FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INSTITUTE OF INFORMATICS
DEPARTMENT OF THEORETICAL COMPUTER SCIENCE
GRADUATE PROGRAM IN COMPUTER SCIENCE

BRUNO MENEGOLA

A Study of the k-way Graph Partitioning Problem

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Marcus Rolf Peter Ritt
Advisor

Porto Alegre, November 2012

CIP – CATALOGING-IN-PUBLICATION

Menegola, Bruno

A Study of the k-way Graph Partitioning Problem / Bruno Menegola. – Porto Alegre: PPGC da UFRGS, 2012.

102 f.: il.

Dissertation (Master) – Federal University of Rio Grande do Sul. Graduate Program in Computer Science, Porto Alegre, BR–RS, 2012. Advisor: Marcus Rolf Peter Ritt.

1. Graph Partitioning. 2. Heuristics. 3. Metaheuristics. 4. Algorithms. I. Ritt, Marcus Rolf Peter. II. A Study of the k-way Graph Partitioning Problem.

FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Pró-Reitor de Coordenação Acadêmica: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir do Nascimento
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGMENTS

First and foremost, I wish to express my love and gratitude to my parents, João and Laurinda, my brothers, Julia and Afonso, for their love, support and encouragement. Without my parents strength, vision and wisdom, I would not get this accomplishment, neither the ones that will succeed.

I own much to my advisor Marcus Ritt for his patience, guidance and support over the past years. I am greatly thankful for his teachings, many academic ones but even some for life. The active interest, methodic manners and perfectionism are some of his many virtues that I wish to see more in the professors of our university. I also would like to thank my former advisor, Luciana Buriol, for her support back in my undergraduate times and also for leading, along with Marcus, a great research group.

I also must thank my numerous friends in Porto Alegre and in Caxias do Sul for their friendship. They contributed much to define the perspectives I have today and many times helped me to keep the clarity of my thoughts. I specially thank my friend Ezequiel Führ for everything we past together in these many years.

This research was developed with resources of the Supercomputing National Center of Federal University of Rio Grande do Sul (CESUP). Whithout these resources my experiments would be impossible to be completed in acceptable time.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

NOMENCLATURE

ARD	Average Relative Deviation
BKV	Best Known Value
GPP	Graph Partitioning Problem
KaFFPa	Karlsruhe Fast Flow Partitioner
KaPPa	Karlsruhe Parallel Partitioner
KaSPar	Karlsruhe Sequential Partitioner
SAG	Strict Aggregation
VLSI	Very-Large-Scale Integration
WAG	Weighted Aggregation

Notation

ϵ	Maximum allowed imbalance for a partition
$\omega(\{u, v\})$	Weight of edge $\{u, v\}$
$\Pi_P(u)$	The subset of V, part of P, that contains the vertex u
$\Theta(P)$	Cut size of a partition P
$c(u)$	Weight of vertex u
E	Set of edges
G	A graph
k	Size of a partition
m	Number of edges
n	Number of vertices
$N^S(u)$	Neighborhood of a vertex u, masked by some vertex set S
$P = \{P_0, \ldots, P_{k-1}\}$	Partition composed of k parts

V	Set of vertices

Algorithms

DG	Differential Greedy
FM	Fiduccia-Mattheyses
GEM	Greedy Edge Matching
GGGP	Greedy Graph Growing Partitioning
GGP	Graph Growing Partitioning
GPA	Global Path Algorithm
GRASP	Greedy Randomized Adaptive Search Procedure
HEM	Heavy Edge Matching
HYPAL	Hybrid k-way Graph Partitioning Algorithm
KL	Kernighan-Lin
LG	Lock-gain
MMG	Min-max Greedy
PR	Path-relinking
SB	Spectral Bisection
SHEM	Sorted Heavy Edge Matching
TFM	Tabu based Fiduccia-Mattheyses

ABSTRACT

The balanced graph partitioning problem asks to find a k-partition of the vertex set of an undirected graph, which minimizes the total cut size and such that the size of no part exceeds $\lfloor \epsilon n/k \rfloor$, for some $\epsilon \in [1, k)$. This dissertation studies this problem, providing a recent review of constructive heuristics, refinement heuristics and multilevel techniques. We also propose a new hybrid algorithm for solving this partitioning problem. We show how several good existing strategies for constructing and improving partitions, as well as some newly proposed ones, can be integrated to form a GRASP with path-relinking. We report computational experiments that show that this approach obtains solutions competitive with state-of-the-art partitioners. In particular, the hybrid algorithm is able to find new best known values in some of the smaller instances, indicating that it can make a qualitative contribution compared to existing methods.

Keywords: Graph Partitioning, Heuristics, Metaheuristics, Algorithms.

Um Estudo do Problema de Particionamento de Grafos em k-partes

RESUMO

O problema de particionamento balanceado de grafos consiste em encontrar uma partição de tamanho k dos vértices de um grafo, minimizando o número de arestas que participam do corte tal que o tamanho de nenhuma parte exceda $\lfloor \epsilon n/k \rfloor$, para algum $\epsilon \in [1, k)$. Essa dissertação estuda esse problema, apresentando uma revisão recente de heurísticas construtivas, heurísticas de refinamento e técnicas multinível. Também propomos um novo algoritmo híbrido para resolver esse problema de particionamento. Nós mostramos como diversas estratégias para construir e aprimorar partições, assim como algumas novas propostas, podem ser integradas para formar um GRASP com path-relinking. Reportamos experimentos computacionais que mostram que essa abordagem obtém soluções competitivas com particionadores no estado-da-arte. Em particular, o algoritmo híbrido é capaz de encontrar novos melhores valores conhecidos em algumas das menores instâncias, indicando que tem uma contribuição qualitativa comparado aos métodos existentes.

Palavras-chave: Particionamento de Grafos, Heurísticas, Metaheurísticas, Algoritmos.

1 INTRODUCTION

Over the last 50 years, there has been much research on combinatorial optimization problems. Such problems usually are derived from real world situations in which it is desired to find a combination of elements that together form one good solution. Since most of these problems have comercial applications, common objective functions are the minimization of costs or the maximization of profits by the correct allocation of resources to perform the required tasks.

An interesting example of such problems is the placement of circuit components onto a set of printed cards that work together to perform tasks of an electronic device. The cards have a restricted size, such that the device does not get too big, and the amount of components in each card is also evidently restricted, since they must be connected to each other by the circuit printed on the card's surface. In this way, if the circuit is big enough it must be split into several cards and which then must be wired interconnected. However, the cost of interconnection is higher than the cost of the printed circuit and, therefore, the number of interconnections must be minimized while the number of components in each card is restricted to some amount.

The previous example is the first application presented into the seminal paper about the balanced graph partitioning problem by Kernighan and Lin (1970), which defines an efficient heuristic to find good solutions. In the example, the circuits can be viewed theoretically as a graph and the cards, as subsets of a partition of the vertices that represent the electronic components.

Balanced graph partitioning has several other real world applications such as partitioning of VLSI circuits (ALPERT; KAHNG, 1995; JOHANNES, 1996; SŁOWIK; BIAŁKO, 2006), road network decomposition (ARORA; RAO; VAZIRANI, 2008), domain decomposition for parallel computing (SIMON, 1991; GILBERT; MILLER; TENG, 1995; SCHLOEGEL; KARYPIS; KUMAR, 2003), image segmentation (SHI; MALIK, 2000), data mining (ZHA et al., 2001), finding ground state magnetization of spin glasses (BARAHONA et al., 1988), structuring cellular networks (TORIL et al., 2010), matrix decomposition (CATALYUREK; AYKANAT, 1996), among others (CHARDAIRE; BARAKE; MCKEOWN, 2007). Extended examples for some of the applications are going to be presented later, in Section 2.3.

1.1 Research Objective and Contributions

This work focuses on the research of the state-of-the-art and the development of new heuristics for the graph partitioning problem.

This dissertation presents two major contributions. The first one is an extended study on graph partitioning – covering fundamental techniques, multilevel technique and a brief research on recent approaches to the problem. The second contribution is a new algorithm for the k-way graph partitioning, which uses the GRASP methodology as the base structure, enhanced by path-relinking technique, recent ideas for heuristics and multilevel partitioning, and some others of our own, that will be detailed later.

1.2 Scope of the Dissertation

The graph partitioning problem has several points of interest. Following the classification of Schloegel, Karypis and Kumar (2003), our research focuses on the study of combinatorial methods and multilevel schemes of static graphs. There exist applications such as numerical simulations for improving the design of helicopter blades (BISWAS; STRAWN, 1994) that requires dynamic adjustments to the mesh that describes the instance to be optimized. We do not cover these dynamic graph partitionings, neither geometric techniques, that solve the problem solely based on coordinate informations, or spectral methods, that solve the problem by the computation of eigenvectors.

For our proposed algorithms, we focus on solving the k-way version of the problem, i.e., algorithms that compute partitions of size k of the graphs. The computed solutions must have its largest part within some maximum limit, specified by a maximum imbalance parameter, explained in Section 2.1, although we do start studing the perfectly balanced bipartitioning.

Our work is restricted to sequential algorithms, but we, at some points, must study parallel algorithms to understand their contributions. However, in such cases we do not give much attention to technical details of parallel computing.

The objective function studied in this work is the minimization of the cut size on unweighted and undirected graphs. Durin the application of multilevel schemes the graphs may be weighted, but the input graphs are not. There exist other alternative objective functions that we discuss in Section 2.2, but is not in the scope of this work to propose optimizations for them.

1.3 Overview of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 introduces a formal definition of the problem with the notation used in this dissertation. It also presents alternative objective functions, some extended examples of real world applications and the definition of benchmark instances and evaluation methodology. Chapter 3 provides a recent study on the state-of-the-art algorithms related to our

scope of work. In Chapter 4 we introduce two Greedy Randomized Adaptive Search Procedures that we studied in order to define and conduct experiments with our k-way Hybrid Graph Partitioning Algorithm in Chapter 5. The most interesting and competitive results are presented in Section 5.3. Concluding remarks are finally given in Chapter 6.

2 THE GRAPH PARTITIONING PROBLEM

The balanced graph partitioning is an NP-hard combinatorial optimization problem (BUI; JONES, 1992) that consists in finding a partition of size k of the vertex set of a graph, subjected to a balance restriction. This restriction limits the maximum cardinality of the weight of each part. For such partitioning, the most common objective is to minimize the sum of the edge weights that connect vertices in distinct parts. Other objective functions have been defined, such as total or maximum communication volume, that will be presented in Section 2.2, but the minimum cut is the most researched function. In this dissertation we focus on the minimum cut case where the input graph is undirected and both vertices and edges have unit weights.

Being a hard problem, exact solutions are found in reasonable time just for small graphs. Using optimized models of integer programming (BOULLE, 2004), for example, experiments show that a state-of-the-art solver can compute a solution for a graph of 500 vertices and 625 edges in about 25 minutes. However, the applications of the problem require to partition much larger graphs and so heuristic solutions are usually indicated. There are some specific heuristics for the problem, some constructive, such as the Min-Max Greedy algorithm (BATTITI; BERTOSSI, 1999) or the Differential Greedy heuristic (BATTITI; BERTOSSI, 1997); and some of iterative refinement, such as the Kernighan-Lin local search (KERNIGHAN; LIN, 1970), the Fiduccia-Mattheyses local search (FIDUCCIA; MATTHEYSES, 1982) or the Lock-Gain local search (KIM; MOON, 2004). Although they provide good results, the refinement algorithms are usually used as local search procedures for other methods, like metaheuristics. Several metaheuristic approaches have already been proposed: Genetic Algorithms (GOLDBERG, 1989; BUI; MOON, 1996; KIM; MOON, 2004), Tabu Search (GLOVER, 1989; ROLLAND; PIRKUL; GLOVER, 1996), Greedy Randomized Adaptive Search Procedure (LAGUNA; FEO; ELROD, 1994), Simulated Annealing (KIRKPATRICK; GELATT; VECCHI, 1983; JOHNSON et al., 1989), Memetic Algorithms (GALINIER; BOUJBEL; FERNANDES, 2011), Large-step Markov Chains (MARTIN; OTTO; FELTEN, 1991) and others (FJäLLSTROM, 1998; CHARDAIRE; BARAKE; MCKEOWN, 2007; SCHLOEGEL; KARYPIS; KUMAR, 2003).

We present, in the next sections, notations and a formal definition of the problem, alternative functions, problem generalizations and some extended examples of applications. We end this chapter detailing our experimental environment, datasets and the methodology used to evaluate our results.

2.1 Notation and Problem Definition

A solution for the graph partitioning problem is a *partition*. A partition is composed by k *parts* and a part is a subset of the graph's vertex set, such that all parts are disjoint sets and their union is equal to the vertex set. Two solutions, or partitions, are equivalent if, for every part of one partition, there is a part in the other that is composed by the same vertices. If two partitions are not equivalent, they are different solutions.

Consider an undirected graph $G = (V, E, c, \omega)$, with $n = |V|$ vertices and $m = |E|$ edges, where $c(u)$ and $\omega(\{u, v\})$ are non-negative weights of a vertex and an edge, respectively. Let P be a collection of k subsets $\{P_0, \ldots, P_{k-1}\}$ of V, we can define the set $C = \{\{u, v\} \in E \mid u \in P_i, v \in P_j, 0 \leq i < j \leq k-1\}$ of cut edges and also the cut size

$$\Theta(P) = \sum_{\{u,v\} \in C} \omega(\{u, v\}), \tag{2.1}$$

which is equal to the cardinality of C if all edges have unitary weights but different otherwise. The problem of k-way graph partitioning with balance $1 \leq \epsilon < k$ can be described by the following model:

$$\textbf{minimize} \quad \Theta(P) \tag{2.2a}$$

$$\textbf{subject to} \quad 0 < \sum_{u \in P_i} c(u) \leq \left\lfloor \frac{\epsilon}{k} \sum_{v \in V} c(v) \right\rfloor \qquad \forall i \in [0, k-1] \tag{2.2b}$$

$$\bigcup_{0 \leq i < k} P_i = V \tag{2.2c}$$

$$P_i \cap P_j = \varnothing \qquad \forall i, j \in [0, k-1] \mid i \neq j \tag{2.2d}$$

The collection P is a balanced partition of V and so a solution of the problem given G. The first restriction (Equation 2.2b) ensures that part P_i does not exceed the size limit imposed by ϵ. The second and third restrictions (Equations 2.2c and 2.2d) ensure that P is a partition of V.

Usually $k = 2^l$, for some $l \in \mathbb{N}^+$ (very often $l \leq 6$, for benchmarking purposes), and so a bipartitioner could be defined and applied recursively, although the quality of these partitions tend to decrease with the increase of partition size (PELLEGRINI, 2007a). For the other possible values of k, the bipartitioner must generate parts with weights proportional to the number of subpartitions it should hold. For example, for $k = 3$ and $\epsilon = 1$, the first iterations should generate a part $|P_1| = n/3$ and another $|P_2| = 2n/3$; while the second iteration should split the part $P_2 = P_3 \cup P_4$ such as $|P_3| = |P_4| = n/3$. High quality partitioners, on the other hand, are designed to handle k-way partitions natively instead of recursively applying a bipartitioner.

If $\epsilon = 1$ the problem is also known as balanced partitioning. In this case, the restriction for each part size should take into account the possibility of n being not divisible by k, and some parts may be bigger than others. A greater imbalance does not change the complexity class of the problem, provided that $1 \leq \epsilon < k$, i.e., the problem remains NP-hard (BUI; JONES, 1992).

To ease the formal definition of some intermediate concepts for the problem, we will define some auxiliary functions. The neighborhood of a vertex u, masked by

some vertex set S is defined as

$$N^S(u) = \{v \mid \{u, v\} \in E, v \in S\}. \tag{2.3}$$

The subset of V, part of some partition P, that contains a vertex u can be retrieved with the function

$$\Pi_P(u) = \{v \mid v \in P_p, u \in P_p\}. \tag{2.4}$$

In this dissertation, all input graphs will be unweighted, i.e., $c(u) = 1$ and $\omega(\{u, v\}) = 1$. Although, intermediate graphs for some algorithms are actually weighted.

2.2 Alternative Objective Functions

Equation 2.1 is one of the possible objective functions for the GPP. The minimization of the cut size accurately reproduces the requirements for many of the problem's applications. However, in some real world problems that can be reduced to the graph partitioning, it was noted that this function did not model all cases correctly. For example, parallel scientific computing applications require a special objective function that models accurately the communication volume among nodes of a distributed computer.

In this case, the computation can be modeled with a graph, with vertices representing atomic computations and edges representing information exchanges. Several atomic computations can be executed by a single processor and the cost of information exchange among these is negligible. Each atomic computation has a computing time cost, or weight, and it is desirable that each processor has about the same amount of work. This way, the utilization of each unit is maximized, i.e., the units shall be idle and waiting for synchronizations or information exchanges the least time possible. Besides, the volume of communication between units must be minimized because this is a very costly task.

It is easy to understand that a partitioning of this graph, minimizing cut size, leads to a minimization of the communication volume, since every edge represents information exchanges. However, if we use the minimum cut size as objective function, we are just estimating this volume, while it could be more precisely defined and optimized by using another function. The *total communication volume* of a part is proportional to the number of neighboring vertices in other parts, and not to the number of edges.

As an example, consider the partition presented in Figure 2.1. The cut size of this partition is 10, assuming that each edge is doubled since the communication can be made in both ways. On the other hand, the *total communication volume* is 7, because one vertex is able to communicate just once with two or more vertices that are in another unit. This happens, for example, with vertices v_1, v_2 and v_7: the communication volume among these vertices is 3, one for v_7 to v_1 and v_2, one for v_1 to v_7 and another for v_2 to v_7.

The major flaw of the minimum cut size, when it is used to optimize the communication volume, is that it overestimates the real volume. The minimum cut size and

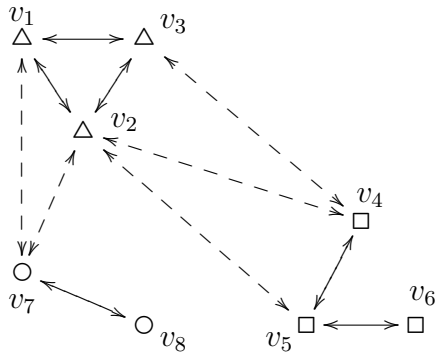


Figure 2.1: Example of total communication volume (HENDRICKSON; KOLDA, 2000). The cut size is 10, with double edges, while the total communication volume is 7, since all edges leaving from the same vertex are counted only once.

the total communication volume are not the same because the former counts each edge in the cut, while messages are required to be delivered just once if two or more edges connect a vertex from a part P_0 to others in another part P_1 (SCHLOEGEL; KARYPIS; KUMAR, 2003).

Another objective function to this application is the *maximum communication volume*, that corresponds to the greatest volume that is produced among two processor units. This function is more interesting in parallel computations because, when barrier synchronizations are required, the unit with the maximum communication volume will determine the communication time.

The maximum communication volume is frequently not treated specifically, but, instead, the total volume is optimized because it is easier to deal with and also because it approximates very well the maximum volume (SCHLOEGEL; KARYPIS; KUMAR, 2003).

The total communication volume can be minimized using relatively simple heuristics (SCHLOEGEL; KARYPIS; KUMAR, 2003), but the most elegant solution uses hypergraphs because they can model more accurately the data communications (HENDRICKSON; KOLDA, 2000).

2.3 Applications

In this section, we present some applications in more detail to illustrate how real world problems can be modeled with graphs and solved by the algorithms that we will present later.

2.3.1 Mobile Cellphone Networks

In a GSM-EDGE Radio Access Network system, key part of GSM cellphone network, the coverage area for mobile phones is divided into cells (TORIL et al., 2010). Each cell is served by a *Base Station* (BS). BSs are assigned to *Package Control Units* (PCU) in order to provide data services. Several PCUs are located in a *Base Station Controller* (BSC) and they deal with several hundreds of BSs. The

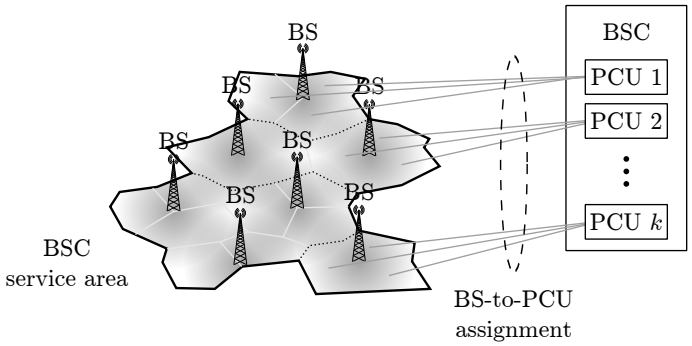


Figure 2.2: Mobile cellphone network basic hierarchy (TORIL et al., 2010). An example of Base Stations (BS) assignment to Package Control Units (PCU).

BSCs are usually in a small number and located in few places. Figure 2.2 gives an example of the hierarchy of the components.

Users are connected with a BS and usually swap connections with other BSs when they are moving around. Internally, a procedure is required to reassign a user to another BS. The time for switching from a BS to another is larger, when they are assigned to different PCUs. This can severely degrade the performance and quality of the services. Therefore, the BSs assignment to PCUs must minimize the number of users that swap BSs in different PCUs. Besides that, the total processing load must be limited to the capacity of the PCUs.

The problem is modeled using an undirected graph, where every vertex represents a BS and edges represent the possible movements of users among BSs. The weight of each edge is the estimated number of users that migrate from one BS to another, while the weight of each BS is the estimated workload required by the PCU to process it. These values can be obtained through a statistical analysis of the network data within a certain interval of time.

Finally, a graph partitioning is retrieved. The size of this partition is equal to the number of PCUs and the vertices of each resulting part are assigned to each PCU. A simple definition would assume that all PCUs have the same processing load limits, but different limits could be modeled into the partitioner.

2.3.2 Scientific Simulations

Many scientific simulations are modeled using irregular 2-D or 3-D meshes and the related computations are executed for each face or face intersection with exchange of information among them after a certain number of iterations.

Figure 2.3 exemplifies the modeling of an air flow simulation for an airfoil using an irregular 2-D mesh. The computation, in this example, is performed at every triangle and must be assigned to several processors. The communication must be minimized and, since every face exchanges information with adjacent elements, this mesh can be modeled as a graph and partitioned minimizing the cut size. It is also important that parts have similar weights, such that every processor performs roughly the same amount of computation. This reduces the total running time by minimizing idle times.

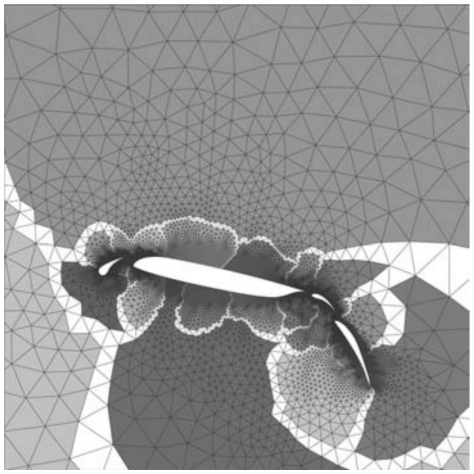


Figure 2.3: Example of partitioning of a 2-D mesh modeling an airfoil simulation (SCHLOEGEL; KARYPIS; KUMAR, 2003). Computations are performed at every face and information is exchanged among adjacent elements. Each shade of gray maps computations to different processors.

This is a clear example of the graph partitioning problem. The model can be extended to consider different computation times for each element, by including vertex weights; or to consider different communication volumes, by including edge weights. Different processors can be considered by adjusting the imbalance parameter or, with more precision, specifying the maximum weight for each part.

2.3.3 VLSI Circuits

The Very-Large-Scale Integration consists in the creation of integrated circuits composed by thousands of transistors on a single chip. An example of a VLSI circuit is a microprocessor of a PC. These circuits are composed of several interconnected subcircuits that are capable of performing one or several operations, similar to functions in a computer program. The problem, though, appears when the circuit design must be transformed into a physical circuit: these interconnections are physical wires and, without proper caution, they increase the circuit size, cost and power usage (ALPERT; KAHNG, 1995).

Partitioning is then used in circuit design for several reasons. It is used to address increasing complexity of VLSI design, dividing the system into smaller and more manageable components. It also has great impact on system performance, since, in currently technology, wire delays dominate gate delays (BAKOGLU, 1990). In electronics, the propagation delay is the length of time between when an input logic state becomes stable and valid and when the output state also does. Wires have and approximate propagation delay of 1 ns for every 15 cm of length, while logic gates can be down to the picosecond range (BALCH, 2003).

Circuits are described on a high-level as the example diagram of Figure 2.4(a). Since a wire is capable of connecting more than two components, hypergraphs are required to precisely represent the circuits diagrams (Figure 2.4(b)). Every wire is represented by a hyperedge, that is composed by all vertices that represent components connected by the wire. A hypergraph partitioner is then required to solve

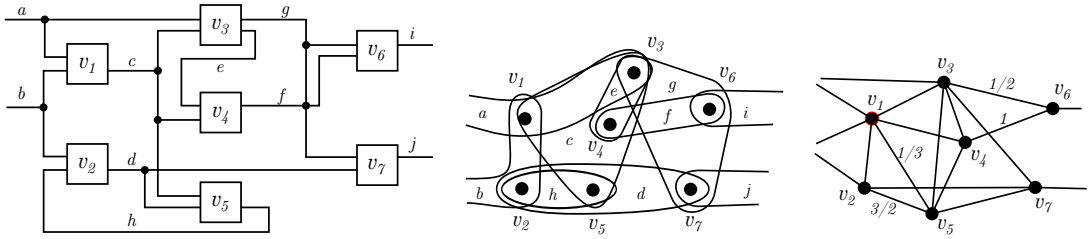


Figure 2.4: Examples of circuit representations (ALPERT; KAHNG, 1995). Figure (a) is the circuit diagram with inputs on the left side and outputs on the right side of the components. Figure (b) is the hypergraph representation of the diagram. Figure (c) is the weighted graph representation using the standard clique model, which replaces every hyperedge with a clique of the related vertices. Not every weights were labeled to simplify the visualization of the example.

the partitioning. Often, the hypergraph is transformed in a simple graph in order to use the better partitioners that exist for this purpose. A standard conversion is to replace every hyperedge by a clique of all its vertices. The resulting edges are weighted with $1/(|e| - 1)$, with $|e|$ being the size of the hyperedge. An example of this conversion can be seen on Figure 2.4(c).

2.4 Experimental Environment and Datasets

In this section we introduce our experimental environment and datasets used throughout this dissertation for testing some of the presented algorithms. Other authors often use one of two datasets: the one from Johnson et al. (1989) and the other from Walshaw (2000). These datasets shall be detailed below because we also use them for our experiments.

The machines used for experiments are detailed by Tables 2.1 and 2.2. We will call them Machine A and Machine B, respectively. Although they have several cores, most of the presented algorithms are sequential. Multi-threaded programs will be mentioned explicitly in their correspondent sections of experimental results.

Table 2.1: Machine A description, used for algorithm experimentation.

Processor:	Intel Core i7 930 @ 2.8 GHz, Quad-core, 8 MiB cache L3
Memory:	12 GiB RAM DDR3 Tripple Channel @ 1333 MHz
Operating System:	Ubuntu 12.04.1 with Linux 3.2.0
C++ compiler:	g++ 4.6.3

2.4.1 Johnson's Dataset

Used by Johnson et al. (1989) to test a simulated annealing algorithm for the GPP and by some other authors later, the dataset consists of two classes of instances.

The first class of instances are composed by random graphs, denoted by $Gn.p$, where n is the number of vertices and p is the probability that any two vertices have

Table 2.2: Machine B description, used for algorithm experimentation. There are two processor models because this configuration represents a node of a cluster with mixed machines. Nodes are selected randomly by the batch system. Each node has two processors of the same model.

Processor:	2x AMD Opteron 6176 SE @ 2.3 GHz, 12 cores, 12 MiB cache L3
	2x AMD Opteron 6238 @ 2.6 GHz, 12 cores, 12 MiB cache L3
Memory:	64 GiB RAM DDR3 @ 1333 MHz, up to 42.7GB/s of memory bandwidth
Operating System:	Novell SUSE Linux Enterprise Server 11-SP1
C++ compiler:	g++ 4.7.0

an edge between them two. These graphs have an expected average vertex degree $E[\bar{\delta}]) = p(n-1)$. The relation between the number of vertices, edge probability, expected average vertex degree and other properties for every instance presented by Johnson et al. is shown by Table 2.3.

Table 2.3: Instances of Johnson et al. (1989). The table shows the number of vertices (n); the number of edges (m); the minimum, average, expected and maximum vertex degree (δ, $\bar{\delta}$, $E[\bar{\delta}]$, Δ); and the number of connected components (C.C.). The first group of instances are random graphs and the second, geometric graphs.

Instance	n	m	δ	$\bar{\delta}$	$E[\bar{\delta}]$	Δ	C.C.
G124.02	124	149	0	2.40	2.50	7	13
G124.04	124	318	1	5.13	5.00	11	1
G124.08	124	620	2	10.00	10.00	20	1
G124.16	124	1271	12	20.50	20.00	30	1
G250.01	250	331	0	2.65	2.50	7	21
G250.02	250	612	0	4.90	5.00	12	3
G250.04	250	1283	3	10.26	10.00	20	1
G250.08	250	2421	10	19.37	20.00	30	1
G500.005	500	625	0	2.50	2.50	9	55
G500.01	500	1223	0	4.89	5.00	12	8
G500.02	500	2355	2	9.42	10.00	18	1
G500.04	500	5120	7	20.48	20.00	34	1
G1000.0025	1000	1272	0	2.54	2.50	8	80
G1000.005	1000	2496	0	4.99	5.00	11	7
G1000.01	1000	5064	2	10.13	10.00	23	1
G1000.02	1000	10107	6	20.21	20.00	34	1
U500.05	500	1282	0	5.13	5.00	16	13
U500.10	500	2355	1	9.42	10.00	20	2
U500.20	500	4549	4	18.20	20.00	30	1
U500.40	500	8793	8	35.17	40.00	56	1
U1000.05	1000	2394	0	4.79	5.00	11	23
U1000.10	1000	4696	0	9.39	10.00	19	3
U1000.20	1000	9339	4	18.68	20.00	39	1
U1000.40	1000	18015	9	36.03	40.00	58	1

The second class of instances are composed by geometric graphs, denoted by $Un.n\pi d^2$, where n is the number of vertices and $n\pi d^2$ is the expected average vertex degree. A geometric graph is built choosing n uniformly random points from the unit square. The edges are placed between vertices that have an Euclidean distance less or equal than d. Properties of the instances in this class are also presented in Table 2.3.

2.4.2 Walshaw's Dataset

The Walshaw's dataset (WALSHAW, 2000) is composed by several real world instances gathered by several authors. Walshaw also keeps a partition archive with the best known values (BKVs) for each instance and several imbalance restrictions ($\epsilon \in \{1.00, 1.01, 1.03, 1.05\}$). This dataset is widely used in literature so it is a good base for comparisons with other approaches.

The properties of the graphs and their sources are presented by Table 2.4. These instances come from several application domains (the source is according to from (KARYPIS; KUMAR, 1998) and (BENLIC; HAO, 2011a)) and we split them into three groups: small, medium and large instances.

2.4.3 Methodology for Experimental Result Evaluations

For our experimental result evaluations, we follow the same methodology of several recent works on the literature (SANDERS; SCHULZ, 2011; BENLIC; HAO, 2011b; BENLIC; HAO, 2011a; OSIPOV; SANDERS, 2010; BENLIC; HAO, 2010; MEYERHENKE; MONIEN; SAUERWALD, 2009).

On this dissertation, our objective is to produce high quality solutions when compared to the state-of-the-art. Some authors of the mentioned works compare themselves to others by counting how many partitions of a benchmark dataset they have improved, generated the same results or have worsened. Some authors also compute relative deviations of their results to their target values, the best known solutions at their time. The relative deviation is defined as

$$(\Theta(P)/\Theta(B) - 1) * 100, \tag{2.5}$$

with P being a partition to be compared to a base solution B of the same graph. The base solution is usually the best known for the graph. By computing the average relative deviation (ARD) of solutions from different instances, we can determine an average distance of each solution in the set of results to their base.

We will benchmark most of our experiments based on the best known solutions stored in the Walshaw's partition archive. The solutions from the archive were computed by several partitioners and most of them have publications available that detail their parameters but some do not, and the time required to compute these best solutions may be unknown. Furthermore, by computing the relative deviations from our method to the BKVs, we are comparing ourselves with several other methods and they are also unable to find all best known solutions by themselves.

The solutions from Walshaw's partition archive have been generated with the imbalance restriction $\forall i, |P_i| \leq \epsilon \lceil n/k \rceil$. However, we defined the problem by limiting the partition size by $\lfloor \epsilon n/k \rfloor$ (see Equation 2.2b). The latter restriction is tighter than the former, so all valid results for the latter are also valid for the former. This difference was performed after most of the experiments finished, so we have to consider it. The experimental results present in this dissertation are valid, but there is a possibility that they could be improved if the Walshaw's restriction was used.

Table 2.4: Instances of Walshaw (2000). The table shows the number of vertices (n); the number of edges (m); the minimum, average and maximum vertex degree (δ, $\bar{\delta}$, Δ); the number of connected components (C.C.) and the source of the instance. Here we divide the instances into three groups: small, medium and large.

Instance	n	m	δ	$\bar{\delta}$	Δ	C.C.	Source
add20	2395	7462	1	6,23	123	1	20-bit Adder
data	2851	15093	3	10,59	17	1	3D Finite Element Mesh
3elt	4720	13722	3	5,81	9	1	2D Nodal Graph
uk	4824	6837	1	2,83	3	1	2D Dual Graph
add32	4960	9462	1	3,82	31	1	32-bit Adder
bcsstk33	8738	291583	19	66,74	140	1	3D Stiffness Matrix
whitaker3	9800	28989	3	5,92	8	1	2D Nodal Graph
crack	10240	30380	3	5,93	9	1	2D Nodal Graph
wing_nodal	10937	75488	5	13,80	28	1	3D Nodal Graph
fe_4elt2	11143	32818	3	5,89	12	1	2D Finite Element Mesh
vibrobox	12328	165250	8	26,81	120	1	Sparse Matrix
bcsstk29	13992	302748	4	43,27	70	28	3D Stiffness Matrix
4elt	15606	45878	3	5,88	10	1	2D Nodal Graph
fe_sphere	16386	49152	4	6,00	6	1	3D Finite Element Mesh
cti	16840	48232	3	5,73	6	1	3D Semi-structured Graph
memplus	17758	54196	1	6,10	573	1	Memory Circuit
cs4	22499	43858	2	3,90	4	1	3D Nodal Graph
bcsstk30	28924	1007284	3	69,65	218	1	3D Stiffness Matrix
bcsstk31	35588	572914	1	32,20	188	2	3D Stiffness Matrix
fe_pwt	36519	144794	0	7,93	15	57	3D Finite Element Mesh
bcsstk32	44609	985046	1	44,16	215	1	3D Stiffness Matrix
fe_body	45087	163734	0	7,26	28	351	3D Finite Element Mesh
t60k	60005	89440	2	2,98	3	1	2D Dual Graph
wing	62032	121544	2	3,92	4	1	3D Dual Graph
brack2	62631	366559	3	11,71	32	1	3D Nodal Graph
finan512	74752	261120	2	6,99	54	1	Stochastic Programming Matrix
fe_tooth	78136	452591	3	11,58	39	1	3D Finite Element Mesh
fe_rotor	99617	662431	5	13,30	125	1	3D Finite Element Mesh
598a	110971	741934	5	13,37	26	1	3D Finite Element Mesh
fe_ocean	143437	409593	1	5,71	6	1	3D Dual Graph
144	144649	1074393	4	14,86	26	1	3D Finite Element Mesh
wave	156317	1059331	3	13,55	44	1	3D Finite Element Mesh
m14b	214765	1679018	4	15,64	40	1	3D Finite Element Mesh
auto	448695	3314611	4	14,77	37	1	3D Finite Element Mesh

3 ALGORITHMS FOR GRAPH PARTITIONING

This chapter presents relevant partitioning techniques along with a short review on some of the state-of-the-art graph partitioners. In order to capture the essence of the high quality partitioners currently proposed, and to justify our own choices, we study their components in the next sections.

3.1 Integer Programs

As a combinatorial problem, the graph partitioning is mostly solved heuristically due to the need of an acceptable computation time. State-of-the-art partitioners require up to two hours to give their best solutions for graphs with up to 500 thousand vertices (SANDERS; SCHULZ, 2011; BENLIC; HAO, 2011b; SANDERS; SCHULZ, 2010; OSIPOV; SANDERS, 2010). We will see that we can find optimal solutions and prove optimality only in small instances. For example, an integer linear program can solve, within a time limit of an hour, only graphs up to 500 vertices. Nevertheless, exact algorithms can help us define bounds on the objetive value and perhaps they can be useful for faster and better heuristics.

Boulle (2004) proposes four different integer programming models for the k-way graph partitioning problem, called A, B, C and D. There is also a simplification of the model B, called $B2$, specifically for the bipartitioning problem. Model A, is the base of all others which evolved by the rewriting of the restrictions as an attempt to improve performance of the MIP solver. Below, we detail models B, $B2$ and D.

The GPP is usually simpler when dealing with the 2-way case, so we start presenting model $B2$, that is used to compute bipartitions, expressed by equations 3.1a to 3.1e. This is an adaptation of the model proposed by Boulle, since the original model does not handle imbalance or weighted edges and vertices.

$$(B2) \quad \textbf{minimize} \quad \sum_{\{u,v\}\in E} f_{u,v}\omega(\{u,v\}) \tag{3.1a}$$

$$\textbf{subject to} \quad \sum_{u\in V} x_{u,1}c(u) \leq \left\lfloor \frac{\epsilon}{k} \sum_{v\in V} c(v) \right\rfloor \tag{3.1b}$$

$$f_{u,v} \geq +x_{u,1} - x_{v,1} \qquad \forall \{u,v\} \in E \tag{3.1c}$$

$$f_{u,v} \geq -x_{u,1} + x_{v,1} \qquad \forall \{u,v\} \in E \tag{3.1d}$$

$$x_{u,1}, f_{u,v} \in \mathbb{B} \qquad \forall u,v \in V \tag{3.1e}$$

Table 3.1: Sizes of integer programming formulations proposed by Boulle (2004) for the graph partitioning problem.

M.	Restrictions	Variables	Binary Variables	Non-zero coefficients
A	$n + 3mk + k$	$nk + mk$	nk	$2nk + 7mk$
B	$n + 2mk + k$	$nk + m$	nk	$2nk + 6mk$
B2	$2m + k$	$n + m$	n	$nk + 6m$
C	$n(1 + k + k\log k) + 2m\log k + k$	$n(k + \log k) + m$	$n\log k$	$n(\log k + 2k + 3k\log k) + 6m\log k$
D	$n(1 + 3(k - 1 - \log k)) + 2m\log k + 2k$	$n(k-1) + m + k$	$n\log k$	$n(2\log k + 8k - 8\log k) + k(k-1) + 6m\log k$

In model $B2$, binary variables $f_{u,v}$ are set to one when the edge $\{u, v\}$ is part of the cut, zero otherwise. Binary variables $x_{u,1}$ are set to one when the second part contains the vertex u, zero when the first part contains u. Equation 3.1b limits the weight of each part. Equations 3.1c and 3.1d guarantee that cutting edges are correctly accounted.

Model B (Equations 3.2a to 3.2f) is a specialization of model $B2$. The variables $x_{u,1}$ from model B are replaced by $x_{u,p}$, meaning that the part indexed by $p \in \{0, \ldots, k-1\}$ contains the vertex u if $x_{u,p} = 1$. Equation 3.2f ensures that a vertex is assigned to exactly one part of the partition.

$$(B) \quad \textbf{minimize} \quad \sum_{\{u,v\}\in E} f_{u,v}\omega(\{u,v\}) \tag{3.2a}$$

$$\textbf{subject to} \quad \sum_{u\in V} x_{u,p}c(u) \le \left\lfloor \frac{\epsilon}{k} \sum_{v\in V} c(v) \right\rfloor \qquad \forall p \tag{3.2b}$$

$$f_{u,v} \ge +x_{u,p} - x_{v,p} \qquad \forall \{u,v\} \in E; \forall p \tag{3.2c}$$

$$f_{u,v} \ge -x_{u,p} + x_{v,p} \qquad \forall \{u,v\} \in E; \forall p \tag{3.2d}$$

$$x_{u,p}, f_{u,v} \in \mathbb{B} \qquad \forall u, v \in V; \forall p \tag{3.2e}$$

$$\sum_{p} x_{u,p} = 1 \qquad \forall u \in V \tag{3.2f}$$

Besides these two, Boulle also formulated models C and D. Model C is an intermediate step from B to D. Model D is more compact than B when $m \ge 3n/2$, with a compactness factor of $\log k/k$ for the binary variables (BOULLE, 2004), where compactness factor stands for the number of binary variables of model D divided by the number of model B. The sizes of all models are presented in Table 3.1.

The basic idea to transform model B into a more compact form is to remove the excess of $x_{u,p}$ variables. Model B formulates the problem but it allows vertices to be contained in several parts, for example, vertex u is in part two and three when $x_{u,2} = x_{u,3} = 1$. Boulle proposes that instead of assigning vertices to parts, parts should be assigned to vertices.

This idea could be modeled using integer variables that represent the index of the part a vertex is assigned to. However, because of the balance restriction, it is required to compute the weight of each part and this cannot be modeled using integer

variables so, to overcome this issue, model D assigns parts to vertices by using the binary representation of the parts' indices. Each assignment is represented by a set of binary variables $b_{u,i}$ and each of these variables corresponds to the bit i of the binary representation of the part's index to which the vertex u is assigned to, with $0 \leq i \leq \lceil \log_2 k \rceil$. For example, if vertex $u = 3$ belongs to part $p = 4_{10} = 100_2$ with $k = 8$, we have $b_{3,0} = 0$, $b_{3,1} = 0$ and $b_{3,2} = 1$.

In order to compute the weight of the parts we need to understand the concept of bit masks. If we sum all variables that represent bit i, we have the number of vertices that belong to parts that bit i is non-zero. For example, for $k = 4$, the parts $2_{10} = 10_2$ and $3_{10} = 11_2$ have bit 1 non-zero and the sum $b_{0,1} + b_{1,1} + \cdots + b_{n-1,1}$ is the number of vertices that belong to parts 2 or 3. With this, we say that the index of some part masks the bit i if this index has its own i bit non-zero. Also, we say that some index p masks a bit mask a if all non-zero bits of a are also non-zero in p. For example, the indices $5_{10} = 101_2$, $7_{10} = 111_2$ and $13_{10} = 1101_2$ mask the bit mask $5_{10} = 101_2$.

With this two concepts of bit masks, Boulle demonstrates that it is possible to compute the weights of the parts based on the binary representation of the indices and a linearization of the variables indexed by these masks.

Model D is presented by the equations 3.3a to 3.3m. Variables $m_{u,a}$ $(0 \leq a \leq k)$ are set to one when the bit mask a is masked by the index of the part that the vertex u belongs. The weights of the vertices are not considered on model D, just as Boulle's. This does not affect the experiments, presented next, because we just used instances with vertices of unitary weight.

$$(D) \quad \textbf{minimize} \quad \sum_{\{u,v\} \in E} f_{u,v}\omega(\{u,v\}) \tag{3.3a}$$

subject to

Each vertex is assigned to one part

$$\sum_i b_{u,i} 2^i \leq k - 1 \qquad \forall u \in V \tag{3.3b}$$

Compute the weight of each part

$$\sum_p s_p \leq n \tag{3.3c}$$

$$\sum_{u \in V} b_{u,i} = \sum_{p | p \text{ masks bit } i} s_p \qquad \forall i \tag{3.3d}$$

$$\sum_{u \in V} m_{u,a} = \sum_{p | p \text{ masks the bitmask } a} s_p \qquad \forall a \ (a \neq 2^q) \tag{3.3e}$$

Maximum size of each part

$$s_p \leq \lfloor \epsilon n / k \rfloor \qquad \forall p \tag{3.3f}$$

Compute cut edges

$$f_{u,v} \geq +b_{u,i} - b_{v,i} \qquad \forall \{u,v\} \in E;\ \forall i \quad (3.3g)$$
$$f_{u,v} \geq -b_{u,i} + b_{v,i} \qquad \forall \{u,v\} \in E;\ \forall i \quad (3.3h)$$

Linearization of quadratic variables m ($m_{u,a} = b_{u,i} m_{u,b}$, $a = i + b$, $i = \lfloor \log_2 a \rfloor$, $i' = \lfloor \log_2 b \rfloor$, replace $m_{u,b}$ by $b_{u,i'}$ if $b = 2^q$)

$$m_{u,a} \leq b_{u,i} \qquad \forall u \in V;\ \forall a\ (a \neq 2^q) \quad (3.3i)$$
$$m_{u,a} \leq m_{u,b} \qquad \forall u \in V;\ \forall a\ (a \neq 2^q) \quad (3.3j)$$
$$m_{u,a} \geq b_{u,i} + m_{u,b} - 1 \qquad \forall u \in V;\ \forall a\ (a \neq 2^q) \quad (3.3k)$$

Decision variables

$$b_{u,i}, f_{u,v}, m_{u,a} \in \mathbb{B} \qquad \forall u, v \in V;\ \forall i;\ \forall a \quad (3.3l)$$
$$s_p \in \mathbb{N} \qquad \forall p \quad (3.3m)$$

We conducted some experiments in order to evaluate the performance of models B and D using a modern mixed integer program solver. We generated several random graphs with $m > 3n/2$ and $k \in \{2, 4, 6\}$ to test the compactness of model D compared to model B. The experiment was run on Machine A, with a time limit of one hour per test. The results can be seen in Figures 3.1(a) and 3.1(b). For the bipartitioning, model D has lower average times than model B. However, for both other partition sizes, the model B was consistently more effective than model D. We conclude, therefore, that model D is the best option for computing bipartitionings and model B, for k-partitions ($k > 2$).

3.2 Fundamental Heuristic Techniques

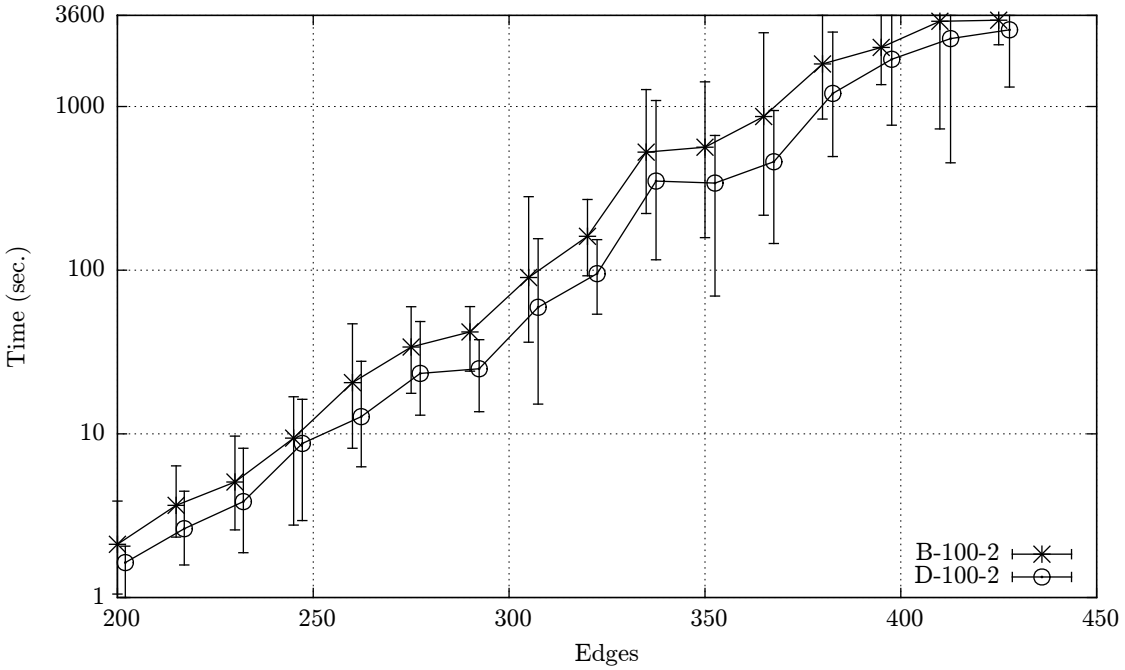
Most of heuristic based partitioners use at least two fundamental techniques: a constructive algorithm, which provides an initial solution for the partitioner or for one of its iterations; and a refinement algorithm, which is usually used as a local search operator in metaheuristics, for example, and intends to improve an existing solution. Both kinds of algorithms are discussed in the next sections but we start with basic notions on what heuristics are based.

3.2.1 The Basics of Heuristic Graph Partitioning

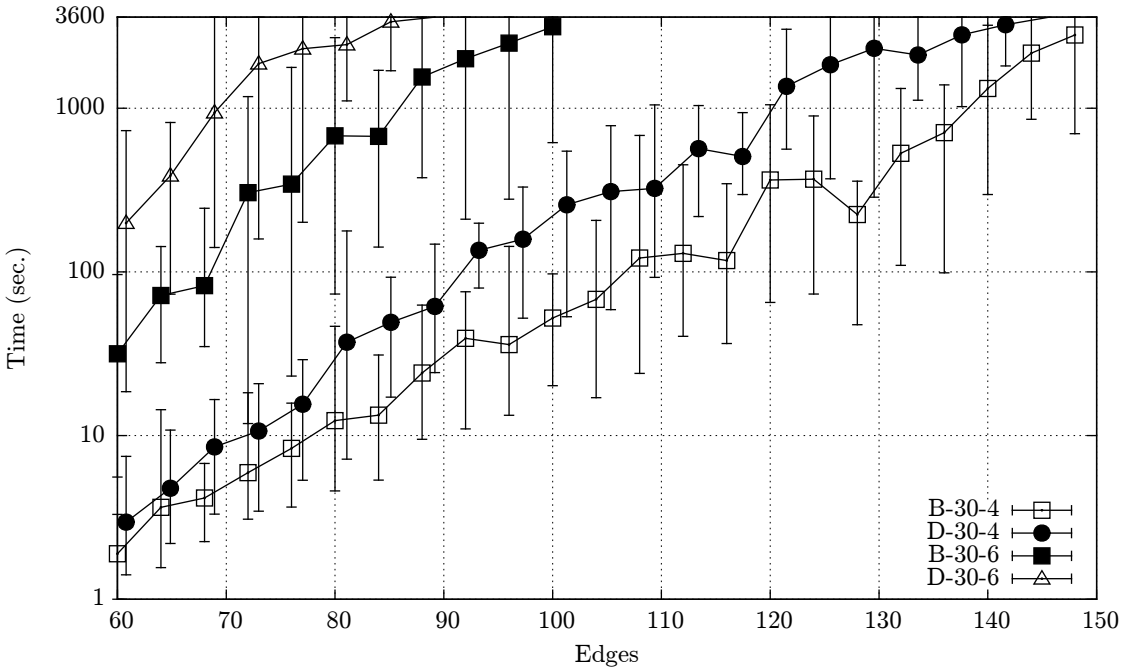
The most common graph partitioning heuristics are based on simple moves of vertices from one part to another. These moves are made one by one but the partition may become partially valid because, despite the partition size is correct, a move causes an increase in the weight of the target part and this may exceed the imbalance limit.

The excess of the imbalance limit is handled by the heuristic, but we can define that, for every partition P, a move of a vertex $v \in P_p$ to a part indexed by p' generates a new partition P', where $v \in P'_{p'}$. Then

$$g_P(v, p') = \Theta(P) - \Theta(P') \quad (3.4)$$

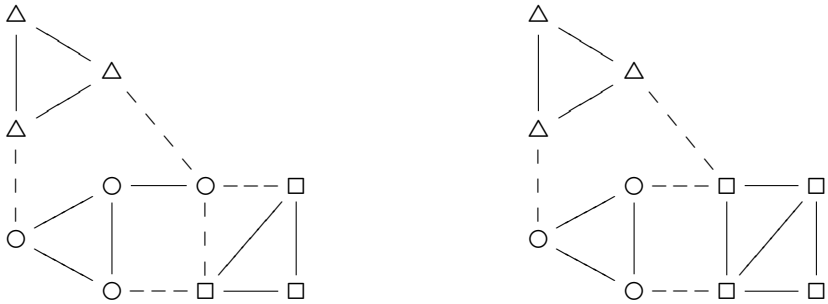


(a) Models B and D, 100 vertices, partitions of size 2.



(b) Models B and D, 30 vertices, partitions of size 4 and 6.

Figure 3.1: Comparison of integer programs B and D for the GPP using small random graphs as input. Each value is composed by the average computing time for ten different instances of approximately equal sizes and the minimum and maximum times as the whiskers. Time limit was one hour. The resulting points for model D tests are slightly moved to the right, so the whiskers of two tests do not overlap, but both models were tested with the same graphs.



(a) Cut size is 5 (b) Cut size is 4

Figure 3.2: Example of a partition refinement. Every vertex can have computed its gain of being moved to another part. The moved vertex in this example has gain −1 in figure (a) because the sum of the edge's weights connecting neighbors in the target part is two and in the source part is one. A gain of −1 means that the cut size will decrease by one if this move is performed.

is the gain in cut size, i.e., how much the cut size changed with this move. The knowledge of the gain of a move is useful to an heuristic because it can make moves based on how much the cut size will be affected with them.

The algorithm that computes the cut size of a graph partition can be trivially implemented to run in time $O(m)$. However, heuristics for the GPP require lots of moves and so this time complexity is too large. In the other hand, the change in cut size can be computed analyzing the edges incident to the moved vertex, since only their status may have changed. Therefore, the time is proportional to the degree of the moved vertex, $O(\Delta)$, and this operation, the gain of a move, can be defined by

$$g_P(v, p') = \sum\{\omega(\{v, u\}) \mid u \in N^{P_{p'}}(v)\} - \sum\{\omega(\{v, u\}) \mid u \in N^{\Pi_P(v)}(v)\}. \quad (3.5)$$

Besides the lower time complexity, function 3.5 has the advantage that it can be computed without actually moving the vertex to the new part, such as required by function 3.4.

An example of a move can be seen in Figure 3.2. We can compute the cut size of each partition (3.2(a) and 3.2(b)) and use equation 3.4 to compute the gain. Also, we can look locally to each vertex of the partition in Figure 3.2(a) and compute the gain to move it to each part using equation 3.5. Observe, through the moved vertex, that we just need to compute the weight of the edges connected to vertices on the source and the target part, while cut edges to other parts will remain the same.

Beyond that, function 3.5 is also useful for constructive heuristics, i.e., algorithms that generate new partitions. We will see in Section 3.2.2 that several heuristics use the gain function to select and assign the vertices in order of decreasing gain.

3.2.2 Constructive Algorithms

Several algorithms require an initial partition to start the optimization. It is very easy to generate a feasible solution for the partitioning problem. The trivial method is to generate a random solution: each vertex is assigned to a random part that does not violate its weight restriction. This does not produce good solutions because it

ignores the fact that real world graphs usually have a structure that can be taken into account. A good constructive algorithm can generate cuts of expected size up to a quarter of the expected size of a random partition for graphs with some structure, as we will see through experimental results in Section 3.2.2.8. Methods more robust than random construction use spectral information of the graph (Section 3.2.2.2) or a greedy function that grows the partition around seed vertices (Sections 3.2.2.4 to 3.2.2.7).

3.2.2.1 Random Partition

For some graph of reasonable size, the method of generating a random partition produces poor solutions on average, a lot greater than the best known cut. However, this algorithm is easy to implement and has time complexity $O(n)$. When used, this method is applied to small graphs, such as the ones of the coarsest level of a multilevel partitioning (Section 3.3) and associated with an iterative refinement algorithm to produce the initial partition.

3.2.2.2 Spectral Bisection (SB) Algorithm

The spectral bisection algorithm produces only graph bipartitionings using eigenvectors computed with Lanczos algorithm (PARLETT, 1987), which is an iterative algorithm and the number of required iterations depends on the desired accuracy. The spectral information obtained is then used to heuristically build an initial partition (KARYPIS; KUMAR, 1998).

3.2.2.3 Greedy Construction

Several greedy constructive algorithms for the GPP have been proposed and most of them have the same algorithmic structure. Algorithm 3.1 defines a procedure used by these constructive heuristics, which differ by their greedy functions.

Algorithm 3.1: Greedy constructive procedure

Input: Graph $G = (V, E, c, \omega)$, Desired size k of the partition
Output: A k-partition of V

```
1  P ← {P_0, ... P_{k-1}}
2  V' ← V
3  for p ∈ [0, k − 1] do
4      u ← random vertex from V'
5      P_p ← {u}
6      V' ← V' ∖ {u}
7  p ← 0
8  while |V'| > 0 do
9      b ← greedy_function(V', P, p, G)
10     P_p ← P_p ∪ {b}
11     V' ← V' ∖ {b}
12     p ← (p + 1) mod k
13 return P;
```

This algorithm assigns k different random vertices (called *seeds*), one to each part of an initially empty partition. The remaining *free* vertices are assigned to parts in circular order selecting vertices greedly. A total of $n - k$ iterations are performed. The greedy function is defined by each constructive heuristic that are going to be exposed. All constructive heuristics were adapted in this work from their original definitions in order to build a k-way initial partition, since they were only able to generate bipartitions. However, the adaptations still lack the ability of generating imbalanced partitions.

This greedy constructive method, however, has a problem of poor quality on bad chosen seeds. In order to soften this problem, one can run the constructive algorithm several times and use the best found solution as initial partition.

We also introduce here, the total gain of assigning a vertex, which is a function common to some greedy constructive heuristics. For a partially formed partition P, the total gain is defined as

$$t_P(v, p') = \sum_{p \neq p'} g_P(v, p), \tag{3.6}$$

that is the increment in cut size for adding a free vertex v to part p' of P. This special function is required because the handled vertex is free and none of its incident edges are accounted in cut size, so we need to compute the whole increment in the cut size after the assignment. Observe that function g_P will sum the weights of the edges connected solely to assigned neighbors of v. Free vertices are filtered at the first sum of equation 3.5 and, since v is also a non-assigned vertex, the second sum will always be zero.

3.2.2.4 Graph Growing Partitioning (GGP) Algorithm

The GGP algorithm (KARYPIS; KUMAR, 1998) is an heuristic for graph partition constructions based on a greedy function. The algorithm performs k iterations. Each iteration selects a random and free vertex and grows a region around it, following a depth first search procedure, until n/k vertices are added to that region. Each region forms a part of the initial partition.

3.2.2.5 Greedy GGP (GGGP) Algorithm

The GGGP algorithm (KARYPIS; KUMAR, 1998) replaces depth first search from GGP by the function that minimizes the total gain at each vertex assignment. Besides that, the vertices are assigned in a circular order of the parts, whereas the GGP algorithm grows a part as a whole at each iteration.

The GGGP algorithm has the same structure of Algorithm 3.1 and uses the greedy function presented in Algorithm 3.2. This function selects the free vertex of minimum total gain and ties are broken randomly.

This heuristic still has the same problem of poor quality on bad chosen seeds. However, experiments show that less iterations are required to achieve similar results as GGP algorithm (KARYPIS; KUMAR, 1998).

Algorithm 3.2: Greedy function for the GGGP algorithm (KARYPIS; KUMAR, 1998)

Input: A list V' of unassigned vertices, the current partition P, part index p where a new vertex should be assigned, graph $G = (V, E, c, \omega)$
Output: A vertex that should be assigned in part P_p

1 $m \leftarrow \min_{v \in V'}(t_P(v, p))$
2 $C \leftarrow \{v \in V' \mid t_P(v, p) = m\}$
3 **return** random vertex from C;

3.2.2.6 Min-max Greedy (MMG) Algorithm

Battiti and Bertossi (1999) found that the greedy function of GGGP algorithm produces lots of ties in real world instances. The MMG algorithm (BATTITI; BERTOSSI, 1999) is based on GGGP algorithm but introduces a new tie breaking rule: among the tied vertices, select one that maximizes the sum of the edge's weights internal to the target part of the assignment, i.e., maximizes $g_P(v, p)$, with p as the target part and v as one of the tied vertices (Algorithm 3.3).

Algorithm 3.3: Greedy function for the MMG algorithm (BATTITI; BERTOSSI, 1999)

Input: A list V' of unassigned vertices, the current partition P, part index p where a new vertex should be assigned, graph $G = (V, E, c, \omega)$
Output: A vertex that should be assigned in part P_p

1 $m \leftarrow \min_{v \in V'}(t_P(v, p))$
2 $C \leftarrow \{v \in V' \mid t_P(v, p) = m\}$
3 $M \leftarrow \max_{v \in C}(g_P(v, p))$
4 $C \leftarrow \{v \in C \mid g_P(v, p) = M\}$
5 **return** random vertex from C;

3.2.2.7 Differential Greedy (DG) Algorithm

The DG algorithm (BATTITI; BERTOSSI, 1997), inspired by the MMG algorithm, adapts the greedy function to be more aware of highly connected vertices. The DG algorithm populates the candidate list with vertices that minimize the difference between total gain and internal gain of the target part, i.e., minimize

$$t_P(v, p') - g_P(v, p'). \tag{3.7}$$

DG algorithm greedy function is presented in Algorithm 3.4.

This function has similar results as MMG algorithm but with a serious consequence: MMG algorithm filters vertices that do not have the minimum total gain — but still have a low total gain — and have a huge number of internal gain, while DG is aware of these highly connected vertices. The usefulness of this function over MMG algorithm's two step procedure can be observed in a simple example: when the assignment targets part p', vertex u has $t_P(u, p') = 1$ and $g_P(u, p') = 10$, vertex v has $t_P(v, p') = 5$ and $g_P(v, p') = 20$, vertex u has a difference of -9 and vertex

Algorithm 3.4: Greedy function for the DG algorithm (BATTITI; BERTOSSI, 1997)

Input: A list V' of unassigned vertices, the current partition P, part index p where a new vertex should be assigned, graph $G = (V, E, c, \omega)$
Output: A vertex that should be assigned in part P_p

1 $m \leftarrow \min_{v \in V'}(t_P(v,p) - g_P(v,p))$
2 $C \leftarrow \{v \in V' \mid t_P(v,p) - g_P(v,p) = m\}$
3 **return** random vertex from C;

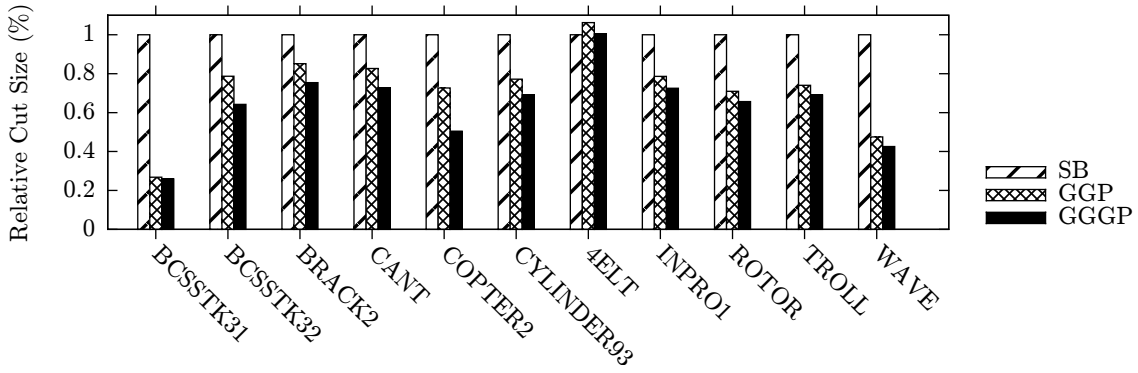


Figure 3.3: Comparison of constructive heuristics results obtained by Karypis and Kumar (1998) for several instances as a relative cut size of SB algorithm.

v has a difference of −15 between outside and inside assigned vertices; the MMG algorithm selects u instead of v because it increments cut size the less, while the DG algorithm selects v because the increment in cut size is not big enough close to the high connectivity of vertex v with their neighbors in part p' and which may become a problem in later iterations. In this sense, the DG algorithm is more aware of the graph structure and tends to choose more wisely the next assigned vertex than MMG algorithm.

3.2.2.8 Experimental Results

Figures 3.3 and 3.4 compare the previously described constructive heuristics. These results were obtained by Karypis and Kumar (1998) and Battiti and Bertossi (1997). Unfortunately, the authors of both papers use different instances, so we cannot directly compare all constructive heuristics, but both test the GGGP heuristic so it can be used as base of comparison.

The authors presented absolute values found for each instance. Instead, we composed the results as a relative cut size, since it allows us to compare better the related algorithms. We have two groups of algorithms to compare: the first, with the Karypis and Kumar data set, we normalized the results by the ones of the SB algorithm; the second, with the Battiti and Bertossi data set, we used the results of the random construction as base of normalization. The GGGP algorithm obtains the best results in all instances within the first group. Besides that, we can see that DG algorithm is better than GGGP for the tested instances and so we conclude that it is the highest quality method of constructive algorithms for the GPP.

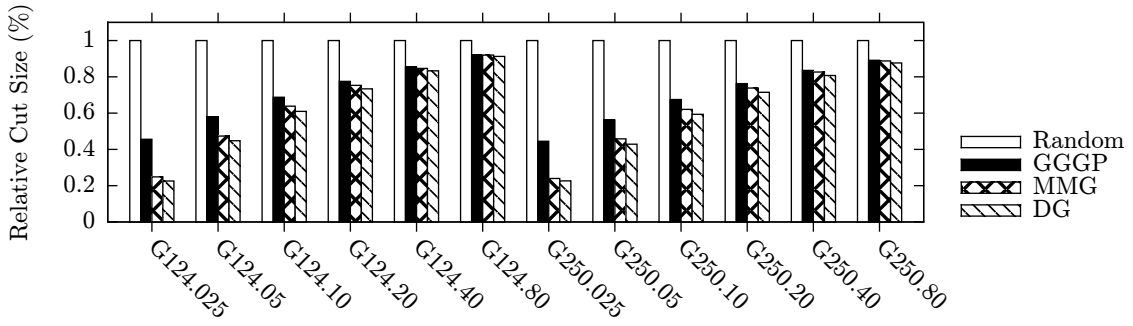


Figure 3.4: Comparison of constructive heuristics results obtained by Battiti and Bertossi (1997) for several instances as a relative cut size of random construction.

3.2.3 Refinement Algorithms

We can improve a perfectly balanced bipartition $\{A, B\}$ by swapping subsets $X \subset A$ and $Y \subset B$ (with $|X| = |Y|$) to opposite sides so that a reduction in cut size is achieved. However, finding these subsets is intractable (SCHLOEGEL; KARYPIS; KUMAR, 2003) but heuristics may be defined. One algorithm of great importance is the Kernighan-Lin local search (KERNIGHAN; LIN, 1970) that later was practically replaced by that of Fiduccia and Mattheyses (1982). Although the latter uses the same heuristic, it is a major improvement over the former because it allows the generation of imbalanced partitions and it reduced time complexity from $O(n^2)$ to $O(m)$ by using a bucket data structure.

We present these refinement algorithms in the next sections along with the tie-breaking rule for Fiduccia-Mattheyses algorithm proposed by Kim and Moon (2004). We also present some adaptations of these heuristics for the k-way GPP, since the methods were originally proposed just for the bipartition problem.

3.2.3.1 Kernighan-Lin (KL) Local Search

Kernighan and Lin (1970) proposed a refinement heuristic for the graph bipartitioning problem which is known as Kernighan-Lin local search. This algorithm performs several iterations, each one swapping two vertices from different parts. Since it uses swaps, this local search is not able to alter the imbalance of a partition. The selection order for the moves is based on the gain function (equation 3.5) using the vertices that produce the greatest gain after the swap, even if it is negative, i.e., increase cut size. Swapped vertices are locked to prevent them of being selected again. The algorithm stops when all vertices are locked and then it restores the best partition found during the search. The KL algorithm can be repeated, using this best partition as input, until no improvements are made.

Formally, the gain of swapping two vertices can be defined as

$$s(u, v) = g_u + g_v - 2\delta(u, v), \tag{3.8}$$

where g_u and g_v are the gains of both u and v, respectively, and

$$\delta(u, v) = \begin{cases} \omega(\{u, v\}) & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases} \tag{3.9}$$

The iterative improvement KL algorithm, as described above, is presented in Algorithm 3.5. Repetitions are usually limited by a constant, although the algorithm frequently converges in a few of them (SCHLOEGEL; KARYPIS; KUMAR, 2003). The (u, v) pair selection (line 7) requires $O(n^2)$ operations and so lines 6 to 11 define the time complexity of the algorithm, $O(n^3)$, since the external loop is iterated a fixed number of times at worst case.

On the other hand, Kernighan and Lin report that it is possible to decrease the selection complexity to linear time if, when computing the gains of u_i and v_i neighbors, three of the best known gains to each part are kept and used later for the selection. This decreases running time by 30% with small degradation in the power of optimization (KERNIGHAN; LIN, 1970).

Algorithm 3.5: Kernighan-Lin Heuristic (KERNIGHAN; LIN, 1970)

Input: Graph $G = (V, E)$; bipartition $P = \{P_0, P_1\}$ of V
Output: A bipartition of V

1 **repeat**
2 **foreach** $u \in P_0$ **do** $g_u \leftarrow g_P(u, 1)$
3 **foreach** $v \in P_1$ **do** $g_v \leftarrow g_P(v, 0)$
4 $L_0 \leftarrow \varnothing$
5 $L_1 \leftarrow \varnothing$
6 **for** $i = 1$ **until** $|V|/2 - 1$ **do**
7 Choose $u_i \in P_0 \setminus L_0$ e $v_i \in P_1 \setminus L_1$ such that $s(u_i, v_i)$ is maximum
8 $L_0 \leftarrow L_0 \cup \{u_i\}$
9 $L_1 \leftarrow L_1 \cup \{v_i\}$
10 **foreach** $u \in P_0 \setminus L_0$ **do** $g_u \leftarrow g_u + 2\delta(u, u_i) - 2\delta(u, v_i)$
11 **foreach** $v \in P_1 \setminus L_1$ **do** $g_v \leftarrow g_v + 2\delta(v, v_i) - 2\delta(v, u_i)$
12 Find a $p \in \{1, \ldots, |V|/2 - 1\}$ that maximizes $l = \sum_{i=1}^{p} s(u_i, v_i)$
13 **if** $l > 0$ **then**
14 **for** $i = 1$ **until** p **do**
15 $P_0 \leftarrow P_0 \cup \{v_i\} \setminus \{u_i\}$
16 $P_1 \leftarrow P_1 \cup \{u_i\} \setminus \{v_i\}$
17 **until** $l \leq 0$ **or** repetition limit exceeded
18 **return** P;

3.2.3.2 Fiduccia-Mattheyses (FM) Heuristic

The KL heuristic has two main problems, the time complexity ($O(n^2)$ in its most optimized version) and the inability to alter the imbalance in order to improve cut size. The FM heuristic, proposed by Fiduccia and Mattheyses (1982), decreases time complexity to $O(m)$, which is better for sparse graphs and equal to KL local search in the worst case, and deals with imbalanced partitions by moving single vertices instead of swapping two of them.

The time complexity is reduced using a data structure called bucket list. Essentially, the bucket list implements a priority queue where insert, delete, update and get minimum (or maximum) operations need time $O(1)$ at the expense of a bucket per priority.

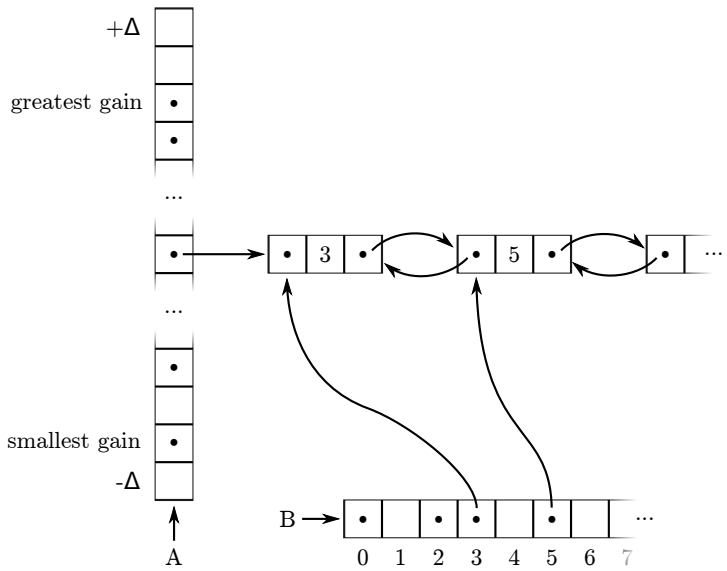


Figure 3.5: The bucket list data structure used by FM heuristic, similar to the one illustrated by Fiduccia and Mattheyses (1982). The B vector holds pointers to vertices with the same indices, stored in some bucket of the A vector. The greatest and smallest gains are kept for quick access of the correspondent buckets. These three structures allow constant time accesses, insertions and deletions of elements in any bucket.

In terms of the FM heuristic, this structure is used to handle the gains of the vertices, which are the priorities for the bucket list. Let Δ be the greatest vertex degree in the graph, there are $2\Delta + 1$ possible values that $g_P(v, p)$ may assume, for any v and p. This way, Fiduccia and Mattheyses propose that the bucket list should have exactly this amount of buckets. The bucket $i \in \{-\Delta, \ldots, 0, \ldots, \Delta\}$ holds the vertices that induce a gain of i when moved to another part. Figure 3.5 illustrates the data structures required for creating a bucket list. Since the absolute value of the gain is limited by the maximum degree, the use of a bucket list is efficient in this algorithm.

The bucket list initialization needs time $O(m)$. The computation of a vertex gain is made in $O(\Delta)$ and, when the gain is altered, the update of a vertex from one bucket to another is made in constant time. The vertex with the greatest gain can be selected also in constant time. The number of vertex moves is limited by $O(m)$ and so the total time complexity of this algorithm.

The FM algorithm uses a bucket list for each part. Each bucket list holds the vertices that can be moved to the part it represents. This leads to a problem of choice when imbalance is allowed: there are two lists that can lose a vertex to be moved. Some strategies were evaluated by Holtgrewe, Sanders and Schulz (2009):

- **Alternating** — The target partition is alternated at every move.
- **MaxLoad** — The part with the greatest weight loses a vertex at every move; a random vertex is selected when all parts have the same weight.
- **TopGain** — The selected vertex is the one that induces the greatest gain

among all possible moves. In case of a tie, a random vertex is selected among the tied ones. An exception is made when the selected vertex produces a part that exceeds the imbalance limit when moved to it. In this case the strategy MaxLoad is used.

- **TopGainMaxLoad** — The same as TopGain but in case of a tie, strategy MaxLoad is used.

Experimental results provided by the authors show that the strategy TopGain produces better results than the others.

3.2.3.3 Lock-gain (LG) Heuristic

Kim and Moon (2004) report that the lack of a good tie breaking rule on FM heuristic make the algorithm choose vertices badly. They propose an additional rule, breaking ties by the lock gain, defined as

$$l_P(v,p') = \sum\{\omega(\{v,u\}) \mid u \in N^{P_{P'}}(v), u \text{ is locked}\} - \sum\{\omega(\{v,u\}) \mid u \in N^{\Pi_P(v)}(v), u \text{ is locked}\}, \quad (3.10)$$

which is the gain in cut size considering only edges connected to locked vertices. Similar to KL heuristic, one can compute the gain of swapping two vertices by

$$s^l(u,v) = l_u + l_v - 2\delta(u,v), \quad (3.11)$$

where l_u and l_v are the lock gains for u and v, respectively, and $\delta(u,v)$ is the same as in Equation 3.9.

The authors also present experiments showing that the tie breaking rule LIFO (last in, first out), which selects the last vertex inserted in a bucket, is better than the rule FIFO (first in, first out; selection of the first inserted vertex) or the rule RANDOM (selection of a random vertex in the same bucket).

Algorithm 3.6, the LG algorithm, uses primarily the lock gain to order the vertex selection. Ties are broken by the free gain, the gain as in Equation 3.5. The LIFO rule is used as a secondary tie breaker.

The authors argue that it is preferable that the moves are made based more on unchangeable gain, since the cut size will be less affected after a number of iterations. It was experimentally demonstrated by Kim and Moon that this order of rule usage, the lock gain and then the free gain, is more effective than the other one.

This algorithm can be implemented using a bucket list that has been enlarged from $2\Delta + 1$ to $(2\Delta + 1)^2$ to be able to store any combination of the possible values for both gains. This way, each vertex v that can be moved to part p with gains l_v and g_v is in the bucket indexed by

$$l_v(2\Delta + 1) + g_v + \lfloor (2\Delta + 1)^2/2 \rfloor \quad (3.12)$$

of the bucket list. Using the same data structures as in FM's, the LG algorithm has time complexity $O(n + m)$ (KIM; MOON, 2004).

Algorithm 3.6: Lock-Gain Refinement Heuristic (KIM; MOON, 2004)

Input: Graph $G = (V, E)$; bipartition $P = \{P_0, P_1\}$ of V
Output: A bipartition of V

```
1  repeat
2      foreach u ∈ P_0 do  g_u ← g_P(u, 1); l_u ← 0
3      foreach v ∈ P_1 do  g_v ← g_P(v, 0); l_v ← 0
4      L_0 ← ∅
5      L_1 ← ∅
6      for i = 1 until |V|/2 − 1 do
7          C_0 ← P_0 ∖ L_0
8          C_1 ← P_1 ∖ L_1
9          m ← max_{u∈C_0, v∈C_1}(s^l(u, v))
10         C_0 ← {u ∈ C_0 | s^l(u, v) = m}
11         C_1 ← {v ∈ C_1 | s^l(u, v) = m}
12         Choose u_i ∈ C_0 e v_i ∈ C_1 such that s(u_i, v_i) is maximum
13         L_0 ← L_0 ∪ {u_i}
14         L_1 ← L_1 ∪ {v_i}
15         foreach u ∈ P_0 ∖ L_0 do
16             l_u ← l_u + 2δ(u, u_i) − 2δ(u, v_i)
17             g_u ← g_u + 2δ(u, u_i) − 2δ(u, v_i)
18         foreach v ∈ P_1 ∖ L_1 do
19             l_v ← l_v + 2δ(v, v_i) − 2δ(v, u_i)
20             g_v ← g_v + 2δ(v, v_i) − 2δ(v, u_i)
21     Find a p ∈ {1, ..., |V|/2 − 1} that maximizes l = Σ_{i=1}^{p} s_P(u_i, v_i)
22     if l > 0 then
23         for i = 1 until p do
24             P_0 ← P_0 ∪ {v_i} ∖ {u_i}
25             P_1 ← P_1 ∪ {u_i} ∖ {v_i}
26 until l ≤ 0 or repetition limit exceeded
27 return P
```

3.2.3.4 k-way Fiduccia-Mattheyses Heuristic

Since the quality of the solutions tend to decay with the increase in partition size with recursive bisection, refinement heuristics capable of natively refining k-way partitions are required for better results. Osipov and Sanders (2010) define a k-way variant of FM algorithm that uses priority queues for storing and ordering the vertices and their gains.

Each part, has a corresponding priority queue holding the vertices that can be moved to it. Therefore a vertex may be contained in several structures, since it may be moved to several parts. Vertices are able to be moved to any part in any iteration. Similar to the TopGain strategy (Section 3.2.3.2), the vertex of greatest gain in all eligible priority queues is selected. A priority queue is eligible if it is non-empty and the part it represents does not reach the maximum allowed weight. Local search is stopped when no eligible priority queues remain.

Osipov and Sanders use priority queues for handling the gains, which may be also implemented by Fiduccia and Mattheyses's bucket list for every part.

3.2.3.5 Improvement of Heuristics

The presented heuristics move all vertices once to some other part than their initial ones before stopping the local search. However, it can be experimentally demonstrated that most of the last moves of a refinement iteration do not help finding a new best solution and the cut size in this state is in fact far from the best solution.

In order to decrease running time, some authors restrict the search by limiting the number of moves to a fixed constant (BUI; MOON, 1996) or a number of continuous failed iterations, i.e., iterations with no improvements over the last one (WALSHAW; CROSS, 2007). Osipov and Sanders (2010) experiment a more adaptive rule for this stopping criterion: the local search stops if some of the last moves indicate that the search will hardly find a new best solution. This criterion is based on a random walk model. It is unlikely that the local search will produce a better cut if

$$p\mu^2 > \alpha\sigma^2 + \beta, \tag{3.13}$$

for some adjustment parameters α and β, where μ is the average gain since the last improvement and σ^2 is the variance of the previous p steps. This is a heuristic rule and the authors mention that it yields about 1% worse cut sizes, compared to the same algorithm without it, but an order of magnitude faster running times.

Another limitation of KL based heuristics is the restriction of moving each vertex at most once. This ensures faster running times but it may decrease the quality of solutions. Another approach, presented by Galinier, Boujbel and Fernandes (2011), locks vertices only for some iterations. The search stops, when all vertices are locked or the number of iterations reaches a limit. This principle is also used in Tabu Search (GLOVER, 1989), where the locked vertices are kept in a tabu list, and the number of iterations they remain there is called tabu tenure. This modification of the FM algorithm we call Tabu based Fiduccia-Mattheyses (TFM).

Galinier, Boujbel and Fernandes use a rather complex tenure function, but give no specific details of the reasoning behind this choice. A vertex moved in iteration i is locked for the next $\tau(i)$ iterations. The function $\tau(i)$ depends on a parameter `maxT` that limits the maximum number of iterations a vertex will be tabu. Given interval points $x_1 = 1$, $x_{j+1} = x_j + \texttt{maxT}\ b_j/2$, $\tau(i) = \texttt{maxT} \times b_j$ for $i \in [x_j, x_{j+1})$, where $(b_j)_{j\in\{1,\dots,15\}} = (1,2,1,4,1,2,1,8,1,2,1,4,1,2,1)$. The sequence b continues by repeating the first 15 values. For example, if `maxT` $= 40$, we have $x = (1, 20, 60, 80, 160, \dots)$ and $\tau = (5, 10, 5, 20, 5, \dots)$, and a vertex that enters the tabu list at iteration $i = 25$ will be locked for $\tau(25) = 10$ iterations. This example can be seen in Figure 3.6. For any `maxT` used, the function has the same form, just scaled differently. Galinier has demonstrated the usefulness of allowing vertices move more than once in the KL local search, and of this particular tenure function.

Some experiments of ours show that the use of a tabu list with this tenure function is very effective on average compared to simpler functions. It is mentioned by (GLOVER; LAGUNA, 1997) that a static tenure function or a random dynamic one may not be the best strategy. A systematic dynamic tenure, as it is, may be viewed as an example of intensification and diversification used to escape local minima in Tabu searches.

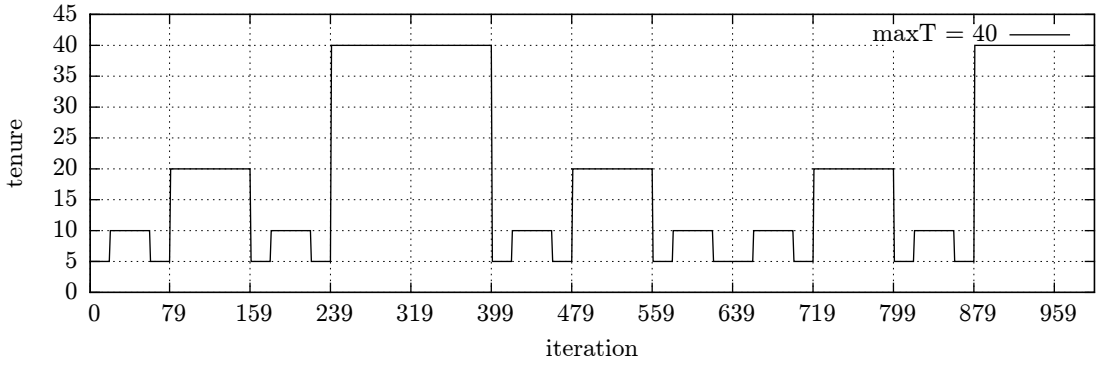


Figure 3.6: An example for the tenure function used by Galinier, Boujbel and Fernandes (2011), here using `maxT` = 40. Observe that this is a periodic function, in this case with a period of 640 iterations.

3.3 Multilevel Technique

We do know by experience that small graphs, even weighted ones, are easy instances for most partitioners, which most of the times are able to find good solutions. Hendrickson and Leland (1995) and Bui and Jones (1993) independently researched a technique to simplify instances so they become small enough for the partitioners find these good solutions. This technique became known as *multilevel graph partitioning*.

Walshaw (2008) presents this technique in a generic form and argues that it can be used to improve the performance of metaheuristics, as long as the required operators may be defined. This generic multilevel technique is implemented by Algorithm 3.7.

Algorithm 3.7: Multilevel refinement technique (WALSHAW, 2008)

Input: An instance P_0 for the problem
Output: A solution for the problem

```
1  l ← 0
2  while P_l too big do
3      P_{l+1} ←coarsen(P_l)
4      l ← l + 1
5  C_l ←initialize(P_l)
6  while l > 0 do
7      l ← l − 1
8      C_l^0 ←uncoarsen(C_{l+1}, P_l)
9      C_l ←refine(C_l^0, P_l)
10 return C_0;
```

A multilevel algorithm is composed of four operators: coarsening, initialization, uncoarsening, and refinement. The coarsening operator defines how an instance is reduced to a simpler one. The initialization builds a solution for the coarsest version of the instance. Uncoarsening transforms the coarse version in its previous one, i.e., the version before being coarsened. Finally, the refinement optimizes the solution of any coarse level.

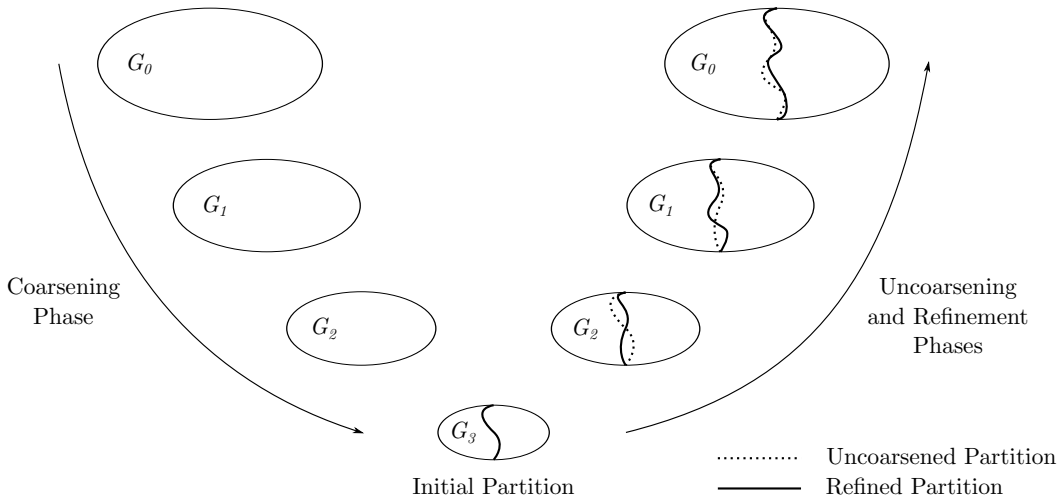


Figure 3.7: Basic scheme of multilevel graph partitioning (SCHLOEGEL; KARYPIS; KUMAR, 2003).

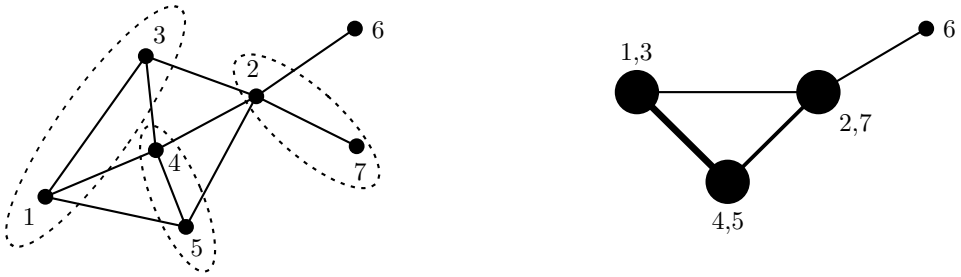
Figure 3.7 illustrates the basic scheme of multilevel graph partitioning. The next sections detail the operators for the multilevel graph partitioning proposed in the literature along with some enhancements for the basic scheme that can improve solution quality.

3.3.1 Graph Coarsening

Graph coarsening can be done in several ways. Hendrickson and Leland (1995) proposed to contract edges at each level. An edge $\{u, v\}$ is contracted by replacing the vertices incident to it with a new vertex w, with weight $c(w) = c(u) + c(v)$ (for an unweighted graph, $\forall u \in V, c(u) = 1$ at the original graph). All edges $\{u, r\}$ and $\{v, s\}$ are replaced by edges $\{w, r\}$ and $\{w, s\}$, the edge $\{u, v\}$ is removed and the weights of the new edges correspond to the old edges' weights or the sum of two edges, in case u and v share a neighbor $r = s$.

The algorithm of Hendrickson and Leland contracts all the edges of a maximum matching at each level. Using this procedure, the number of coarse levels is $O(\log_2 n)$. However, finding an optimal maximum matching requires time $O(n^{2.376})$ and space $O(n^2)$ (MUCHA; SANKOWSKI, 2004). This is impracticable for the multilevel technique since real world graphs are too large for these requirements. This way, approximation or heuristic algorithms are used to order the edge selection for contraction. Some of the studied algorithms are:

- **HEM** (*Heavy Edge Matching*) – given a list of vertices in a random order, this algorithm matches every unmatched vertex u with its first unmatched neighbor v, if existent, such as the weight of the edge $\{u, v\}$ is maximum among all incident edges to u (BENLIC; HAO, 2011a). This method has time complexity $O(m)$ (KARYPIS; KUMAR, 1998) and has no guarantees of quality.
- **SHEM** (*Sorted HEM*) – used in the METIS partitioner (KARYPIS; KUMAR,



(a) level i (b) level $i+1$

Figure 3.8: String Aggregation scheme (CHEVALIER; SAFRO, 2009). The dotted ellipses (left graph) represent pairs of vertices that shall be contracted on the coarser level (right graph).

1995), this algorithm is the same as HEM except for the initial order of the vertex list: they are arranged in a non-decreasing order of degrees. Vertices with the same degree are ordered randomly. It has better results but the complexity and quality guarantees are the same as HEM.

- **GEM** (*Greedy Edge Matching*) – the edges are ordered in a non-increasing order of edge weight and read from the first to the last, one by one, i.e., it performs a scan of this list. An edge is added to the matching if both of its vertices are unmatched. The time complexity of this algorithm is $O(m \log n)$ in general, linear with integer weights and it guarantees a 1/2-approximation of the optimal value (MAUE; SANDERS, 2007).

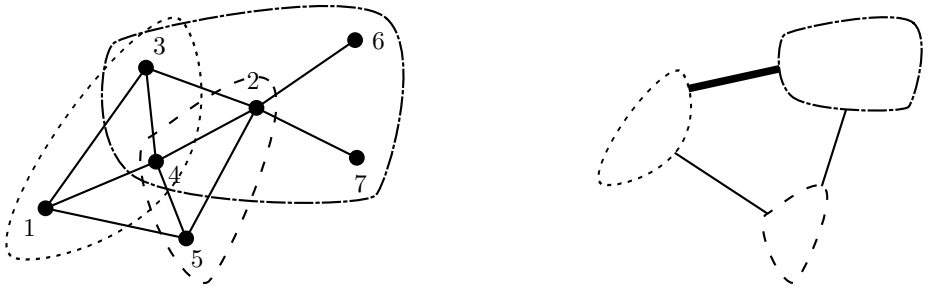
- **GPA** (*Global Path Algorithm*) – proposed by (MAUE; SANDERS, 2007), this algorithm is similar to GEM because it also performs a scan of the edges' list using this non-increasing order of edge weights but, instead of building the matching directly, it first builds a collections of paths and cycles of odd length. With them, the matching is computed using dynamic programming. It also guarantees a 1/2-approximation and time complexity $O(m \log n)$ in general and linear for integer weights. Despite the approximation guarantee, this method has, empirically, considerably better results.

These methods are part of a contraction scheme known as *Strict Aggregation* (SAG, Figure 3.8). They are simple and adopted by several multilevel partitioners.

Strictly speaking, the described maximum matching algorithms process the edge list ordered by a rank function. This function can be any one with an image that may form a partially ordered set, just as the function that returns the weight of an edge ($\omega(\{u,v\})$) or a function that return a random number for any edge. In fact, other functions are more interesting for the matching inside a multilevel graph partitioner because of the balance restriction that competes with the cut size. Several functions have been proposed by Holtgrewe, Sanders and Schulz (2009):

$$\text{expansion}(\{u,v\}) = \frac{\omega(\{u,v\})}{c(u)+c(v)}; \tag{3.14}$$

$$\text{expansion}^*(\{u,v\}) = \frac{\omega(\{u,v\})}{c(u)c(v)}; \tag{3.15}$$



(a) level i (b) level $i+1$

Figure 3.9: Weighted Aggregation scheme (CHEVALIER; SAFRO, 2009). The three dotted curves in the left graph represent vertex subsets that constitute aggregations in the coarser level (right graph). The vertices part of several subsets and are split into fractions, added to the aggregations in the coarser level.

$$\text{expansion}^{*2}(\{u,v\}) = \frac{\omega(\{u,v\})^2}{c(u)c(v)}; \tag{3.16}$$

$$\text{innerOuter}(\{u,v\}) = \frac{\omega(\{u,v\})}{\text{out}(v) + \text{out}(u) - 2\omega(\{u,v\})}, \tag{3.17}$$

where $\text{out}(v) = \sum_{u \in N(v)} \omega(\{v,u\})$. These functions are based on the principle that it is good to contract heavy edges, so the cut size do not get incremented too much. Moreover, it is also good to contract edges with light vertices, so every coarse level has a certain homogeneity on its vertices' weights. Homogeneity is interesting because it gets easier to move vertices from one part to another without breaking the imbalance rule.

Another contraction scheme, known as *Weighted Aggregation* (WAG), allows vertices to be split into fractions (Figure 3.9) and every fraction belongs to different aggregations, which are the new vertices at the coarser level. This implies that the vertex set is covered by, possibly small, subsets that have a non-empty intersection.

The WAG scheme is interesting because express a predisposition of relationship among the vertices. These predispositions are going to accumulate in the coarsest level, suggesting that the aggregations are connected in finest levels. The SAG scheme, on the other hand, can conflict the local and global decisions.

Chevalier and Safro (2009) evaluate which contraction scheme gives the best results. Tests were made using instances of the Walshaw dataset (WALSHAW, 2000) to compare the results of the Scotch partitioner package (PELLEGRINI, 2007b) (that uses the HEM algorithm by default in the coarsening phase) and an implementation of the WAG scheme (SAFRO; RON; BRANDT, 2009) replacing the default scheme of Scotch. In both cases, the optimizations of the coarsening and uncoarsening phases were turned off and the WAG scheme was implemented in a simple version, although the authors mention several optimizations for it.

Results show that the WAG scheme contributes to cut sizes 15% better than the ones of the partitioner using the SAG scheme. Even without optimizations, the relative deviation of the partitioning using the WAG scheme was about 0.6% of the best known value (at that time) for the tested instances and always produces, in average, better partitions than the one with a SAG scheme. These results lead the

authors to conclude that the WAG scheme is the most appropriate. Unfortunately, other SAG methods, more efficient that the HEM algorithm, were not evaluated (in Holtgrewe, Sanders and Schulz (2009) three SAG methods were compared: SHEM, GEM and GPA; and the last one was considered to have the best cost/benefit of all but the author did not tested any WAG method).

3.3.2 Initial Partition

When the resulting graph of the coarsening phase reaches a small enough size, an initial partition must be defined. A simple criterion for the coarsening phase is to stop when only k vertices remain, and require that each is assigned to a different part. However, without any special care at the coarsening phase, the vertices most certainly will be too much heterogeneous in their weights and so it may be impossible to build a feasible solution respecting the imbalance constraints.

Several methods were already proposed to solve this problem, called coarsening homogeneity (WALSHAW, 2008). One of them is to coarsen but stopping with a number of vertices much larger than the partition size (HENDRICKSON; LELAND, 1995). Another method consists in including balance techniques to the refinement phase, so the initial partition can be infeasible but the final one is not (KARYPIS; KUMAR, 1998). Also, it is possible to restrain the space of allowed vertices for contraction. For example, Osipov and Sanders (2010) do not allow a vertex to be part of a contraction if its weight is larger than $1.5n/(20k)$.

After the coarsening, an initial partition is obtained by some algorithm, e.g. one of the constructive algorithms of Section 3.2.2.

3.3.3 Graph Uncoarsening

This phase undoes the coarsening level by level. The usual coarsening methods use one of strict aggregation, so the uncoarsening is simple and just the opposite of this operation: if two or more vertices were contracted into one, it is enough to remove this new vertex and restore the original ones and their edges.

On the other hand, if a weight aggregation method is used in the coarsening phase, the extension is more complex. When contracting an edge of some level, the new vertex may join parts of a vertex from the original graph and this fractions are summed up. When the uncoarsening is made, the knowledge of the fractions of the immediately higher level is missing. Thus, is required to interpolate the values based on the fractions of their neighbors (CHEVALIER; SAFRO, 2009).

3.3.4 Partition Refinement

The refinement is done using an algorithm such as some of the ones presented in Section 3.2.3. The Fiduccia-Mattheyses heuristic is the most often applied by multilevel partitioners, but any method that is able to refine an existing solution can be used. In this case, it is required that the refinement algorithm is capable of dealing with graphs weight on both edges and vertices, since the coarse graphs have these properties.

3.3.5 Multilevel Enhancements

Besides the mentioned coarsening homogeneity, other enhancements can be applied to the multilevel partitioning process. We present two in the next sections: imbalance relaxation and recoarsening.

3.3.5.1 Imbalance Relaxation

Imbalance relaxation allows greater imbalances at the coarsest levels, so the refinement algorithm is able to move heavy vertices to other parts, tightening this relaxation as the uncoarsening proceeds.

There may be cases that the coarsening phase results in a coarse graph that will inhibit the finding of the minimum cut, even after all uncoarsenings and refinements, because of the imbalance constraint, that prevents large vertices to be moved from one part to another. For example, suppose that a vertex v of the coarsest graph must be assigned to the part P_1 in order that the cut of the partition $P = \{P_0, P_1\}$ is minimum. A coarsening phase may produce a graph such that the constructive algorithm, building the initial partition, is forced to assign vertex v to part P_0. If this vertex and the vertices it represents are too large to be moved to part P_1 in the first refinements without breaking the imbalance rule, it is possible that the later refinements get stuck in a local minimum, unable to move all vertices that belong to v to part P_1. Therefore, the minimum cut is never found because the imbalance constraint prevented the move of a large vertex in the first moves.

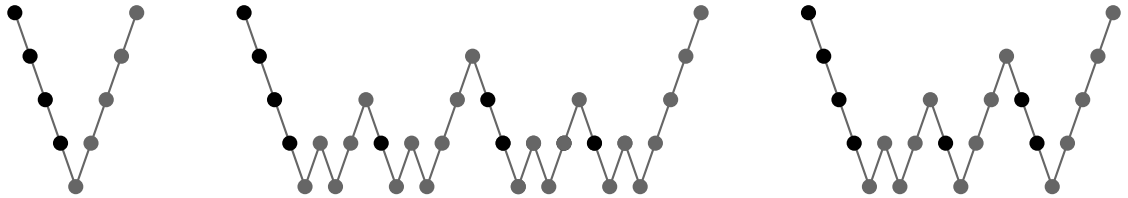
Walshaw and Cross (2000) studied the imbalance relaxation and report that this adjustment helps for better results decreasing cut size between 1% and 3%.

3.3.5.2 Recoarsening

Another enhancement applied to multilevel partitioners is the recoarsening. Very often the ranks that order the edges for selection get tied and a random edge is selected. This may contract edges that lead to poor partitions, and lock some moves in the coarsest levels. With this random element inside coarsening phase, two coarsenings of the same graph may lead to different results.

The recontraction can basically happen in two ways: the first one, called external recoarsening, considers the multilevel partitioner as a black box, repeating the hole partitioning several times until it reaches some limit or no better solutions are found. The second one, called internal recoarsening, coarsens and uncoarsens the graphs inside the process in order to obtain different solutions with less effort. In both cases, the best found solution is restored at the end of the process.

An example of partitioner that use the internal recoarsening technique is the n-level algorithm (OSIPOV; SANDERS, 2010), which uses what the so-called Trial Trees. For each coarsening level, the algorithm makes two attempts of coarsening, building two coarse graphs in the lower level. This process is recursive, so the coarsest levels have much more attempts than the finest ones. This technique is similar to that used in the Karger-Stein randomized algorithm for finding the minimum cut of an undirected graph (KARGER; STEIN, 1996). The reasoning behind this ap-



(a) V-cycle (b) W-cycle (c) F-cycle

Figure 3.10: Recoarsening strategies (SANDERS; SCHULZ, 2010). Dark dots represent independent trials of coarsening.

proach is the increase in probability of making a bad choice as the graph decreases in size. Since the finest levels have a low probability picking the wrong edge for contraction, fewer attempts are required and so execution time is reduced.

In Sanders and Schulz (2010) a recoarsening is defined as a global search strategy. A strategy without any recoarsening, i.e., the multilevel partitioner coarsens the graph up to the lowest level and uncoarsens back to the top is called V-cycle. The name reflects that the coarsening movement is similar to a "V" in the tree, that goes straight to the bottom and then straight to the top (see Figure 3.10). A strategy that executes two independent attempts for each level, as long as a random element is present in edge selection for contraction, is called W-cycle. Sanders and Schulz propose one more strategy, called F-cycle, that is similar to W-cycle but the global number of attempts for each level is limited by two.

A proposed enhancement to speed up the partitioners that use these strategies is to not recoarse at every level but every couple or more. To improve the quality of the solutions, the edges part of the best known cut are not allowed to be contracted.

3.4 Partitioning Approaches

This section presents some relevant and state-of-the-art graph partitioners as examples of how the subprocedures of a partitioning algorithm connect together and how they must be configured. The following partitioners are important because they advanced the state-of-the-art.

3.4.1 KaPPa

Proposed by Holtgrewe, Sanders and Schulz (2009), the *Karlsruhe Parallel Partitioner* (KaPPa), is a k-way multilevel graph partitioner. It had important results but it was later surpassed by KaSPar and KaFFPa, which follow it in this line of evolution. However, it introduced some interesting optimizations for the multilevel scheme and some experiments that help tuning some of the partitioner parameters.

As its name reveals, this is a parallel partitioner but, since parallel algorithms are not the purpose of this work, we will not detail the parallelization. However, a brief description of general function is given below.

As a multilevel partitioner, KaPPa has the three described phases of Section 3.3.

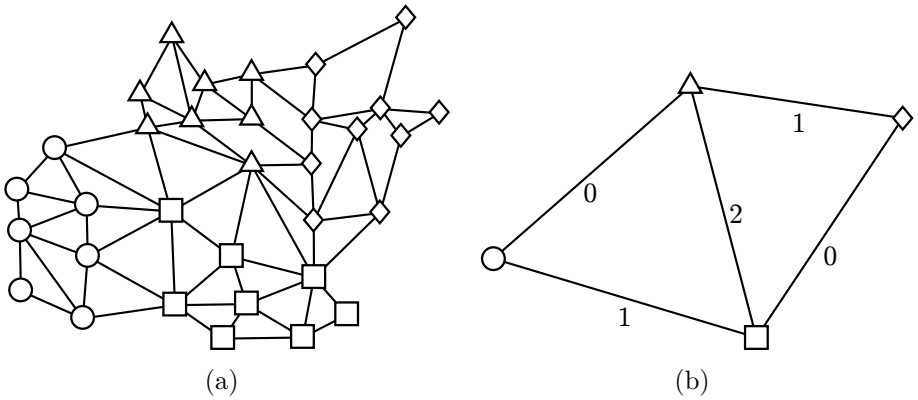


Figure 3.11: A quotient graph example (HOLTGREWE; SANDERS; SCHULZ, 2009). The graph on the left is partitioned in four parts. On the right graph, its correspondent quotient graph. The numbers of the edges in the quotient graph are added after an edge coloring is performed. Edges with the same number (color) form a matching and can be processed in parallel.

At coarsening phase, a combination of local matching and parallel matching is used to increase the speedup of the algorithm. This phase starts with a fast initial partitioning, just to distribute one part for each processing unit. The algorithm proceeds with a matching of the edges internal to each part, using the sequential algorithm GPA of Section 3.3.1. Then the *gap graph* is considered for a parallel matching. This graph contains the edges $\{u, v\}$ that have vertices in distinct parts and weight $\omega(\{u, v\}) \geq \omega(\{u, r\}) + \omega(\{v, s\})$, with $\{u, r\}$ and $\{v, s\}$ being the heaviest outgoing local edges of u and v. The matching algorithms select edges for contraction ordered by the rank function expansion^{*2} (Equation 3.16).

The coarsening phase ends when a coarse graph has few than $\max\{20, n/(\alpha k^2)\}$ vertices, for some parameter α. The coarsest graph is then partitioned using the Scotch partitioner package (PELLEGRINI, 2007b). Several initial partitioning trials are made in parallel because the coarsest graph is replicated to all processing units. The best solution found among all trials is used as initial partition.

At uncoarsening and refinement phases, each processing unit handles a single part of the partition. If the vertices could be moved to any part, there should be lots of synchronization restrictions that could overload the communication and decrease the algorithm's parallelization degree. In order to decrease synchronization requirements, the authors propose the use of a scheme to exchange vertices among processing units. This scheme requires a structure called quotient graph. This graph have its vertices representing parts of the coarse graph and edges are inserted if there exist edges connecting vertices in different parts of the coarse graph. See Figure 3.11 as an example. This quotient graph is computed at every level of KaPPa.

Given a matching M_Q of Q, two processing units synchronize to refine the graph relative to its two assigned parts only if the edge that connects them in Q is in M_Q. In other words, at start, vertices can only be swapped among two parts. Obviously, this does not lead to the best global cut because all combination of part's pairs must be exploited. This is solved used an edge coloring of Q. An edge coloring assigns colors (or numbers) to edges such as no pair of edges incident to the same vertex

have the same color. In Q, each color corresponds to a matching. Each matched edge can be processed in parallel, refining the border between the two incident parts. The colors are processed sequentially to alternate between the pairs of parts that can be refined.

3.4.2 KaSPar

The KaSPar acronym comes from *Karlsruhe Sequential Partitioner*, proposed by Osipov and Sanders (2010) from the same group that introduced the KaPPa, and is a multilevel partitioner. KaSPar (Algorithm 3.8) contracts a single edge at each coarse level, and is also known as an *n-level algorithm*. The edge contraction order is also ruled by the rank function expansion* (Equation 3.15). Besides that, no vertex weighting more than $1.5n/20k$ is allowed to be part of a contraction.

Algorithm 3.8: n-level algorithm (OSIPOV; SANDERS, 2010)

Input: Graph $G = (V, E, c, \omega)$, desired size k of the partition, imbalance limit ϵ
Output: A k-partition of V

1 **if** G is small enough **then return** `initial_partition`(G, ϵ);
2 pick the edge $e = u, v$ with the highest rank;
3 contract e; $P \leftarrow n$-level(G, ϵ); uncontract e;
4 `activate`(u); `activate`(v); `local_search()`;
5 **return** P;

The coarsening phase ends when a coarse graph have less than $20k$ vertices, no vertex is eligible for contraction or there are less edges than vertices, which may happen when the graph has several components. The initial partition is the best among several executions ($100/\log_2 k$ in KaSPar's slowest and highest quality configuration) of the Scotch partitioner (PELLEGRINI, 2007b).

The refinement is based on the Fiduccia-Mattheyses heuristic. Just the vertices of an extended edge are the initial candidates to be moved to other part. This is the definition of the `activate` functions in the algorithm. Only the active vertices have their gains computed using the function $g_P(v, p)$ (Equation 3.5) and each part has its own priority queue with vertices that can be moved to it. The local search is then similar to the k-way FM, detailed in Section 3.2.3.4, but every moved vertex is deactivated and marked. The unmarked neighbors of a marked vertex are also activated and added to the priority queues. The local search is interrupted when no vertex is eligible for moving or the random walk based stop criterion (Section 3.2.3.5) is satisfied. Finally, the best solution found is restored and all vertices get unmarked. This local search is repeated until no improvements are made.

The recoarsening technique, explained in Section 3.3.5.2, is applied to the n-level using the W-cycle scheme. Since the search tree has $O(n)$ levels, recoarsening happens only at levels n/c^i, for $i \in \mathbb{N}$ and some contraction factor c.

When compared to KaPPa, the KaSPar has better minimum and average results, despite consuming more time. That is because it is a sequential algorithm, but is interesting to note that the KaPPa parallelism does not help at solution quality, even with more local searches than KaSPar.

The authors' implementation does not allow to build perfectly balanced partitions, but, for the other commonly used values for ϵ, KaSPar does very well, improving many best known values of the standard benchmark (WALSHAW, 2000) and reproducing lots of others.

3.4.3 KaFFPa

The evolution of the ideas from KaPPa and KaSPar lead to the *Karlsruhe Fast Flow Partitioner* (KaFFPa) (SANDERS; SCHULZ, 2010), also a multilevel partitioner. The coarsening phase is based on the GPA matching algorithm (Section 3.3.1), and the rank function expansion^{*2} (Equation 3.16) and innerOuter (Equation 3.17). The latter is used at the first level because there are too much ties of the former in this case. The former is used in the remaining levels since it is the best one for this algorithm, as demonstrated by Holtgrewe, Sanders and Schulz (2009). Moreover, KaFFPa also uses the KaSPar rule that no vertex can be part of a contraction if it weighs more than $1.5n/20k$.

The coarsening phase stops if there are less than $\max\{60k, n/(60k)\}$ vertices. The initial partition is also the best among $100/\log k$ runs of the Scotch partitioner (PELLEGRINI, 2007b).

The refinement uses a quotient graph Q, explained in Section 3.4.1, to decide how to exchange vertices among parts. Two refinement methods are used for each pair of parts related by matchings of Q. These methods will be explained below, but we explain first how the pairs are selected.

The refinements are initially among two parts only, and are applied repeatedly among selected parts. To improve the partition, the selected pair of parts must be connected. When a pair of parts is refined and an improvement is made, then a global search attempt is made using a k-way FM algorithm that allows moves among any parts. This process is repeated until no improvements are made and then the algorithm proceeds to the next uncoarsened level.

The first 2-way refinement method tries to improve a solution by its maximum flow within the region around the cut. The second method is a modified FM heuristic that starts the optimization trial by the border vertices. Moved vertices have their neighbors activated and the search continues.

The recoarsening is also applied to KaFFPa, but the F-cycle (Section 3.3.5.2) is used. Although this partitioner has $O(\log_2 n)$ levels, because it uses a matching to contract edges, a full F-cycle would require too much time and so the authors proposed that the recoarsening should be made only at every second level.

3.5 Other Partitioners

Besides the three mentioned, many other partitioners have also been proposed. Recent ones include the KaFFPaE, from Sanders and Schulz (2011); MMA01, from Benlic and Hao (2010); MMA02, from Benlic and Hao (2011b); MTSA, from (BENLIC; HAO, 2011a); and others.

4 GRASP WITH PATH-RELINKING FOR THE GRAPH PARTITIONING PROBLEM

In this chapter we report our experiences with the graph partitioning problem. Some algorithms were implemented and tested, such as the Differential Greedy constructive heuristic, Kernighan-Lin, Fiduccia-Mattheyses and Lock-Gain refinement heuristics. We start introducing the metaheuristic Greedy Randomized Adaptive Search Procedure (GRASP) and path-relinking. Next, we propose some algorithms for the GPP using these techniques and give experimental results.

4.1 GRASP and Path-relinking Techniques

GRASP was developed in late 1980s by Feo and Resende (1989) and is a multi-start heuristic that, in each iteration, constructs an initial solution and refines it by a local search.

The initial solution must have some random element such that the space of possible solutions can be explored properly. The constructive algorithm also must be greedy, i.e., it chooses one element among the best from a restricted candidate list for insertion in the solution, and adaptive, because this choice depends on the previously inserted items.

The resulting solution of the construction phase is not guaranteed to be locally optimal, so it must be improved by a local search procedure. As a multi-start method, GRASP keeps the best found solution of all its iterations. As an effort to find the global optimum, the local search procedure may attempt to avoid local optima by including some acceptance mechanism for neighbors that deteriorate the current solution.

Some problems may require other local minima avoidance techniques. Path-relinking (PR) is an enhancement for GRASP that can lead to significant improvements in solution time and quality (RESENDE; RIBEIRO, 2003). It was originally proposed by Glover (1996) as an intensification strategy for Tabu and Scatter Search, but later proposed to be used within GRASP by Laguna and Marti (1999). It explores trajectories connecting solutions to find better ones. One of them is often chosen from a set of good (elite) solutions found during the search. The idea for connecting two solutions is to find a path through their neighborhoods by locally modifying the source solution until the target solution is reached. Only modifica-

tions which make the source more similar to the target are allowed. If a better solution is found, then the PR is successful and this new solution replaces the one of the current iteration. Algorithm 4.1 describes the generic structure of a GRASP with PR.

Algorithm 4.1: Generic GRASP with PR

```
1  E ← ∅
2  while no stop criterion is satisfied do
3      S ← construct a solution
4      S ← apply local search to S
5      if elite set E has some solution then
6          S ← best solution from path-relinking using S and some solution from E
7      if S is the best known solution or S may be added to E then E ← E ∪ {S}
8      adjust elite set E
9  return best solution in E
```

The stop criterion in line 2 is often just a time limit or maximum number of iterations, but can also be a target solution value or solution convergence, if detectable. Line 6 invokes the path-relinking using the current solution S and one or more solutions of the elite set E. Path-relinking strategies are presented below. If no better solutions among S and the path-relinked solutions with E are found, S is not altered. The best known solution is always added to elite set (line 7). The current solution may or may not be added to the elite set if it is worst than the best one. Some usual rules for this decision and the adjustment of the elite set at line 8 are presented next.

The elite set is initially empty and usually has a size limit. If it reaches its maximum capacity, an entrance rule must be applied to new solutions. A usual initial rule is to drop the new solution if its value is worse than all of the elite ones. If not, other strategies must be applied such as dropping the worst solution from elite set and adding the new one to it. Another strategy is to replace the elite solution that is most similar to the new solution. The similarity measure must be specifically defined for each problem and it is also useful to increase diversity in the elite set by allowing only solutions which differ enough from the others to enter the pool.

Path-relinking uses an elite solution and the one being optimized. An elite solution can be selected randomly. However, a strategy that selects one of the most different solutions from the other been relinked is usually better (RESENDE; WERNECK, 2004).

PR can be viewed as a local search strategy with a limited set of movements. Given two solutions, we can define their neighborhoods and restrain the set of neighbors to those which modify elements which are not common in both solutions. These common neighbors connect the two initial solutions by one or more paths of movements. These paths can be extended to two initial solutions that have no intersecting neighbors, but can be connected through intermediate ones. If some of these intermediate solutions have an objective cost lower than the two initial ones, a new local solution has been found. No restrictions are made to the objective cost, that can grow as much as needed until one source solution is altered to match a target

solution. The PR can be explored in many forms (RESENDE; RIBEIRO, 2003):

- *periodical relinking* – PR is applied at some iterations only;
- *forward relinking* – PR path of neighbors is built from the worst solution to the best;
- *backward relinking* – PR path of neighbors is built from the best solution to the worst;
- *back and forward relinking* – both trajectories are explored;
- *mixed relinking* – two trajectories are explored simultaneously, one starting from the best solution and another from the worst, and the PR ends when it reaches an intermediate and equidistant solution from the two initial ones;
- *randomized relinking* – movements are not greedy, but selected randomly among a list of best movements;
- *truncated relinking* – the full trajectory between the two initial solutions is not explored completely.

The exploration of two trajectories usually takes twice of the running time with no much improvements than a simpler relinking, as observed by Ribeiro, Uchoa and Werneck (2001). These same authors also observe that the backward relinking is usually the best among the single trajectories strategies.

The graph partitioning problem, as an application of GRASP, will be briefly explained in the next section, where we also define one partitioner of our own, targeted to solve the perfectly balanced bipartitioning problem.

4.2 Experimenting with GRASP and Path-relinking for the 2-way Equicut

Despite of several metaheuristics had been experimented to build graph partitioners, even including one GRASP (LAGUNA; FEO; ELROD, 1994), we decided to apply GRASP with more recent algorithms, such as the Differential-Greedy constructive heuristic from Battiti and Bertossi (1997) and the Lock-gain refinement heuristic from Kim and Moon (2004). We initially restricted ourselves to study the balanced bipartitioning problem, or the 2-way equicut, because these heuristics were proposed specifically for it and our objective was to find how their interaction would behave within a metaheuristic. Our initial propositions are presented next, along with some experimental results and their analysis.

Our partitioner for the 2-way equicut problem is defined in Algorithm 4.2. It is a GRASP with path-relinking based on the generic structure presented by Algorithm 4.1, but specialized to this problem. The generic methodology requires constructive and refinements algorithms. In our approach they are applied in lines 2 to 9. The path-relinking operator is applied in lines 10 to 16 and the elite set is managed in lines 17 to 18.

Algorithm 4.2: GRASP with path-relinking for the 2-way equicut

```
1  while no stop criterion is satisfied do
2      P ← diff_greedy()
3      repeat
4          Q ← diff_greedy()
5          Q ← local_search(Q)
6          R ← combined_dg(P, Q)
7          R ← iterative_improvement_local_search(R)
8          if R is better than P then P ← R
9      until P not improved
10     if elite set has some solution then
11         repeat
12             Q ← random solution from the first RandLimit in elite set
13             R ← path_relink(P, Q)
14             if R is better than P and Q then
15                 P ← iterative_improvement_local_search(R)
16         until P not improved
17     elite_appeal(P)
18     if current iteration multiple of Regen then regenerate()
19 return best solution in elite set
```

The Differential Greedy constructive algorithm (Section 3.2.2.7) was used to build new solutions at every GRASP iteration. This is an appropriate algorithm for this task since it has random elements, such as the seed vertices for each part and random tie breaking rule for vertices with the same gain; it is greedy, by the inclusion of vertices with the highest gain; and it is adaptive, because the gain function is updated at every vertex inclusion. The DG algorithm is used at lines 2, 4 and inside of the `combined_dg` operator of line 6.

As the local search procedure, we apply the Kernighan-Lin algorithm (Section 3.2.3.1) to search the neighborhood for better solutions. Our implementation of the KL algorithm uses the bucket list structure from Fiduccia and Mattheyses (1982) as detailed in Section 3.2.3.2. This way, the swap operator of the KL algorithm can be implemented in constant time. To improve the performance of the algorithm, we stop the local search after $n/6$ vertices have been swapped, supported by preliminary experiments showing that the solution quality does not degrade much in this way. The local search was defined in two versions: single or iterative improvement. The *single improvement* version used in line 5 and inside of `path_relink` operator performs a single execution of the KL algorithm. The *iterative improvement* version used in lines 7 and 15, on the other hand, executes the KL algorithm as long as it can improve the current solution.

Lines 2 to 9 are a mixed constructive and refinement phase that attempts to build good partitions by the combination of initial solutions from the DG algorithm and some refined ones, coming from the local search of previous iterations of this operator itself. The phase is based on the `combined_dg` operator, described by Algorithm 4.3. This is a modified DG algorithm that, instead of building a new partition starting from random vertices, it uses common ones from two input partitions. Lines 2 to 6 ensures that the maximum number of vertices is reused in the new partition. The

following lines are the same as in the original DG algorithm.

Algorithm 4.3: Combined Differential-Greedy

Input: Graph $G = (V, E)$; Two bipartitions P and Q of V.
Output: A bipartition of V

```
1  R ← {R_0, R_1}
2  R_0 ← {v | v ∈ P_0, v ∈ Q_0}
3  R_1 ← {v | v ∈ P_1, v ∈ Q_1}
4  if |R_0| + |R_1| < |V|/2 then
5      R_0 ← {v | v ∈ P_0, v ∈ Q_1}
6      R_1 ← {v | v ∈ P_1, v ∈ Q_0}
7  V' ← V \ R_0 \ R_1
8  p ← 0
9  while |V'| > 0 do
10     b ← differential_greedy_function(V', R, p, G)
11     R_p ← R_p ∪ {b}
12     V' ← V' \ {b}
13     p ← (p + 1) mod k
14 return R
```

Back to Algorithm 4.2, the construction and refinement phase attempts to overcome a flaw of DG algorithm, the bad selection of seed vertices by reusing vertices of refined partitions to build new ones. The output of this phase is always a refined solution and so the GRASP may continue.

The next phase is the *path-relinking* (lines 10 to 16) and it is also repeated as long as improvements are made. The elite solution Q is selected randomly among the first `RandLimit` solutions of the elite set for path-relinking. The PR operator at line 13 locks the vertices of P common to Q in a similar manner of lines 2 to 6 of Algorithm 4.3. The remaining vertices are swapped following the Kernighan-Lin greedy function of gains until the partition P becomes equivalent to Q. If a partition better than P and Q is found, a local search is applied to it and it becomes the partition P of the current GRASP iteration (lines 14 and 15).

The partition P, output of the path-relinking phase, is added to the elite set following Algorithm 4.4. If it is not the best known solution, it must be better than the worst solution in the set and sufficiently different from the others (line 14). This difference between two partitions is computed by

$$d(P, Q) = \frac{|V| - |R_0| - |R_1|}{|V|}, \tag{4.1}$$

with R being computed the same way as lines 2 to 6 of Algorithm 4.3. The minimum difference that a solution must have to the others to be considered a new elite is given by the parameter `MinDiff`.

When the elite set is bigger than the expected size, limited by the parameter `MaxSize`, a solution must be dropped (line 7 of Algorithm 4.4). The dropped solution is the most similar to the recently inserted one.

Sometimes, this procedure may populate an elite set with solutions too similar with each other because of the condition that every new best known solution must be

Algorithm 4.4: Elite Appeal Procedure

Input: A partition P that appeals to be considered elite

```
1  if P is the best solution then
2      add P to the elite set
3  if P is better than some other elite then
4      if d(P,Q) ≥ MinDiff, for all Q in elite set then
5          add P to the elite set
6  if elite set greater than MaxSize then
7      drop the most similar solution with P from the elite set
8  return True if P is a new elite, False otherwise
```

accepted, preventing new solutions to replace some of them. Thus, the small entropy may prejudice the performance of the path-relinking and local minima can be hard to escape. This way, every `Regen` iterations, the elite set is regenerated (line 18) by Algorithm 4.5. The restriction of minimum difference, inside the `elite_appeal` procedure, guarantees that the resulting elite set has enough entropy.

Algorithm 4.5: Elite regeneration

Input: Global elite set $\{E_0, E_1, \dots\}$, sorted by non-decreasing cut size

```
1  E' ← {E_0, ...}
2  clear the global elite set
3  foreach P ∈ {E_i ∈ E' | i < EliteRandLimit} do
4      foreach Q ∈ E' do
5          R ←path_relink(P,Q)
6          R ←iterated_local_search(R)
7          elite_appeal(R)
```

4.2.1 Experimental Results

We conducted some experiments to evaluate the quality of the partitioner. We used the instances from Walshaw's partition archive (Section 2.4.2). Some preliminary tests were conducted to determine the following used parameters: `MaxSize` = 10, `MinDiff` = 10%, `Regen` = 32 and `RandLimit` = 3. The running time was limited to 200, 600 and 1800 seconds for the small, medium and large instances, respectively. A comparison with the best known value of each instance can be seen in Figure 4.1. The figure also shows the results of the same GRASP but with the Lock-gain local search, that will be detailed below.

The results show that the GRASP with Kernighan-Lin local search were unable to obtain any new best known value. However, this approach could match the BKV in 20 instances. For the remaining 14, results were 1.48% worse on average than the BKV. The instance `t60k` had the worst result, being 7.59% above BKV. These results, although far from excellent, gave us a hint that this approach was promising, since most of the results were very close to the best ones.

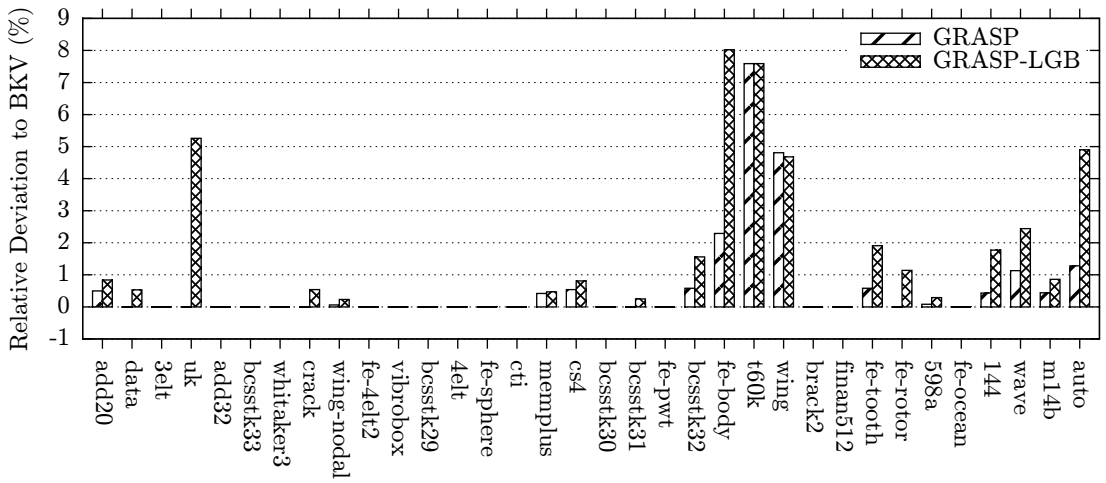


Figure 4.1: Comparison of two implementations of a GRASP with path-relinking for the 2-way equicut problem. Our results are shown as the relative deviations to each instance of the Walshaw package. In the figure, GRASP stands for the version that uses the Kernighan-Lin local search, while GRASP-LGB stands for the one that uses the Lock-gain local search.

4.2.2 Lock-gain as GRASP's Local Search

To compare with other tie breaking rules for the KL algorithm associated to other partitioning techniques, such as metaheuristics, we implemented the Lock-Gain heuristic, based on Algorithm 3.6. Some tests have been made to verify that our implementation's results are equivalent to the ones presented by the authors. We tested our implementation using some of the instances used by Kim and Moon for evaluation of the Lock-Gain heuristic. The same tests proposed by the authors were performed with the available instances in order to compare the refinement results of KL and LG heuristic. The same minimum cut sizes reported by the authors were also found by our implementation for every tested instance and the average results are comparable.

The usage of this recent heuristic for local search was motivated by the better results obtained when compared to Fiduccia-Mattheyses and also because of how the local search is applied inside our path-relinking operator. In our approach, the path-relinking locks the vertices common to both partitions that are being relinked. The Lock-gain could use this information of locked vertices to move the remaining ones with higher potential of finding better solutions. The higher quality of Lock-gain is based in the knowledge of locked vertices, that imply an immutable weight in the cut size. With this, we believed that the PR could be more effective if this knowledge was considered.

Our second version of the GRASP with path-relinking is, therefore, the same as described in Algorithm 4.2 but we replace the local search with the Lock-gain algorithm in every occurrence of the Kernighan-Lin's.

We conducted the same experiments as with the first version of the GRASP, with the same parameters and time limit. In Figure 4.1 it is possible to compare

the final minimum relative deviations of both GRASP versions. As one can see, the results of GRASP with LG heuristic are worse on average than the ones using the KL algorithm. Except for a single instance, `wing`, that yields a better cut size with the LG heuristic, 16 instances had the same results and 17 others worsened 1.37% on average when compared to the version with KL heuristic. If we compare the LG version with the BKV of each instance, 19 instances had worse results, 2.32% of relative deviation on average, 0.84% higher than the KL version.

A possible explanation for the GRASP with LG heuristic results being worse than the GRASP with KL heuristic is that the locked vertices have different sources in GRASP and the pure LG algorithm proposed by Kim and Moon. This heuristic uses the locked vertices to compute a distinguished gain for the unlocked vertices. However, this locking is made by the LG algorithm itself and, initially, no vertex is locked. On the other hand, the LG algorithm combined with the path-relinking operator starts with a non empty set of locked vertices, because they were common to the related solutions. It happens that many vertices can be blocked by the PR operator, building small vertex groups that are not allowed to be moved by the local search. The remaining vertices may be placed in such a way that they depend on the swapping of some of the locked vertices, or else they are unable to make improvements. The procedure to avoid this problem is still uncertain, but an option could be the contraction of the common vertices from the related solutions. The small groups formed by locked neighbors would be contracted into a weighted vertex and they could be allowed be moved to other parts by the PR operator. The LG algorithm would loose its relation to the locked vertices inside the operator, but it still could be used to enhance the results.

By the end of the experiments, we concluded that the Lock-gain algorithm does not help to improve our results, at least for the instances of the Walshaw's partition archive. The Differential Greedy constructive algorithm, on the other hand, had shown to be helpful for the good results of the GRASP with path-relinking using the Kernighan-Lin refinement heuristic.

4.3 A GRASP with Path-relinking for the k-way GPP

In this section we detail a new GRASP with path-relinking for the k-way graph partitioning problem. This algorithm has the same structure of Algorithm 4.2 and we highlight their differences below.

We used the k-way versions of the Differential Greedy constructive algorithm (Section 3.2.2.7) and the Tabu based Fiduccia-Mattheyses refinement heuristic (Sections 3.2.3.4 and 3.2.3.5). The random walk stop criterion, explained in the latter section, was also applied to the TFM heuristic.

The definition of set R in lines 2 to 6 of Algorithm 4.3, had to be replaced since the number of possible mappings for k-partitions would turn this approach prohibitive. A new method was then required to compute the mapping that maximizes partition similarity.

Gusfield (2002) defines *partition distance* as the number of vertices that need to change parts so a partition becomes equivalent to another. Given two partitions P

and Q and some bijection mapping the parts of P to the parts of Q, $\sigma : \{0, \ldots, k-1\} \rightarrow \{0, \ldots, k-1\}$, let $S(\sigma) = \{v \in V \mid v \in P_i, v \in Q_j, \sigma(i) = j\}$ be the set of vertices that makes the two partitions similar based on the correspondence of parts imposed by σ. Two partitions are said to be equivalent when $|S(\sigma)| = n$. The critical issue in this case is defining σ when two partitions differ. We are in fact interested in the mapping σ_* that maximizes the similarity, i.e., $|S(\sigma_*)| = \max_\sigma(|S(\sigma)|)$. The partition distance is then defined as $d(P, Q) = n - |S(\sigma_*)|$.

In order to define the mapping σ_*, a greedy algorithm that does not guarantee optimality is introduced in Galinier, Boujbel and Fernandes (2011). Gusfield (2002) and Konovalov, Litow and Bajema (2005) compute the partition distance optimally, by its reduction to the Assignment Problem (LAWLER, 1976), with time complexity $O(k^3)$. We preferred to define σ_* optimally through a maximal matching in a bipartite graph, which may also be reducted from the Assignment Problem. This graph is defined as follows: each part P_i can be related to each part Q_j; the weight $\omega(\{i, j\})$ of this edge corresponds to the number of similar vertices, i.e., $\omega(\{i, j\}) = |\{v \in V \mid v \in P_i, v \in Q_j\}|$. The maximal matching is obtained via the Kuhn-Munkres (Hungarian) algorithm (MUNKRES, 1957) and the solution corresponds to the mapping σ_*. The complexity of this algorithm is also $O(k^3)$. We have no evidence that an optimal solution for the partition distance is better than an heuristical one for the final results of the partitioning, but computing the optimal solution is not very costly compared to the rest of the partitioning.

The `combined_dg`, `path_relink` and `elite_appeal` procedures were adapted to use the mapping function σ_* in order to lock common vertices or to compute partition distances.

The unlocked vertices at path-relinking are only allowed to be moved to their equivalent parts. For example, if a vertex $u \in A_1$ and $u \in B_3$, but $\sigma_*(2) = 3$, the only allowed move for u is from part A_1 to A_2 because A_2 is equivalent to B_3, as defined by σ_*. The order of movements is guided by Fiduccia-Mattheyses heuristic explained above and all allowed movements are performed once.

The `elite_appeal` procedure also uses the mapping function to compute the difference among two partitions. All remaining elements of the partitioner were kept with the same structure.

We conducted a series of experiments in order to evaluate the quality of the solutions generated by this GRASP. Our experiments were run on Machine A with a time limit of two hours for each instance of the Walshaw package. These times are acceptable since state-of-the-art approaches impose limits from one to five hours (OSIPOV; SANDERS, 2010; SANDERS; SCHULZ, 2010; BENLIC; HAO, 2011b). The parameters used in the experiments are: $\alpha = 100$, $\beta = \log_2 n$, `maxT` $= k\sqrt{m}$, local search movements limit $= 10n$, `MinDiff` $= 5\%$, `MaxSize` $= 5$, `RandLimit` $= 5$, `Regen` $= 32$. Tests were replicated three times for each combination of $k \in \{4, 8\}$ and $\epsilon \in \{1.03, 1.05\}$. The best and average cut size found for each instance in all repetitions is shown in Tables 4.1 and 4.2, for the cases $\epsilon = 1.03$ and $\epsilon = 1.05$ respectively.

An overall view of the results shows us that the GRASP is more effective for the cases where $k = 4$ in both tested imbalances. Compared to best known values, our

algorithm achieves an average relative deviation of 3.9%. The deviation is 1.6% for the small instances, 9.7% for the medium instances and 2.1% for the large instances. We were able to improve the results of eleven cases and achieved the best known value in 40 of them.

The large relative deviations are mostly due to some graphs that the procedure has difficulties to deal with, such as `uk`, `memplus`, `cs4`, `fe_body`, `t60k`, `wing` and `auto`. We suppose that the problem with the last one is due to the fact that only few iterations were able to be performed in the given time limit. This also supports our first finding, based on analysis of the partitionings, that the path-relinking is responsible for most of the improvements on cut size. The large instances that had a low number of iterations were not able to take full advantage of the newly proposed operator of PR. For the instances `fe_body`, `t60k` and `wing`, our constructive algorithm turned out to be a lot less effective than expected when compared to the remaining instances in the dataset. With these results we observe that good initial partitions play an important role on the whole procedure and we should search for better methods.

Table 4.1: Best and average results found by our GRASP among all repetitions compared with the best known values (BKV) for ϵ = 1.03. Bold values mark best known values. Relative deviation is presented as a percentage of our cuts relative to BKV. We also provide some statistics such as the number of improved results, matched BKV or had worse cuts, the average and standard deviation of the relative deviation and the average of relative deviation counting only worse results.

	k = 4					k = 8				
				Rel. Dev.					Rel. Dev.	
Instance	BKV	Best	Avg	Best	Avg	BKV	Best	Avg	Best	Avg
add20	1158	**1135**	1142.7	-2.0	-1.3	**1689**	1693	1701.7	0.2	0.7
data	**369**	**369**	369.0	0.0	0.0	**638**	**638**	638.3	0.0	0.1
3elt	**198**	**198**	198.0	0.0	0.0	**334**	335	335.7	0.3	0.5
uk	**39**	42	42.3	7.7	8.5	**78**	85	90.0	9.0	15.4
add32	**33**	**33**	33.0	0.0	0.0	**66**	**66**	66.3	0.0	0.5
bcsstk33	20854	**20762**	20763.3	-0.4	-0.4	34078	**34065**	34067.3	-0.0	-0.0
whitaker3	**378**	**378**	378.0	0.0	0.0	**650**	653	653.3	0.5	0.5
crack	**360**	**360**	360.0	0.0	0.0	**671**	673	673.7	0.3	0.4
wing_nodal	**3538**	**3538**	3539.3	0.0	0.0	**5361**	5366	5367.0	0.1	0.1
fe_4elt2	**342**	**342**	342.7	0.0	0.2	**595**	597	598.7	0.3	0.6
vibrobox	**18736**	18742	18742.3	+0.0	+0.0	**24170**	24227	24231.3	0.2	0.3
bcsstk29	**7971**	7993	8026.7	0.3	0.7	**13717**	13791	13794.7	0.5	0.6
4elt	**319**	**319**	319.0	0.0	0.0	**522**	523	523.0	0.2	0.2
fe_sphere	**764**	**764**	764.0	0.0	0.0	**1152**	**1152**	1152.0	0.0	0.0
cti	**916**	**916**	916.0	0.0	0.0	**1714**	**1714**	1716.0	0.0	0.1
memplus	**9362**	10473	10577.3	11.9	13.0	**11624**	12928	13113.3	11.2	12.8
cs4	**917**	988	1005.3	7.7	9.6	**1424**	1546	1558.7	8.6	9.5
bcsstk30	16399	**16371**	16392.3	-0.2	-0.0	**34137**	34179	34232.0	0.1	0.3
bcsstk31	**7150**	7171	7191.3	0.3	0.6	**12985**	13201	13234.3	1.7	1.9
fe_pwt	**700**	**700**	700.0	0.0	0.0	**1410**	1415	1415.0	0.4	0.4
bcsstk32	**8725**	8880	9449.0	1.8	8.3	**19956**	20827	21202.0	4.4	6.2
fe_body	**598**	661	674.7	10.5	12.8	**1016**	1147	1226.7	12.9	20.7
t60k	**203**	252	268.3	24.1	32.2	**449**	856	930.7	90.6	107.3
wing	**1593**	1803	1822.7	13.2	14.4	**2451**	3088	3462.7	26.0	41.3
brack2	**2834**	**2834**	2834.0	0.0	0.0	**6800**	6835	6860.3	0.5	0.9
finan512	**324**	**324**	324.0	0.0	0.0	**648**	**648**	648.0	0.0	0.0
fe_tooth	**6764**	6777	6784.3	0.2	0.3	**11274**	11464	11481.3	1.7	1.8
fe_rotor	7118	**7103**	7138.0	-0.2	0.3	**12445**	12886	12928.7	3.5	3.9
598a	**7816**	7858	7869.0	0.5	0.7	**15613**	15875	15900.3	1.7	1.8
fe_ocean	**1693**	**1693**	1693.0	0.0	0.0	**3920**	3949	3962.3	0.7	1.1
144	**15078**	15151	15196.7	0.5	0.8	**25092**	26515	26687.3	5.7	6.4
wave	**16665**	16728	16736.0	0.4	0.4	**28495**	28966	29007.0	1.7	1.8
m14b	**12948**	13281	13332.3	2.6	3.0	**25390**	26564	26613.0	4.6	4.8
auto	**25789**	26621	26904.3	3.2	4.3	**44785**	49207	49909.3	9.9	11.4
Improved:				4	3				1	1
Matched:				14	12				5	2
Worst:				16	19				28	31
Average (%):				2.42	3.19				5.81	7.48
Std (%):				5.38	6.74				15.94	19.46
Average of worsts (%):				5.31	5.80				7.05	8.20

Table 4.2: Best and average results found by our GRASP among all repetitions compared with the best known values (BKV) when $\epsilon = 1.05$.

	k = 4					k = 8				
				Rel. Dev.					Rel. Dev.	
Instance	BKV	Best	Avg	Best	Avg	BKV	Best	Avg	Best	Avg
add20	1149	**1128**	1128.0	-1.8	-1.8	**1675**	1682	1684.3	0.4	0.6
data	**363**	**363**	363.0	0.0	0.0	**628**	**628**	628.0	0.0	0.0
3elt	**197**	**197**	197.0	0.0	0.0	**329**	**329**	329.7	0.0	0.2
uk	**39**	40	41.7	2.6	6.8	**75**	84	88.3	12.0	17.8
add32	**33**	**33**	33.0	0.0	0.0	**63**	**63**	63.3	0.0	0.5
bcsstk33	**20167**	**20167**	20167.0	0.0	0.0	33919	**33916**	33916.0	0.0	0.0
whitaker3	377	**376**	376.3	-0.3	-0.2	**644**	648	649.0	0.6	0.8
crack	**360**	**360**	360.0	0.0	0.0	**666**	**666**	667.0	0.0	0.2
wing_nodal	**3521**	3522	3522.7	+0.0	+0.0	**5341**	5345	5346.3	0.1	0.1
fe_4elt2	**335**	337	337.0	0.6	0.6	**578**	583	583.7	0.9	1.0
vibrobox	**18690**	**18690**	18693.3	0.0	0.0	**23924**	23944	23957.3	0.1	0.1
bcsstk29	**7925**	7986	7992.0	0.8	0.8	**13540**	13556	13615.0	0.1	0.6
4elt	**315**	**315**	315.3	0.0	0.1	**515**	516	516.0	0.2	0.2
fe_sphere	**762**	**762**	762.0	0.0	0.0	**1152**	**1152**	1152.0	0.0	0.0
cti	**889**	**889**	889.0	0.0	0.0	**1684**	**1684**	1684.0	0.0	0.0
memplus	**9292**	10387	10468.0	11.8	12.7	**11543**	12796	12882.3	10.9	11.6
cs4	**909**	977	995.7	7.5	9.5	**1420**	1521	1543.7	7.1	8.7
bcsstk30	16186	**16169**	16180.0	-0.1	-0.0	**34071**	34121	34149.0	0.1	0.2
bcsstk31	7086	**7079**	7096.7	-0.1	0.2	**12853**	13129	13161.3	2.1	2.4
fe_pwt	**700**	**700**	700.0	0.0	0.0	**1405**	**1405**	1405.3	0.0	0.0
bcsstk32	**8441**	8468	9071.7	0.3	7.5	**19411**	20290	20386.7	4.5	5.0
fe_body	**588**	675	707.0	14.8	20.2	**1013**	1129	1200.7	11.5	18.5
t60k	**195**	231	238.0	18.5	22.1	**443**	786	854.3	77.4	92.9
wing	**1590**	1716	1767.0	7.9	11.1	**2440**	3058	3182.0	25.3	30.4
brack2	**2731**	**2731**	2731.7	0.0	0.0	**6592**	6609	6615.3	0.3	0.4
finan512	**324**	**324**	324.0	0.0	0.0	**648**	**648**	648.0	0.0	0.0
fe_tooth	**6688**	6704	6705.7	0.2	0.3	**11154**	11297	11338.7	1.3	1.7
fe_rotor	6899	**6804**	6824.0	-1.4	-1.1	**12309**	12775	12808.3	3.8	4.1
598a	**7728**	7762	7764.7	0.4	0.5	**15414**	15676	15712.0	1.7	1.9
fe_ocean	**1686**	**1686**	1686.3	0.0	+0.0	**3893**	3961	3963.7	1.7	1.8
144	**14982**	15102	15122.7	0.8	0.9	**24767**	25899	26066.7	4.6	5.2
wave	**16533**	16633	16636.7	0.6	0.6	**28489**	28733	28835.3	0.9	1.2
m14b	**12945**	13062	13187.7	0.9	1.9	**25143**	26236	26527.7	4.3	5.5
auto	**25271**	25867	26031.3	2.4	3.0	**44206**	47488	49172.7	7.4	11.2
Improved:				5	4				1	1
Matched:				13	9				8	4
Worst:				16	21				25	29
Average (%):				1.95	2.82				5.27	6.61
Std (%):				4.63	5.83				13.79	16.68
Average of worsts (%):				4.38	4.71				7.17	7.75

5 A HYBRID ALGORITHM FOR THE k-WAY GRAPH PARTITIONING PROBLEM

In this chapter we propose and detail a new hybrid GRASP with path-relinking and a multilevel algorithm for the k-way graph partitioning problem. This partitioner, defined in Section 5.1, is an evolution of our previous work, presented in Chapter 4. Several experiments, presented in Section 5.2, were conducted to define its final form and the set of parameter values used to produce our best results, presented in Section 5.3.

5.1 HYPAL

Here we present our new approach for the problem, the HYbrid k-way graph PArtitioning ALgorithm (HYPAL). We first give an overview of the partitioner and then we detail its procedures and structures in the next subsections. A diagram of HYPAL can be seen in Figure 5.1, while more details are presented next.

Our approach for the k-way graph partitioning problem is shown in Algorithm 5.1, which is similar to Algorithm 4.2 but with enhanced operators. The basic heuristics are the same as previously used: the Differential-Greedy constructive algorithm of Section 3.2.2.7 and the Tabu based Fiduccia-Mattheyses refinement heuristic of Section 3.2.3.5. Different from Algorithm 4.2, HYPAL does not use the random walk stop criterion inside the local search operator because we observed that it provided no contributions in solution quality for the partitioner (see Section 5.2).

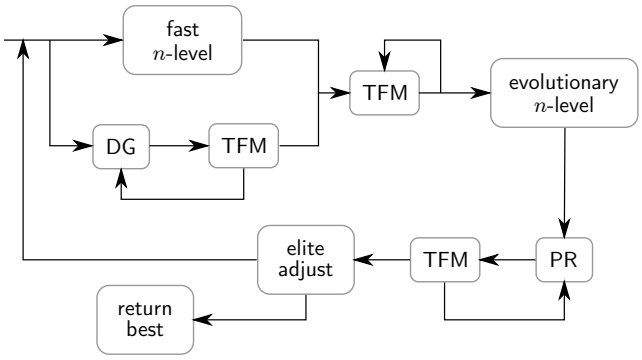


Figure 5.1: A diagram of HYPAL which shows how the different partitioning techniques were combined to form a GRASP with PR.

Algorithm 5.1: Hybrid k-way Graph Partitioning Algorithm (HYPAL)

```
1  while stop criterion is not satisfied do
2      P ← fast_n_level();
3      for i ∈ {1,...,MaxDG} do
4          Q ← diff_greedy();
5          Q ← local_search(Q);
6          if Q is better than P then P ← Q;
7      P ← iterative_improvement_local_search(P);
8      P ← evolutionary_n_level(P);
9      if elite set has some solution then
10         repeat
11             Q ← random solution from the first EliteRandLimit in elite set;
12             R ← path_relink(P, Q);
13             if R is better than P and Q then
14                 P ← iterative_improvement_local_search(R);
15         until P not improved;
16     if elite_appeal(P) then
17         P ← evolutionary_n_level(P);
18         elite_appeal(P);
19     else if denied appeals greater than EliteMaxDenied then regenerate();
20     update_pheromone(P);
21 return best solution in elite set;
```

The construction phase of each GRASP iteration (lines 2 to 6) chooses as the initial solution the best among the result of a modified n-level partitioning (described in Section 5.1.2) and the execution of `MaxDG` iterations of the DG algorithm and then applies TFM once. The refinement phase consists of two local searches. The first is an iterative TFM, the second is a probabilistic n-level algorithm, which was designed as a refinement algorithm.

The path-relinking phase (lines 9 to 19) first tries to improve the current solution by path-relinking to a randomly selected solution among the best `RandLimit` elite solutions. If the solution could be improved, TFM is applied. This is repeated until no more improvement can be made (lines 10 to 15). The PR operator is the same as described in Section 4.3. The elite set is updated in lines 16 to 19. The new solution is added to the elite set, if it is of sufficient quality and satisfies a minimum distance criterion with respect to the other elite solutions, as in Algorithm 4.4. If the current solution enters the elite pool, a new probabilistic n-level search is executed because it may have been modified since the last search. Line 20 is part of the probabilistic n-level adjustment and will be detailed in Section 5.1.2.

5.1.1 Constructive and Refinement Operators

The reasoning behind our approach of using the combination of a fast n-level and some repetitions of DG algorithm for the construction phase derives from the empirically measured bad solution quality of the DG algorithm in some instances. These poor solutions would affect the final result of a GRASP iteration spending processing time with partitions that most likely would not even be elite ones. Bat-

titi and Bertossi (1997) showed that the DG algorithm should be executed several times in order to improve the solution quality, since the random seed selection could eventually make poor choices. However, some experiments with our GRASP showed that the DG algorithm was slightly ineffective in some instances, generating worse partitions when compared to its performance on other graphs. This way, we decided to include a simplified n-level algorithm to overcome such limitations. This n-level was designed to be faster than its original form, proposed by Osipov and Sanders (2010), and it has random elements that make it suitable for a constructive algorithm.

The refinement phase of our algorithm is composed of two local searches. The first one is a Fiduccia-Mattheyses algorithm associated with a Tabu list that allows vertices to change parts more than once. Preliminary experiments (Section 5.2) had shown that this local search provides better solutions than the original FM algorithm, even being more time consuming. Our second local search is based on a modified n-level algorithm and will be presented next.

5.1.2 n-level

A promising avenue for obtaining better partitioners are hybrid heuristics, which systematically combine features from different methods. Two recent approaches by Benlic and Hao (2011a), Benlic and Hao (2011b) combine a multilevel scheme with a Tabu Search or a Memetic Algorithm, which replace the initial partition and refinement phases, and obtain good results. Inspired by these hybrid metaheuristic algorithms, we decided to use the n-level algorithm to enhance our basic operators.

Different from other hybrid algorithms that use the multilevel technique as an environment for applying metaheuristics at every coarse level, we use the GRASP as external algorithm and the n-level as a constructive and a refinement heuristic. GRASP relies on multiple starts to explore the search space, therefore, the required operators for this procedure must be the least time consuming as possible and so, in order to use a multilevel partitioner, it must be fast.

The solution quality of a multilevel algorithm can be improved if recoarsening is applied, at the price of increasing runtime. To save time, our *fast n-level* algorithm drops recoarsening, using the V-cycle, explained in Section 3.3.5.2. The resulting partitions are worse but still serve our purpose to enhance the construction phase of the GRASP. The random tie breaking rule for the edge selection in the n-level algorithm is also useful for this case, since randomization is required for a constructive algorithm in GRASP.

For the GRASP's refinement phase, we propose a modified n-level, designed to be an evolutionary local search operator called *evolutionary n-level* algorithm. The recoarsening scheme used in this algorithm is the F-cycle that makes two global attempts at every level that has a coarse graph shrunken to the half of the last recontraction level. An F-cycle is faster than the W-cycle and experiments have shown that it is not much worse.

Since this algorithm must be a local search operator, it must refine the current solution or return the initial one if it cannot be improved. This behavior is handled

by the coarsening scheme, that does not contract cut edges of the best solution. In this way, the current solution of a GRASP iteration is set as the best one at the beginning of the n-level operator.

At refinement phase, the original n-level activates only the vertices incident to a recently expanded edge and these are the seeds for its local search (see Section 3.4.2). We decided to alter this behavior and, instead of immediately activating these vertices, put them in a pool of vertices, for a lazy activation. When this pool reaches a limit, defined by `LazyActivSize`, its vertices get activated and a local search is started. We ignore the activation of vertices that no longer exist since they had been extended and deleted by an edge uncoarsening. We decided to use this lazy activation pool to avoid overheads and also because the solutions had shown to be a bit better with this modification, as seen in Section 5.2.

Despite of these simple modifications, what makes this algorithm an evolutionary local search is the inclusion of probabilities of edges being part of the optimum cut based on previously observed partitions. These probabilities are useful for a better edge selection rule in the contraction phase. We used some ideas of the Ant Colony metaheuristic, so every edge has a new parameter, an amount of pheromone. When a good solution is defined the pheromones are updated (line 20 of Algorithm 5.1) as

$$\tau_{u,v} \leftarrow (1-\rho)\tau_{u,v} + \delta, \tag{5.1}$$

where $\tau_{u,v}$ is the amount of pheromone of edge $\{u,v\}$, ρ is the evaporation rate and δ is one if the edge is part of the current cut, zero otherwise.

With the edge pheromones we could follow an ant colony optimization and define the probability $p_{u,v}$ of an edge to be part of the cut as

$$p_{u,v} = \frac{(\tau_{u,v}^{\alpha})(\eta_{u,v}^{\beta})}{\sum(\tau_{u,v}^{\alpha})(\eta_{u,v}^{\beta})}, \tag{5.2}$$

where α and β are adjustment parameters and $\eta_{u,v}$ is "a priori" knowledge that such edge is part of the cut — we defined $\eta_{u,v} = \text{expansion}^*(\{u,v\})^{-1}$. However, the computation of $p_{u,v}$ is costly because of the sum of knowledge used for normalization. Since we are interested in a rank function to rule the edge selection order, the values are not required to be normalized and thus we use just the inverse of the numerator as a new rank function

$$\text{ant}(\{u,v\}) = (\tau_{u,v}^{-\alpha})(\text{expansion}^*(\{u,v\})^{\beta}). \tag{5.3}$$

When selecting an edge for contraction, this function is used to rank the edges and the highest ranks are selected first. The inverse of the numerator is used because the more pheromone an edge has, the greater is its probability to be part of the optimal cut, and we want these edges to be contracted in the coarsest levels, since it is easier to refine these partitions. In case of ties, expansion* is used as a first-level tie-breaking rule and remaining ties are broken randomly. When contracting an edge, the edges incident to the contracted vertices must be reassigned to the new vertex. The pheromone of these edges is the same as the edges they represent or the maximum of the two edges they represent, if a common neighbor is found.

5.2 Experiments

The goal of this phase of our research was to enhance partition quality, remove ineffective parts of our algorithm and tune the set of parameters. We started with the GRASP of Section 4.3 as our base version and we made several incremental adaptations until we reached the definition of the algorithm presented above, in Section 5.1.

The number of required experiments to test all possible configurations of the algorithm is prohibitive. For this reason we opted for a one-factor-at-a-time (MONTGOMERY, 2008) methodology to execute and evaluate the results. The experiments reported here in this section were executed in Machine A, single runs, with time limit of 30 minutes and computing the 4-way partitioning with maximum imbalance of 3%. The single run is a problem to remark, because the decisions that we will expose were taken based on the result's average relative deviation in each run. This deviation is very likely to change in new runs because of the evolutionary aspect of the algorithm. However, we know by experience that these averages do not change too much in our algorithm, so decisions based on two very distinct deviations would be reliable. On the other hand, decisions based on close deviations may be incorrectly taken, but as the cost to reduce computing time to complete the experiments.

Altogether, 12 experiments were conducted, each of one composed of several tests. Each experiment intends to answer specific questions. An overview of the results can be seen in Figure 5.2, which presents a quartile plot that depicts minimum and maximum values as the whiskers, the median as the points and the lower and upper quartiles as the spaces between the points and the whiskers. The maximum values were scaled differently than the other values, such that the most interesting part of relative deviations in the range $[0, 4]$ is better visible. The best average relative deviations found for each experiment is also reported as a line plot. Results are presented for all experiments and our base results, computed with a slightly modified version of the GRASP algorithm presented in Section 4.3.

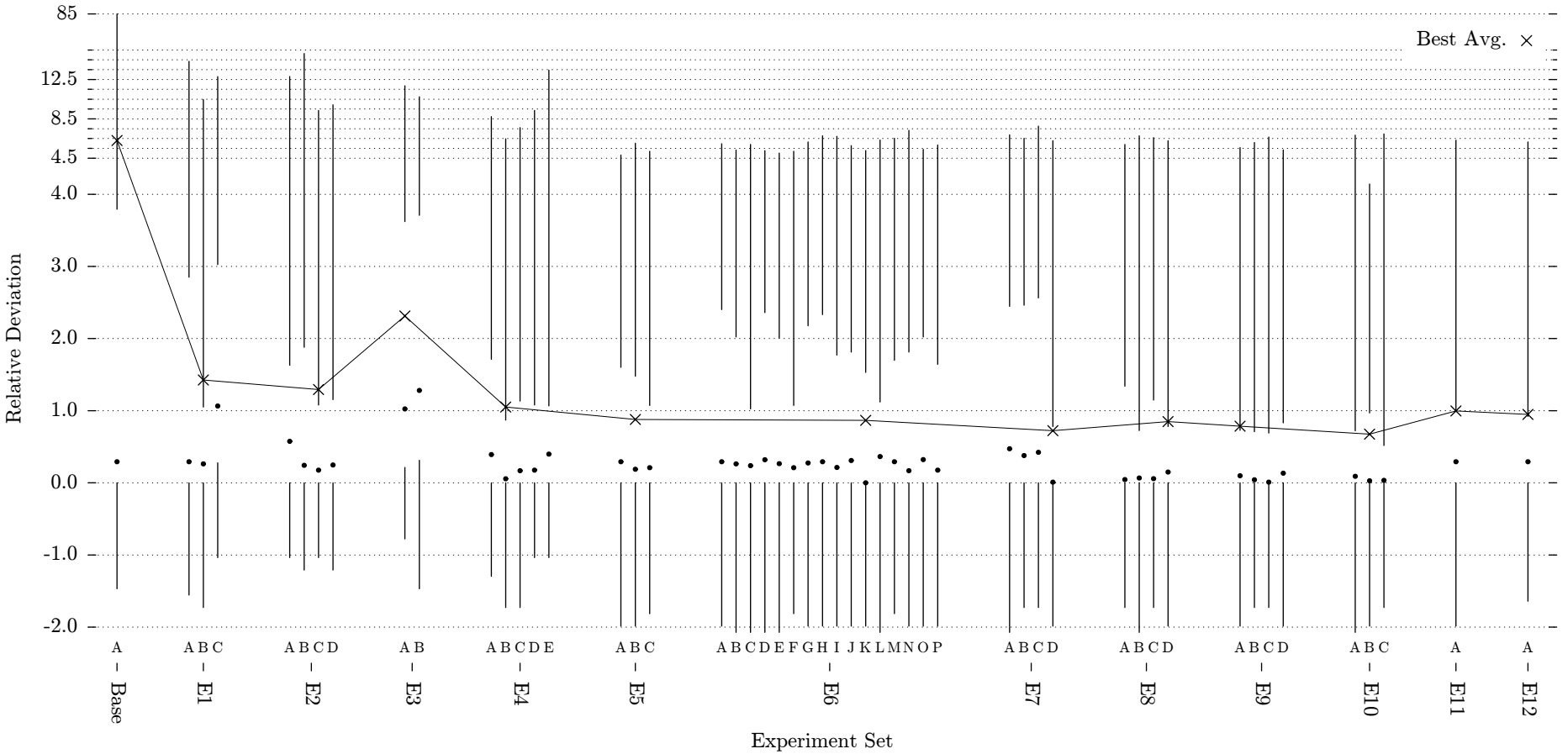


Figure 5.2: An overview of the experimental results used to tune our algorithm. The quartile plot exposes minimum, lower quartile, median, upper quartile and maximum results of each test. Tests A to P are relative to each experiment and distinct from each other. Observe that values greater than 4.5 are scaled differently so the lower values can be better visualized. The best average relative deviation among the tests in each experiment is also presented. Each experiment inherits the improvements of the previous ones.

The experiments were made sequentially so each new improvement discovered was carried to the next ones and the results of such enhancement was considered a new baseline. This can be observed by the decreasing average, median and upper quartile of relative deviations of most tests in Figure 5.2. The experiments are detailed next and their tests are abbreviated by E$e.t$, with e as an experiment identification and t as a trial identification relative to e.

Base Our base version for experimentation is the algorithm described in Section 4.3 with a small adaptation of the constructive phase: instead of a single execution of the DG algorithm at line 2, the resulting solution is the best among `MaxDG` repetitions of it and a simple local search (as in lines 3 to 6 of Algorithm 5.1). The parameters used for this experiment are: `MaxDG` = 3, α = 100, $\beta = \log_2 n$, `maxT` = $k\sqrt{m}$, local search moves limit = $10n$, `MinDiff` = 5%, `MaxSize` = 5, `RandLimit` = 3, `Regen` = 32. This experiment resulted in an average relative deviation of 6.30% above the best known values.

Experiment 1 (E1) The goal of this experiment was to enhance local search by testing other heuristic combinations. We first introduced the evolutionary n-level, described above, as a second local search operator and tested three refinement heuristics for the first operator: Fiduccia-Mattheyses (E1.A), Tabu-based Fiduccia-Mattheyses (E1.B) and Lock-Gain (E1.C). These tests required extra parameters, because of the evolutionary n-level: $\alpha_{n\text{-level}}$ = 100, $\beta_{n\text{-level}} = \log_2 n$, `MinVertices` = $20k$, `MaxVertexWeight` = $1.5n/(20k)$, `InitParts` = 50, `LazyActivSize` = 60, $\alpha_{\texttt{ant}}$ = 0.5, $\beta_{\texttt{ant}}$ = 1, $\rho_{\texttt{ant}}$ = 0.1. A summary of the meaning of parameters can be found in Table 5.2. The resulting average relative deviations for tests A to C were, respectively: 1.77%, 1.43% and 2.78%. Interestingly, the LG heuristic produced very poor results inside our heuristic, different than the comparison to FM heuristic made by its authors, Kim and Moon (2004), which demonstrated the usefulness of such method as local search of a genetic algorithm. Compared to the pure FM heuristic, we kept the TFM as our local search algorithm since it provides the best solution quality.

Experiment 2 (E2) In this experiment we tested the effectiveness of the `combined_dg` procedure (line 6 of Algorithm 4.2) for the construction phase of our partitioner and variations of the `MaxDG` parameter. The base version of this experiment is the resulting of E1.B. Four tests were executed: setting `MaxDG` = 1 (E2.A); fixing the number of iterations of the loop starting at line 3 at 100 (E2.B), always keeping the best solution found; replacing lines 3 to 9 with solely line 7, i.e., dropping the `combined_dg` and applying just an iterative local search, also setting `MaxDG` = 10 (E2.C); using the `combined_dg` but with `MaxDG` = 10 (E2.D). The resulting average relative deviations for tests A to D were, respectively: 2.09%, 1.89%, 1.29% and 1.30%. The base version had `MaxDG` = 3 and it can be observed that increasing this parameter to 10 made the average relative deviation decrease by 0.1%, approximately. Also, tests C and D show us that the `combined_dg` does not alter the resulting quality, so we opted for the simpler version and kept E2.C as our next base version.

Experiment 3 (E3) Since the constructive phase changed after E2, we wondered if the refinement heuristic was chosen correctly, so in E3 we tested again the same local search strategies as in E1. The tabu based FM algorithm results are already available as E2.C. The simple FM and the LG heuristics were tested as E3.A and E3.B, respectively. Results had shown that the TFM is still better than the other two, which had 2.31% and 2.44% of average relative deviation, and so we kept our previously chosen heuristic.

Experiment 4 (S4) With the simplified constructive phase of our base version, E2.C, we tested the value of the `MaxDG` parameter. We tested each `MaxDG` $\in \{3, 5, 6, 10, 20\}$ as E4.A to E4.E, respectively, which resulted in average relative deviations of 1.25%, 1.05%, 1.17%, 1.29% and 1.56%. We kept the `MaxDG` = 5 as our best result and base version for the next experiment.

Experiment 5 (E5) Even with all the previous improvements, some instances still had poor results at constructive phase, such as noted in Section 4.3. This experiment had the purpose of testing a combination of the DG algorithm with versions of the n-level of Section 3.4.2 as a constructive operator. Trial E5.A introduced the line 2 of Algorithm 5.1 but this n-level used the F-cycle recoarsening scheme. Trial E5.B uses the same parameter settings as E5.A but with `MaxDG` = 3, as an attempt to decrease construction times. Trial E5.C uses the same parameter settings as E5.B but with the n-level modified to be faster, as described in Section 5.1.2. The resulting average relative deviations for these trials were, respectively: 1.01%, 0.88%, 0.90%. We decided to use the fast n-level algorithm as the new base version of our partitioner because it may improve the performance and the saved time may be used to perform more GRASP iterations.

Experiment 6 (E6) The purpose of this experiment was to simplify and calibrate the elite related options. Two regeneration schemes were tested. The first scheme is already described in Section 4.3. The other scheme replaces the rule to start regeneration to every `MaxDenied` times that the `elite_appeal` denies the inclusion of the current solution as a new elite. We expected that this would add more variability to the elite set faster than with the other scheme, and this could enhance solution quality. We also tested other values for the parameters `MinDiff` and `Regen`. The parameter details and resulting average relative deviations of each trial can be seen in Table 5.1. The averages for both regeneration schemes are similar, but the upper quartiles are lower in general for the new proposed scheme. This way, we decided to maintain E6.K as our new base version because it had the lowest average and median.

Experiment 7 (E7) The purpose of this experiment was to compare different combinations of parameters for the random walk criterion inside our Tabu based Fiduccia-Mattheyses local searches. Several combinations were tested: $\alpha = 30(m/n)$ and $\beta = \log n$ (E7.A), which resulted in an average relative deviation of 1.45%; $\alpha = 1$ and $\beta = \log n$ (E7.B), deviation of 1.34%; $\alpha = 4$ and $\beta = \log n$ (E7.C), deviation of 1.43%; $\alpha = 1$ and $\beta = m$ (E7.D), deviation of 0.72%. The random walk stop criterion

Exp.	Parameters	ARD (%)	Exp.	Parameters	ARD (%)
A	`MinDiff` = 1, `Regen` = 8	1.07	I	`MinDiff` = 1, `MaxDenied` = 5	0.95
B	`MinDiff` = 5, `Regen` = 8	0.99	J	`MinDiff` = 5, `MaxDenied` = 5	1.06
C	`MinDiff` = 1, `Regen` = 16	0.98	K	`MinDiff` = 1, `MaxDenied` = 8	**0.87**
D	`MinDiff` = 5, `Regen` = 16	1.04	L	`MinDiff` = 5, `MaxDenied` = 8	0.96
E	`MinDiff` = 1, `Regen` = 32	0.97	M	`MinDiff` = 1, `MaxDenied` = 16	1.01
F	`MinDiff` = 5, `Regen` = 32	0.90	N	`MinDiff` = 5, `MaxDenied` = 16	1.10
G	`MinDiff` = 1, `Regen` = 64	1.07	O	`MinDiff` = 1, `MaxDenied` = 32	1.09
H	`MinDiff` = 5, `Regen` = 64	1.07	P	`MinDiff` = 5, `MaxDenied` = 32	1.03

Table 5.1: Parameter configuration and resulting average relative deviations for each trial in E6. For trials with parameter `MaxDenied`, instead of `Regen`, the new regeneration scheme is applied.

was apparently a good idea, since it could cut moves that were unlikely to lead to better solutions. However, we then observed that high quality solutions would be achieved only with large values of α and this was an indication that this criterion was reducing the solution quality. With trial E7.D, we proved that this criterion is ineffective in our approach. By setting $\beta = m$, the Equation 3.13 is always false, since there may be no larger gain than m. Based on these results we chose to not use this criterion.

Experiment 8 (E8) With our previous finding, we investigated if the random walk criterion affects the solutions quality of the evolutionary n-level operator. Based on version E7.D, we tested $\alpha = 1$ and $\beta = \log n$ (E8.A), which resulted in an average relative deviation of 0.93; $\alpha = 4$ and $\beta = \log n$ (E8.B), deviation of 0.87%; $\alpha = 1$ and $\beta = m$ (E8.C), deviation of 0.82%; and $\alpha = 100$ and $\beta = \log n$ (E8.D), deviation of 0.85%. The results were very similar, and we identified no pattern related to this parameter, so we kept our parameters unchanged.

Experiment 9 (E9) This experiment had the goal of demonstrating the effectiveness of the lazy activation lists in our n-level operators and to tune its parameters. We tested the lazy activation list with size 60 (E9.A), as in E8.D, which resulted in an average relative deviation of 0.79%. In this experiment the trials with sizes 30 (E9.B), 90 (E9.C) and 2 (E9.D), produced average relative deviations of 0.88%, 0.82% and 0.96%. Therefore we kept this parameter at value 60 and observed that the lazy activation is useful for this multilevel strategy.

Experiment 10 (E10) The purpose of this experiment was to callibrate the evaporation rate ρ of our evolutionary n-level algorithm. Trial E9.A uses $\rho = 0.1$ and we tested $\rho = 0.05$ (E10.A), 0.15 (E10.B) and 0.2 (E10.C). They resulted in average relative deviation of 0.75%, 0.68% and 0.77%, respectively. We decided to use $\rho = 0.15$ since this value produced the best solutions.

Experiment 11 (E11) With this experiment we were trying to enhance the performance of our partitioner by reducing the number of initial partitions built by the n-level operators on their coarsest levels. We tested the parameter `InitParts` = 25

Table 5.2: Parameters and respective values used in the HYPAL.

Target	Parameter	Value	Short Description
DG	`MaxDG`	3	Number of iterations that DG algorithm is performed
n-level	α	100	Random walk stop criterion adjustment (evolutionary n-level algorithm)
	α	1	Random walk stop criterion adjustment (fast n-level algorithm)
	β	$\log n$	Random walk stop criterium adjustment
	`MinVertices`	$20k$	Coarsening stop criterium
	`MaxVertexWeight`	$1.5n/(20k)$	Avoid too heavy vertices
	`InitParts`	50	Number of attempts to find and initial partition
	`LazyActivSize`	60	Number of uncontracted vertices until activation is performed
Tabu	`maxT`	$k\sqrt{m}$	Maximum tabu tenure
	`LSMaxIter`	$10n$	Maximum number of iterations a LS may perform
Elite	`MinDiff`	1%	Minimum difference a solution must have to enter elite set
	`MaxSize`	5	Elite set size
	`RandLimit`	3	Number of best elite solutions that can be selected for PR
	`MaxDenied`	8	Number of denied appeals until a elite regeneration is performed
Ant	α	0.5	Importance of pheromone
	β	1	Importance of a priori knowledge
	ρ	15%	Evaporation rate

(E11.A) opposed to the 50 used by E10.B. This trial resulted in an average relative deviations of 1.00%, so we kept the settings of trial E10.B.

Experiment 12 (E12) This experiment aims at quantifying the contribution of the path-relinking to the average results. E12.A executes the GRASP of E10.B without the path-relinking phase, just keeping the best solution found among all its iterations. It results in an average relative deviation of 0.95%, which is worse than our base version.

5.3 Experimental Results

We conducted a series of experiments in order to evaluate the quality of the solutions generated by the HYPAL compared to the state-of-the-art. We used the graphs from Walshaw's partition archive and their best known partition values. All combinations of $k \in \{2, 4, 8, 16, 32, 64\}$ and $\epsilon \in \{1.01, 1.03, 1.05\}$ have been tested. Our experiments were run on Machine B and we set a time limit of two hours for each instance in each configuration of partition size and imbalance limit. With experiments of Section 5.2, we defined the resulting values for our parameters, which can be seen in Table 5.2.

We replicated each test 10 times. The minimum and average relative deviation results of all tests can be seen in Tables 5.3, 5.4 and 5.5 for imbalance limits of 1.01, 1.03 and 1.05, respectively. Absolute values can be found in Appendix B.

An overall view of the results shows us that the HYPAL is more effective for a small number of parts for all tested imbalances. This behavior can also be seen in Figure 5.3. The solution quality is also slightly higher as imbalance increases. We were able to improve the results of 11 cases, achieved the best known value in 164 of them and the remaining 430 of them had worse results than the BKV. The overall average relative deviation of HYPAL is 2.90% above BKV with the minimum results

Table 5.3: Minimum and average relative deviations of resulting cut sizes among all repetitions when $\epsilon = 1.01$. Improved or matched results are set in bold face. Some instances could not compute a result within the imposed time limit, so the number of repetitions that produce some result is marked close to the average result.

	$k = 2$		$k = 4$		$k = 8$		$k = 16$		$k = 32$		$k = 64$	
Instance	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg
add20	0,17	0,44	**-0,95**	**-0,95**	0,06	0,30	0,73	1,61	1,86	2,66	1,35	2,12
data	**0,00**	**0,00**	0,27	0,27	0,30	0,41	1,43	2,72	1,95	3,89	3,03	3,80
3elt	**0,00**	**0,00**	**0,00**	**0,00**	0,29	0,59	0,35	0,69	2,20	4,09	4,18	5,01
uk	**0,00**	**0,00**	2,50	2,50	2,50	3,50	7,64	9,38	10,04	13,73	12,99	15,96
add32	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,27
bcsstk33	**0,00**	**0,00**	**0,00**	0,00	**0,00**	**0,00**	0,34	0,56	1,51	2,36	2,56	3,25
whitaker3	**0,00**	**0,00**	**0,00**	**0,00**	0,15	0,23	0,83	1,08	2,28	2,99	4,59	5,47
crack	**0,00**	**0,00**	**0,00**	**0,00**	0,15	0,18	0,74	0,95	2,28	4,40	4,80	5,92
wing_nodal	**0,00**	**0,00**	0,08	0,11	0,26	0,41	0,70	1,02	2,63	2,88	4,75	6,63
fe_4elt2	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,61	1,20	1,75	3,73	5,54	4,33	6,14
vibrobox	**0,00**	**0,00**	0,04	0,11	0,14	0,63	2,05	2,48	5,39	5,89	4,90	5,91
bcsstk29	**0,00**	**0,00**	**0,00**	0,05	2,36	3,56	2,54	3,74	6,78	8,87	11,34	13,66
4elt	**0,00**	**0,00**	**0,00**	0,12	0,38	0,41	1,29	2,79	3,06	4,93	6,12	7,74
fe_sphere	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,12	0,25	1,53	1,94	5,35	6,77
cti	**0,00**	**0,00**	**0,00**	**0,00**	0,34	0,54	1,52	3,39	5,35	8,16	13,98	15,89
memplus	**-0,40**	0,21	2,25	3,07	3,69	4,13	5,00	7,33	10,08	13,32	8,54	9,51
cs4	**0,00**	0,25	2,38	3,56	4,88	6,86	8,39	12,14	18,70	25,96	23,03	25,61
bcsstk30	**0,00**	**0,00**	**0,00**	0,19	0,18	0,58	4,02	6,79	7,14	9,83	10,04	14,65
bcsstk31	**0,00**	**0,00**	0,01	0,35	0,97	1,60	8,26	13,20	13,67	19,08	15,21	18,56
fe_pwt	**0,00**	**0,00**	**0,00**	**0,00**	0,49	0,54	0,18	1,28	4,21	6,11	5,55	7,08
bcsstk32	**0,00**	**0,00**	0,45	0,68	2,05	4,47	5,73	8,42	16,17	21,19	14,20	27,59
fe_body	**0,00**	**0,00**	**0,00**	0,20	3,42	4,99	6,59	11,75	37,09	43,83	16,09	20,94
t60k	**0,00**	1,87	1,44	2,36	3,74	5,73	10,81	13,58	19,95	26,04	28,76	32,21
wing	**0,00**	0,45	3,66	4,84	6,18	9,21	13,77	15,41	16,89	20,49	36,52	41,22
brack2	**0,00**	**0,00**	**0,00**	0,04	1,59	2,15	2,79	3,61	5,56	7,57	9,03	11,02
finan512	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	5,45	13,57
fe_tooth	0,03	0,05	0,23	0,36	1,97	2,24	2,64	3,68	4,62	5,68	6,47	7,57
fe_rotor	**0,00**	0,02	0,49	2,36	1,53	3,25	2,33	3,49	4,08	5,36	6,69	9,07
598a	**0,00**	0,01	0,39	0,55	0,61	1,01	1,28	1,88	2,73	3,60	3,71	4,57(8)
fe_ocean	**-0,26**	**-0,03**	0,22	0,58	1,77	2,91	2,61	4,36	4,18	6,79	9,53	12,66
144	0,06	0,13	0,49	0,91	1,08	1,72	1,90	2,22	2,49	3,53	4,14	4,27(3)
wave	0,23	0,44	0,21	0,39	0,90	1,45	1,29	3,13	2,79	3,54	3,68	4,16(4)
m14b	0,03	0,09	1,52	1,95	2,31	2,69	2,11	3,28	2,94	4,28	–	– (0)
auto	0,29	0,46	0,81	1,20	2,07	5,11	1,55	2,91(7)	–	– (0)	–	– (0)
Improved:	2	1	1	1	0	0	0	0	0	0	0	0
Matched:	26	21	15	9	5	4	2	2	2	2	1	0
Worse:	6	12	18	24	29	30	32	32	31	31	31	32
Average:	0,00	0,13	0,49	0,76	1,36	2,12	3,02	4,44	6,78	9,05	9,09	11,53
STD:	0,11	0,34	0,94	1,26	1,56	2,29	3,36	4,35	7,68	9,43	7,99	9,40
Avg. of worses:	0,13	0,37	0,97	1,11	1,60	2,40	3,21	4,71	7,22	9,63	9,38	11,53
Minimum:	-0,40	-0,03	-0,95	-0,95	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,27
Lower quartile:	0,00	0,00	0,00	0,00	0,16	0,41	0,76	1,37	2,28	3,54	4,29	5,35
Median:	0,00	0,00	0,03	0,19	0,76	1,23	1,73	3,02	4,08	5,54	5,84	7,66
Upper quartile:	0,00	0,08	0,48	0,85	2,07	3,44	3,71	6,18	7,14	9,83	11,75	14,96
Maximum:	0,29	1,87	3,66	4,84	6,18	9,21	13,77	15,41	37,09	43,83	36,52	41,22

Table 5.4: Minimum and average relative deviations of resulting cut sizes among all repetitions when $\epsilon = 1.03$. Improved or matched results are set in bold face. Some instances could not compute a result within the imposed time limit, so the number of repetitions that produce some result is marked close to the average result.

	$k = 2$		$k = 4$		$k = 8$		$k = 16$		$k = 32$		$k = 64$	
Instance	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg
add20	**0,00**	0,27	**-1,22**	**-1,16**	**-0,12**	0,16	0,79	1,43	1,53	2,08	2,19	3,25
data	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,05	1,38	2,02	2,88	4,01	4,96	5,80
3elt	**0,00**	**0,00**	**0,00**	0,05	0,30	0,54	0,36	0,55	1,48	2,34	4,43	4,77
uk	**0,00**	**0,00**	**0,00**	1,28	**0,00**	2,05	4,32	7,70	6,25	8,50	8,82	10,38
add32	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,62	0,81
bcsstk33	**0,00**	**0,00**	**0,00**	0,01	0,03	0,04	0,22	0,39	1,13	1,99	2,65	3,26
whitaker3	**0,00**	**0,00**	**0,00**	**0,00**	0,62	0,80	0,74	1,26	1,81	2,62	4,39	5,06
crack	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,37	0,56	0,87	1,57	3,26	4,01	5,01
wing_nodal	**0,00**	**0,00**	0,08	0,10	0,07	0,14	0,79	1,02	2,57	2,90	3,73	4,68
fe_4elt2	**0,00**	**0,00**	0,29	0,47	0,34	0,62	1,21	1,56	3,52	4,69	5,30	6,24
vibrobox	**0,00**	**0,00**	**0,00**	0,01	0,10	0,35	1,03	1,86	3,76	5,35	4,43	5,24
bcsstk29	**0,00**	**0,00**	0,01	0,15	0,58	1,04	1,63	3,15	3,52	6,62	4,16	6,09
4elt	**0,00**	**0,00**	**0,00**	**0,00**	0,19	0,19	0,11	2,09	1,91	3,19	5,77	6,93
fe_sphere	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,24	0,35	1,22	1,58	3,85	4,96
cti	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,12	0,95	2,51	4,05	6,05	10,95	12,68
memplus	0,02	0,11	3,08	3,47	4,64	5,29	4,81	6,87	5,23	11,65	8,28	9,42
cs4	**0,00**	0,03	2,40	3,32	3,65	5,39	6,67	9,50	10,92	13,84	20,84	22,59
bcsstk30	**0,00**	**0,00**	0,02	0,16	0,01	0,11	1,04	4,75	6,54	10,99	7,66	13,74
bcsstk31	**0,00**	**0,00**	**0,00**	0,11	0,83	1,21	4,13	7,97	11,01	16,54	15,54	16,63
fe_pwt	**0,00**	**0,00**	**0,00**	**0,00**	0,28	0,36	0,94	1,13	4,24	5,81	6,42	7,08
bcsstk32	**0,00**	**0,00**	0,03	0,17	0,62	1,34	2,48	3,78	7,30	9,34	9,87	13,65
fe_body	**0,00**	**0,00**	**0,00**	**0,00**	0,59	1,78	4,35	5,87	14,80	17,24	19,84	21,76
t60k	**0,00**	0,28	0,49	2,91	2,67	4,23	10,35	11,53	16,82	18,35	23,72	26,49
wing	0,39	1,20	2,45	4,34	5,02	6,90	10,39	13,67	17,13	19,05	19,98	28,79
brack2	**0,00**	**0,00**	**0,00**	0,11	0,27	1,24	2,23	3,57	4,52	6,14	7,88	8,96
finan512	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,96	3,32
fe_tooth	0,03	0,05	0,24	0,48	2,22	2,67	2,55	3,76	4,85	6,08	6,53	7,77
fe_rotor	**0,00**	**0,00**	0,50	1,60	0,87	2,67	2,31	3,19	5,09	6,05	6,08	7,68
598a	**0,00**	0,00	0,40	0,53	1,01	1,35	1,73	2,10	2,69	3,52	4,35	4,72(9)
fe_ocean	**0,00**	**0,00**	**0,00**	**0,00**	0,15	0,89	2,73	4,05	4,26	6,02	9,06	11,73
144	0,03	0,21	0,40	0,90	1,99	2,47	1,61	2,20	3,27	4,29	3,73	4,30(6)
wave	0,40	0,47	0,37	0,53	0,48	1,38	1,73	2,83	2,17	3,43	3,29	3,81(3)
m14b	**0,00**	0,10	2,06	2,83	2,34	2,68	2,48	3,23	1,76	4,20(8)	4,08	4,08(1)
auto	0,22	0,45	1,07	2,27	1,55	4,82	1,35	4,40	5,11	5,11(1)	–	– (0)
Improved:	0	0	1	1	1	0	0	0	0	0	0	0
Matched:	28	23	17	11	7	3	2	2	2	2	0	0
Worse:	6	11	16	22	26	31	32	32	32	32	33	33
Average:	0,03	0,09	0,37	0,73	0,92	1,57	2,30	3,56	4,85	6,55	7,53	9,14
STD:	0,10	0,23	0,86	1,27	1,35	1,82	2,56	3,29	4,41	5,18	5,94	6,99
Avg. of worses:	0,18	0,29	0,87	1,17	1,21	1,72	2,44	3,79	5,15	6,96	7,53	9,14
Minimum:	0,00	0,00	-1,22	-1,16	-0,12	0,00	0,00	0,00	0,00	0,00	0,62	0,81
Lower quartile:	0,00	0,00	0,00	0,00	0,01	0,17	0,79	1,31	1,84	3,20	4,01	4,72
Median:	0,00	0,00	0,00	0,11	0,32	0,97	1,49	2,67	3,64	5,23	5,30	6,24
Upper quartile:	0,00	0,05	0,39	0,81	0,98	2,37	2,53	4,31	5,20	8,03	8,82	11,73
Maximum:	0,40	1,20	3,08	4,34	5,02	6,90	10,39	13,67	17,13	19,05	23,72	28,79

Table 5.5: Minimum and average relative deviations of resulting cut sizes among all repetitions when $\epsilon = 1.05$. Improved or matched results are set in bold face. Some instances could not compute a result within the imposed time limit, so the number of repetitions that produce some result is marked close to the average result.

	$k = 2$		$k = 4$		$k = 8$		$k = 16$		$k = 32$		$k = 64$	
Instance	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg	Min	Avg
add20	**0,00**	0,13	**-0,62**	**-0,62**	**-0,36**	0,01	0,69	1,35	1,79	2,52	1,06	1,76
data	0,55	0,55	**0,00**	**0,00**	**0,00**	**0,00**	0,28	0,69	2,64	3,08	3,56	4,56
3elt	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,24	0,18	0,41	1,83	2,57	3,00	3,51
uk	**0,00**	**0,00**	**0,00**	**0,00**	1,33	3,73	5,84	6,93	7,63	8,60	8,63	9,90
add32	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,19
bcsstk33	**0,00**	**0,00**	0,04	0,04	0,02	0,03	0,45	0,59	1,30	2,20	2,18	2,65
whitaker3	**0,00**	**0,00**	**0,00**	0,19	0,16	0,71	1,12	1,43	2,14	2,73	3,69	4,89
crack	**0,00**	**0,00**	**0,00**	**0,00**	0,15	0,24	0,19	0,69	1,45	2,12	3,01	4,39
wing_nodal	0,06	0,06	**0,00**	0,04	0,09	0,18	1,08	1,38	2,52	2,79	4,42	4,80
fe_4elt2	**0,00**	**0,00**	**0,00**	0,18	0,17	0,52	1,43	2,04	3,18	4,52	5,45	7,04
vibrobox	**0,00**	**0,00**	**0,00**	0,02	0,03	0,10	1,46	2,13	4,24	5,87	4,33	4,84
bcsstk29	**0,00**	**0,00**	**0,00**	0,15	0,49	0,76	0,53	2,66	4,29	6,53	4,49	6,29
4elt	**0,00**	**0,00**	**0,00**	**0,00**	0,19	0,21	0,90	1,38	2,68	3,40	5,51	6,59
fe_sphere	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,24	0,39	0,49	1,93	3,91	5,10
cti	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,05	1,44	2,27	3,58	5,73	10,29	11,70
memplus	**-0,09**	0,15	2,63	3,32	4,90	5,45	4,80	6,71	8,13	11,33	9,58	10,06
cs4	**0,00**	0,14	1,98	3,16	2,39	4,54	7,39	9,79	11,96	13,27	19,81	21,63
bcsstk30	**0,00**	**0,00**	**0,00**	0,07	0,03	0,16	2,73	5,44	6,48	9,04	10,56	12,54
bcsstk31	**-0,34**	**-0,12**	**-0,04**	0,06	0,16	0,44	3,45	6,10	12,04	16,13	12,54	14,51
fe_pwt	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	0,02	0,29	0,90	4,00	5,50	5,74	6,36
bcsstk32	**0,00**	0,68	**0,00**	0,48	**-0,61**	1,24	2,22	4,02	4,21	6,83	8,33	13,82
fe_body	**0,00**	**0,00**	1,70	1,70	0,59	0,87	3,21	4,99	10,68	13,10	14,47	16,34
t60k	**0,00**	**0,00**	1,54	2,51	4,31	6,24	9,78	11,92	15,41	18,30	23,31	25,54
wing	0,26	0,43	1,76	3,49	4,26	6,45	10,57	13,06	16,06	20,75	18,26	24,79
brack2	**0,00**	**0,00**	**0,00**	0,05	0,44	0,66	1,90	2,55	4,91	6,20	6,10	7,86
finan512	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	3,26	4,66
fe_tooth	**0,00**	0,01	0,18	0,31	1,07	2,09	3,63	4,37	5,10	6,32	6,20	7,35
fe_rotor	**0,00**	**0,00**	0,06	1,14	1,08	2,44	0,76	2,69	3,47	5,69	5,66	7,65(8)
598a	**0,00**	**0,00**	0,36	0,49	0,90	1,42	1,44	2,02	2,49	3,45	4,32	5,09(9)
fe_ocean	**0,00**	**0,00**	**0,00**	0,02	1,49	1,74	2,17	3,57	4,99	6,07	8,72	10,33
144	0,03	0,29	0,43	1,18	1,68	2,06	2,09	2,72	2,34	3,77	4,00	4,57(5)
wave	0,29	0,38	0,54	0,64	0,65	1,35	2,17	2,91	2,74	3,56	4,02	4,25(5)
m14b	**0,00**	0,15	1,07	2,16	2,24	2,70	2,77	3,92	2,95	4,44	–	– (0)
auto	0,22	0,43	1,31	2,19	1,60	5,45	1,50	3,21(8)	–	– (0)	–	– (0)
Improved:	2	1	2	1	2	0	0	0	0	0	0	0
Matched:	26	21	19	10	7	4	2	2	2	2	1	0
Worse:	6	12	13	23	25	30	32	32	31	31	31	32
Average:	0,03	0,10	0,38	0,68	0,87	1,53	2,32	3,39	4,78	6,31	7,14	8,61
STD:	0,14	0,19	0,74	1,11	1,35	1,97	2,61	3,22	4,16	5,05	5,45	6,28
Avg. of worses:	0,24	0,28	1,05	1,03	1,22	1,74	2,46	3,60	5,09	6,72	7,37	8,61
Minimum:	-0,34	-0,12	-0,62	-0,62	-0,61	0,00	0,00	0,00	0,00	0,00	0,00	0,19
Lower quartile:	0,00	0,00	0,00	0,00	0,01	0,12	0,57	1,36	2,34	2,79	3,85	4,64
Median:	0,00	0,00	0,00	0,07	0,18	0,69	1,45	2,61	3,47	5,50	5,48	6,47
Upper quartile:	0,00	0,14	0,41	1,01	1,27	2,09	2,76	4,28	5,10	6,83	8,93	10,67
Maximum:	0,55	0,68	2,63	3,49	4,90	6,45	10,57	13,06	16,06	20,75	23,31	25,54

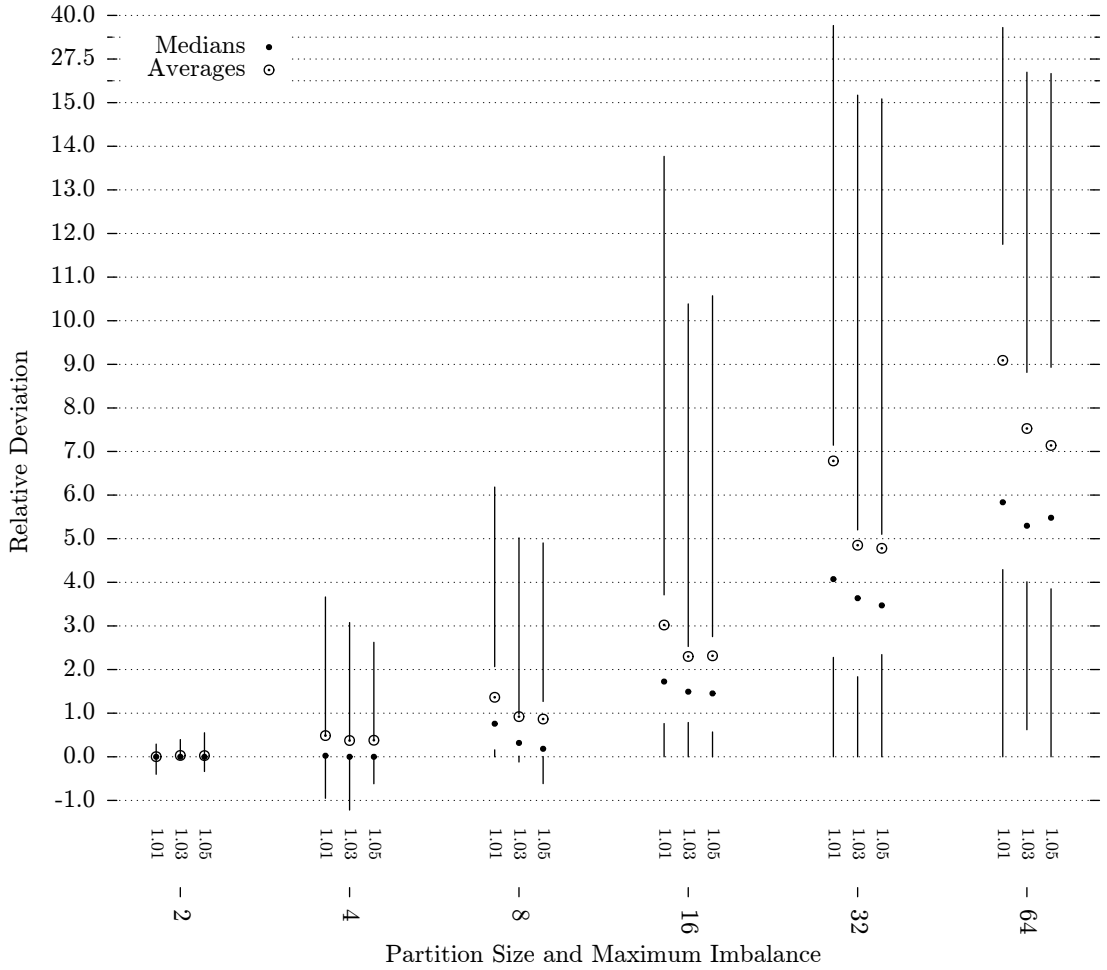


Figure 5.3: Quartiles and average of minimum results for all tested configurations of partition size and imbalance. Values above 15 are scaled different for better visualization.

and 3.91% above BKV with the average results.

The increase of average results, for increasing k, is directly related to the number of GRASP iterations, which decreases as the partition size increases. This is still open to improvements and gains in performance that can be obtained. However, we cannot confirm if this would be enough to make the HYPAL better in such cases. On the other hand, we have some evidence that indicates that the combinations of approaches in our algorithm has some merit. If we analyze the executions with a single partition size, the number of GRASP iterations also decreases as the graph size increases. The small graphs, that allow a relative large number of iterations to be performed, have good results and even some improvements. If we analyze the executions with a single graph, the increase in partition size decreases the number of iterations and increases the average results.

The problem with poor results for some specific instances, as highlighted in Section 4.3, have been reduced significantly but it still remains. This can be seen in instances `cs4`, `fe_body`, `t60k` and `wing` for $k \geq 16$. The inclusion of a multilevel algorithm as constructive and refinement operator improved the average results, as seen in Section 5.2, but apparently it was not enough to overcome the some poor

results. These instances must be investigated more closely to understand where our algorithm is failing.

The results presented in Tables 5.3, 5.4 and 5.5 were based on the best known partitions published on Walshaw's partition archive in mid October, 2012. All experiments before these have been compared with results published earlier, with last archive update by mid October, 2011. If we compare the HYPAL minimum results with such older best known partitions, our algorithm would have an overall average relative deviation of 2.35% above BKVs; with 45 improved cases, 166 partitions that we achieved the same BKV, and 394 worsened cases. Although the number of improved cases has decreased, the results using the older BKVs are not much better than the ones based on recent approaches, therefore, we are still competitive with them.

6 CONCLUDING REMARKS

In this dissertation, we studied a metaheuristic for the graph partitioning problem. This problem may be defined with one of several objective functions but we focused on the minimization of cut size. We presented a review of the state-of-the-art algorithms for graph partitioning such as constructive and refinement heuristics and multilevel technique. We implemented some of these algorithms, among others are the Differential Greedy constructive heuristic, the Fiduccia-Mattheyses refinement heuristic and an n-level graph partitioner proposed by Osipov and Sanders (2010); and proposed a GRASP with path-relinking for the k-way GPP.

The HYPAL, our approach to solve the problem is a hybrid metaheuristic of a GRASP with path-relinking and an adapted n-level partitioner, used in constructive and refinement phases of the GRASP. We proposed an evolutionary n-level algorithm, based on ideas of the Ant Colony metaheuristic, that guides the partitioning cycle using information of previous solutions. The minimum cut sizes of our algorithm have an average relative deviation of 2.9% above the best known values of 34 benchmark instances from the Walshaw's Graph Partition Archive with partition sizes $k \in \{2, 4, 8, 16, 32, 64\}$ and maximum imbalance of 1%, 3% or 5%. We found new best known solutions for 11 cases and achieved equal results in 164 of them.

The improvements, especially for the small instances, are, in our opinion, significant since these instances requires less machine resources than larger ones and have been tested with other partitioners for almost a decade, after their best known values have been found. Almost all best known values at this time have been obtained by Sanders and Schulz (2011), with a time limit of two hours using a parallel algorithm in 16 processors, by Sanders and Schulz (2012), with a time limit from 30 minutes to one hour on 32 processors, or a still unpublished implementation of MAGP by Frank Schneider.

6.1 Ideas for future research

The poor results for large partition sizes compared to the small sizes make us believe that the number of GRASP iterations must be increased. Performance optimizations are required in our implementation to confirm this hypothesis. As a preliminary test, we can increase the time limit to be proportional to the partition size. Depending on the results, a parallelized version of the algorithm could be defined and implemented.

There are some instances, mentioned in Section 5.3, that always have poor results. A closer investigation may show points of failure of the algorithms and maybe lead to propose new heuristics that circumvent such difficulties.

The Tabu based Fiduccia-Mattheyses had proven to be more effective than other tested refinement heuristics. However, the research of good stopping criteria to avoid ineffective moves would be interesting since the saved time could be used in other parts of the algorithm.

REFERENCES

ALPERT, C. J.; KAHNG, A. B. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, v. 19, n. 1–2, p. 1–81, 1995.

ARORA, S.; RAO, S.; VAZIRANI, U. Geometry, flows, and graph-partitioning algorithms. *Commun. ACM*, ACM, New York, NY, USA, v. 51, p. 96–105, October 2008.

BAKOGLU, H. *Circuits, interconnections, and packaging for VLSI*. [S.l.]: Addison-Wesley Pub. Co., 1990. (VLSI systems series).

BALCH, M. *Complete Digital Design: A Comprehensive Guide to Digital Electronics and Computer System Architecture*. New York: McGraw-Hill, 2003.

BARAHONA, F. et al. An application of combinatorial optimization to statistical physics and circuit layout design. *Oper. Res.*, INFORMS, v. 36, p. 493–513, May 1988.

BATTITI, R.; BERTOSSI, A. Differential greedy for the 0-1 equicut problem. In: *in Proceedings of the DIMACS Workshop on Network Design: Connectivity and Facilities Location*. [S.l.]: American Mathematical Society, 1997. p. 3–21.

BATTITI, R.; BERTOSSI, A. A. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 48, p. 361–385, 1999.

BENLIC, U.; HAO, J. An effective multilevel memetic algorithm for balanced graph partitioning. In: *22nd IEEE International Conference on Tools with Artificial Intelligence*. [S.l.]: IEEE Computer Society, 2010. p. 121–128.

BENLIC, U.; HAO, J. An effective multilevel tabu search approach for balanced graph partitioning. *Computers & OR*, v. 38, n. 7, p. 1066–1075, 2011.

BENLIC, U.; HAO, J. A multilevel memetic approach for improving graph k-partitions. *IEEE Transactions on Evolutionary Computation*, IEEE Computer Society, Los Alamitos, CA, USA, v. 15, p. 624–642, 2011.

BISWAS, R.; STRAWN, R. C. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, v. 13, n. 6, p. 437–452, 1994.

BOULLE, M. Compact mathematical formulation for graph partitioning. *Optimization and Engineering*, v. 5, p. 315–333, 2004.

BUI, T. N.; JONES, C. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 42, p. 153–159, May 1992.

BUI, T. N.; JONES, C. A heuristic for reducing fill-in in sparse matrix factorization. In: *SIAM Conference on Parallel Processing for Scientific Computing 1993*. [S.l.: s.n.], 1993. p. 445–452.

BUI, T. N.; MOON, B. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 45, p. 841–855, 1996.

CATALYUREK, U.; AYKANAT, C. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In: *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*. London, UK: Springer-Verlag, 1996. (IRREGULAR '96), p. 75–86.

CHARDAIRE, P.; BARAKE, M.; MCKEOWN, G. P. A probe-based heuristic for graph partitioning. *IEEE Transactions on Computers*, IEEE Computer Society, Washington, DC, USA, v. 56, p. 1707–1720, December 2007.

CHEVALIER, C.; SAFRO, I. Comparison of coarsening schemes for multilevel graph partitioning. In: STüTZLE, T. (Ed.). *LION*. [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5851), p. 191–205.

FEO, T. A.; RESENDE, M. G. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, v. 8, n. 2, p. 67–71, 1989.

FIDUCCIA, C. M.; MATTHEYSES, R. M. A linear-time heuristic for improving network partitions. In: *Proceedings of the 19th Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 1982. (DAC '82), p. 175–181.

FJäLLSTROM, P. Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science*, v. 3, n. 10, 1998.

GALINIER, P.; BOUJBEL, Z.; FERNANDES, M. C. An efficient memetic algorithm for the graph partitioning problem. *Annals of Operations Research*, Springer Netherlands, v. 191, p. 1–22, 2011.

GILBERT, J.; MILLER, G.; TENG, S. Geometric mesh partitioning: implementation and experiments. *Parallel Processing Symposium, International*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 418, 1995.

GLOVER, F. Tabu search — part I. *ORSA Journal on Computing*, v. 1, p. 190 – 206, 1989.

GLOVER, F. Tabu search and adaptive memory programing – advances, applications and challenges. In: *Interfaces in Computer Science and Operations Research*. [S.l.]: Kluwer, 1996. p. 1–75.

GLOVER, F.; LAGUNA, M. *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

GUSFIELD, D. Partition-distance: A problem and class of perfect graphs arising in clustering. *Information Processing Letters*, v. 82, n. 3, p. 159–164, May 2002.

HENDRICKSON, B.; KOLDA, T. G. Graph partitioning models for parallel computing. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 26, p. 1519–1534, November 2000.

HENDRICKSON, B.; LELAND, R. A multilevel algorithm for partitioning graphs. In: *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1995. (Supercomputing '95).

HOLTGREWE, M.; SANDERS, P.; SCHULZ, C. Engineering a scalable high quality graph partitioner. *CoRR*, abs/0910.2004, 2009. Informal publication.

JOHANNES, F. M. Partitioning of VLSI circuits and systems. In: *Proceedings of the 33rd annual Design Automation Conference*. New York, NY, USA: ACM, 1996. (DAC '96), p. 83–87.

JOHNSON, D. S. et al. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 37, p. 865–892, October 1989.

KARGER, D. R.; STEIN, C. A new approach to the minimum cut problem. *Journal of the ACM*, ACM, New York, NY, USA, v. 43, p. 601–640, July 1996.

KARYPIS, G.; KUMAR, V. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System*. 1995. http://www.cs.umn.edu/~metis.

KARYPIS, G.; KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 20, p. 359–392, December 1998.

KERNIGHAN, B. W.; LIN, S. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, v. 49, n. 2, p. 291–307, February 1970.

KIM, Y.; MOON, B. Lock-gain based graph partitioning. *Journal of Heuristics*, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, p. 37–57, January 2004.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*, v. 220, n. 4598, p. 671–680, 1983.

KONOVALOV, D. A.; LITOW, B.; BAJEMA, N. Partition-distance via the assignment problem. *Bioinformatics*, v. 21, n. 10, p. 2463–2468, 2005.

LAGUNA, M.; FEO, T. A.; ELROD, H. C. A greedy randomized adaptive search procedure for the two-partition problem. *Operations Research*, v. 42, n. 4, p. 677–687, July 1994.

LAGUNA, M.; MARTI, R. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS J. on Computing*, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 11, n. 1, p. 44–52, jan. 1999.

LAWLER, E. *Combinatorial optimization: networks and matroids*. [S.l.]: Holt, Rinehart and Winston, 1976.

MARTIN, O.; OTTO, S. W.; FELTEN, E. W. Large-step markov chains for the traveling salesman problem. *Complex Systems*, v. 5, p. 299–326, 1991.

MAUE, J.; SANDERS, P. Engineering algorithms for approximate weighted matching. In: DEMETRESCU, C. (Ed.). *Experimental Algorithms*. [S.l.]: Springer Berlin / Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4525). p. 242–255.

MEYERHENKE, H.; MONIEN, B.; SAUERWALD, T. A new diffusion-based multilevel algorithm for computing graph partitions. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 69, p. 750–761, September 2009.

MONTGOMERY, D. C. *Design and Analysis of Experiments*. [S.l.]: John Wiley & Sons, 2008. (Student solutions manual).

MUCHA, M.; SANKOWSKI, P. Maximum matchings via gaussian elimination. In: *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 2004. p. 248–255.

MUNKRES, J. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, SIAM, v. 5, n. 1, p. 32–38, 1957.

OSIPOV, V.; SANDERS, P. n-level graph partitioning. In: *Proceedings of the 18th annual European conference on Algorithms: Part I*. Berlin, Heidelberg: Springer-Verlag, 2010. (ESA'10), p. 278–289.

PARLETT, B. *The Symmetric Eigenvalue Problem*. [S.l.]: Society for Industrial and Applied Mathematics, 1987. (Classics in Applied Mathematics).

PELLEGRINI, F. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In: KERMARREC, A.; BOUGé, L.; PRIOL, T. (Ed.). *Euro-Par 2007 Parallel Processing*. [S.l.]: Springer Berlin / Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4641). p. 195–204.

PELLEGRINI, F. *SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package*. 2007. http://www.labri.fr/perso/pelegrin/scotch.

RESENDE, M. G. C.; RIBEIRO, C. C. GRASP and path-relinking: Recent advances and applications. In: IBARAKI, T.; YOSHITOMI, Y. (Ed.). *Proceedings of the Fifth Metaheuristics International Conference (MIC2003)*. [S.l.: s.n.], 2003.

RESENDE, M. G. C.; WERNECK, R. F. A hybrid heuristic for the p-median problem. *Journal of Heuristics*, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, n. 1, p. 59–88, jan. 2004.

RIBEIRO, C. C.; UCHOA, E.; WERNECK, R. F. A hybrid grasp with perturbations for the steiner problem in graphs. *INFORMS JOURNAL ON COMPUTING*, v. 14, p. 200–2, 2001.

ROLLAND, E.; PIRKUL, H.; GLOVER, F. Tabu search for graph partitioning. *Annals of Operations Research*, Springer Netherlands, v. 63, p. 209–232, 1996.

SAFRO, I.; RON, D.; BRANDT, A. Multilevel algorithms for linear ordering problems. *J. Exp. Algorithmics*, ACM, New York, NY, USA, v. 13, p. 4:1.4–4:1.20, February 2009.

SANDERS, P.; SCHULZ, C. Engineering multilevel graph partitioning algorithms. *CoRR*, abs/1012.0006, 2010.

SANDERS, P.; SCHULZ, C. Distributed evolutionary graph partitioning. *CoRR*, abs/1110.0477, 2011.

Sanders, P.; Schulz, C. Think Locally, Act Globally: Perfectly Balanced Graph Partitioning. *ArXiv e-prints*, out. 2012. ArXiv:1210.0477.

SCHLOEGEL, K.; KARYPIS, G.; KUMAR, V. Graph partitioning for high-performance scientific simulations. In: DONGARRA, J. et al. (Ed.). *Sourcebook of parallel computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. p. 491–541.

SHI, J.; MALIK, J. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 22, p. 888–905, August 2000.

SIMON, H. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, v. 2, n. 2-3, p. 135 – 148, 1991. Parallel Methods on Large-scale Structural Analysis and Physics Applications.

SŁOWIK, A.; BIAŁKO, M. Partitioning of VLSI circuits on subcircuits with minimal number of connections using evolutionary algorithm. In: RUTKOWSKI, L. et al. (Ed.). *Artificial Intelligence and Soft Computing – ICAISC 2006*. [S.l.]: Springer Berlin / Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4029). p. 470–478.

TORIL, M. et al. An adaptive multi-start graph partitioning algorithm for structuring cellular networks. *Journal of Heuristics*, Springer Netherlands, p. 1–21, 2010.

WALSHAW, C. *The Graph Partitioning Archive*. 2000. http://staffweb.cms.gre.ac.uk/~c.walshaw/partition.

WALSHAW, C. Multilevel refinement for combinatorial optimisation: Boosting metaheuristic performance. In: BLUM, C. et al. (Ed.). *Hybrid Metaheuristics*. [S.l.]: Springer, 2008, (Studies in Computational Intelligence, v. 114). p. 261–289.

WALSHAW, C.; CROSS, M. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 22, p. 63–80, January 2000.

WALSHAW, C.; CROSS, M. *JOSTLE: parallel multilevel graph-partitioning software — an overview*. 2007.

ZHA, H. et al. Bipartite graph partitioning and data clustering. In: *Proceedings of the tenth international conference on Information and knowledge management*. New York, NY, USA: ACM, 2001. (CIKM '01), p. 25–32.

APPENDIX A UM ESTUDO DO PROBLEMA DE PARTICIONAMENTO DE GRAFOS EM k-PARTES

O problema de particionamento balanceado de grafos consiste em encontrar uma partição de tamanho k dos vértices de um grafo, sujeito a uma função objetivo e uma restrição de balanceamento. Essa restrição limita a cardinalidade máxima do tamanho de cada parte e, usualmente, objetiva-se minimizar a soma dos pesos das arestas que ligam vértices em partições distintas. Outras funções objetivo estão definidas na literatura mas o corte mínimo é a função mais estudada. O particionamento de grafos é um problema de otimização combinatória NP-difícil (BUI; JONES, 1992).

Formalmente, considere um grafo $G = (V, E, c, \omega)$ (onde $c(u)$ e $\omega(\{u, v\})$ representam as funções de peso positivo de um vértice e de uma aresta, respectivamente), conjuntos $P_0, \ldots, P_{k-1}$ tal que $P_i \subseteq V$, um número real $1 \leq \epsilon < k$, o conjunto de arestas do corte $C = \{\{u, v\} \in E \mid u \in P_i, v \in P_j, 0 \leq i < j \leq k-1\}$ e o tamanho do corte $\Theta(P) = \sum_{\{u,v\} \in C} \omega(\{u, v\})$. O problema do particionamento balanceado de grafos pode ser descrito com o seguinte modelo:

$$\textbf{minimize} \quad \Theta(P) \tag{A.1a}$$

$$\textbf{subject to} \quad 0 < \sum_{u \in P_i} c(u) \leq \left\lfloor \frac{\epsilon}{k} \sum_{v \in V} c(v) \right\rfloor \qquad \forall i \in [0, k-1] \tag{A.1b}$$

$$\bigcup_{0 \leq i < k} P_i = V \tag{A.1c}$$

$$P_i \cap P_j = \varnothing \qquad \forall i, j \in [0, k-1] \mid i \neq j \tag{A.1d}$$

Esse modelo descreve o problema para k partes. Entretanto, na literatura é mais comum encontrar técnicas que resolvem o caso $k = 2$. Nesse caso o problema é conhecido por biparticionamento de grafos e é mais simples de resolver. Para $k > 2$, é possível realizar uma implementação que considera esses casos diretamente ou então aplicar o biparticionador recursivamente, embora nesse caso a qualidade da partição tende a decair com o aumento do número de partes (PELLEGRINI, 2007a).

No particionamento recursivo, quando $k = 2^l$, para $l \in \mathbb{N}^+$, não é preciso nenhum cuidado especial para usar o biparticionador. Nos demais casos, as partes precisam ser proporcionais ao número de subdivisões que irão abrigar. Por exemplo, para $k = 3$ e $\epsilon = 1$, a primeira iteração deverá gerar uma parte $|P_0| = n/3$ e outra $|P_1| = 2n/3$; a segunda iteração irá dividir a parte $P_1 = P_2 \cup P_3$ tal que $|P_2| = |P_3| = n/3$.

Além do número de partes, é possível lidar com o balanceamento de cada uma

delas. O modelo apresentado é genérico pois consegue estipular o limite do tamanho de cada partição. No caso em que $\epsilon = 1$ o problema é chamado de particionamento totalmente balanceado. Normalmente, o caso balanceado é mais demorado por ser muito restrito e muitas vezes desnecessário, já que a maioria das aplicações permite um certo desbalanceamento das partes. Desde que $1 \leq \epsilon < k$, um desbalanceamento maior não altera a classe de complexidade do problema (BUI; JONES, 1992).

Existem outras funções objetivo além do corte mínimo, como, por exemplo, a minimização do volume de comunicação total e do volume de comunicação máximo. Essas funções definem melhor o custo de comunicação entre processadores quando o particionamento é aplicado para este fim. Entretanto, essas funções não possuem boas heurísticas que as resolvem, sendo mais comum a sua modelagem de comunicação por meio de hipergrafos, que são mais pesquisados do que essas funções.

O Capítulo 2 apresenta essas funções objetivos mais detalhadamente, uma definição formal do problema, assim como alguns exemplos extendidos de aplicações. Explicamos o uso do particionamento de grafos em redes de telefonia móvel, simulações científicas e particionamento de circuitos VLSI.

No Capítulo 3 relatamos diversos algoritmos utilizados para criar particionadores. Técnicas fundamentais como a construção e refinamento de soluções são detalhadas com a definição de algoritmos como o Differential Greedy (BATTITI; BERTOSSI, 1997) e o Kernighan-Lin (KERNIGHAN; LIN, 1970), bem como a sua versão mais otimizada por Fiduccia e Mattheyses (1982). No mesmo capítulo também é apresentada a técnica multinível de particionamento, onde um grafo é contraído diversas vezes para versões mais simplificadas, sofre um particionamento inicial e então é descontraído e refinado. Além disso, explicamos o funcionamento de alguns particionadores para compreender a sua essência e propor as nossas ideias.

O Capítulo 4 relata nossas experiências iniciais através da definição de duas versões de um GRASP com path-relinking, uma para o problema de bipparticionamento perfeitamente balanceado de grafos e outra para o problema de k-particionamento balanceado.

Um particionador híbrido, que une um GRASP com path-relinking com um particionador n-level, é proposto no Capítulo 5. Ele é chamado de HYPAL (HYbrid k-way graph Partitioning ALgorithm) e evoluiu de ideias detalhadas no capítulo anterior. Diversos experimentos foram executados e relatados de forma a fundamentar as nossas escolhas para os elementos do nosso algoritmo. Apresentamos resultados experimentais utilizando 34 instâncias disponíveis no arquivo de partições de Walshaw (WALSHAW, 2000). Todas as combinações de $k \in \{2, 4, 8, 16, 32, 64\}$ e $\epsilon \in \{1.01, 1.03, 1.05\}$ foram testadas. Obtivemos novos melhores resultados conhecidos em 11 casos e alcançamos os mesmos melhores valores em 164 outros. Utilizando os valores mínimos, o HYPAL teve um desvio relativo médio de 2.9% acima dos melhores valores conhecidos. O bom desempenho geral é devido a uma combinação de diversas técnicas. Porém, para grafos maiores, a implementação atual ainda é lenta, embora tenha potencial para melhorar. Pretendemos otimizá-lo ainda mais e também realizar mais experimentos visando aprimorar algumas de suas falhas, mas os resultados são animadores. Acreditamos que podemos realizar uma contribuição qualitativa já que os demais melhores valores conhecidos foram obtidos principalmente com algoritmos paralelos que utilizam mais de 16 processadores.

APPENDIX B ABSOLUTE VALUES OF HYPAL EXPERIMENTAL RESULTS

The Tables B.1, B.2 and B.3 present the minimum and average cut sizes found in 10 repetitions of the HYPAL for each instance and configuration of partition size and maximum imbalance. Best known values are as published in mid October, 2012, on Walshaw's partition archive.

Table B.1: Minimum and average cut size results among all repetitions compared to the best known values (BKV) when $\epsilon = 1.01$. Best known values are set in bold face.

	$k = 2$			$k = 4$			$k = 8$			$k = 16$			$k = 32$			$k = 64$		
Instance	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg
add20	**586**	587	588,60	1158	**1147**	1147,00	**1689**	1690	1694,00	**2047**	2062	2080,00	**2369**	2413	2432,00	**2966**	3006	3028,80
data	**188**	**188**	188,00	**376**	377	377,00	**656**	658	658,70	**1121**	1137	1151,50	**1799**	1834	1868,90	**2839**	2925	2947,00
3elt	**89**	**89**	89,00	**199**	**199**	199,00	**340**	341	342,00	**568**	570	571,90	**954**	975	993,00	**1532**	1596	1608,70
uk	**19**	**19**	19,00	**40**	41	41,00	**80**	82	82,80	**144**	155	157,50	**249**	274	283,20	**408**	461	473,10
add32	**10**	**10**	10,00	**33**	**33**	33,00	**66**	**66**	66,00	**117**	**117**	117,00	**212**	**212**	212,00	**486**	**486**	487,30
bcsstk33	**10097**	**10097**	10097,00	**21338**	**21338**	21338,20	**34175**	**34175**	34175,00	**54505**	54693	54810,00	**77195**	78361	79013,70	**106932**	109670	110407,40
whitaker3	**126**	**126**	126,00	**380**	**380**	380,00	**654**	655	655,50	**1084**	1093	1095,70	**1667**	1705	1716,80	**2483**	2597	2618,80
crack	**183**	**183**	183,00	**362**	**362**	362,00	**676**	677	677,20	**1081**	1089	1091,30	**1669**	1707	1742,50	**2523**	2644	2672,40
wing_nodal	**1695**	**1695**	1695,00	**3559**	3562	3562,80	**5401**	5415	5423,10	**8302**	8360	8386,70	**11733**	12042	12070,80	**15736**	16483	16778,60
fe_4elt2	**130**	**130**	130,00	**349**	**349**	349,00	**603**	**603**	606,70	**1000**	1012	1017,50	**1608**	1668	1697,10	**2472**	2579	2623,90
vibrobox	**10310**	**10310**	10310,00	**18943**	18950	18962,90	**24422**	24456	24576,90	**31791**	32443	32580,80	**39477**	41604	41802,00	**46541**	48821	49291,40
bcsstk29	**2818**	**2818**	2818,00	**8029**	**8029**	8033,30	**13891**	14219	14385,40	**21694**	22244	22505,90	**34739**	37095	37819,60	**55147**	61399	62682,30
4elt	**138**	**138**	138,00	**320**	**320**	320,40	**532**	534	534,20	**927**	939	952,90	**1538**	1585	1613,80	**2549**	2705	2746,30
fe_sphere	**386**	**386**	386,00	**766**	**766**	766,00	**1152**	**1152**	1152,00	**1708**	1710	1712,20	**2479**	2517	2527,20	**3534**	3723	3773,30
cti	**318**	**318**	318,00	**944**	**944**	944,00	**1746**	1752	1755,50	**2768**	2810	2861,80	**4001**	4215	4327,50	**5594**	6376	6483,10
memplus	5478	**5456**	5489,40	**9448**	9661	9738,20	**11776**	12211	12262,50	**12928**	13575	13876,10	**13992**	15403	15855,10	**16269**	17659	17815,60
cs4	**366**	**366**	366,90	**925**	947	957,90	**1434**	1504	1532,40	**2087**	2262	2340,30	**2903**	3446	3656,50	**3982**	4899	5001,60
bcsstk30	**6335**	**6335**	6335,00	**16583**	**16583**	16614,30	**34565**	34628	34764,00	**69915**	72726	74664,70	**112941**	121010	124045,40	**170385**	187493	195350,70
bcsstk31	**2699**	**2699**	2699,00	**7282**	7283	7307,80	**13153**	13281	13363,90	**23607**	25557	26722,70	**37310**	42410	44427,80	**57090**	65775	67688,50
fe_pwt	**340**	**340**	340,00	**704**	**704**	704,00	**1432**	1439	1439,70	**2797**	2802	2832,90	**5514**	5746	5850,90	**8142**	8594	8718,60
bcsstk32	**4667**	**4667**	4667,00	**9186**	9227	9248,10	**20200**	20614	21102,90	**35619**	37660	38616,50	**60053**	69761	72777,50	**90590**	103452	115581,30
fe_body	**262**	**262**	262,00	**598**	**598**	599,20	**1023**	1058	1074,00	**1714**	1827	1915,40	**2788**	3822	4009,90	**4723**	5483	5712,10
t60k	**75**	**75**	76,40	**208**	211	212,90	**454**	471	480,00	**805**	892	914,30	**1313**	1575	1654,90	**2062**	2655	2726,20
wing	**784**	**784**	787,50	**1610**	1669	1688,00	**2474**	2627	2701,90	**3857**	4388	4451,40	**5584**	6527	6728,30	**7643**	10434	10793,10
brack2	**708**	**708**	708,00	**3013**	**3013**	3014,20	**7029**	7141	7179,80	**11492**	11813	11906,30	**17151**	18104	18449,10	**25617**	27931	28439,50
finan512	**162**	**162**	162,00	**324**	**324**	324,00	**648**	**648**	648,00	**1296**	**1296**	1296,00	**2592**	**2592**	2592,00	**10560**	11135	11993,50
fe_tooth	**3814**	3815	3816,00	**6846**	6862	6870,70	**11366**	11590	11620,10	**17265**	17720	17900,50	**24832**	25978	26242,90	**34240**	36456	36832,90
fe_rotor	**2031**	**2031**	2031,50	**7160**	7195	7328,80	**12641**	12834	13052,00	**20177**	20647	20881,10	**30990**	32253	32651,30	**45341**	48375	49452,90
598a	**2388**	**2388**	2388,20	**7948**	7979	7991,90	**15838**	15935	15997,50	**25624**	25952	26105,70	**38431**	39480	39815,20	**55896**	57968	58450,38
fe_ocean	387	**386**	386,90	**1813**	1817	1823,50	**4063**	4135	4181,40	**7617**	7816	7948,80	**12525**	13049	13375,30	**19855**	21748	22368,80
144	**6478**	6482	6486,70	**15140**	15214	15277,90	**25240**	25512	25672,90	**37417**	38127	38248,00	**55417**	56799	57370,90	**77006**	80195	80294,33
wave	**8657**	8677	8695,30	**16747**	16783	16811,80	**28758**	29018	29174,90	**42354**	42901	43679,20	**60633**	62326	62778,20	**83478**	86547	86947,50
m14b	**3826**	3827	3829,40	**12973**	13170	13226,40	**25666**	26258	26356,10	**42173**	43064	43557,70	**64807**	66715	67579,40	—	—	—
auto	**9949**	9978	9994,90	**26614**	26829	26933,80	**45442**	46384	47764,50	**76542**	77732	78773,14	—	—	—	—	—	—

Table B.2: Minimum and average cut size results among all repetitions compared to the best known values (BKV) when $\epsilon = 1.03$. Best known values are set in bold face.

	$k = 2$			$k = 4$			$k = 8$			$k = 16$			$k = 32$			$k = 64$		
Instance	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg
add20	**560**	**560**	561,50	1148	**1134**	1134,70	1679	**1677**	1681,70	**2035**	2051	2064,10	**2351**	2387	2399,90	**2925**	2989	3020,20
data	**185**	**185**	185,00	**369**	**369**	369,00	**638**	**638**	638,30	**1088**	1103	1110,00	**1768**	1819	1838,90	**2783**	2921	2944,50
3elt	**87**	**87**	87,00	**198**	**198**	198,10	**334**	335	335,80	**561**	563	564,10	**944**	958	966,10	**1512**	1579	1584,10
uk	**18**	**18**	18,00	**39**	**39**	39,50	**78**	**78**	79,60	**139**	145	149,70	**240**	255	260,40	**397**	432	438,20
add32	**10**	**10**	10,00	**33**	**33**	33,00	**66**	**66**	66,00	**117**	**117**	117,00	**212**	**212**	212,00	**483**	486	486,90
bcsstk33	**10064**	**10064**	10064,00	**20762**	**20762**	20764,50	**34065**	34076	34078,60	**54354**	54474	54568,60	**76750**	77617	78279,70	**105767**	108565	109218,60
whitaker3	**126**	**126**	126,00	**378**	**378**	378,00	**649**	653	654,20	**1076**	1084	1089,60	**1655**	1685	1698,40	**2459**	2567	2583,40
crack	**182**	**182**	182,00	**360**	**360**	360,00	**671**	**671**	673,50	**1072**	1078	1081,30	**1657**	1683	1711,00	**2491**	2591	2615,80
wing_nodal	**1678**	**1678**	1678,00	**3534**	3537	3537,70	**5360**	5364	5367,70	**8244**	8309	8328,40	**11632**	11931	11969,00	**15613**	16196	16343,60
fe_4elt2	**130**	**130**	130,00	**341**	342	342,60	**595**	597	598,70	**990**	1002	1005,40	**1593**	1649	1667,70	**2435**	2564	2587,00
vibrobox	**10310**	**10310**	10310,00	**18736**	**18736**	18738,10	**24153**	24177	24237,40	**31494**	31818	32080,00	**39201**	40673	41296,70	**46299**	48350	48722,90
bcsstk29	**2818**	**2818**	2818,00	**7971**	7972	7982,80	**13714**	13794	13857,20	**21258**	21605	21928,20	**33842**	35033	36082,00	**54486**	56755	57802,60
4elt	**137**	**137**	137,00	**319**	**319**	319,00	**522**	523	523,00	**903**	904	921,90	**1519**	1548	1567,40	**2514**	2659	2688,30
fe_sphere	**384**	**384**	384,00	**764**	**764**	764,00	**1152**	**1152**	1152,00	**1696**	1700	1701,90	**2463**	2493	2501,90	**3507**	3642	3681,10
cti	**318**	**318**	318,00	**916**	**916**	916,00	**1714**	**1714**	1716,10	**2728**	2754	2796,50	**3948**	4108	4186,70	**5524**	6129	6224,70
memplus	**5353**	5354	5358,90	**9362**	9650	9687,30	**11624**	12163	12238,60	**12888**	13508	13772,80	**13927**	14656	15549,20	**16162**	17501	17685,20
cs4	**360**	**360**	360,10	**917**	939	947,40	**1424**	1476	1500,70	**2055**	2192	2250,30	**2885**	3200	3284,40	**3982**	4812	4881,60
bcsstk30	**6251**	**6251**	6251,00	**16372**	16375	16398,80	**34137**	34140	34175,00	**69393**	70113	72686,70	**110709**	117947	122874,90	**168299**	181193	191427,70
bcsstk31	**2676**	**2676**	2676,00	**7150**	**7150**	7157,80	**12974**	13082	13131,50	**23232**	24191	25083,30	**36655**	40692	42719,50	**56209**	64943	65558,50
fe_pwt	**340**	**340**	340,00	**700**	**700**	700,00	**1410**	1414	1415,10	**2754**	2780	2785,10	**5403**	5632	5717,00	**8036**	8552	8605,00
bcsstk32	**4667**	**4667**	4667,00	**8725**	8728	8740,00	**19888**	20012	20154,30	**34932**	35797	36253,90	**58741**	63031	64226,20	**89927**	98800	102206,20
fe_body	**262**	**262**	262,00	**598**	**598**	598,00	**1016**	1022	1034,10	**1702**	1776	1801,90	**2709**	3110	3176,00	**4527**	5425	5512,20
t60k	**71**	**71**	71,20	**203**	204	208,90	**449**	461	468,00	**792**	874	883,30	**1302**	1521	1540,90	**2036**	2519	2575,40
wing	**773**	776	782,30	**1593**	1632	1662,10	**2451**	2574	2620,10	**3784**	4177	4301,30	**5559**	6511	6618,00	**7561**	9072	9738,00
brack2	**684**	**684**	684,00	**2834**	**2834**	2837,10	**6781**	6799	6865,40	**11262**	11513	11664,40	**17031**	17801	18076,50	**25368**	27366	27641,60
finan512	**162**	**162**	162,00	**324**	**324**	324,00	**648**	**648**	648,00	**1296**	**1296**	1296,00	**2592**	**2592**	2592,00	**10560**	10661	10910,30
fe_tooth	**3788**	3789	3790,00	**6756**	6772	6788,70	**11241**	11491	11540,80	**17108**	17544	17751,30	**24649**	25844	26147,30	**33805**	36012	36430,00
fe_rotor	**1959**	**1959**	1959,00	**7050**	7085	7162,80	**12445**	12553	12777,20	**19920**	20380	20555,40	**30647**	32206	32502,60	**44831**	47555	48273,20
598a	**2367**	**2367**	2367,10	**7816**	7847	7857,80	**15613**	15771	15823,60	**25391**	25830	25924,70	**38122**	39148	39462,60	**55376**	57785	57990,00
fe_ocean	**311**	**311**	311,00	**1693**	**1693**	1693,00	**3920**	3926	3955,00	**7405**	7607	7704,70	**12291**	12814	13031,40	**19521**	21289	21810,00
144	**6432**	6434	6445,40	**15075**	15136	15210,70	**24911**	25407	25526,60	**37189**	37786	38007,80	**54813**	56604	57161,90	**76570**	79429	79859,00
wave	**8591**	8625	8631,20	**16640**	16702	16728,00	**28494**	28632	28888,60	**42141**	42872	43333,30	**60418**	61728	62490,20	**82847**	85574	86001,33
m14b	**3823**	**3823**	3826,70	**12948**	13215	13314,20	**25390**	25984	26069,30	**41778**	42813	43128,60	**64488**	65622	67199,50	**95647**	99545	99545,00
auto	**9673**	9694	9716,60	**25789**	26064	26375,20	**44768**	45463	46924,30	**75719**	76738	79053,50	**119157**	125249	125249,00	—	—	—

Table B.3: Minimum and average cut size results among all repetitions compared to the best known values (BKV) when $\epsilon = 1.05$. Best known values are set in bold face.

	$k = 2$			$k = 4$			$k = 8$			$k = 16$			$k = 32$			$k = 64$		
Instance	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg	BKV	Min	Avg
add20	**536**	**536**	536,70	1132	**1125**	1125,00	1668	**1662**	1668,10	**2029**	2043	2056,40	**2343**	2385	2402,10	**2925**	2956	2976,50
data	**181**	182	182,00	**363**	**363**	363,00	**628**	**628**	628,00	**1076**	1079	1083,40	**1743**	1789	1796,60	**2750**	2848	2875,30
3elt	**87**	**87**	87,00	**197**	**197**	197,00	**329**	**329**	329,80	**557**	558	559,30	**930**	947	953,90	**1499**	1544	1551,60
uk	**18**	**18**	18,00	**39**	**39**	39,00	**75**	76	77,80	**137**	145	146,50	**236**	254	256,30	**394**	428	433,00
add32	**10**	**10**	10,00	**33**	**33**	33,00	**63**	**63**	63,00	**117**	**117**	117,00	**212**	**212**	212,00	**483**	**483**	483,90
bcsstk33	**9914**	**9914**	9914,00	**20158**	20167	20167,00	**33908**	33916	33917,80	**54119**	54365	54440,80	**76080**	77071	77751,60	**105347**	107644	108136,40
whitaker3	**126**	**126**	126,00	**376**	**376**	376,70	**644**	645	648,60	**1069**	1081	1084,30	**1633**	1668	1677,50	**2441**	2531	2560,30
crack	**182**	**182**	182,00	**360**	**360**	360,00	**666**	667	667,60	**1063**	1065	1070,30	**1655**	1679	1690,10	**2491**	2566	2600,30
wing_nodal	**1668**	1669	1669,00	**3520**	**3520**	3521,30	**5339**	5344	5348,60	**8160**	8248	8272,50	**11536**	11827	11858,10	**15515**	16200	16259,50
fe_4elt2	**130**	**130**	130,00	**335**	**335**	335,60	**578**	579	581,00	**979**	993	999,00	**1572**	1622	1643,00	**2420**	2552	2590,40
vibrobox	**10310**	**10310**	10310,00	**18690**	**18690**	18693,30	**23924**	23932	23948,80	**31234**	31691	31900,80	**38866**	40514	41146,10	**46104**	48098	48336,50
bcsstk29	**2818**	**2818**	2818,00	**7925**	**7925**	7936,50	**13540**	13606	13642,80	**20924**	21035	21481,10	**33455**	34889	35640,60	**53987**	56412	57381,20
4elt	**137**	**137**	137,00	**315**	**315**	315,00	**515**	516	516,10	**887**	895	899,20	**1494**	1534	1544,80	**2486**	2623	2649,80
fe_sphere	**384**	**384**	384,00	**762**	**762**	762,00	**1152**	**1152**	1152,00	**1678**	1682	1684,50	**2427**	2439	2473,90	**3456**	3591	3632,30
cti	**318**	**318**	318,00	**889**	**889**	889,00	**1684**	**1684**	1684,90	**2703**	2742	2764,40	**3914**	4054	4138,20	**5479**	6043	6120,10
memplus	5266	**5261**	5273,80	**9292**	9536	9600,50	**11543**	12109	12171,70	**12874**	13492	13738,40	**13899**	15029	15473,80	**16044**	17581	17658,70
cs4	**353**	**353**	353,50	**908**	926	936,70	**1420**	1454	1484,50	**2043**	2194	2243,00	**2860**	3202	3239,40	**3962**	4747	4819,10
bcsstk30	**6251**	**6251**	6251,00	**16165**	**16165**	16176,20	**34068**	34079	34121,80	**68788**	70664	72529,30	**109716**	116827	119630,10	**167084**	184721	188040,80
bcsstk31	2669	**2660**	2665,80	7085	**7082**	7089,50	**12851**	12871	12908,10	**22871**	23661	24265,10	**36440**	40828	42316,40	**55423**	62375	63466,90
fe_pwt	**340**	**340**	340,00	**700**	**700**	700,00	**1405**	**1405**	1405,30	**2737**	2745	2761,70	**5306**	5518	5597,60	**7959**	8416	8465,20
bcsstk32	**4622**	**4622**	4653,50	**8441**	**8441**	8481,30	19375	**19256**	19614,90	**34374**	35137	35755,60	**58387**	60843	62377,20	**88609**	95987	100857,60
fe_body	**262**	**262**	262,00	**588**	598	598,00	**1012**	1018	1020,80	**1683**	1737	1767,00	**2679**	2965	3030,00	**4512**	5165	5249,30
t60k	**65**	**65**	65,00	**195**	198	199,90	**441**	460	468,50	**787**	864	880,80	**1291**	1490	1527,20	**2016**	2486	2530,80
wing	**770**	772	773,30	**1590**	1618	1645,50	**2440**	2544	2597,30	**3775**	4174	4268,10	**5516**	6402	6660,70	**7535**	8911	9403,10
brack2	**660**	**660**	660,00	**2731**	**2731**	2732,40	**6592**	6621	6635,70	**11092**	11303	11375,30	**16803**	17628	17844,20	**25108**	26639	27081,10
finan512	**162**	**162**	162,00	**324**	**324**	324,00	**648**	**648**	648,00	**1296**	**1296**	1296,00	**2592**	**2592**	2592,00	**10560**	10904	11051,70
fe_tooth	**3773**	**3773**	3773,20	**6687**	6699	6707,70	**11154**	11273	11387,50	**16988**	17605	17729,80	**24282**	25521	25816,90	**33406**	35478	35862,00
fe_rotor	**1940**	**1940**	1940,00	**6781**	6785	6858,20	**12309**	12442	12609,10	**19680**	19830	20209,50	**30356**	31410	32082,30	**44400**	46915	47796,00
598a	**2336**	**2336**	2336,00	**7725**	7753	7762,60	**15414**	15553	15633,10	**25206**	25570	25714,30	**37653**	38591	38952,20	**54684**	57049	57466,22
fe_ocean	**311**	**311**	311,00	**1686**	**1686**	1686,30	**3886**	3944	3953,80	**7338**	7497	7599,80	**12044**	12645	12775,50	**19400**	21091	21404,10
144	**6345**	6347	6363,40	**14982**	15046	15158,10	**24195**	24601	24693,60	**36613**	37380	37607,70	**54203**	55473	56244,60	**75782**	78816	79246,20
wave	**8524**	8549	8556,60	**16533**	16622	16638,60	**28489**	28674	28874,90	**42031**	42942	43252,20	**59631**	61262	61754,60	**82024**	85323	85512,80
m14b	**3802**	**3802**	3807,80	**12863**	13000	13141,20	**25143**	25705	25821,90	**41113**	42250	42722,60	**63480**	65355	66296,80	—	—	—
auto	**9450**	9471	9490,20	**25271**	25602	25825,40	**44206**	44913	46617,40	**74304**	75416	76689,13	—	—	—	—	—	—