7137.30 (74.34)	2.55
		4	11811.26 (75.34)	11866.70 (56.64)	11424.10 (57.11)	11358.40 (47.68)	3.83
		8	14546.19 (79.11)	14541.60 (50.16)	14009.30 (50.70)	13942.80 (36.86)	4.15

Table C.9. Execution time averages (and standard deviations) for geometric graphs

PROBLEM			EXECUTION TIME AVERAGES (in seconds)				RATIO
N	D	K	PLM ₁	PFM ₁	PFM ₂	PFM ₃	
250	2	2	0.24 (0.08)	0.31 (0.14)	0.32 (0.12)	0.64 (0.20)	2.63
		4	0.51 (0.11)	0.78 (0.18)	2.24 (0.81)	4.91 (1.56)	9.59
		8	1.26 (0.25)	2.31 (0.67)	10.22 (3.45)	48.97 (12.05)	38.83
	3	2	0.42 (0.11)	0.45 (0.09)	0.73 (0.20)	0.95 (0.39)	2.27
		4	0.79 (0.11)	1.59 (0.32)	3.59 (0.65)	11.48 (4.66)	14.59
		8	1.76 (0.31)	3.30 (0.78)	13.86 (4.68)	91.38 (30.67)	51.98
	4	2	0.46 (0.10)	0.65 (0.16)	1.13 (0.26)	2.14 (0.60)	4.70
		4	0.94 (0.16)	1.69 (0.52)	5.14 (1.38)	18.95 (7.92)	20.14
		8	2.05 (0.38)	4.86 (0.79)	20.50 (8.50)	124.16 (49.97)	60.54
	8	2	0.85 (0.18)	0.85 (0.22)	1.26 (0.31)	2.55 (0.88)	3.00
		4	1.63 (0.35)	2.45 (0.94)	7.73 (2.73)	16.75 (4.40)	10.26
		8	3.24 (0.70)	5.53 (2.31)	31.57 (12.00)	159.04 (60.45)	49.13
	16	2	1.27 (0.26)	1.19 (0.16)	2.09 (0.39)	4.43 (1.03)	3.47
		4	2.68 (0.68)	3.42 (0.50)	9.62 (2.78)	33.99 (10.56)	12.67
		8	5.37 (1.26)	11.14 (4.39)	36.26 (13.40)	248.35 (71.44)	46.24
500	2	2	0.50 (0.12)	0.69 (0.22)	1.05 (0.36)	1.43 (0.41)	2.88
		4	1.11 (0.16)	3.05 (0.71)	5.48 (2.11)	9.91 (3.25)	8.97
		8	2.63 (0.40)	6.56 (0.97)	19.56 (5.80)	83.93 (34.61)	31.89
	3	2	0.77 (0.17)	1.28 (0.32)	2.14 (0.43)	3.85 (1.17)	4.97
		4	1.56 (0.23)	3.23 (0.57)	9.84 (2.34)	35.60 (8.29)	22.86
		8	3.38 (0.51)	8.00 (2.32)	38.92 (10.72)	199.93 (67.19)	59.17
	4	2	1.05 (0.21)	1.37 (0.31)	2.93 (1.42)	5.57 (1.53)	5.31
		4	1.81 (0.25)	3.80 (1.23)	11.93 (4.84)	38.90 (14.91)	21.49
		8	3.97 (0.62)	12.30 (2.76)	54.47 (17.12)	275.37 (74.08)	69.33
	8	2	1.84 (0.35)	2.18 (0.39)	4.41 (1.18)	4.77 (1.20)	2.60
		4	3.29 (0.66)	6.21 (2.36)	16.26 (3.32)	48.30 (17.55)	14.67
		8	6.75 (1.35)	14.14 (3.42)	66.32 (22.83)	433.21 (153.51)	64.15
	16	2	3.24 (0.62)	2.56 (0.38)	4.82 (1.09)	7.49 (0.94)	2.31
		4	5.99 (1.59)	7.82 (3.13)	19.82 (6.93)	70.17 (18.72)	11.71
		8	12.39 (3.10)	23.45 (10.54)	98.38 (32.52)	662.38 (158.62)	53.47
1000	2	2	1.86 (0.48)	2.04 (0.37)	3.70 (1.02)	5.81 (1.32)	3.12
		4	3.17 (0.54)	5.80 (0.99)	15.74 (3.60)	42.03 (7.93)	13.28
		8	5.58 (0.85)	13.50 (4.36)	62.23 (19.13)	310.97 (120.53)	55.75
	3	2	2.30 (0.49)	3.59 (0.85)	6.32 (1.16)	10.25 (3.78)	4.46
		4	4.19 (0.53)	7.47 (2.10)	25.23 (8.68)	56.49 (25.46)	13.48
		8	7.21 (0.95)	21.65 (6.82)	65.12 (26.22)	578.78 (174.87)	80.30
	4	2	2.78 (0.73)	4.58 (1.32)	8.71 (2.46)	14.31 (4.13)	5.15
		4	4.07 (0.68)	11.24 (2.56)	33.61 (12.07)	80.77 (31.21)	19.84
		8	8.05 (1.31)	26.16 (6.21)	125.18 (44.17)	595.98 (185.86)	74.03
	8	2	5.27 (1.20)	6.03 (0.98)	7.89 (2.52)	12.97 (3.83)	2.46
		4	7.15 (1.36)	13.42 (4.03)	41.25 (15.20)	115.26 (39.62)	16.11
		8	13.62 (2.30)	35.37 (8.65)	203.11 (89.76)	1065.79 (319.43)	78.27
	16	2	9.20 (2.23)	7.23 (1.41)	10.26 (2.35)	16.70 (2.68)	1.82
		4	12.91 (2.57)	21.67 (7.47)	41.90 (10.72)	154.20 (43.94)	11.95
		8	25.22 (5.25)	46.65 (17.93)	200.76 (72.59)	1376.01 (293.27)	54.56

Table C.10. Cutsizes averages (and standard deviations) for geometric graphs

PROBLEM			CUTSIZE AVERAGES				IMP. (%)
N	D	K	PLM ₁	PFM ₁	PFM ₂	PFM ₃	
250	2	2	5.92 (2.73)	2.20 (1.94)	0.30 (0.64)	0.20 (0.40)	96.62
		4	18.09 (7.51)	8.10 (5.17)	0.80 (1.17)	0.30 (0.46)	98.34
		8	25.56 (8.99)	13.20 (6.85)	2.00 (2.05)	1.60 (1.11)	93.74
	3	2	15.42 (7.00)	2.80 (3.25)	2.20 (2.82)	1.30 (1.49)	91.57
		4	35.94 (12.29)	10.10 (4.35)	5.60 (3.38)	2.60 (1.62)	92.77
		8	56.60 (13.67)	24.10 (9.69)	11.80 (4.71)	8.00 (2.90)	85.87
	4	2	28.10 (13.38)	18.40 (12.27)	16.70 (10.31)	9.80 (8.94)	65.12
		4	75.51 (18.34)	51.20 (9.71)	31.00 (11.10)	20.20 (10.17)	73.25
		8	110.48 (19.71)	74.80 (17.85)	50.60 (5.08)	43.00 (6.07)	61.08
	8	2	86.72 (29.30)	105.60 (35.08)	77.20 (27.50)	73.50 (23.93)	15.24
		4	230.07 (49.39)	193.10 (32.43)	133.80 (20.34)	130.20 (21.83)	43.41
		8	351.68 (60.27)	308.40 (32.24)	238.40 (35.11)	235.00 (37.68)	33.18
	16	2	284.03 (50.70)	313.00 (64.33)	305.50 (48.24)	356.40 (68.87)	-25.48
		4	644.89 (79.11)	610.60 (49.82)	607.30 (67.70)	605.20 (73.61)	6.15
		8	1000.84 (68.53)	937.00 (17.20)	935.10 (18.95)	929.80 (17.83)	7.10
500	2	2	20.50 (8.90)	2.40 (2.54)	0.40 (1.20)	0.00 (0.00)	100.00
		4	39.54 (9.99)	11.30 (6.29)	0.40 (0.49)	0.30 (0.46)	99.24
		8	53.94 (11.45)	19.10 (6.69)	1.90 (1.70)	1.20 (1.78)	97.78
	3	2	52.59 (13.56)	19.10 (8.18)	13.40 (5.10)	6.20 (5.69)	88.21
		4	91.59 (19.64)	44.40 (9.88)	12.80 (6.27)	2.60 (2.46)	97.16
		8	124.43 (20.93)	56.80 (15.43)	16.50 (6.38)	8.00 (3.52)	93.57
	4	2	82.69 (21.60)	45.80 (22.77)	27.50 (10.84)	15.80 (5.11)	80.89
		4	153.53 (24.25)	89.90 (20.36)	34.90 (11.93)	17.00 (4.34)	88.93
		8	188.44 (22.17)	112.80 (16.83)	38.40 (7.28)	34.20 (10.10)	81.85
	8	2	157.32 (60.02)	211.70 (44.19)	125.00 (24.27)	107.60 (33.71)	31.60
		4	408.36 (80.83)	303.60 (62.00)	239.20 (50.16)	217.70 (58.97)	46.69
		8	579.66 (83.67)	464.90 (43.46)	383.30 (33.84)	347.30 (39.97)	40.09
	16	2	415.50 (94.53)	613.80 (165.58)	591.80 (147.82)	620.30 (151.32)	-49.29
		4	1065.02 (169.93)	1088.60 (107.28)	977.20 (159.66)	1032.20 (137.85)	3.08
		8	1622.46 (160.29)	1561.80 (195.83)	1534.00 (103.68)	1426.30 (83.80)	12.09
1000	2	2	39.72 (9.96)	13.90 (6.85)	6.10 (5.22)	3.10 (2.62)	92.20
		4	75.49 (15.60)	26.40 (5.83)	9.20 (3.87)	1.80 (1.33)	97.62
		8	104.95 (16.70)	44.80 (10.81)	7.50 (3.75)	1.60 (1.36)	98.48
	3	2	101.63 (25.16)	55.60 (8.36)	31.00 (7.76)	25.80 (10.22)	74.61
		4	174.41 (28.71)	85.20 (14.20)	46.60 (15.72)	25.40 (10.05)	85.44
		8	216.67 (30.21)	113.50 (16.21)	47.50 (12.05)	26.00 (7.46)	88.00
	4	2	186.93 (40.10)	99.40 (20.03)	63.30 (17.11)	32.10 (5.84)	82.83
		4	330.25 (44.88)	163.60 (23.53)	69.40 (15.14)	39.30 (8.84)	88.10
		8	398.12 (43.77)	224.20 (33.01)	69.50 (12.11)	46.70 (11.93)	88.27
	8	2	450.14 (109.99)	322.90 (48.79)	321.30 (65.57)	257.90 (50.01)	42.71
		4	913.73 (138.88)	604.10 (42.23)	469.90 (62.07)	439.30 (63.05)	51.92
		8	1173.00 (136.09)	806.20 (63.73)	610.90 (92.12)	578.20 (60.09)	50.71
	16	2	781.89 (229.76)	1228.70 (238.00)	1037.90 (229.03)	1171.40 (292.21)	-49.82
		4	1996.35 (293.39)	1937.50 (189.19)	1901.50 (323.77)	1765.30 (210.51)	11.57
		8	2727.82 (272.50)	2386.70 (200.10)	2409.40 (226.18)	2488.50 (157.38)	8.77

Table C.11. Execution time averages (and standard deviations) for grid graphs

PROBLEM			EXECUTION TIME AVERAGES (in seconds)				RATIO
N	D	K	PLM ₁	PFM ₁	PFM ₂	PFM ₃	
250	1	2	0.18 (0.05)	0.20 (0.00)	0.28 (0.11)	0.46 (0.13)	2.50
		4	0.46 (0.09)	0.68 (0.10)	1.49 (0.32)	3.41 (1.51)	7.40
		8	1.18 (0.18)	2.07 (0.57)	6.38 (1.29)	35.96 (14.27)	30.40
	2	2	0.31 (0.07)	0.57 (0.19)	0.77 (0.22)	1.69 (0.34)	5.37
		4	0.63 (0.11)	1.35 (0.37)	4.13 (1.60)	11.96 (3.79)	18.98
		8	1.48 (0.27)	3.27 (0.81)	13.27 (3.90)	89.05 (24.13)	60.13
	3	2	0.54 (0.12)	0.57 (0.23)	1.17 (0.25)	1.50 (0.38)	2.78
		4	0.98 (0.19)	2.13 (0.54)	5.53 (2.53)	16.49 (5.98)	16.83
		8	2.26 (0.46)	5.92 (1.46)	19.24 (5.17)	126.40 (45.28)	55.90
500	1	2	0.47 (0.10)	0.62 (0.15)	0.92 (0.20)	1.37 (0.51)	2.94
		4	0.93 (0.17)	1.55 (0.32)	6.51 (1.56)	9.13 (3.74)	9.84
		8	2.21 (0.36)	4.65 (0.81)	18.29 (3.50)	113.35 (39.00)	51.29
	2	2	0.87 (0.22)	1.15 (0.32)	2.63 (0.80)	4.52 (1.39)	5.21
		4	1.30 (0.22)	3.22 (1.17)	13.60 (4.33)	32.04 (12.63)	24.59
		8	2.99 (0.47)	11.11 (2.55)	43.03 (14.53)	237.66 (98.49)	79.59
	3	2	1.18 (0.26)	1.87 (0.52)	3.20 (0.80)	4.62 (1.85)	3.91
		4	1.96 (0.32)	5.63 (2.10)	15.44 (3.31)	38.72 (11.35)	19.73
		8	4.32 (0.68)	13.89 (3.31)	55.59 (11.69)	303.29 (163.98)	70.22
1000	1	2	1.05 (0.18)	1.29 (0.16)	2.24 (0.27)	4.68 (0.94)	4.44
		4	1.90 (0.31)	3.80 (0.93)	9.44 (1.70)	26.54 (5.66)	13.99
		8	4.34 (0.58)	7.54 (2.00)	46.29 (10.34)	218.71 (54.84)	50.37
	2	2	2.05 (0.35)	3.40 (0.68)	6.03 (0.91)	11.36 (2.99)	5.55
		4	2.91 (0.49)	10.01 (2.21)	30.38 (9.97)	94.25 (39.63)	32.43
		8	6.62 (1.05)	22.93 (5.38)	88.30 (18.74)	589.44 (171.68)	89.07
	3	2	2.87 (0.63)	5.05 (1.20)	7.68 (3.15)	16.02 (5.60)	5.58
		4	4.01 (0.56)	13.43 (3.41)	38.66 (7.57)	113.78 (35.62)	28.37
		8	8.87 (1.34)	47.01 (12.50)	150.28 (49.90)	936.34 (445.42)	105.54

Table C.12. Cutsizes averages (and standard deviations) for grid graphs

PROBLEM			CUTSIZE AVERAGES				IMP. (%)
N	D	K	PLM1	PFM1	PFM2	PFM3	
250	1	2	3.57 (2.10)	2.40 (1.02)	0.80 (0.60)	0.50 (0.50)	85.99
		4	11.83 (4.44)	8.80 (1.89)	1.80 (1.08)	0.20 (0.40)	98.31
		8	18.05 (5.40)	11.70 (2.83)	1.50 (1.36)	0.70 (0.90)	96.12
	2	2	19.98 (6.84)	11.50 (4.46)	10.60 (4.25)	4.50 (1.43)	77.48
		4	58.38 (12.06)	34.40 (3.41)	15.40 (4.10)	7.90 (3.05)	86.47
		8	78.09 (9.87)	53.80 (8.61)	22.00 (3.52)	14.90 (2.70)	80.92
	3	2	35.10 (9.88)	36.30 (12.55)	27.00 (7.94)	18.90 (8.14)	46.15
		4	114.94 (21.46)	81.60 (15.45)	54.80 (13.02)	42.30 (3.85)	63.20
		8	173.24 (23.23)	120.00 (17.11)	90.00 (7.56)	75.60 (6.36)	56.36
500	1	2	10.45 (3.07)	8.30 (3.95)	3.10 (1.97)	0.30 (0.46)	97.13
		4	27.10 (6.54)	19.80 (3.94)	1.00 (1.18)	0.40 (0.66)	98.52
		8	36.96 (7.60)	21.30 (6.83)	3.60 (1.56)	1.50 (1.12)	95.94
	2	2	49.34 (11.69)	36.30 (8.58)	20.60 (5.89)	13.40 (5.64)	72.84
		4	117.08 (19.35)	71.10 (8.60)	30.00 (6.15)	22.20 (4.58)	81.04
		8	155.41 (19.66)	87.70 (10.91)	33.60 (6.59)	25.30 (3.44)	83.72
	3	2	83.24 (25.92)	67.90 (20.47)	37.00 (12.26)	23.40 (5.80)	71.89
		4	214.17 (34.32)	127.90 (22.64)	75.60 (15.96)	52.20 (7.26)	75.63
		8	301.71 (41.35)	204.40 (12.26)	112.20 (9.09)	103.50 (12.63)	65.70
1000	1	2	17.93 (4.73)	17.00 (3.16)	6.90 (2.91)	1.70 (0.90)	90.52
		4	41.57 (10.93)	32.60 (3.75)	6.40 (2.20)	2.30 (1.00)	94.47
		8	60.29 (10.43)	49.50 (7.49)	6.30 (1.27)	3.70 (2.90)	93.86
	2	2	89.97 (17.87)	49.80 (9.53)	29.70 (6.34)	20.30 (7.52)	77.44
		4	209.06 (42.53)	112.90 (8.83)	46.90 (9.76)	34.90 (9.87)	83.31
		8	279.61 (37.01)	139.60 (16.01)	55.80 (8.21)	38.70 (8.65)	86.16
	3	2	155.97 (36.68)	111.00 (31.26)	81.40 (20.32)	53.20 (20.73)	65.89
		4	482.95 (56.49)	221.70 (36.59)	122.10 (23.54)	87.30 (14.72)	81.92
		8	627.35 (51.30)	319.30 (41.48)	177.10 (19.90)	137.00 (19.67)	78.16

Table C.13. Execution time averages (and standard deviations) for ladder graphs

PROBLEM			EXECUTION TIME AVERAGES (in seconds)				RATIO
N	D	K	PLM1	PFM1	PFM2	PFM3	
250	1	2	0.19 (0.06)	0.13 (0.05)	0.33 (0.06)	0.37 (0.11)	1.90
		4	0.42 (0.09)	0.50 (0.10)	1.11 (0.40)	2.40 (0.60)	5.70
		8	1.16 (0.16)	1.60 (0.28)	5.12 (1.28)	27.83 (14.41)	24.05
	2	2	0.40 (0.08)	0.47 (0.11)	0.74 (0.15)	1.25 (0.42)	3.16
		4	0.67 (0.15)	1.65 (0.44)	3.49 (0.76)	11.25 (6.06)	16.69
		8	1.59 (0.24)	3.81 (1.04)	17.14 (5.73)	100.91 (36.15)	63.47
500	1	2	0.60 (0.11)	0.58 (0.07)	0.81 (0.14)	1.37 (0.26)	2.30
		4	1.00 (0.15)	1.83 (0.37)	4.51 (1.27)	5.54 (1.83)	5.55
		8	2.38 (0.32)	3.86 (0.56)	12.84 (2.79)	67.00 (27.99)	28.10
	2	2	0.87 (0.16)	1.53 (0.32)	2.51 (0.60)	3.90 (1.79)	4.48
		4	1.35 (0.19)	3.87 (0.54)	10.10 (2.42)	30.24 (14.04)	22.35
		8	3.10 (0.36)	11.88 (3.62)	36.46 (11.64)	198.66 (65.59)	64.08
1000	1	2	1.01 (0.14)	1.06 (0.15)	1.82 (0.33)	2.41 (0.47)	2.39
		4	1.83 (0.28)	3.21 (0.85)	8.53 (1.61)	20.84 (9.31)	11.38
		8	4.28 (0.49)	7.10 (0.91)	36.28 (9.14)	200.73 (42.85)	46.92
	2	2	1.93 (0.42)	3.28 (0.82)	6.66 (2.05)	9.28 (3.08)	4.81
		4	3.05 (0.40)	8.29 (2.26)	22.57 (6.46)	74.73 (25.96)	24.53
		8	6.69 (0.76)	24.98 (6.92)	101.09 (34.00)	422.97 (149.78)	63.19

Table C.14. Cutsizes averages (and standard deviations) for ladder graphs

PROBLEM			CUTSIZE AVERAGES				IMP. (%)
N	D	K	PLM1	PFM1	PFM2	PFM3	
250	1	2	1.82 (1.27)	5.70 (1.19)	2.00 (0.63)	0.40 (0.49)	78.02
		4	7.94 (2.88)	6.70 (1.42)	1.00 (1.10)	0.00 (0.00)	100.00
		8	15.00 (5.56)	13.20 (4.17)	0.90 (0.83)	0.90 (1.64)	94.00
	2	2	14.85 (7.19)	9.10 (6.46)	3.10 (2.95)	0.90 (0.83)	93.94
		4	51.04 (16.03)	20.10 (6.11)	7.10 (2.91)	2.30 (2.53)	95.49
		8	72.57 (15.39)	42.20 (10.23)	10.00 (4.05)	6.20 (2.27)	91.46
500	1	2	5.40 (2.35)	8.20 (1.83)	2.40 (1.28)	0.30 (0.46)	94.44
		4	21.65 (6.95)	15.80 (3.60)	3.60 (3.10)	0.00 (0.00)	100.00
		8	27.94 (7.91)	21.90 (5.66)	2.80 (2.36)	1.20 (1.54)	95.71
	2	2	24.51 (7.45)	20.90 (6.88)	8.40 (4.82)	3.20 (2.27)	86.94
		4	73.64 (15.19)	41.40 (6.92)	9.00 (3.10)	6.60 (3.26)	91.04
		8	114.05 (17.10)	54.80 (16.27)	11.40 (3.95)	6.30 (2.24)	94.48
1000	1	2	8.75 (2.84)	7.80 (2.48)	3.70 (2.37)	0.30 (0.64)	96.57
		4	27.93 (9.05)	26.00 (5.67)	4.70 (2.57)	0.60 (1.02)	97.85
		8	42.24 (9.91)	36.70 (5.48)	5.60 (2.15)	3.20 (1.60)	92.42
	2	2	55.17 (14.16)	30.50 (5.45)	12.70 (5.66)	8.20 (5.00)	85.14
		4	130.27 (22.96)	62.70 (11.34)	18.30 (4.34)	12.30 (3.07)	90.56
		8	172.82 (26.36)	86.10 (14.90)	18.70 (3.87)	13.90 (2.77)	91.96

Table C.15. Execution time averages (and standard deviations) for tree graphs

PROBLEM			EXECUTION TIME AVERAGES (in seconds)				RATIO
N	D	K	PLM1	PFM1	PFM2	PFM3	
250	2	2	0.31 (0.10)	0.36 (0.12)	0.69 (0.28)	1.15 (0.55)	3.69
		4	0.63 (0.12)	1.34 (0.42)	3.48 (0.97)	9.81 (4.57)	15.65
		8	1.50 (0.27)	3.61 (0.71)	10.64 (3.64)	85.34 (30.83)	56.74
500	2	2	0.72 (0.22)	1.05 (0.21)	2.21 (0.56)	4.45 (1.94)	6.22
		4	1.39 (0.29)	2.96 (0.86)	8.55 (3.46)	21.78 (6.97)	15.65
		8	3.09 (0.53)	7.40 (2.69)	40.21 (15.48)	220.73 (68.48)	71.43
1000	2	2	1.80 (0.57)	2.75 (0.55)	6.88 (0.79)	12.04 (3.64)	6.70
		4	2.85 (0.61)	6.61 (1.72)	25.38 (4.75)	72.15 (30.08)	25.31
		8	6.29 (1.17)	13.61 (3.47)	109.28 (37.32)	361.30 (124.25)	57.44

Table C.16. Cutsizes averages (and standard deviations) for tree graphs

PROBLEM			CUTSIZE AVERAGES				IMP. (%)
N	D	K	PLM1	PFM1	PFM2	PFM3	
250	2	2	79.68 (14.69)	82.20 (17.29)	62.60 (20.17)	47.40 (19.35)	40.51
		4	143.78 (16.33)	137.60 (12.63)	98.40 (7.16)	95.10 (7.30)	33.86
		8	194.06 (15.08)	177.00 (9.40)	135.90 (6.62)	130.50 (5.35)	32.75
500	2	2	159.72 (44.73)	174.30 (14.35)	133.30 (37.45)	51.80 (44.98)	67.57
		4	286.01 (37.77)	278.00 (25.76)	199.60 (23.72)	181.40 (17.80)	36.58
		8	400.15 (25.35)	372.70 (16.28)	277.60 (9.24)	261.40 (4.50)	34.67
1000	2	2	302.34 (89.16)	324.10 (35.12)	262.60 (77.08)	203.20 (72.47)	32.79
		4	598.87 (59.39)	566.00 (32.69)	385.60 (31.46)	356.20 (11.38)	40.52
		8	789.63 (37.61)	773.00 (29.16)	532.90 (14.21)	511.40 (7.80)	35.24

Table C.23. Cutsizes averages for random graphs (for K-PLM-like algorithms)

PROBLEM		CUTSIZE AVERAGES											
D	K	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
2	2	1.00	0.87	0.68	0.88	0.79	0.76	0.65	0.94	0.93	0.64	0.66	1.01
	4	1.00	0.77	0.70	0.91	0.68	0.63	0.58	0.90	0.62	0.56	0.53	0.87
	8	1.00	0.80	0.74	0.91	0.70	0.66	0.55	0.85	0.64	0.59	0.47	0.84
	16	1.00	0.84	0.76	0.87	0.72	0.64	0.56	0.77	0.65	0.61	0.50	0.74
3	2	1.00	0.95	0.95	0.92	0.93	0.91	0.83	0.89	0.98	0.88	0.85	0.89
	4	1.00	0.90	0.85	0.88	0.91	0.83	0.78	0.82	0.87	0.80	0.75	0.78
	8	1.00	0.94	0.88	0.88	0.90	0.86	0.79	0.83	0.86	0.82	0.77	0.80
	16	1.00	0.97	0.90	0.89	0.93	0.86	0.81	0.82	0.89	0.86	0.79	0.80
4	2	1.00	0.95	0.93	0.90	0.94	0.89	0.89	0.90	0.94	0.89	0.86	0.88
	4	1.00	0.97	0.94	0.91	0.97	0.92	0.89	0.91	0.94	0.91	0.86	0.88
	8	1.00	0.95	0.93	0.92	0.95	0.92	0.88	0.88	0.92	0.90	0.86	0.85
	16	1.00	0.96	0.94	0.92	0.95	0.93	0.89	0.88	0.93	0.91	0.88	0.86
8	2	1.00	0.99	0.99	0.97	1.01	1.00	0.97	0.96	1.01	0.98	0.97	0.96
	4	1.00	0.99	0.97	0.95	0.98	0.96	0.95	0.94	0.97	0.97	0.94	0.92
	8	1.00	0.99	0.97	0.96	0.99	0.97	0.95	0.94	0.97	0.97	0.94	0.93
	16	1.00	0.99	0.97	0.97	0.99	0.98	0.95	0.94	0.97	0.96	0.95	0.93
16	2	1.00	1.00	0.99	0.98	1.00	0.99	0.99	0.98	1.00	0.98	0.98	0.98
	4	1.00	1.00	1.00	0.98	1.00	0.99	0.98	0.97	1.00	0.99	0.98	0.96
	8	1.00	0.99	0.99	0.98	0.99	0.99	0.98	0.97	0.99	0.99	0.97	0.96
	16	1.00	1.00	0.99	0.98	0.99	0.98	0.98	0.97	0.99	0.98	0.97	0.96

Table C.24. Cutsizes averages for geometric graphs (for K-PLM-like algorithms)

PROBLEM		CUTSIZE AVERAGES											
D	K	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
2	2	1.00	1.17	0.37	2.08	1.15	0.46	0.06	1.87	0.94	0.44	0.71	1.56
	4	1.00	0.70	0.45	0.92	0.54	0.28	0.19	0.87	0.27	0.16	0.21	0.83
	8	1.00	0.76	0.56	0.74	0.45	0.31	0.28	0.62	0.18	0.14	0.34	0.58
	16	1.00	0.80	0.62	0.57	0.29	0.34	0.32	0.56	0.17	0.11	0.23	0.50
3	2	1.00	0.79	0.37	0.88	0.77	0.41	0.23	0.60	0.78	0.29	0.28	0.77
	4	1.00	0.62	0.54	0.56	0.55	0.48	0.28	0.36	0.53	0.41	0.22	0.38
	8	1.00	0.83	0.60	0.47	0.59	0.55	0.37	0.34	0.40	0.41	0.26	0.39
	16	1.00	0.86	0.77	0.59	0.55	0.48	0.40	0.43	0.36	0.37	0.30	0.34
4	2	1.00	1.03	0.81	0.78	1.19	0.86	0.39	0.81	1.12	0.50	0.49	0.84
	4	1.00	0.81	0.71	0.73	0.68	0.48	0.52	0.51	0.48	0.37	0.45	0.48
	8	1.00	0.90	0.73	0.61	0.63	0.62	0.54	0.53	0.53	0.52	0.45	0.48
	16	1.00	0.97	0.82	0.75	0.71	0.69	0.59	0.63	0.64	0.62	0.55	0.57
8	2	1.00	0.78	0.85	1.05	0.70	0.75	0.72	0.91	0.99	0.68	0.65	0.80
	4	1.00	0.80	0.75	0.74	0.80	0.76	0.61	0.56	0.69	0.67	0.52	0.58
	8	1.00	0.87	0.75	0.73	0.78	0.68	0.61	0.59	0.65	0.61	0.58	0.53
	16	1.00	1.03	0.88	0.91	0.88	0.86	0.81	0.80	0.83	0.83	0.79	0.78
16	2	1.00	0.89	0.92	0.96	0.90	0.86	0.93	1.07	1.01	0.85	0.92	0.95
	4	1.00	0.95	0.85	0.86	0.90	0.87	0.78	0.83	0.84	0.82	0.77	0.77
	8	1.00	0.94	0.98	0.94	0.95	0.95	0.94	0.93	0.94	0.94	0.93	0.92
	16	1.00	0.98	0.96	0.96	0.96	0.95	0.94	0.93	0.95	0.94	0.93	0.92

Table C.25. Execution time averages (and standard deviations) for benchmark circuits

PROBLEM		EXECUTION TIME AVERAGES (in seconds)				RATIO
Name	K	PLM1	PFM1	PFM2	PFM3	
balu	2	2.42 (0.29)	3.64 (0.67)	4.45 (1.15)	6.89 (1.40)	2.84
	4	2.70 (0.53)	5.34 (1.60)	27.02 (6.48)	66.59 (24.45)	24.62
	6	4.40 (0.90)	8.76 (2.15)	43.92 (14.38)	238.89 (78.58)	54.29
	8	6.51 (1.43)	11.03 (4.10)	62.44 (21.18)	376.89 (182.80)	57.85
	10	7.67 (1.23)	13.55 (2.73)	90.32 (35.21)	694.05 (273.85)	90.49
sioo	2	1.36 (0.36)	1.02 (0.19)	2.13 (0.60)	7.07 (1.68)	5.20
	4	1.82 (0.36)	1.83 (0.52)	9.14 (4.66)	43.31 (12.90)	23.86
	6	2.94 (0.44)	2.85 (0.80)	16.74 (7.37)	84.87 (31.54)	28.82
	8	5.29 (0.78)	4.16 (0.48)	33.58 (15.46)	178.73 (55.98)	33.79
	10	7.50 (1.47)	6.36 (0.81)	33.93 (11.38)	364.58 (121.26)	48.61
primary1	2	2.81 (0.38)	3.90 (1.07)	5.24 (1.92)	10.85 (5.01)	3.86
	4	4.45 (1.02)	11.25 (2.43)	24.25 (9.12)	80.12 (20.76)	18.02
	6	7.11 (1.43)	13.98 (3.37)	51.58 (15.38)	168.95 (41.90)	23.75
	8	9.97 (2.74)	22.50 (6.65)	84.51 (29.24)	536.11 (175.66)	53.80
	10	13.56 (3.83)	33.40 (10.24)	163.91 (47.14)	761.19 (296.71)	56.11
struct	2	7.63 (1.50)	23.28 (7.12)	30.03 (7.84)	61.27 (14.64)	8.03
	4	9.86 (2.24)	51.60 (22.13)	116.20 (45.64)	341.01 (117.99)	34.57
	6	16.33 (4.42)	78.37 (21.92)	257.47 (83.85)	1006.45 (321.86)	61.65
	8	29.22 (7.70)	110.13 (19.89)	479.20 (170.87)	2383.57 (863.01)	81.59
	10	41.73 (11.78)	165.47 (27.82)	780.43 (357.21)	4478.32 (1455.08)	107.32
industry1	2	10.20 (3.80)	16.00 (3.52)	32.50 (14.07)	53.96 (13.30)	5.29
	4	11.38 (2.44)	53.49 (19.85)	152.29 (59.16)	559.79 (231.35)	49.21
	6	20.08 (4.80)	39.17 (12.75)	293.19 (120.12)	1023.61 (391.90)	50.99
	8	27.21 (7.19)	62.60 (15.47)	565.87 (236.42)	2220.35 (989.41)	81.60
	10	40.74 (9.24)	102.82 (23.41)	586.58 (217.13)	3499.45 (1772.60)	85.91
primary2	2	15.44 (3.39)	23.80 (10.09)	55.26 (21.18)	85.16 (28.65)	5.52
	4	23.56 (5.75)	76.55 (27.81)	252.34 (66.08)	878.96 (304.19)	37.32
	6	37.31 (6.40)	89.61 (26.17)	507.98 (195.96)	1577.86 (417.87)	42.28
	8	53.72 (13.16)	123.60 (56.23)	900.58 (261.92)	3353.91 (1082.74)	62.43
	10	80.36 (14.84)	172.52 (61.50)	1050.40 (322.83)	7027.23 (2320.50)	87.44

Table C.26. Cutsizes averages (and standard deviations) for benchmark circuits

PROBLEM		CUTSIZE AVERAGES				IMP. (%)
Name	K	PLM1	PFM1	PFM2	PFM3	
balu	2	32.10 (6.19)	37.80 (11.12)	42.90 (6.76)	39.80 (7.61)	-23.99
	4	177.60 (7.52)	155.60 (14.79)	105.10 (16.36)	93.70 (19.46)	47.24
	6	203.60 (7.98)	186.90 (10.85)	150.00 (9.62)	128.00 (15.12)	37.13
	8	225.00 (8.46)	213.20 (7.40)	174.80 (14.03)	162.60 (13.18)	27.73
	10	244.50 (6.26)	233.40 (5.52)	202.20 (12.81)	187.10 (12.00)	23.48
sioo	2	50.50 (9.56)	69.10 (2.34)	63.70 (3.13)	28.30 (9.90)	43.96
	4	101.45 (8.17)	97.00 (4.05)	90.20 (10.78)	74.80 (7.30)	26.27
	6	128.00 (4.42)	125.60 (6.22)	123.00 (7.50)	113.60 (7.24)	11.25
	8	147.95 (4.52)	150.00 (4.40)	134.60 (6.07)	131.40 (5.71)	11.19
	10	160.15 (7.21)	153.60 (7.34)	150.40 (5.31)	142.60 (6.96)	10.96
primary1	2	75.20 (9.34)	78.90 (7.29)	69.80 (8.60)	72.20 (8.26)	3.99
	4	198.15 (14.18)	152.80 (12.11)	129.10 (8.35)	107.90 (7.46)	45.55
	6	232.80 (14.31)	192.80 (17.51)	143.30 (8.23)	133.90 (10.64)	42.48
	8	263.15 (17.73)	209.30 (16.51)	158.90 (9.72)	143.50 (7.13)	45.47
	10	291.40 (21.00)	222.00 (23.73)	173.10 (15.06)	168.60 (11.02)	42.14
struct	2	57.00 (9.31)	81.70 (22.93)	66.90 (14.39)	55.00 (9.52)	3.51
	4	301.20 (19.89)	166.10 (25.60)	126.20 (16.63)	111.50 (12.04)	62.98
	6	449.45 (42.76)	304.70 (31.01)	211.40 (49.34)	185.30 (18.17)	58.77
	8	587.60 (40.17)	411.80 (49.82)	346.60 (40.40)	312.00 (37.90)	46.90
	10	624.80 (41.18)	498.50 (25.37)	404.00 (39.50)	362.70 (45.67)	41.95
industry1	2	74.65 (34.96)	128.30 (15.56)	106.90 (32.33)	98.40 (12.56)	-31.82
	4	449.60 (29.45)	360.90 (39.95)	243.40 (34.00)	193.70 (33.68)	56.92
	6	552.40 (22.55)	500.60 (27.92)	351.90 (43.59)	289.60 (27.75)	47.57
	8	596.45 (20.04)	536.10 (12.72)	374.60 (46.85)	359.00 (44.96)	39.81
	10	630.90 (18.72)	554.20 (20.22)	432.70 (17.03)	409.80 (25.91)	35.05
primary2	2	259.05 (36.07)	261.00 (33.65)	235.30 (26.53)	222.20 (32.10)	14.23
	4	739.70 (36.66)	610.70 (54.28)	455.20 (22.58)	401.60 (23.32)	45.71
	6	945.80 (17.78)	869.90 (36.00)	568.30 (42.60)	538.90 (17.04)	43.02
	8	986.85 (23.65)	923.50 (41.76)	642.80 (34.75)	633.60 (32.72)	35.80
	10	1069.15 (15.90)	1010.00 (29.15)	721.50 (57.00)	668.90 (52.86)	37.44

Table C.27. Minimum Cutsizes for benchmark circuits

PROBLEM		MINIMUM CUTSIZES				IMP. (%)
Name	K	PLM ₁	PFM ₁	PFM ₂	PFM ₃	
balu	2	27	27	29	27	0.00
	4	166	128	77	66	60.24
	6	185	171	137	105	43.24
	8	208	199	153	134	35.58
	10	229	227	169	166	27.51
sioo	2	34	67	58	25	26.47
	4	84	87	74	64	23.81
	6	119	118	108	97	18.49
	8	139	144	126	124	10.79
	10	147	143	145	133	9.52
primary1	2	59	67	59	58	1.69
	4	170	125	115	95	44.12
	6	197	170	128	118	40.10
	8	226	190	142	129	42.92
	10	255	189	146	148	41.96
struct	2	43	37	37	38	11.63
	4	256	126	94	91	64.45
	6	379	246	114	157	58.58
	8	516	330	282	257	50.19
	10	553	458	358	280	49.37
industry1	2	23	90	42	86	-273.91
	4	391	304	191	139	64.45
	6	507	421	252	218	57.00
	8	563	519	282	287	49.02
	10	600	529	405	366	39.00
primary2	2	178	214	201	158	11.24
	4	658	505	412	364	44.68
	6	908	809	521	517	43.06
	8	943	859	598	597	36.69
	10	1035	969	631	604	41.64

Table C.28. Cutsizes averages for some benchmark circuits

PROBLEM		CUTSIZE AVERAGES				
Name	K	PLM ₁	PLM ₁₁	PFM ₁	PFM ₂	PFM ₃
balu	2	32.10	35.00	37.80	42.90	39.80
	4	177.60	120.50	155.60	105.10	93.70
	6	203.60	164.20	186.90	150.00	128.00
	8	225.00	185.70	213.20	174.80	162.60
	10	244.50	206.70	233.40	202.20	187.10
struct	2	57.00	51.10	81.70	66.90	55.00
	4	301.20	166.40	166.10	126.20	111.50
	6	449.45	198.40	304.70	211.40	185.30
	8	587.60	316.30	411.80	346.60	312.00
	10	624.80	438.60	498.50	404.00	362.70
industry1	2	74.65	60.50	128.30	106.90	98.40
	4	449.60	305.90	360.90	243.40	193.70
	6	552.40	404.30	500.60	351.90	289.60
	8	596.45	467.30	536.10	374.60	359.00
	10	630.90	486.50	554.20	432.70	409.80

Table C.29. Minimum Cutsizes for some benchmark circuits

<i>PROBLEM</i>		<i>MINIMUM CUTSIZES</i>				
<i>Name</i>	<i>K</i>	<i>PLM1</i>	<i>PLM11</i>	<i>PFM1</i>	<i>PFM2</i>	<i>PFM3</i>
balu	2	27	27	27	29	27
	4	166	85	128	77	66
	6	185	133	171	137	105
	8	208	161	199	153	134
	10	229	197	227	169	166
struct	2	43	43	37	37	38
	4	256	133	126	94	91
	6	379	165	246	114	157
	8	516	249	330	282	257
	10	553	392	458	358	280
industry1	2	23	45	90	42	86
	4	391	251	304	191	139
	6	507	363	421	252	218
	8	563	430	519	282	287
	10	600	454	529	405	366

Bibliography

- [1] D. Adler. Switch-level simulation using dynamic graph algorithms. *IEEE Transactions on Computer-Aided Design*, 10(3):346–355, March 1991.
- [2] S. N. Bhatt and F. T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28:300–343, 1984.
- [3] D. E. Van Den Bout and T. K. Miller. Graph partitioning using annealed neural networks. *IEEE Transactions on Neural Networks*, 1(2):192–203, June 1990.
- [4] M. A. Breuer. Min-cut placement. *Journal of Design Automation and Fault Tolerant Computing*, 1(4):343–362, October 1977.
- [5] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.
- [6] T. N. Bui, C. Heigham, C. Jones, and F. T. Leighton. Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 775–778, 1989.
- [7] T. Bultan and C. Aykanat. A new mapping heuristic based on mean field annealing. *Journal of Parallel and Distributed Computing*, 16:292–305, 1992.
- [8] W.-K. Chen, M. F. M. Stallmann, and E. F. Gehringer. Hypercube embedding heuristics: An evaluation. *International Journal of Parallel Programming*, 18(6):505–549, November 1989.
- [9] L. I. Corrigan. A placement capability based on partitioning. In *Proceedings of the 16th Design Automation Conference*, pages 406–413, 1979.

- [10] P. Cox, R. Burch, and B. Epler. Circuit partitioning for parallel processing. In *Proceedings of the International Conference on Computer-Aided Design*, pages 186–189, Santa Clara, California, 1986.
- [11] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 4(1):92–98, January 1985.
- [12] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.
- [13] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman and Co., New York, New York, 1979.
- [14] J. A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.
- [15] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 16(6):427–449, November 1987.
- [16] L. Herault and J.-J. Niez. Neural networks and graph k-partitioning. *Complex Systems*, 3:531–575, 1989.
- [17] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, November 1989.
- [18] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37:79–100, 1988.
- [19] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.
- [20] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, May 1984.
- [21] U. Lauther. A min-cut placement algorithm for general cell assemblies based on a graph representation. In *Proceedings of the 16th Design Automation Conference*, pages 1–10, 1979.

- [22] C. Leiserson. Area-efficient graph layout (for VLSI). In *Proceedings of the 21st Annual Symposium on the Foundations of Computer Science*, pages 270–281, New York, 1980. IEEE.
- [23] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, Inc., Chichester, West Sussex, England, 1990.
- [24] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.
- [25] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.
- [26] J. W. H. Liu. A graph partitioning algorithm by node separators. *ACM Transactions on Mathematical Software*, 15(3):198–219, September 1989.
- [27] V. B. Rao and T. N. Trich. Network partitioning and ordering for MOS VLSI circuits. *IEEE Transactions on Computers*, 6(1):128–144, January 1987.
- [28] R. L. Rivest. The "PI" (placement and interconnect) system. In *Proceedings of the 19th Design Automation Conference*, pages 475–481. IEEE, 1982.
- [29] Y. G. Saab and V. B. Rao. An evolution-based approach to partitioning ASIC systems. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 767–770, 1989.
- [30] P. Sadayappan and F. Ercal. Cluster-partitioning approaches to mapping parallel programs onto a hypercube. In *Proceedings of the International Conference on Supercomputing*, Athens, Greece, June 1987.
- [31] L. A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, January 1989.
- [32] D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th ACM/IEEE Design Automation Conference*, pages 57–62, 1972.
- [33] A. Sen, H. Deng, and S. Guha. On a graph partition problem with applications to VLSI layout. *Information Processing Letters*, 43(2):87–94, August 1992.

- [34] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):144–220, June 1991.
- [35] H. Shiraishi and F. Hirose. Efficient placement and routing techniques for master-slice LSI. In *Proceedings of the 17th Design Automation Conference*, pages 458–464, 1980.
- [36] P. Suaris and G. Kedem. A new approach to standard cell layout. In *Proceedings of the International Conference on Computer-Aided Design*, pages 474–477. IEEE, 1987.
- [37] O. Tejayadi and I. N. Hajj. Dynamic partitioning methods for piecewise-linear VLSI circuit simulation. *International Journal of Circuit Theory and Applications*, 16:457–472, 1988.
- [38] K. Thulasiraman and M. N. S. Swamy. *Graphs: theory and algorithms*. John Wiley and Sons, Inc., New York, NY, USA, 1992.
- [39] A. Vanelli and K. R. Kumar. A method for finding minimal bottleneck cells for grouping part-machine families. *International Journal of Production Research*, 24:387–400, 1986.
- [40] G. Vijayan. Partitioning logic on graph structures to minimize routing cost. *IEEE Transactions on Computer-Aided Design*, 9(12):1326–1334, December 1990.
- [41] J. White and A. Sangiovanni-Vincentelli. Partitioning algorithms and parallel implementation of waveform relaxation for circuit simulation. In *Proceedings of the International Symposium on Circuits and Systems*, pages 221–224. IEEE, June 1985.

Additions and Corrections

Page No.	Line No.	The Additon (A) or Modification (M)
xvii	-	(A) \leftarrow assignment operator
xvii	-	(A) \wedge logical and operator
10	last	(M) ... both end vertices lie in P_k .
21	first	(M) The cost $\chi(\Pi)$ of the partition is ...
40	16	(M) ... weight w_T , and the total net weight c_T ,
42	4	(M) Suppose thah $H = (V, E)$ is a hypergraph with ...
48	28	(M) ... the item (3) establishes the basis of ...
49	19	(M) ... is given in Figure 4.7. The algorithm employs two ...
50	-	(M) 2.2.5. if $K_2 < N$ then free the bucket list nodes
50	-	(M) 2.4. find the maximum prefix sum <i>gainsum</i> of move gains of $K_1 K_2$ moves
53	12	(M) in a different concept from all the other ...
56	-	(M) 2.6. find the maximum prefix sum <i>gainsum</i> of move gains of K_1 moves
56	-	(A) 2.9. free the bucket list nodes
87	12	(M) partitioning algorithms by changing the parameters ...