Randomized Greedy Algorithms for the Hypergraph Partitioning Problem

R. Battiti, A. Bertossi, and R. Rizzi

Abstract. We propose a series of randomized greedy construction schemes for the hypergraph partitioning problem. While the final results are inferior to those obtained by recent multi-level methods, the advantages of our greedy schemes are their simplicity and low computational complexity. The best greedy algorithms considered obtain low cut values and large standard deviations of the results. Therefore, when independent repetitions are considered, the quality of the best solution greatly improves and, in some cases, it is superior to the variable-depth Fiduccia-Mattheyses (FM) algorithm, for smaller CPU times. Furthermore, the algorithms can be used as building blocks in more complex schemes. For example, we successfully employ our greedy schemes to produce initial partitions for *improvement-based* heuristics. In particular, if FM is run starting from partitions generated by our greedy schemes, instead of random initial solutions, a significant improvement of the average solution quality is achieved for comparable computation times.

1. Introduction

The hypergraph partitioning problem is defined as follows (in this paper we consider only the case of partitioning into two sets, also called bisection). Let $H(V, E)$ be an hypergraph, that is a set V of *nodes* plus a set E of subsets of V called *hyperedges* (or simply *edges*), see also Fig. 1. One is required to partition the nodes in V into two sets, say Set_0 and Set_1, aiming at minimizing the number of *broken* edges, i.e, edges in E containing nodes on both sides of the partition, while obeying a constraint on the smallest possible sizes (min_size) of Set_0 and Set_1. Equivalently, one is given a percentage value, for example 45%, and is required to return a partition in which the cardinalities of Set_0 and Set_1 both exceed $0.45 * |V|$. Because of its physical realization (nodes are modules and a hyperedge is an electrical network connecting a set of modules) one often uses the term "circuit partitioning."

Hypergraph partitioning has recently emerged as a central issue in VLSI design [**1, 9**]. First of all, in VLSI placement, a divide and conquer approach is taken

---

1991 *Mathematics Subject Classification.* Primary 68M10 , 68R10 ; Secondary 68P05 , 90B12, 90C35, 90C27.

Key words and phrases. greedy algorithms, hypergraph-partitioning, data structures, computer implementation .

where the circuit is hierarchically divided into smaller components using hypergraph partitioning [**16**]. The formulation also finds applications in *rapid prototyping* where the goal typically is to partition to a minimal number of FPGAs (Field Programmable Gate Arrays) under different constraints such as available number of pins and routing resources [**4, 12, 13, 14, 22**]. Another application area is the *design for testability* of VLSI circuits where the circuit is partitioned into smaller parts to facilitate testing [**21**]. However the importance of the hypergraph partitioning problem goes beyond VLSI design. For example, electrical circuits with multiple-pin nets are readily modeled as hypergraphs. Other applications include data mining [**18**], efficient storage of large databases on disks [**19**], clustering and partitioning of the roadmap database for routing applications [**20**], de-clustering data in parallel databases [**19**].

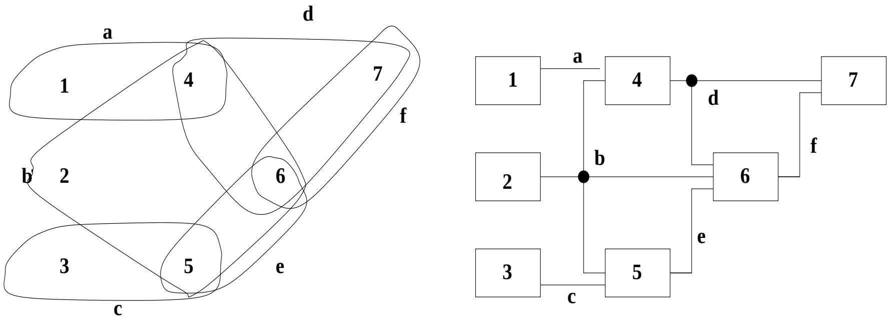

FIGURE 1. A hypergraph (left) and a concrete realization as electrical circuit (right).

In the last years one has observed a blossoming of graph and hypergraph partitioning algorithms and software packages (METIS, MELO, Paraboli, SCOTCH). The solution quality and reliability improvements which have come along is remarkable and would have been unpredictable only a few years ago [**1, 9, 5, 6**]. We refer to [**1**] for a recent survey of the problem and for detailed references.

Because graph bisection is a special case of hypergraph bisection and is NP-hard, hypergraph bisection is NP-hard as well. However, because of the importance of the problem, many heuristic algorithms have been developed [**1**]. A widely used class of *iterative improvement* partitioning algorithms start by randomly generating an initial partition and then attempt to progressively reduce the cut value of the partition by repeatedly moving vertices between the two parts of the partition. The most used among these algorithms is the Fiduccia-Mattheyses [**7, 8**] (FM) algorithm, derived from the seminal Kernighan-Lin algorithm for graph partitioning [**10**]. In the FM algorithm a vertex (that is not *locked*) is moved that results in the greatest reduction in the edge-cuts, which is also called the *gain* for moving the vertex. Because more nodes can have the same gain, Krishnamurty [**11**] proposed to look ahead up to r levels of gains before making moves as a tie breaking rule to be used when more nodes achieve the same reduction of the edge-cuts, a frequent case of real-world partitioning problems.

In this paper we analyze the performance of greedy strategies in the context of hypergraph bisection. The basic motivation of our research is to investigate whether one could extend to hypergraphs the successful performance of greedy schemes previously obtained on graphs [**2**]. The results obtained are not competitive with state-of-the-art multi-level algorithms [**9**]. Nonetheless, our analysis shows greedy schemes to be a practical and viable choice for applications characterized by strong trade-offs between solution quality and computation time, like for example declustering data in parallel databases and other applications in parallel computing. Moreover, our schemes can be used as building blocks of more complex schemes. For example, we compared the classical *improvement-based* Fiduccia-Mattheyses heuristic, starting from a random initial solution, against the same heuristic, but starting from solutions produced by a greedy scheme. While the computation time of the two algorithms is comparable, the improvement in terms of average solution quality is significant.

The following part of this paper is organized as follows. The benchmark graphs considered for the VLSI applications are described in Sec. 2, some simple greedy approaches are presented in Sec. 3, while the more effective Likelihood Greedy algorithm is presented in Sec. 4. The results obtained when Fiduccia-Mattheyses starts from our greedy solutions are discussed in Sec. 5. Finally, the implementation details, computational complexity, and measured CPU times are presented in Sec. 6.

2. Benchmarks

The computational tests in this paper are executed on the collection of hypergraphs (nets) proposed as benchmarks for the hypergraph partitioning problem by Chuck Alpert[1]. All nets are in *netlist* format.

The nets are collected into three major groups:

- **Cheng's nets**, made available by C.K. Cheng at UCSD.
- **VPNR nets**, obtained by translating in netlist format some instances first represented in VPNR format.
- **Large nets**, made available by Lars Hagen (lars@cadence.com).

The characteristics of these hypergraphs are shown in Table 1. The column labeled "Best" lists the heuristically best value from Table 7 of [**9**]. Most of these results are obtained with a finely tuned multi-level algorithms (hMETIS) that is considered the state of the art for hypergraph problems of interest in the VLSI community. The two additional columns list quantities involved in the computational complexity of some greedy schemes ($|h|$ is the cardinality of hyperarc h), see Sec 6.

Chuck Alpert has made available an efficient implementation of the Fiduccia-Mattheyses Algorithm [2]. The implementation uses a LIFO (last-in-first-out) bucket structure which many researchers discovered to be significantly more effective than either FIFO (first-in-first-out) or random bucket structures [**8**]. This FM implementation provides a standard and basic benchmark against which other partitioning codes can be measured in terms of runtimes and solution quality.

We have run our Greedy algorithms 100 times on each single net. In the tables one reports the minimum value found among all runs (Min), the average values obtained (Ave) and the standard deviation of the values (Sdev). The standard

[1] These nets are publically available on the "Circuit Partitioning Page" at URL `http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html` .

[2] Available on WWW at URL `http://vlsicad.cs.ucla.edu/~cheese/codes.html`.

graph	vertices	hyperedges	pins	"Best"	$\Sigma_{h\in E(H)} \lvert h\rvert^2$	$\Sigma_{h\in E(H)} \lvert h\rvert^2 \log \lvert V\rvert$
balu	801	735	2697	27	33637	97669
p1	833	902	2908	47	15422	45042
bm1	882	903	2910	47	15426	45436
t4	1515	1658	5975	48	258673	822686
t3	1607	1618	5807	57	136691	438233
t2	1663	1720	6134	87	217016	698985
t6	1752	1541	6638	60	251541	815881
structP	1952	1920	5471	33	22555	74216
t5	2595	2750	10076	71	558422	1906529
19ks	2844	3282	10547	104	382443	1320931
p2	3014	3029	11219	139	85723	298242
s9234P	5866	5844	14065	40	46076	173630
biomedP	6514	5742	21040	83	2582834	9850535
s13207P	8772	8651	20606	53	77706	306402
s15850P	10470	10383	24712	42	90199	362595
industry2	12637	13419	48158	168	1786670	7328284
industry3	15406	21923	65791	241	425483	1781790
s35932	18148	17828	48145	41	2476306	10546163
s38584	20995	20717	55203	47	342383	1479819
avq.small	21918	22124	76231	127	63318301	274852139
s38417	23849	23843	57613	50	230903	1011190
avq.large	25178	25384	82751	127	63331341	278722576
golem3	103048	144949	338419	1424	940085	4712683

TABLE 1. The characteristics of the hypergraphs used to evaluate the algorithms. The "Best" column lists the heuristically best value from Table 7 of [**9**].

deviation of the estimated averages can be obtained by dividing the above Sdev value by the square root of the number of tests, i.e. by 10.

The computational tests discussed in this paper have been executed on a machine with the following characteristics:

- CPU: Pentium Pro
- Clock: 200 MHz
- Configuration: 160 MByte
- Operating System: Red Hat Linux release 4.1 (Vanderbilt)
- Compiler: Gnu `g++` version 2.7.2.1 with maximum optimization

In this preliminary investigation, the focus of attention is on the average results and standard deviations obtained by the different approaches and on their computational complexity more than on the the obtained computing times. Nonetheless, CPU times for the given machines are reported in Sec. 6.3. The standard deviation is of interest when independent repetitions of the algorithms are considered and the minimum is returned.

3. Simple greedy approaches

Motivated by the previous success of some greedy heuristics for the graph partitioning problem [**3, 2**], we investigate in this paper the behavior of a greedy approach for the hypergraphs partitioning problem.

Our general scheme is the following: At the beginning both Set_0 and Set_1 are empty and all nodes are marked as *free*. Then free nodes are inserted one by one into either Set_0 or Set_1. Insertions are definitive: once a node has been added to

Set_i ($i = 0, 1$) it will stay there until the end of the algorithm. A node inserted into Set_i is no longer free but *fixed* to i.

To ease the comparisons among different strategies, we have adopted the following rule. First, nodes are added alternately to Set_0 and Set_1 until the sizes of both sets reach the *min_size* value. Finally, the remaining free nodes are inserted one by one into either set, aiming at obtaining the smallest increase of newly broken edges.

The algorithms considered in this paper correspond to the above scheme, differing only in the way they choose the next node to be inserted. Each greedy scheme can be described with the help of a *gain* function which expresses the *pricing policy* of the heuristic. For each node v, $gain(v, 0)$ is the heuristic's estimate of the advantage of assigning v to Set_0. Similarly $gain(v, 1)$ is the gain for assigning v to Set_1. At each move the scheme greedily strives to maximize the gain.

All nodes of a certain gain are stored into a list and the following criterion [**8**] is used for inserting and deleting nodes. When the gain of a node is increased the node is inserted at the beginning of the corresponding list. When the gain of a node is decreased the node is inserted at the end. At each move the greedy scheme chooses the first node in the list of nodes of maximum gain.

A common characteristic of the schemes considered here is that the gain of node v is the sum of many terms, one for every hyperedge containing v:

$$gain(v; i) = \sum_{h \ni v} g_h(v; i)$$

A scheme is defined by specifying $g_h(v; i)$ for a generic hyperedge h. When $g(v; 0) = -g(v; 1)$ for each node v, the scheme is said *consistent*. Consistent schemes turned out to be most effective. When a scheme is consistent we can define the gain of a node as an absolute value (not depending on a specific side of the bipartition), precisely as $g(v) = g(v; 1)$. When choosing a node to be inserted into Set_1, the heuristic strives to maximize the gain. When searching for a node to be inserted into Set_0, a node of minimum gain is chosen. More precisely, all consistent schemes obey the following rules:

(i) when we seek for a node to be inserted into Set_1, then we choose randomly and with uniform probability a node of maximum gain.

(ii) when we seek for a node to be inserted into Set_0 then we choose randomly and with uniform probability a node of minimum gain.

(iii) Assume we have to choose a node to be inserted in whichever set of the partition. If max gain $> -$min gain then we decide to insert into Set_1 and go to (ii). If max gain $= -$min gain then we choose at random and with equal probability whether to insert into Set_1 and go to (ii) or to insert into Set_0 and go to (i). Otherwise we decide to insert into Set_0 and go to (i).

We are interested in heuristics which are robust and can be executed repeatedly to improve the solution quality. Therefore *randomization* is desirable in our greedy schemes. At the beginning, when all nodes are still free and all gains are zero, all nodes are inserted into the zero-gain list in a random order. The experiments show that this source of randomness suffices to derive a robust heuristic. The standard deviation of the cut value returned for a given input hypergraph is significant, so that much better results can be achieved by repeating the greedy construction with a different random seed.

Let us now describe how the gains are calculated in the different schemes.

3.1. Δf Greedy. The method is the most straightforward greedy approach, based only on the consideration of the function to be minimized. In each step of the algorithm, let f be the number of broken edges. At termination f is the value of the partition obtained. An edge is said to be *broken* when it contains nodes both fixed to 0 and fixed to 1.

The idea behind the Δf Greedy algorithm is to choose randomly between those nodes which produce the smallest possible increase of the number of broken edges (the value of f) when they are added to the current partition.

Let v be a free node and h an hyperedge. If $v \notin h$ then $g_h(v;i) = 0$ for $i = 0, 1$. Otherwise $g_h(v;i)$ is given as follows.

$$g_h(v;i) = \begin{cases} -1 & \text{if } |h|_{1-i} > 0 \text{ and } |h|_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

where $|h|_i = |h \cap Set_i|$ (for $i = 0, 1$). When we look for a node to be added to Set_i we search for a free node v which maximizes $gain(v;i)$. When i is not given, that is, when $|Set_0| \geq min\_size$ and $|Set_1| \geq min\_size$, then one maximizes the gain both over v and over $i \in \{0, 1\}$. If there are ties, a random choice among the winning nodes is executed. The implementation uses a standard "buckets" structure.

As expected, and as noted independently by many previous researchers, the results obtained by this immediate greedy algorithm are very unsatisfactory, see Table 2. The problem is caused by the large number of ties that are present during the greedy construction: the algorithm is forced to make a "blind" choice among a very large set of winning nodes.

3.2. Critical Edges Greedy. In our previous work about graph partitioning [**2**] a greedy approach with a gain function given by the *difference* between connections of node v to Set_0 and connections of node v to Set_1 was the most successful option. Therefore we were motivated to modify the method for the case of hypergraphs.

The modifications must take into account the fact that different hyperedges h containing a node v should not be given the same weight when deciding about an addition. In particular, if h contains already nodes in both sets (see situation (a) in Fig. 2) it should not be considered: there is no way to undo its cut. If at least two nodes of h are free (situations (b) and (c) in Fig. 2) and all other nodes, if any, are fixed to the same set, then one cannot decide the fate of h during a single addition. For example, in situation (c), even if node u is added to Set_0, node v may be added to Set_1 in the future steps, therefore breaking the hyperedge. The cases in which the fate can be decided in one step are those corresponding to situation (d), where only one node is free and all other nodes have been added to the same set. If the free node is added to the same set, h is not broken, otherwise it is. An edge is said 0-*critical* when all of its nodes are fixed to 0, except for a single free node. An edge is 1-*critical* when all of its nodes are fixed to 1, except for a single free node.

Critical Edges Greedy is our first example of a consistent scheme resulting from the adaptation of the Differential-Greedy algorithm [**2**] to hypergraphs. In each step of the algorithm, the *gain* of a free node v is defined as the number of 1-*critical* edges containing v minus the number of 0-*critical* edges containing v. The implementation uses the "buckets" data structure.

When the results are compared with those of Δf Greedy, see Table 2, one observes a much better performance for most graphs. In particular, macroscopic

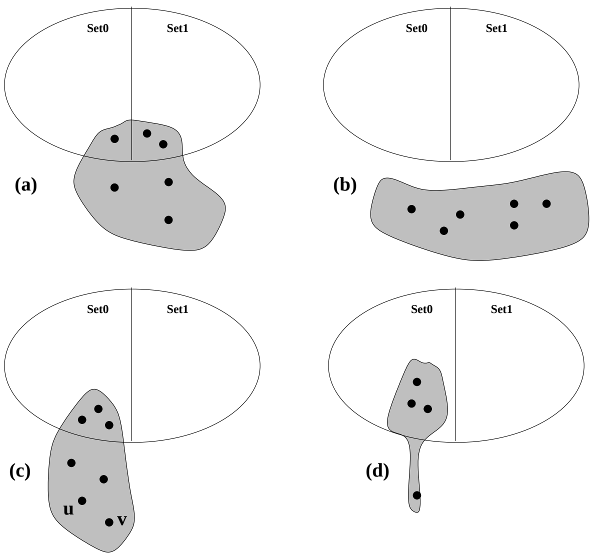

FIGURE 2. Critical and non-critical hyperedges (see text for explanation).

differences are observed for the larges graphs. For example, when the minimum value is considered, one passes from 5,748 to 1,168 broken edges for `industry3`, from 32,034 to 2,383 for `golem3`. Nonetheless, the average and minimum results are very far from those of the FM algorithm.

3.3. Weighted Greedy. The Weighted Greedy algorithm is a second consistent scheme with the aim of considering in a more gradual way the effect of non-critical edges. The previous approach does not give any preference information when the weights obtained by considering only critical edges is the same for different nodes.

Let v be a free node and h an hyperedge. If $v \notin h$ or if h is broken then $g_h(v) = g_h(v;1) = 0$. Otherwise $g_h(v) = \frac{|h|_1}{|h|} - \frac{|h|_0}{|h|}$, where $|h|$ is the number of nodes in h and $|h|_i = |h \cap Set_i|$ for $i = 0, 1$. (In particular either $|h|_1 = 0$ or $|h|_0 = 0$ because h is not broken). The implementation uses binary heaps to store the gain values.

The last column in Table 2 presents the results obtained by the Weighted Greedy algorithm. A notable improvement is obtained with respect to the Critical Edges Greedy. In particular, both the average and the minimum results become roughly comparable to those obtained by FM. In some cases FM results are beaten (these cases are written in boldface in the table). In the other cases the difference is often small.

4. The Likelihood Greedy algorithm

Weighted Greedy clearly outperforms all competitors examined in Table 2. Critical Edges Greedy suffers from the following weakness: only critical edges play a role in the choices of the algorithm. Our first attempt to consider non-critical edges was represented by the Weighted Greedy algorithm. However the policy of Weighted Greedy can be ameliorated. In particular the gain calculation can be

Graph	Fiduccia		Δf Greedy		Critical Edges Greedy		Weighted Greedy	
	Min	Ave (Sdev)	Min	Ave (Sdev)	Min	Ave (Sdev)	Min	Ave (Sdev)
baluP	30	66.3 (18.6)	152	198.1 (17.0)	160	196.8 (11.8)	**28**	**63.1 (23.1)**
p1	47	74.9 (13.7)	202	249.1 (18.6)	147	192.4 (15.2)	57	90.2 (16.9)
bm1	49	75.8 (14.9)	192	242.5 (18.8)	164	197.4 (15.7)	59	87.9 (14.6)
t4	80	135.6 (25.4)	352	426.3 (28.8)	259	307.7 (20.3)	**55**	**114.6 (28.0)**
t3	62	108.7 (22.1)	305	422.0 (33.8)	237	308.4 (22.4)	67	128.1 (34.8)
t2	124	175.5 (24.2)	395	472.9 (26.5)	258	324.1 (17.3)	**100**	**149.3 (26.7)**
t6	60	91.0 (14.8)	417	475 (26.7)	183	212.6 (17.4)	62	117.6 (30.0)
structP	41	55.4 (8.6)	406	444.4 (18.4)	346	418.6 (26.9)	**41**	106.6 (35.6)
t5	104	179.4 (35.6)	521	688.9 (43.7)	371	446.8 (33.4)	**86**	**151.6 (36.8)**
19ks	130	178.9 (26.8)	583	696.3 (40.8)	416	479.2 (32.0)	**124**	221.6 (55.1)
p2	182	278.5 (40.3)	832	928.5 (47.0)	752	808.9 (21.5)	232	319.9 (54.6)
s9234P	51	89.9 (25.2)	1122	1182.9 (25.4)	326	372.2 (16.9)	60	123.9 (44.9)
biomedP	83	126.5 (40.1)	1259	1569.4 (172.2)	890	944.7 (23.2)	128	223.1 (34.6)
s13207P	78	126.2 (21.3)	1553	1670.2 (34.3)	456	499.4 (16.4)	118	181.0 (37.7)
s15850P	104	184.0 (29.4)	1934	2030.4 (34.3)	558	627.9 (20.9)	**104**	**167.9 (33.0)**
industry2	264	602.1 (161.8)	2815	3233.0 (157.1)	1702	1885.1 (73.3)	306	715.3 (295.7)
industry3	263	484.6 (161.0)	5748	6167.1 (112.6)	1168	2648.3 (815.6)	555	961.5 (335.0)
s35932	85	203.5 (53.7)	3170	3302.0 (52.1)	1137	1245.6 (42.2)	**45**	**187.8 (92.9)**
s38584	63	258.8 (120.3)	4300	4476.6 (64.7)	1788	1889.9 (42.4)	97	**253.5 (106.7)**
avq.small	343	620.3 (116.3)	4904	5573.6 (233.7)	3454	3637.0 (70.2)	**328**	**590.7 (120.3)**
s38417	147	368.0 (92.3)	4747	4870.3 (58.4)	1546	1647.3 (36.8)	**95**	**198.2 (48.6)**
avq.large	373	759.3 (123.9)	5335	6166.7 (295.0)	3449	3625.1 (79.6)	**325**	**694.9 (157.5)**
golem3	2175	3217.9 (304.0)	32034	32379.5 (160.1)	2383	3449.6 (714.2)	5634	6109.6 (214.5)

TABLE 2. Simple greedy schemes compared with Fiduccia-Mattheyses (FM)

unfair if two hyperedges h_1 and h_2 have widely different numbers of *nodes already added*. For example, suppose that $|h_1| = 3$, $|h_2| = 6$, while $|h_1|_1 = 2$ and $|h_2|_1 = 4$ (and no node in Set_0). The contribution to the gain from h_1 and from h_2 will be equal, while h_1 is a critical edge and the addition of the single remaining free node to Set_1 should be preferred to the addition of a free node in h_2, with the danger of a subsequent cut of h_2 when its second node is added. In other words, the number of nodes already added to hyperedge should *not* influence the addition of the remaining nodes. Our first greedy approaches have been modified to take these comments into account and the experimental results obtained confirm the intuitive expectations.

Likelihood Greedy is a consistent scheme. Let v be a free node and h an hyperedge. If $v \notin h$ or if h is broken then $g_h(v) = g_h(v;1) = 0$. Otherwise $g_h(v)$ is given as follows.

$$g_h(v) = \begin{cases} \frac{1}{2^{|h|_f - 1}} & \text{if } |h|_1 > 0 \text{ and } |h|_0 = 0 \\ -\frac{1}{2^{|h|_f - 1}} & \text{if } |h|_0 > 0 \text{ and } |h|_1 = 0 \\ 0 & \text{otherwise.} \end{cases}$$

where $|h|_i = |h \cap Set_i|$ (for $i = 0, 1$) and $|h|_f$ denotes the number of free nodes of h.

The rationale for such a choice of gains is the following. Let u be any free node of h. The *a priori* probability of node u being assigned to Set_0 is $\frac{1}{2}$ and it is the same as the probability of node u being assigned to Set_1. Let us assume that $|h|_1 > 0$ and $|h|_0 = 0$. If v is assigned to Set_0 then hyperedge h gets broken. To favor additions to Set_1 the gain $g_h(v)$ is positive. However, assigning node v to Set_1 is not sufficient to guarantee that h will not be in the final cut set

Graph	Fiduccia		(3)-Likelihood Greedy		(10)-Likelihood Greedy		Likelihood Greedy	
	Min	Ave (Sdev)	Min	Ave (Sdev)	Min	Ave (Sdev)	Min	Ave (Sdev)
baluP	30	66.3 (18.6)	33	62.2 (15.1)	29	51.6 (13.8)	**29**	**51.4 (14.1)**
p1	47	74.9 (13.7)	65	89.2 (11.3)	67	83.7 (10.0)	63	82.9 (11.0)
bm1	49	75.8 (14.9)	61	88.8 (11.3)	59	87.0 (9.9)	62	86.2 (9.9)
t4	80	135.6 (25.4)	74	128.0 (19.9)	69	112.2 (17.3)	**67**	**104.4 (19.8)**
t3	62	108.7 (22.1)	70	123.4 (30.2)	60	100.3 (27.6)	**60**	**93.9 (25.7)**
t2	124	175.5 (24.2)	108	142.2 (16.7)	104	138.6 (19.3)	**103**	**141.4 (18.1)**
t6	60	91.0 (14.8)	93	122.3 (12.9)	64	89.8 (11.7)	67	**84.9 (9.6)**
structP	41	55.4 (8.6)	42	82.9 (19.5)	46	84.2 (19.3)	46	82.3 (21.7)
t5	104	179.4 (35.6)	109	145.0 (22.0)	85	123.6 (22.9)	**76**	**127.0 (22.4)**
19ks	130	178.9 (26.8)	163	248.2 (36.7)	166	225.4 (27.6)	165	225.1 (27.4)
p2	182	278.5 (40.3)	252	339.1 (31.0)	232	313.0 (36.6)	222	307.5 (40.1)
s9234P	51	89.9 (25.2)	67	130.3 (35.1)	61	114.6 (33.1)	63	129.5 (28.1)
biomedP	83	126.5 (40.1)	156	214.2 (36.1)	123	194.5 (33.7)	91	159.6 (44.1)
s13207P	78	126.2 (21.3)	115	168.0 (12.4)	98	158.7 (16.9)	97	151.0 (17.3)
s15850P	104	184.0 (29.4)	90	150.7 (29.4)	89	133.6 (25.4)	**92**	**131.7 (26.2)**
industry2	264	602.1 (161.8)	601	881.0 (114.0)	499	762.7 (112.6)	310	768.2 (166.4)
industry3	263	484.6 (161.0)	618	1047.7 (308.4)	454	873.7 (274.1)	334	819.0 (341.1)
s35932	85	203.5 (53.7)	85	155.2 (39.8)	74	105.0 (25.8)	**73**	**103.6 (33.9)**
s38584	63	258.8 (120.3)	165	275.4 (64.5)	93	202.1 (73.1)	73	**202.6 (74.6)**
avq.small	343	620.3 (116.3)	312	535.6 (82.6)	228	516.3 (74.5)	**241**	**513.3 (73.1)**
s38417	147	368.0 (92.3)	91	180.8 (38.9)	82	144.6 (28.1)	**74**	**143.6 (24.8)**
avq.large	373	759.3 (123.9)	376	584.8 (87.7)	348	567.0 (89.3)	382	**567.1 (87.4)**
golem3	2175	3217.9 (304.0)	5084	5736.0 (229.1)	1935	3077.8 (312.3)	2405	**3009.5 (313.4)**

TABLE 3. Different versions of the Likelihood Greedy scheme

returned by the algorithm. In fact, this will require that each free node of h is assigned to Set_1. Therefore, if node v is assigned to Set_1, then the probability of h not being broken is $\frac{1}{2^{|h|_f-1}}$, in the crude assumption that each node has equal probability of being added to either set and that the individual additions are done independently. In fact, let x_h be a random variable associated to our random heuristic and whose value is 1 if edge h is in the final cutset returned by the algorithm and 0 otherwise. Because assigning v to Set_0 breaks h then the expected value $E[x_h|v \to Set_0] = 1$. On the other side $E[x_h|v \to Set_1] = 1 - \frac{1}{2^{|h|_f-1}}$. Thus $g_h(v) = g_h(v;1) = E[x_h|v \to Set_0] - E[x_h|v \to Set_1] = \frac{1}{2^{|h|_f-1}}$.

This greedy scheme is not to be confused with the probability-based approach proposed by Dutt and Deng [**5**] in the framework of iterative-improvement methods. It is of interest to observe that rough estimates of probabilities to end up in the different sets are useful in both algorithms to greatly improve the performance.

In order to consider in more detail the effect of this probabilistic strategy we consider also a version called (k)-Likelihood Greedy, in which the gain is set to zero if the number of free nodes in an hyperedge is greater than k.

In Table 3 the three columns in addition to that for FM (that is always reported to ease the comparison) list the results obtained by the (3)-Likelihood Greedy, (10)-Likelihood Greedy, and the Likelihood Greedy without limitations on the size of the hyperarcs. When k increases in (k)-Likelihood Greedy the performance improves rapidly at the beginning ($k = 2, 3$) and then reaches a "performance plateau", explained by the fact that hyperedges with large number of nodes are rare in the benchmark. When one considers the average values, the performance of the Likelihood Greedy scheme without limitations is in general better than that

Graph	Fiduccia		Greedy (Likelihood)		Greedy + Fiduccia		Perf. Ratio
	Min	Ave (Sdev)	Min	Ave (Sdev)	Min	Ave (Sdev)	Min / "Best"
baluP	30	66.3 (18.6)	29	51.4 (14.1)	**27**	**32.9 (7.5)**	**1**
p1	47	74.9 (13.7)	63	82.9 (11.0)	**47**	**69.5 (8.4)**	**1**
bm1	49	75.8 (14.9)	62	86.2 (9.9)	**48**	**72.9 (9.0)**	1.02
t4	80	135.6 (25.4)	67	104.4 (19.8)	**60**	**89.0 (14.5)**	1.25
t3	62	108.7 (22.1)	60	93.9 (25.7)	**60**	**82.2 (17.9)**	1.05
t2	124	175.5 (24.2)	103	141.4 (18.1)	**94**	**127.0 (16.8)**	1.08
t6	60	91.0 (14.8)	67	84.9 (9.6)	**60**	**76.5 (8.7)**	**1**
structP	41	55.4 (8.6)	46	82.3 (21.7)	**41**	57.1 (9.0)	1.24
t5	104	179.4 (35.6)	76	127.0 (22.4)	**74**	**102.2 (16.6)**	1.04
19ks	130	178.9 (26.8)	165	225.1 (27.4)	**127**	181.3 (21.9)	1.22
p2	182	278.5 (40.3)	222	307.5 (40.1)	**171**	**246.7 (34.4)**	1.23
s9234P	51	89.9 (25.2)	63	129.5 (28.1)	**49**	106.5 (28.5)	1.22
biomedP	83	126.5 (40.1)	91	159.6 (44.1)	84	**120.5 (33.9)**	1.01
s13207P	78	126.2 (21.3)	97	151.0 (17.3)	91	128.2 (16.2)	1.71
s15850P	104	184.0 (29.4)	92	131.7 (26.2)	**76**	**117.3 (24.7)**	1.80
industry2	264	602.1 (161.8)	310	768.2 (166.4)	**216**	**517.2 (128.0)**	1.28
industry3	263	484.6 (161.0)	334	819.0 (341.1)	**253**	555.9 (223.2)	1.04
s35932	85	203.5 (53.7)	73	103.6 (33.9)	**73**	**101.6 (29.5)**	1.78
s38584	63	258.8 (120.3)	73	202.6 (74.6)	**56**	**138.8 (59.3)**	1.19
avq.small	343	620.3 (116.3)	241	513.3 (73.1)	**220**	**474.6 (79.7)**	1.73
s38417	147	368.0 (92.3)	74	143.6 (24.8)	**68**	**128.8 (22.5)**	1.36
avq.large	373	759.3 (123.9)	382	567.1 (87.4)	**270**	**514.1 (70.4)**	2.12
golem3	2175	3217.9 (304.0)	2405	3009.5 (313.4)	**1633**	**2367.2 (265.7)**	1.14

TABLE 4. FM starting from Likelihood Greedy solution

of Weighted Greedy. On the contrary, when one considers the *minimum* values, some of the advantage is lost, and Weighted Greedy obtains superior results in about 50% of the cases. This effect is caused by the smaller standard deviation of results obtained by Likelihood Greedy: a more accurate selection of the node to be added improves the average results but decreases the amount of "diversification." In the last columns of Table 3 a bold value indicates that FM results are beaten or duplicated. Let us note that the performance is particularly significant if one considers that the CPU times requires by the Likelihood Greedy scheme are less than those required by FM, in some cases by a large factor, see Table 5.

5. Fiduccia-Mattheyses starting from our greedy solutions

Because greedy schemes are quite effective and fast, it is worth considering the solution quality obtained if one runs the Fiduccia-Mattheyses algorithm starting from a solution generated by a greedy scheme instead that from a random partition.

Table 4 displays the results obtained by executing the Fiduccia-Mattheyses procedure starting from a solution generated by the *Likelihood Greedy* heuristic. In addition, the last column shows the "performance ratio" of the combination, defined as the "Min" value (obtained in the 100 runs) divided by the "Best" heuristic value of Table 1.

6. Implementation and computational complexity

A main contribution of Fiduccia and Mattheyses [7] was the use of the now standard *buckets* data structure: nodes with a same gain are collected into the same bucket, so that the nodes with maximum (or minimum) gain values are immediately available. Motivated by the results in [**8**], our buckets are implemented as doubly linked lists and we confirmed experimentally that the following policy is most convenient: when $g(v; i)$ increases then v is inserted at the beginning of the bucket corresponding to the new value; when $g(v; i)$ decreases then v is inserted at the end of the new bucket. When we are extracting a node of maximum gain (for insertion into Set_0 or Set_1) then we take the first node in the corresponding bucket. For a consistent scheme the above policy translates as follows: when $g(v)$ is increasing then v is inserted at the beginning of the bucket corresponding to the new value; when $g(v)$ is decreasing then v is inserted at the end of the new bucket. When we are extracting a node of maximum gain (for insertion into Set_1) then we take the first node in the bucket of maximum gain; when we extract a node of minimum gain (for insertion into Set_0) then we take the last node in the bucket of minimum gain nodes. As suggested in [**8**], the randomization is introduced by permuting the nodes randomly at the beginning of the algorithm, just before insertion into the buckets. The remaining steps of the algorithms are deterministic with only one exception: when both sides of the partition contain at least *min_size* nodes and the maximum gains achievable by inserting into Set_0 and into Set_1 are the same, then we choose at random and with equal probability whether to perform the next insertion into Set_0 or into Set_1.

For the additional implementational details it is convenient to distinguish between the various schemes considered, as it is done in the next two subsections.

6.1. Δf Greedy and Critical Edges Greedy. Given an hypergraph $H(V, E)$, the cardinality $|e|$ of an edge $e \in E$ is the number of nodes contained in e. We denote by $card(H)$ the maximum cardinality of an edge in E. Let p denote the number of pins in the hypergraph, that is $p = \sum_{e \in E} |e|$. The degree of node v is the number of hyperedges containing v. We denote by $deg(H)$ the maximum degree of a node in H.

Storing an hypergraph requires an amount of memory which is linear in p, the number of pins. The time needed to input the hypergraph is hence $\Theta(p)$. We will show that Δf and Critical Edges Greedy require $\Theta(p)$ time to return a partition together with its cut value. One cannot ask for more, because computing the cut value of a given partition requires $\Theta(p)$ time on its own.

Let us denote by γ the range of different possible gain values during execution. Let us note that the value of γ depends on the single greedy scheme under consideration. In particular, both in Δf Greedy and Critical Edges Greedy, the gain of node v is an integer with absolute value bounded by the degree of node v, hence $\gamma \leq 2\ deg(H) + 1$. This allows for the standard buckets data structure, the one introduced by Fiduccia and Mattheyses [**7**]. In particular, buckets for all possible gain values are allocated, together with two additional arrays. The first stores for each node v the gain of v (and hence which bucket contains v). The second records the position of v inside the bucket associated to $g(v)$. This ensemble can be thought of as a two-coordinate system which results in an efficient (constant time) implementation of the needed operations. We refer to [**7, 8, 2**] and [**1**] for a detailed introduction to the standard use of buckets. Let us only mention that the data structure permits to update the gains of the nodes in constant time while keeping the buckets of minimum and maximum gains always at disposal. In particular, choosing the next node to be fixed (the first or last node into the minimum or maximum gain bucket) is a constant time operation whose cost can be considered as $O(|V|)$ for the overall execution. Therefore it remains to bound the complexity of updating the gains. As we said above the gain of node v can be seen as the sum of many terms, one for every hyperedge containing v:

$$gain(v) = \sum_{h \ni v} g_h(v)$$

Moreover, let us note that $g_h(v_1)$ and $g_h(v_2)$ have the same value for any two free nodes $v_1, v_2 \in h$. When this common value $g_h(.)$ changes, we say that edge h *changes status*. Each time a node is fixed we must look at all edges h containing v and check if they change their status. If this happens then the gain values of all free nodes in h must be updated. In detail, the following operations are executed:

```
1.  for each node v
2.      fix v either to 0 or 1
3.      for each edge h containing v
4.          if h changes status by fixing v then
5.          update the gain of each free node in h.
```

In the case of Δf Greedy each edge h undergoes one or two status changes: when a first node of h is fixed (either to 0 or 1) and when h gets eventually broken. Thus the total number of gain updates is $\Theta(p)$.

In the case of Critical Edges Greedy each edge h undergoes at most one status change, if it eventually becomes 0-critical or 1-critical. However when this happens h contains precisely one free node and hence the total number of gain updates is $O(|E|)$.

Let us now consider the computational complexity cause by the control structure in lines 1. — 4. For each edge h we store $|h|_0$, $|h|_1$ and $|h|_f$. Each time we test *"if h changes status"* we first update the values of $|h|_0$, $|h|_1$ and $|h|_f$. From the knowledge of $|h|_0$, $|h|_1$ and $|h|_f$ we can readily decide if h gets broken or fixed to either 0 or 1 (Δf Greedy). We can also check if h becomes critical (Critical Edges Greedy). We conclude that testing *"if h changes status"* requires constant time in all three schemes. The cost caused by the control structure of lines 1. — 4. is therefore $\Theta(p)$. To summarize, the complexity of Δf and Critical Edges Greedy is $\Theta(p)$.

The above asymptotic complexity analysis is complemented by the measured average PU times of the different greedy schemes in Table 5, where the effects of non-asymptotic terms is also visible.

6.2. Weighted Greedy and Likelihood Greedy. The situation is significantly different when considering Weighted Greedy and Likelihood Greedy. In fact, standard buckets permit to update the gains of the nodes in constant time if the gain values are integers of bounded value. If this is not the case, then binary heaps can be used to store the gain values. Our implementation of Weighted and Likelihood Greedy employs a binary heap structure. In particular, the binary heap template "sortseq" from LEDA [**17**] has been used. Here, the amortized time cost for a single bucket operation (node update, extract max or min) is $O(\log b)$, where b is the maximum number of buckets ever contained in the list. Obviously $b \leq |V|$.

Graph	Fiduccia Ave	Δf Ave	Critical Ave	Weighted Ave	Likely(3) Ave	Likely(10) Ave	Likely Ave	Likely + Fid Ave
baluP	0.049	0.012	0.008	0.014	0.007	0.042	0.054	0.083
p1	0.071	0.013	0.005	0.016	0.008	0.040	0.053	0.082
bm1	0.072	0.011	0.008	0.017	0.008	0.043	0.054	0.091
t4	0.182	0.027	0.010	0.045	0.016	0.209	0.151	0.226
t3	0.259	0.027	0.013	0.085	0.015	0.190	0.237	0.302
t2	0.200	0.027	0.012	0.042	0.018	0.207	0.129	0.211
t6	0.270	0.028	0.013	0.051	0.017	0.049	0.212	0.288
structP	0.208	0.027	0.014	0.028	0.019	0.043	0.059	0.172
t5	0.433	0.048	0.017	0.079	0.024	0.238	0.277	0.406
19ks	0.648	0.049	0.023	0.116	0.027	0.084	0.428	0.701
p2	0.619	0.050	0.021	0.081	0.029	0.088	0.327	0.583
s9234P	1.302	0.100	0.049	0.073	0.054	0.093	0.175	0.605
biomedP	1.750	0.127	0.049	0.140	0.057	0.123	0.460	1.081
s13207P	1.960	0.159	0.081	0.121	0.088	0.138	0.294	0.966
s15850P	2.207	0.194	0.101	0.144	0.113	0.179	0.377	0.895
industry2	4.323	0.310	0.128	0.945	0.164	0.343	3.134	4.263
industry3	4.202	0.429	0.171	0.407	0.229	0.444	1.493	3.748
s35932	6.052	0.387	0.169	0.385	0.193	0.315	0.578	1.097
s38584	6.284	0.489	0.226	0.455	0.247	0.421	1.386	2.995
avq.small	7.943	0.547	0.216	1.530	0.250	0.422	5.772	7.079
s38417	5.977	0.511	0.261	0.409	0.262	0.415	1.014	2.430
avq.large	8.632	0.612	0.259	1.686	0.292	0.490	8.181	8.283
golem3	43.129	3.072	1.676	2.254	2.806	4.422	5.250	39.729

TABLE 5. Average CPU times obtained for the different algorithms

Not only the cost of updating a single gain value is now $O(\log |V|)$ as discussed before, but the total number of gain updates increases.

The crucial point is that in these greedy schemes edges change status each time one of their nodes is fixed. For any edge h, this happens precisely $|h|$ times, as many as the nodes in h. When edge h changes status, we update the gain of at most $|h|$ nodes. The total number of gain updates is therefore $\Theta(\sum_{h\in E(H)} |h|^2)$. The complexity of these greedy schemes is therefore $O(\sum_{h\in E(H)} |h|^2 \log |V|)$.

6.3. CPU times. The above described computational complexities agree in a qualitative way with the measured CPU times, although the non-asymptotic terms in the complexity functions are still observable for the small size problems. In particular, the terms proportional to $|V|$, the number of nodes, are not negligible with respect to the higher order terms.

When one analyzes the computation times, see Table 6.3, one observes a roughly comparable (and in some cases smaller) average computation times for the combined option (Likelihood Greedy followed by Fiduccia-Mattheyses) than for MF starting from random solutions, especially for the larger graphs. This phenomenon is due to the fact that, while our greedy schemes are not much more expensive than what is required for a random assignment and function evaluation, the time subsequently spent in the Fiduccia-Mattheyses refinement phase tends to be smaller if one starts from a reasonably good initial partition. Let us remember that, when the average and minimum values of the combined option are considered, the results are better than those of FM in most cases, as it is shown in Table 4.

7. Conclusions

Motivated by a previous success with some greedy heuristics for the graph partitioning problem [**3, 2**], we investigated in this paper the behavior of a greedy approach to the hypergraph partitioning problem.

The conclusions to be derived from our experimental tests are that these techniques are *not* competitive with the state of the art, that is given by multi-level algorithms especially tuned to the hypergraph partitioning instances of interest for VLSI applications. The best values obtained are in most cases within 20-30% but in some cases up to two times larger than the best heuristic values.

Nonetheless, two advantages of our greedy schemes against more sophisticated methods are simplicity and low computational complexity. More surprisingly, if we look at the quality of the best solution over independent repetitions and within a same amount of allotted CPU time, some of our greedy schemes compare well against heuristics which have been the state of the art until some years ago, in particular the Fiduccia-Mattheyses algorithm. The use of independent repetitions is also at the basis of the popular "greedy randomized adaptive search procedure" (GRASP) approach [**15**]. The algorithms considered in this paper are simpler, because no criteria for the construction of candidate lists is considered, beyond the criterion given by having the same gain values (ties). Even the average quality of solutions returned after a single run could be of interest for some applications, considering that the time requirements are in some cases comparable with those required for a random assignment plus objective function (cut value) evaluation. Moreover, our schemes can be employed to produce initial partitions for *improvement-based* heuristics. For example, we compared the classical Fiduccia-Mattheyses heuristic, starting from a random initial solution, against the same heuristic, but starting from solutions produced by a greedy scheme. While the computation time of the two algorithms is roughly the same, the improvement in terms of solution quality is significant.

Of course, the present work is a preliminary investigation. The next step is to integrate our proposed greedy techniques into state-of-the-art multi-level techniques.

References

[1] C.J. Alpert and A.B. Kahng, *Recent directions in netlist partitioning: A survey*, UCLA Computer Science Department, Los Angeles, 1996.

[2] R. Battiti and A. A. Bertossi, *Differential greedy for the 0–1 equicut problem*, Network Design: Connectivity and Facilities Location (D.Z. Du and P.M. Pardalos, eds.), DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 40, American Mathematical Society, 1997, pp. 3–22.

[3] ———, *Greedy and prohibition–based heuristics for graph-partitioning*, Tech. Report UTM-97-512, Dip. di Matematica, Univ. di Trento, 1997, revised version to appear in IEEE Trensactions on Computers.

[4] N.C. Chou, L.T. Liu, and C.K. Cheng, *Circuit partitioning for huge logic emulation systems*, Proc. 31th ACM/IEEE Design Automation Conference, 1994, pp. 244–249.

[5] S. Dutt and W. Deng, *A probability-based approach to VLSI circuit partitioning*, Proc. 31th ACM/IEEE Design Automation Conference, June 1996.

[6] ———, *VLSI circuit partitioning by cluster-removal using iterative improvement techniques*, Proc. IEEE/ACM International Conference on CAD, November 1996.

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

[8] L.W. Hagen, D.J.-H. Huang, and A.B. Kahng, *On implementation choices for iterative improvement partitioning algorithms*, IEEE Trans. Computer-Aided Design (1995), submitted.

[9] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *Multilevel hypergraph partitioning: Applications in VLSI domain*, Tech. report, University of Minnesota, Department of Computer Science, April 1997.

[10] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Technical J. **49** (1970), 291–307.

[11] B. Krishnamurthy, *An improved min-cut algorithm for partitioning VLSI networks*, IEEE Trans. Computers **33** (1984), no. 5, 438–446.

[12] R. Kuznar, F. Brglez, and K. Kozminski, *Cost minimization of partitions into multiple devices*, Proc. 30th ACM/IEEE Design Automation Conference, 1993, pp. 315–320.

[13] ______, *Multi-way netlist partitioning into heterogeneus: fpgas and minimization of total device cost and interconnect*, Proc. 31th ACM/IEEE Design Automation Conference, 1994, pp. 238–243.

[14] ______, *A unified cost model for min-cut partitioning with replication applied to optimization of large heterogeneous fpga partitions*, Proc. 31th ACM/IEEE Design Automation Conference, 1994, pp. 271–276.

[15] M. Laguna, T. A. Feo, and H. C. Elrod, *A greedy randomized adaptive search procedure for the two-partition problem*, Oper. Res. **42** (1994), 677.

[16] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*, John Wiley and Sons, 1990.

[17] K. Mehlhorn and S. Naeher, *LEDA, a platform for combinatorial and geometric computing*, Communications of the ACM **38** (1995), no. 1, 96–102.

[18] B. Mobasher, N. Jain, E.H. Han, and J. Srivastava, *Web mining: Pattern discovery from world wide web transactions*, Tech. Report TR-96-050, University of Minnesota, Department of Computer Science, 1996.

[19] S. Shekhar and R. Aggarwal, *Partitioning similarity graphs: A framework for declustering problems*, Tech. Report TR-94-18, University of Minnesota, Department of Computer Science, 1994.

[20] ______, *Clustering roadmaps for routing: A hypergraph approach*, Tech. report, University of Minnesota, Department of Computer Science, 1997.

[21] S. Tragoudas, R. Farrell, and F. Makedon, *Circuit partitioning into small sets: A tool to support testing with further applications*, Proc. 28th ACM/IEEE Design Automation Conference, 1991, pp. 518–521.

[22] N.S. Woo and J. Kim, *An efficient method of partitioning circuits for multiple-fpga implementation*, Proc. 30th ACM/IEEE Design Automation Conference, 1993, pp. 202–207.

(R. Battiti) Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Povo (Trento) Italy

E-mail address, R. Battiti: `battiti@science.unitn.it`

(A. Bertossi) Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Povo (Trento) Italy

E-mail address, A. Bertossi: `bertossi@science.unitn.it`

(R. Rizzi) Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Povo (Trento) Italy

E-mail address, R. Rizzi: `rrizzi@rtm.science.unitn.it`