Greedy, Prohibition, and Reactive Heuristics for Graph Partitioning

Roberto Battiti, *Member*, *IEEE Computer Society*, and Alan Albert Bertossi

Abstract—New heuristic algorithms are proposed for the Graph Partitioning problem. A greedy construction scheme with an appropriate *tie-breaking* rule (MIN-MAX-GREEDY) produces initial assignments in a very fast time. For some classes of graphs, independent repetitions of MIN-MAX-GREEDY are sufficient to reproduce solutions found by more complex techniques. When the method is not competitive, the initial assignments are used as starting points for a prohibition-based scheme, where the prohibition is chosen in a randomized and *reactive* way, with a bias towards more successful choices in the previous part of the run. The relationship between prohibition-based diversification (Tabu Search) and the variable-depth Kernighan-Lin algorithm is discussed. Detailed experimental results are presented on benchmark suites used in the previous literature, consisting of graphs derived from parametric models (random graphs, geometric graphs, etc.) and of "real-world" graphs of large size. On the first series of graphs, a better performance for equivalent or smaller computing times is obtained, while, on the large "real-world" instances, significantly better results than those of multilevel algorithms are obtained, but for a much larger computational effort.

Index Terms—Graph bisection, graph partitioning, heuristic algorithms, iterative improvement, local search, reactive search.

1 Introduction

The graph partitioning problem on a graph $G = (V, E)$, V being the set of vertices and E the set of edges, consists of dividing the vertices into disjoint subsets such that the number of edges whose endpoints are in different subsets is minimized. We consider the *balanced* partitioning problem, where the difference of cardinalities between the largest and the smallest subset is at most one. When the number of subsets is equal to two, equivalent terms are *graph bisection*, or *graph bipartitioning* problem.

In detail, a $0/1$ *balanced bipartition* or *bisection* of a graph $G(V, E)$ is an unordered pair $(set0, set1)$ of subsets of V such that $set0 \cup set1 = V$ and $set0 \cap set1 = \emptyset$. In addition, the difference between the cardinalities of the two sets, i.e., $||set0| - |set1||$ is as small as possible: zero if V contains an even number of vertices, one otherwise. An edge $(i, j) \in E$ is *cut* by a bisection if its endpoints belong to different subsets, i.e., if $i \in set0$ and $j \in set1$ or $i \in set1$ and $j \in set0$. In the minimum bisection problem, one aims at minimizing the *cut size*, denoted as $f(set0, set1)$, given by the number of edges that are cut by the given partition.

The partitioning problem arises in many areas of computer science, like parallel computing, sparse matrix factorization, network partitioning, and VLSI circuit placement. In particular, the partitioning problem models the placement of data onto a multiprocessor computer with distributed memory, where the computation load has to be balanced among the different processors and the amount of communication has to be minimized [53], [24], [55], [60], [32], [26]. The bisection problem is of great relevance in VLSI design because it models the problem of optimally placing "standard cells" to minimize the "routing area" required to connect the cells [19]. In addition, the bisection problem models the problem of minimizing the number of holes on a circuit board, subject to pin preassignment and layer preferences, and also has applications in physics to find the ground state magnetization of spin glasses [3]. A review of the related and more general hypergraph partitioning problem appeared in [2].

The graph bisection problem is a fundamental problem and has been studied extensively in the past [9], [10], [13]. The problem is NP-hard for general graphs, as well as for bipartite graphs [25], and even finding good approximation solutions for general graphs or arbitrary planar graphs is NP-hard [11]. Approximation algorithms for the bipartitioning problem, into two bounded but not necessarily equal-sized sets, are available, but they are not practical [39]. Therefore, heuristics appear as the only viable option for the solution of real-world partitioning problems in acceptable computing times.

While it is not the purpose of this paper to present an exhaustive review of heuristics for the problem, let us mention some references that are of particular interest for our work. In particular, different aspects of greedy heuristics are described in [20], [21], [55], [38], [5], while [33] presents a detailed empirical study of Simulated Annealing (SA), following the application proposed in [36], [37]. The starting point of our investigation was the recent paper by Bui and Moon [14]. It contains a detailed discussion of previous studies and heuristics for the problem, presents a state-of-the-art Genetic Algorithm (GA) for its solution, called BFS-GBA, with extensive experimental comparisons with Simulated Annealing approaches [37], [54], [51], [57], [33], and with the classic Kernighan-Lin (KL) algorithm [35]. Different implementa-

• *The authors are with the Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Povo (Trento), Italy.*
E-mail: {battiti, bertossi}@science.unitn.it.

Manuscript received 20 Feb. 1997; revised 23 Mar. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 103605.

tion aspects of the KL algorithm are discussed in [23], [31], [17], while a combination of KL with SA is proposed in [41].

While the reader is referred to [14] for a detailed discussion of the KL, SA, and GA heuristics, we will briefly summarize some Tabu Search approaches for the problem in Section 4. Tabu Search (TS), introduced by Glover [27] and, independently, by Hansen and Jaumard [29] with the term SAMD ("steepest ascent mildest descent"), is based upon local search and it adopts a simple *prohibition* scheme to obtain diversification. As will become clear in Section 4, the method is deeply related to the KL heuristic. Recent applications of TS to graph partitioning include [50] and [16]. In particular, [16] argues that the Genetic Algorithm presented in [45], [49] developed for the general bipartitioning is not appropriate for the 0-1 bipartitioning problem because it is dominated by the randomized greedy procedure (GRASP) proposed in [38]. Furthermore, the TS procedure proposed in [16] tends to beat GRASP on random graphs. It is therefore of interest to extend the computational comparison of [14] to include prohibition-based diversification methods.

This was the starting motivation of our work. In the process of comparing the different heuristic alternatives, a very fast and effective greedy procedure (MIN-MAX-GREEDY) was discovered. In particular, *independent* repetitions of MIN-MAX-GREEDY reach competitive results on a subset of the graphs considered in [14], while graphs with a geometric structure and, especially, random graphs require more refined algorithms. When the MIN-MAX-GREEDY procedure is used to generate starting points for short runs of TS and a simple *randomized* and *reactive* scheme is used to change the crucial parameter of TS, the *prohibition period*, a new heuristic with a superior performance with respect to BFS-GBA [14] is obtained.

When the size of the graphs becomes very large, a recent stream of research advocates the use of *multilevel* algorithms, in which the original graph is approximated by a sequence of increasingly smaller graphs. The smallest graph is then partitioned using an efficient technique, and this partition is propagated back through the hierarchy of graphs and refined. Multilevel techniques have been proposed in [12] and in [31], inspired by the work of [4]. In [31], the edge and vertex weights are modified during the coarsening. The effectiveness of different coarsening schemes and choices for the partitioning and refinement heuristics is investigated in [34]. A recent comparison of multilevel techniques with the *Helpful Sets* (HS) heuristic, where sets of vertices take part in repeated exchanges aiming at reducing the cut size, is presented in [43]. The state-of-the-art multilevel techniques consist of highly tuned methods designed to reduce the computational effort needed to partition large graphs. As expected, the multilevel techniques are much faster than our algorithm when the dimension of the (sparse) graphs becomes very large ($\approx$ 100,000-500,000 nodes). On the other hand, the solutions obtained by our algorithm are significantly better in most cases.

The remaining part of this paper is organized as follows: The benchmark graphs of [14], [33], [43] are briefly summarized in Section 2. Then, the work is subdivided into two main sections. The first (Section 3) is dedicated to the MIN-MAX-GREEDY heuristic. After a description of the motivation and of the improved experimental results, the effects of independent repetitions (Section 3.4) and of detailed implementation choices (Section 3.5) are analyzed.

The second main section (Section 4) is dedicated to *prohibition-based* heuristics. These techniques have a long history (the well-known Kernighan-Lin [35] algorithm can be interpreted as a simple prohibition-based scheme) and are now mostly known with the term "Tabu Search." The study proceeds in steps, by first studying a basic realization (Section 5) and the effect of the crucial parameter, the *prohibition period*, on the performance (Section 5.1). Two remedies are then proposed to avoid an explicit tuning phase: a simple but surprisingly effective *randomization* (Section 6) and a *reactive* (feedback) loop to bias the random choice in a way that depends on the previous history of the current search (Section 7). The final algorithm proposed (REACTIVE-RANDOMIZED-TS) integrates the new greedy construction with the use of a randomized and reactive choice of the prohibition and demonstrates a significantly better performance.

Finally, a comparison of the proposed algorithm with the state-of-the-art methods for very large sparse graphs is presented in Section 8.

2 Test Beds and Experimental Setup

The computational tests in this paper are executed on two groups of graphs. The first group consists of the same collection of graphs used in [14], obtained from the authors of that paper. The collection is composed of eight random graphs and eight geometric graphs originally proposed in [33], with a number of vertices ranging from 500 to 1,000, plus 24 graphs proposed in [14], with a number of vertices from 134 to 5,252 (eight random regular graphs, eight caterpillar graphs, and eight grid graphs). We refer to the original papers for a detailed discussion of the properties and results obtained on these graphs. The different classes are briefly described below.

- $Gn.d$: A random graph with n vertices, where an edge between any two vertices is created with probability p such that the expected vertex degree, $p(n-1)$, is d.
- $Un.d$: A random geometric graph with n vertices uniformly scattered in the unit square. Two vertices are connected by an edge if and only if their Euclidean distance is t or less, where $d = n\pi t^2$ is the expected vertex degree.
- $breg.n.b$: A random regular graph with n vertices of degree 3, whose optimal bisection size is b with probability $1 - o(1)$, see [10].
- $cat.n$: A caterpillar graph with n vertices. It is constructed by starting with a straight line (the *spine*), where each vertex has degree two except the outermost vertices. Each vertex on the spine is then connected to six new vertices [14], the legs of the caterpillar. With an even number of vertices on the spine, the optimal bisection size is 1.

TABLE 1
"Real-World" Benchmark Graphs Used in [43] (see text for details)

Graph	Number of nodes	Number of edges
airfoil1	4253	12289
big	15606	45878
wave	156317	1059331
nasa4704	4704	50026
bcspwr09	1723	2394
bcsstk13	2003	40940
DEBR12	4096	8189
DEBR18	262144	524285

- *rcat.n*: A caterpillar graph with n vertices, where each vertex on the spine has $\sqrt{n}$ legs. All caterpillar graphs have optimal bisection size of 1.
- *grid.n.b*: A grid graph with n vertices, whose optimal bisection size is b.
- *w-grid.n.b*: The same grid graph as above, but the boundaries are wrapped around.

While the first group contains graphs derived from simple parametric models, the second group contains graphs related to applications in the area of parallel computing that have been used in a recent comparison of multilevel partitioning techniques [43]. It contains four numerical grids (*airfoil1, big, wave, nasa4704*), two graphs obtained from the Harwell-Boeing collection [18] (*bcspwr09, bcsstk13*), and two De Bruijn networks (*DEBR12, DEBR18*). The numerical grids and the Harwell-Boeing graphs are widely used in the literature to show the performance of different partitioning methods on "real-world" problems, see, for example, [46], [28], while the De Bruijn networks are 4-regular Cayley graphs defined by shuffle- and shuffle-exchange permutations. All graphs have been obtained from the authors of [43]. The dimensions of these graphs are listed in Table 1.

Because we are not interested in sacrificing fast development times and easy reuse capabilities to obtain the best possible CPU times, all code has been developed in a high-level object-oriented language (C++). The compiler used is the g++ compiler from GNU, the target machine is a Digital AlphaServer 2100 Model 5/250 with four CPU Alpha, 1 GB RAM, 12 GB Hard Disk, with the OSF/1 vers. 4.0 operating system. No parallel processing has been used, all times are for a single CPU usage. A recent benchmark dedicated to integer operations is the SPECint92 set: The value obtained for the machine is of 277.1 SPECint92 for a single CPU. Therefore, our CPU is approximately 12.7 times faster than the Sun SPARC IPX used in [14] for integer operations. Although the machine is recent, it is not the state-of-the-art at the time of writing: New models are already available that are approximately five times faster for integer operations. The times listed do not include the input/output times and the times to create the initial graph data structures because these are the same for all algorithms and independent of the number of iterations.

While the CPU times are reported because they are of interest for users of the proposed algorithms and because they permit a comparison with previous approaches, the number of incremental changes (iterations) executed by the

MIN-MAX-GREEDY
1 $in0 \leftarrow$ random vertex $\in \{1, ..., n\}$
2 $in1 \leftarrow$ random vertex $\in \{1, ..., n\} \setminus \{in0\}$
3 $set0 \leftarrow \{in0\}$
4 $set1 \leftarrow \{in1\}$
5 $tobeadded \leftarrow V \setminus \{in0, in1\}$
6 $f \leftarrow 0$
7 if $(in0, in1) \in E$ then $f \leftarrow 1$
8 $addset \leftarrow 1$
9 while $|tobeadded| > 0$ do
10 $addset \leftarrow (1 - addset)$
11 $otherset \leftarrow (1 - addset)$
12 $minedges \leftarrow \min_{i \in tobeadded} E(i, otherset)$
13 $candidates \leftarrow \{i \in tobeadded$ such that $E(i, otherset) = minedges\}$
14(*) $maxedges \leftarrow \max_{i \in candidates} E(i, addset)$
15(*) $candidates \leftarrow \{i \in candidates$ such that $E(i, addset) = maxedges\}$
16 $bestvertex \leftarrow$ random vertex $\in candidates$
17 $addset \leftarrow addset \cup \{bestvertex\}$
18 $f \leftarrow f + minedges$
19 $tobeadded \leftarrow tobeadded \setminus \{bestvertex\}$
20 return f

Fig. 1. The MIN-MAX-GREEDY algorithm. (*) lines are not present in the standard greedy algorithm.

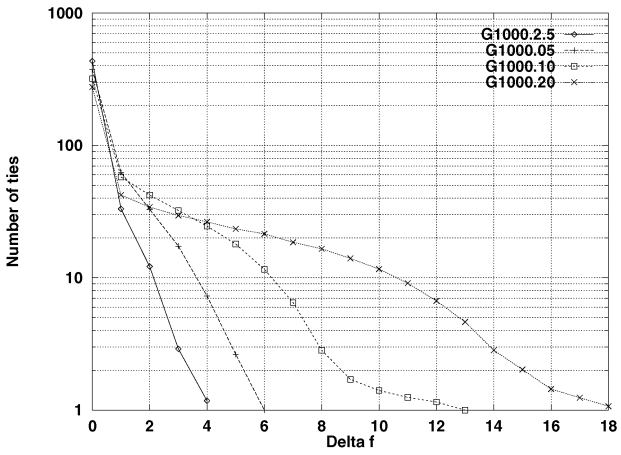

Fig. 2. Number of ties while executing the greedy algorithm on random graphs of 1,000 vertices and different densities. Averages of 100 runs.

local-search-based heuristics is a more natural measure of computational effort. This number is proportional to the CPU time with a factor that depends on machine, language, and compiler, but is approximately independent of the particular algorithm tested, provided that all algorithms are based on the same local search method (i.e., the neighborhood evaluation has the same computational cost, as it is the case for our prohibition-based algorithms).

3 MIN-MAX GREEDY

A simple greedy construction algorithm for the bipartitioning problem is immediately obtained by placing two random "seed" vertices into the two sets of the partition and by repeatedly adding to the two sets a vertex that produces the minimum possible increase Δf of the *cut size* f, the function to be minimized. Variations of this basic greedy approach are used for example in [38], where one vertex at a time is added, randomly selected among the first k vertices with smallest Δf, k being a fixed parameter, and in [16], where one starts from two "seed" vertices for the two sets and adds at each iteration a couple of vertices with minimal Δf. The modification introduced in our algorithm is minor, although with a crucial effect, therefore, Fig. 1 can be used also to describe the standard greedy algorithm. In fact, the standard algorithm is obtained by canceling line nos. 14-15 in the figure.

Let us introduce some notation. Let n be the number of vertices in the graph, $n = |V|$ and m be the number of edges, $m = |E|$. Given a subset set of the vertex set V and a vertex $i \in V$, let us define as $E(i, set)$ the number of edges incident on vertex i whose other endpoint is in the given set:

$$E(i, set) \equiv |\{(i, j) \in E \text{ such that } j \in set\}|. \quad (1)$$

For $j \in V$, let $X(j)$ be the set vertex j belongs to, $set0$ or $set1$. When the meaning is clear from the context, we will denote a given set by the digit 0 or 1, e.g., see line 8 in Fig. 1. The complete assignment is therefore represented by a binary string $X \in \{0, 1\}^n$. While the assignment is being constructed, a third value, e.g., $X(j) = -1$, can be used to signify that a vertex j is not yet assigned. Without loss of generality, let us assume that n is even: In this case, a *legal* assignment, i.e., such that the two sets are equal-sized, is represented by a binary string such that the number of bits set to 1 is equal to the number of bits set to 0. Let f be the function to be minimized:

$$f(X) = |\{(i, j) \in E \text{ such that } X(j) \neq X(i)\}|,$$

$addset$ the set of the partition one is adding a vertex to, and $otherset$ the other one. After adding to $addset$ an arbitrary vertex i not already contained in the two sets, the function increases by $\Delta f = E(i, otherset)$. A greedy addition to $addset$ consists, therefore, of adding a vertex i that minimizes the quantity $E(i, otherset)$. Let us now comment on the standard greedy algorithm in Fig. 1. The two random and different seed vertices are chosen and added to the two sets (lines 1-4). The initial set of yet unassigned vertices ($tobeadded$) is initialized to contain all vertices apart from the two seeds (line 5). The initial f value is 0 if the two seed vertices are not connected, 1 otherwise (lines 6-7). Then, the main loop follows (lines 10-19). In the loop, additions alternate between $set1$ and $set0$. First, the set where the vertex is to be added ($addset$) is "flipped" and, consequently, $otherset$ is updated (lines 10-11). Then, the minimum number of additional edges in the cut introduced by the new addition is determined ($minedges$ in line 12). Finally, the candidate vertices are determined as those producing the given $minedges$ value (line 13), lines 14-15 are skipped, and a random vertex in the $candidates$ set is chosen ($bestvertex$, line 16) and added to the given set (line 17). The cut size f is updated (line 18) and the vertices to be added loose $bestvertex$ (line 19).

3.1 Experimental Motivation

Let us now describe the motivation for introducing the modified algorithm called MIN-MAX-GREEDY. In a preliminary series of experiments, we observed a large number of *ties* (vertices producing the same Δf), especially for low-density graphs. Fig. 2 shows the average number of ties, i.e., the size of the $candidates$ set at line 13 of Fig. 1, as a function of the $minedges$ value during execution of the standard greedy algorithm of Fig. 1. Because the algorithm is randomized, see line 16, 100 runs with different seeds for the random number generator have been executed for each of the $G1000$ files described in Section 2. It can be noted that a very large number of candidates exists when the $minedges$ or, equivalently, Δf values are zero or very small, an event that happens from the beginning of the greedy algorithm until the last steps, where larger Δf values must be picked. The number of ties decreases for larger Δf values, but it remains large, especially in the denser graphs (see graph $G1000.20$). For the given $G1000$ graphs, during most iterations, the algorithm is forced to make a random choice between a large number of candidates, of a size comparable to that of the $tobeadded$ set.

The experimental findings can be explained by a simple model. Let us assume that d is the average degree of the nodes in a random graph and let us assume that nodes are added independently to the two sets without considering their degrees. After t additions, $t/2$ nodes will be added to $set0$ and $t/2$ to $set1$ (for simplicity, let us assume that t is even). On average, the nodes in $set0$ will have $d\,t/2$ edges to

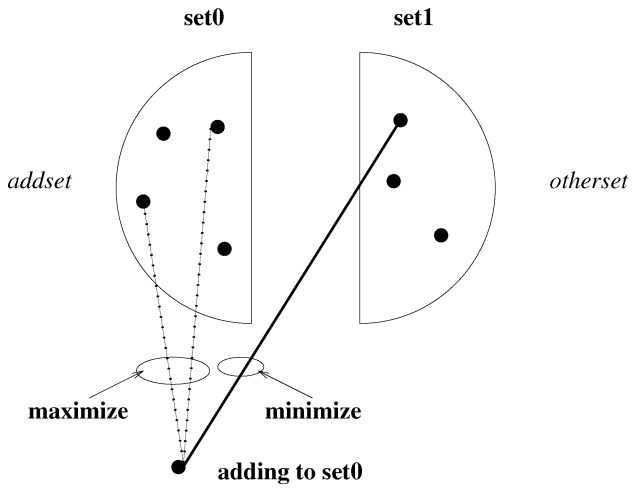

Fig. 3. MIN-MAX-GREEDY in action: situation before adding a new vertex to $set0$.

nodes in the $tobeadded$ set. Some of these edges can connect to the same nodes in the $tobeadded$ set, although, if t is very small with respect to n, all connections will be to different nodes with high probability. In any case, a set of *at least* $n - t - (d\,t/2)$ nodes in the $tobeadded$ set have no edges to nodes in $set0$ and will cause Δf equal to zero if added to $set1$. In other words, the number of ties in the initial part of the greedy construction, with the given assumptions, is equal to or larger than $n - t - (d\,t/2)$, see also [5].

3.2 A Tie-Breaking Rule

The MIN-MAX-GREEDY algorithm adds a tie-breaking rule as follows: Among the candidates with the same $minedges$ value, a subset with the largest number of edges to $addset$ is extracted (lines 14-15). The criterion is illustrated in Fig. 3. A random extraction is then executed if more than one candidate survives the sieve (line 16). The rationale behind this choice is that, when vertex i is added to $addset$, the internal edges connecting i to members of $addset$, numbering $E(i, addset)$ will never be part of the cut in the later phase of the greedy construction. The more edges become internal during the initial iterations, the less the probability that one will be forced to place large number of edges across the cut during later steps. Now, for a given number $minedges$ of new edges added across the cut, the maximum number of edges that can be packed as internal edges is $maxedges$ (derived in line 14, Fig. 1). The term "Min-Max" reflects the two-step selection process, where the primary goal is to minimize $E(i, otherset)$, the secondary one to maximize $E(i, addset)$.

3.3 Experimental Tests

The MIN-MAX-GREEDY algorithm has been tested on the entire benchmark suite of [14] and compared with the standard greedy algorithm and with the extraction of a random configuration, where a random subset of $n/2$ vertices is set to 1. The last algorithm has a computational complexity of $O(|E|)$, where the dominant contribution comes from the evaluation of f on the randomly generated string: f is initialized to 0, then all edges are examined, and, when the endpoints are in different sets, f is incremented.

The results of 1,000 runs on each task are collected in Table 2. For each method, the table lists the minimum f value found by all 1,000 runs, the average, with standard deviation of the distribution, and the CPU time in seconds for an individual run, for constructing the assignment and for calculating the f value. The "Best" column lists the heuristically best values from [14]. These values are either the globally optimal ones, for the caterpillar, grid, and regular graphs (with a probability close to one for these last graphs), or the best values obtained on the graphs by all algorithms considered in [14], including multistart KL, SA, GBA, and BFS-GBA. In the table, a value in boldface means that the value is the lowest minimum or average value obtained when one considers all algorithms summarized in the given table.

It can be noted that MIN-MAX-GREEDY reaches average f values that are significantly lower than the values obtained by the standard greedy algorithm (the *standard error* [48] of the average is equal to the standard deviation divided by the square root of the number of values, i.e., $\sqrt{1000} \approx 31$ in our case of 1,000 independent tests). Furthermore, results of interest are obtained by considering the Min values: While these are close, but still significantly larger than the Best value for the random G graphs, they are much closer and, in some cases, coincident with the Best value, for the geometric U graphs. In addition, almost all optimal values are reproduced for the regular, caterpillar, and grid graphs. Only for Breg500.20 and Breg5000.16 is a larger value is obtained. Let us note the dramatic improvement of the Min values for some classes of graphs. For example, the Min value for U1000.40 is 8,788 with a random assignment, 1,486 with the standard greedy, 741, close to the Best value of 737, for MIN-MAX-GREEDY. For Breg5000.16, the three Min values are: 3,600, 1,390, and 18, close to the Best value of 16.

The conclusion is that the *repetition of independent runs* of MIN-MAX-GREEDY easily solves most of the regular, caterpillar, and grid graphs. Let us note that the CPU time of each run is very small, ranging from about 1 millisecond to 100 milliseconds, and of the same order of magnitude as the CPU time required to build a random configuration and to evaluate its *cut size*. Therefore, even 1,000 repetitions require less than 60 seconds in most cases.

The reason why the average is much larger than the Min value and the standard deviation is large is related to the random choice of seeds (by chance, the two seeds may belong to the same optimal set) and to the remaining "blind" breaking of ties executed in line 16 of Fig. 1. Independent repetitions are therefore crucial to obtain competitive solutions.

In Section 3.4, we investigate how independent repetitions of MIN-MAX-GREEDY can provide competitive results for the regular, caterpillar, and grid graphs. In Section 7, it will be shown how the embedding of the MIN-MAX-GREEDY algorithm into a *prohibition-based* local search scheme achieves competitive results also for the G and U graphs.

3.4 Independent Repetitions of Min-Max Greedy

In order to measure how many independent repetitions are necessary in order to reach a certain performance, the statistical distribution of the results obtained by MIN-MAX-

GREEDY has been investigated. A total of 100,000 repetitions have been executed for each task. Some representative histograms are shown in Fig. 4, where the count in each bin is divided by the total number of repetitions to obtain an estimate of the probability.

In all cases, one observes a nonnegligible probability for values of the cut size that are very close to the optimal one. For example, on the Breg500.0 task, one has a probability close to 0.5 of obtaining the minimum value, while the remaining probability is clustered for a value of about 90. A larger spread of the distribution is present for the regular graphs with larger degrees, for the caterpillar graphs ((e), (f), (g), and (h) in Fig. 4), and for the grid graphs ((i), (j), (k), and (l) in Fig. 4).

The probability distribution affects the "best-so-far" value found during a sequence of repetitions in an immediate way,[1] see also [33]. For example, Fig. 5 shows how the "best-so-far" value evolves as a function of the repetitions k for the Breg500 tasks. The values reported are averages of 100 runs, each run consisting of a sequence of 1,000 repetitions.[2] To avoid cluttering the figure, the standard deviation bars are shown only for the Breg500.20 task. As predicted from the probability distribution, a couple of independent repetitions are sufficient to reach very low cut sizes for the Breg500.0 task, while a larger number is required for the denser graphs. In all cases, values very close to the optimal ones are reached in about 1,000 iterations, corresponding to a couple of seconds of CPU time, see Table 2.

The purpose of the following analysis is that of assessing the expected number of repetitions necessary to obtain either the minimum value or a very close approximation (Min plus Δ). From the 100,000 repetitions of MIN-MAX-GREEDY executed to study the probability distribution one derives the fraction of times such that the cut size obtained is less than or equal to Min + Δ, for different Δ values ranging from 0 to 5. The fraction, estimate of the probability, is then inverted to get an estimate of the expected number of repetitions necessary to reach a certain performance, reported in Table 3. It can be observed that all Min values, reached in 100,000 runs, coincide with those of [14]. Furthermore, in most cases, a very good approximation to the minimum is reached within acceptable numbers of repetitions for a total CPU time of the order of some seconds. The conclusion of this investigation is that the simple repetition of *independent* runs of MIN-MAX-GREEDY achieves results that are competitive with those obtained by having a population of *interacting* solutions in the Genetic Algorithm proposed in [14].

1. The probability that a given target value is obtained in at least one of k repetitions is equal to one minus the probability that all runs fail, in turn equal to the product of the individual failure probabilities, because of the independence assumption. All individual failure probabilities are equal and they are estimated by summing the frequencies of the bins in the histograms for values larger than the target.

2. Given the speed of each repetition, we did not use the more efficient, but less reliable, way described in [33] and based on large number of $m >> k$ of repetitions, used in a subsequent phase to extract k random samples.

3.5 Implementation of MIN-MAX-GREEDY

The CPU times shown in Table 2 are obtained through a careful implementation of the policy described in a high-level pseudolanguage in Fig. 1.

While the algorithm runs, a support "buckets" data structure is maintained, see Fig. 6. Two arrays of buckets, one for each set of the partition are maintained. $bucket[set][value]$ contains the set of nodes j such that the number of edges $E(j, set)$ is equal to $value$. This can be done in the case of the 0-1 (unweighted) partitioning problem because the range of possible $E(j, set)$ values is limited by the maximum vertex degree d_{max}, in turn upper bounded by n. A similar structure has been used in [23]. Let us consider the example of Fig. 6: The shaded areas represent vertices stored in the buckets. In a simple realization, requiring $O(n^2)$ memory, each bucket is an array of maximum size n containing the vertices with the given value in the first positions. If the available memory is scarce, a linked-list implementation can be used, requiring $O(n)$ memory. We will consider the array implementation because it is simpler to explain. The number of vertices contained for each value is stored in two $size$ vectors, for $set0$ and $set1$. The offsets of a given vertex in the two buckets is stored in two $offset$ vectors. The maximum and minimum value of $E(i, set)$ over all vertices $min[set]$ and $max[set]$ are maintained so that the first and last nonempty buckets are immediately retrieved when needed. For example, in Fig. 6, vertex 9 has $E(9, set0) = 2$, and $E(9, set1) = 6$. It appears in the bucket for $set0$ corresponding to value 2 with offset 2 (offsets start from 0) and in the bucket for $set1$ corresponding to value 6 with offset 1. Vertex 9 reaches the maximum number of connections (six) to $set1$.

To derive the computational complexity of the MIN-MAX-GREEDY algorithm, it is useful to separate the operations required to maintain the "buckets" data structure from the operations required to select the best vertex to add; that will be discussed in the following sections. For the analysis, let us assume that $|E|$ is $\Omega(|V|)$, the usual case for applications of partitioning algorithms.

The following operations are executed in $O(1)$ steps:

- INSERTION(v): For each set, vertex v is inserted at the bucket for value $E(v, set)$ in the first free position: $bucket[set][value][\ size[set][value]\ ]$. Therefore, $offset[set][v]$ becomes $size[set][value]$, then the given position of the $size$ array is incremented by one.
- DELETION(v): For each set, the vertex in the last bucket position is copied into the position containing v, then $size[set][E(v, set)]$ is decremented.
- INCREASE(set, v): Executed when $E(v, set)$ increases by one during the greedy algorithm. It is immediately realized by first deleting v from the appropriate bucket and then inserting it into the bucket corresponding to a value incremented by one.

The only nontrivial analysis to derive the complexity to maintain the "buckets" structure is related to the operations needed to update the max and min values. Now, the worst-case computational complexity required to update the max and min values is $O(1)$ after a single INSERTION(v),

TABLE 2
MIN-MAX-GREEDY Algorithm Compared with Standard Greedy and Random Initialization

Graph	Best	**Min-Max Greedy**			Greedy without Tie Breaking			Random Initialization		
		Min	Ave (St.Dev.)	CPU	Min	Ave (St.Dev.)	CPU	Min	Ave (St.Dev.)	CPU
G500.2.5	49	**61**	**75.6 (4.8)**	0.0035	113	138.0 (7.1)	0.0020	270	313.1 (12.6)	0.0007
G500.05	218	**248**	**275.3 (8.5)**	0.0052	309	343.5 (10.4)	0.0026	550	612.9 (17.0)	0.0008
G500.10	626	**678**	**711.3 (11.1)**	0.0072	737	778.9 (14.0)	0.0036	1106	1181.1 (24.0)	0.0011
G500.20	1744	**1828**	**1884.1 (17.8)**	0.0105	1878	1947.7 (19.9)	0.0058	2458	2564.6 (35.2)	0.0018
G1000.2.5	95	**125**	**149.7 (7.4)**	0.0105	240	276.7 (10.4)	0.0040	578	637.0 (18.2)	0.0019
G1000.05	445	**518**	**559.0 (12.0)**	0.0148	646	694.6 (14.2)	0.0050	1179	1249.8 (25.1)	0.0021
G1000.10	1362	**1487**	**1550.0 (17.2)**	0.0309	1618	1690.4 (20.5)	0.0072	2414	2533.4 (36.5)	0.0027
G1000.20	3382	**3598**	**3679.0 (25.0)**	0.0411	3714	3815.4 (27.9)	0.0110	4926	5059.1 (47.3)	0.0039
U500.05	2	**2**	**14.1 (9.1)**	0.0029	79	139.9 (19.0)	0.0025	590	643.1 (17.4)	0.0009
U500.10	26	**28**	**55.8 (22.8)**	0.0038	174	315.2 (41.5)	0.0034	1082	1178.9 (24.1)	0.0012
U500.20	178	**186**	**267.9 (50.6)**	0.0061	348	654.9 (97.1)	0.0053	2139	2278.4 (33.1)	0.0018
U500.40	412	**412**	**542.1 (101.3)**	0.0094	581	1300.7 (250.7)	0.0088	4247	4407.5 (42.0)	0.0028
U1000.05	1	**1**	**15.0 (9.4)**	0.0052	206	285.7 (24.1)	0.0047	1114	1196.9 (24.8)	0.0020
U1000.10	39	**40**	**90.5 (24.8)**	0.0072	429	622.1 (58.5)	0.0068	2222	2350.1 (34.5)	0.0025
U1000.20	222	**237**	**387.6 (78.1)**	0.0125	909	1331.6 (142.3)	0.0102	4502	4674.0 (47.9)	0.0035
U1000.40	737	**741**	**1064.8 (185.4)**	0.0190	1486	2615.5 (364.7)	0.0167	8788	9015.5 (65.6)	0.0060
Breg500.0	0	**0**	**47.5 (46.9)**	0.0030	124	150.5 (7.4)	0.0021	336	375.3 (14.1)	0.0009
Breg500.12	12	**12**	**62.7 (23.0)**	0.0027	126	149.0 (7.5)	0.0021	332	375.8 (13.6)	0.0009
Breg500.16	16	**16**	**61.3 (24.4)**	0.0026	124	149.6 (7.4)	0.0021	330	375.0 (13.2)	0.0009
Breg500.20	20	**26**	**70.1 (18.9)**	0.0026	128	150.6 (7.2)	0.0022	340	376.2 (13.4)	0.0010
Breg5000.0	0	**0**	**430.2 (438.4)**	0.0940	1392	1467.7 (23.4)	0.0225	3616	3749.4 (43.0)	0.0082
Breg5000.4	4	**4**	**378.9 (384.5)**	0.0622	1392	1468.5 (24.2)	0.0223	3624	3752.4 (44.4)	0.0087
Breg5000.8	8	**8**	**375.2 (355.7)**	0.0571	1402	1469.6 (23.2)	0.0225	3572	3751.7 (43.3)	0.0086
Breg5000.16	16	**18**	**397.6 (322.3)**	0.0568	1390	1468.8 (24.1)	0.0224	3600	3748.4 (42.0)	0.0093
Cat.352	1	**1**	**28.0 (25.3)**	0.0015	72	95.1 (7.3)	0.0014	143	175.5 (9.2)	0.0005
Cat.702	1	**1**	**56.2 (54.0)**	0.0028	155	188.2 (11.1)	0.0024	303	351.0 (12.8)	0.0011
Cat.1052	1	**1**	**85.4 (84.7)**	0.0043	233	281.4 (13.6)	0.0042	479	525.8 (15.4)	0.0018
Cat.5252	1	**1**	**422.4 (432.7)**	0.0259	1308	1404.2 (28.5)	0.0265	2473	2627.6 (36.2)	0.0096
RCat.134	1	**1**	**12.7 (8.9)**	0.0004	27	44.5 (5.3)	0.0005	46	67.2 (5.6)	0.0002
RCat.554	1	**1**	**55.2 (51.4)**	0.0025	161	211.7 (14.2)	0.0023	241	276.5 (11.7)	0.0009
RCat.994	1	**1**	**101.0 (98.6)**	0.0047	325	398.5 (19.1)	0.0041	457	496.8 (15.6)	0.0015
RCat.5114	1	**1**	**538.4 (546.6)**	0.0353	2074	2247.8 (48.3)	0.0218	2419	2557.2 (35.5)	0.0087
Grid100.10	10	**10**	**15.2 (2.9)**	0.0005	24	38.9 (5.1)	0.0004	66	90.9 (6.6)	0.0002
Grid500.21	21	**21**	**39.4 (8.7)**	0.0025	158	200.89 (13.5)	0.0025	435	478.0 (15.1)	0.0008
Grid1000.20	20	**20**	**66.2 (14.8)**	0.0056	347	405.6 (19.5)	0.0046	905	965.3 (21.4)	0.00220
Grid5000.50	50	**50**	**127.4 (35.5)**	0.0305	1953	2088.9 (44.6)	0.0248	4776	4925.8 (49.4)	0.0088
W-grid100.20	20	**20**	**26.0 (3.7)**	0.0007	30	46.2 (5.7)	0.0005	76	100.9 (6.9)	0.0002
W-grid500.42	42	**42**	**50.5 (9.7)**	0.0026	174	216.9 (14.5)	0.0022	450	501.1 (15.9)	0.0010
W-grid1000.40	40	**40**	**43.5 (5.8)**	0.0049	372	430.5 (19.8)	0.0048	928	1001.1 (22.3)	0.0019
W-grid5000.100	100	**100**	**225.3 (44.8)**	0.0285	1988	2137.8 (45.7)	0.0248	4844	5000.3 (48.5)	0.0083

Averages are on 1,000 tests. "Best" lists the heuristically best values from [14].

$O(d_{max})$ after a single DELETION(v), d_{max} being the maximum vertex degree (if a bucket becomes empty, the max or min value may jump to another value in the range 0, ..., d_{max}), and $O(1)$ after a single INCREASE(set, v). At a single iteration of MIN-MAX-GREEDY, the node $bestvertex$ is deleted and INCREASE must be called for all *neighbors* of $bestvertex$ still in $tobeadded$, for a total of $O(|E|)$ INCREASE operations during the entire greedy construction.

Now, if one considers the entire *sequence* of DELETION and INCREASE operations executed during MIN-MAX-GREEDY, one can demonstrate that the total computational complexity to update the max and min values is $O(|E|)$. Let us consider the min value first: Its value is nondecreasing so that a total of at most d_{max} changes happen. If one considers also the checks needed to see whether a bucket becomes empty, one obtains a total of $O(|E|)$ checks because of the $O(|E|)$ INCREASE operations, for a total complexity $O(|E|)$. The max value can both increase by one unit, after some calls of INCREASE, and decrease by a step of up to d_{max} units, after some calls of DELETION. Fortunately, by using *amortized analysis* techniques, one demonstrates that the complexity for the whole sequence is again $O(|E|)$, see, for example, [5]. In particular, by using the "accounting" method, one can "prepay" one unit of credit when max increases, so that the accumulated credit can be used when max decreases, for a total complexity $O(|E|)$, considering also the checks.

Therefore, if one considers that a total of $O(|E|)$ INCREASE operations, and that a total of $|V|$ INSERTION and DELETION operations are executed, the total computa-

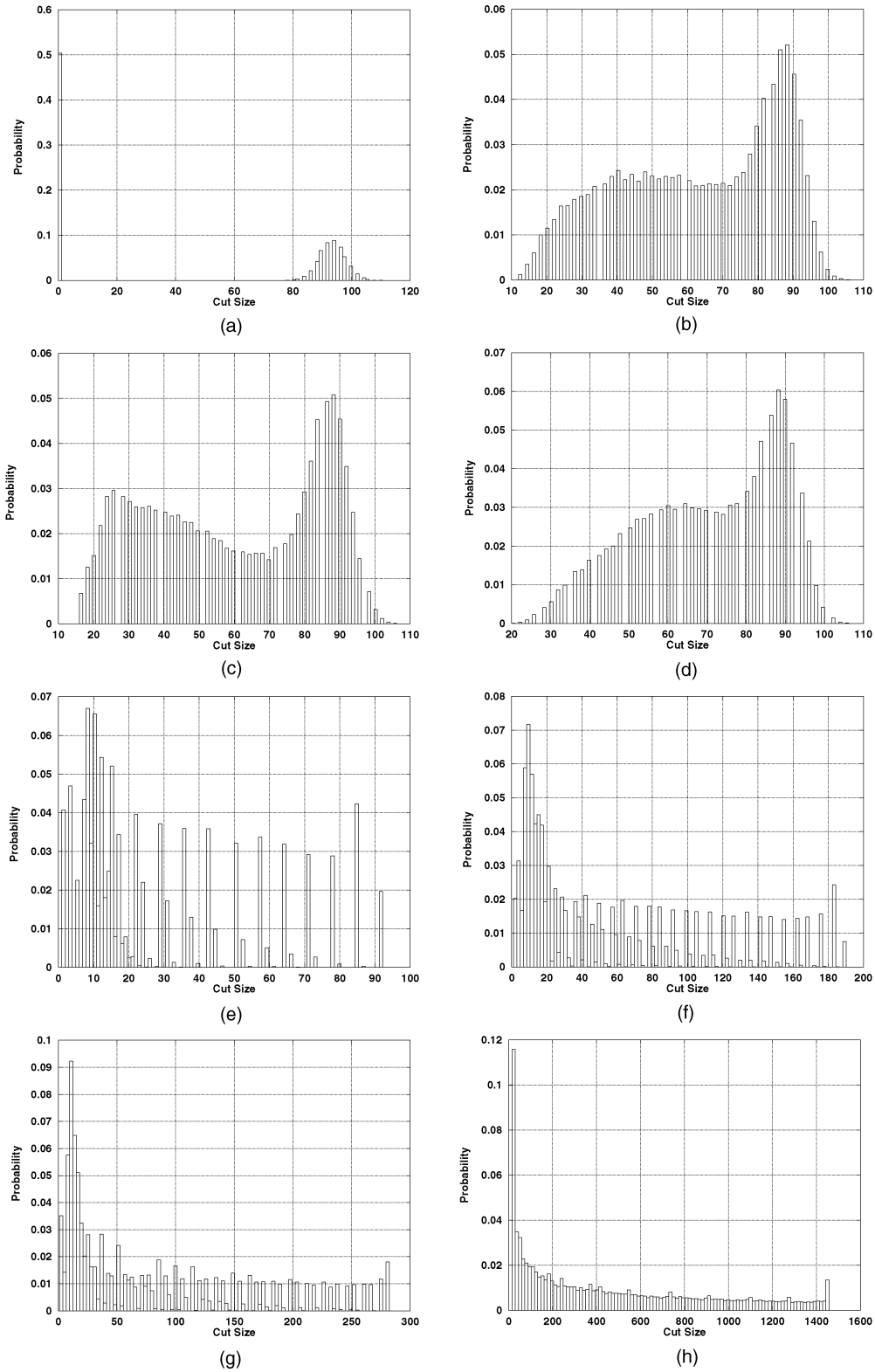

Fig. 4. MIN-MAX-GREEDY distribution of cut sizes (probability measured from 100,000 repetitions). (a) Breg500.0, (b) Breg500.12, (c) Breg500.16, (d) Breg500.20, (e) Cat.352, (f) Cat.702, (g) Cat.1052, (h) Cat.5252.

tional complexity required to maintain the "buckets" data structure (including max and min) during the MIN-MAX-GREEDY algorithm is $O(|E|)$.

Let us now consider how the choice of the next vertex to add in the MIN-MAX-GREEDY algorithm (lines 12-16 in Fig. 1) can be efficiently implemented. The first design choice is that of substituting the randomized selection among the winning $candidates$ (line 16) with a single randomization of the indices executed before the main loop starts.

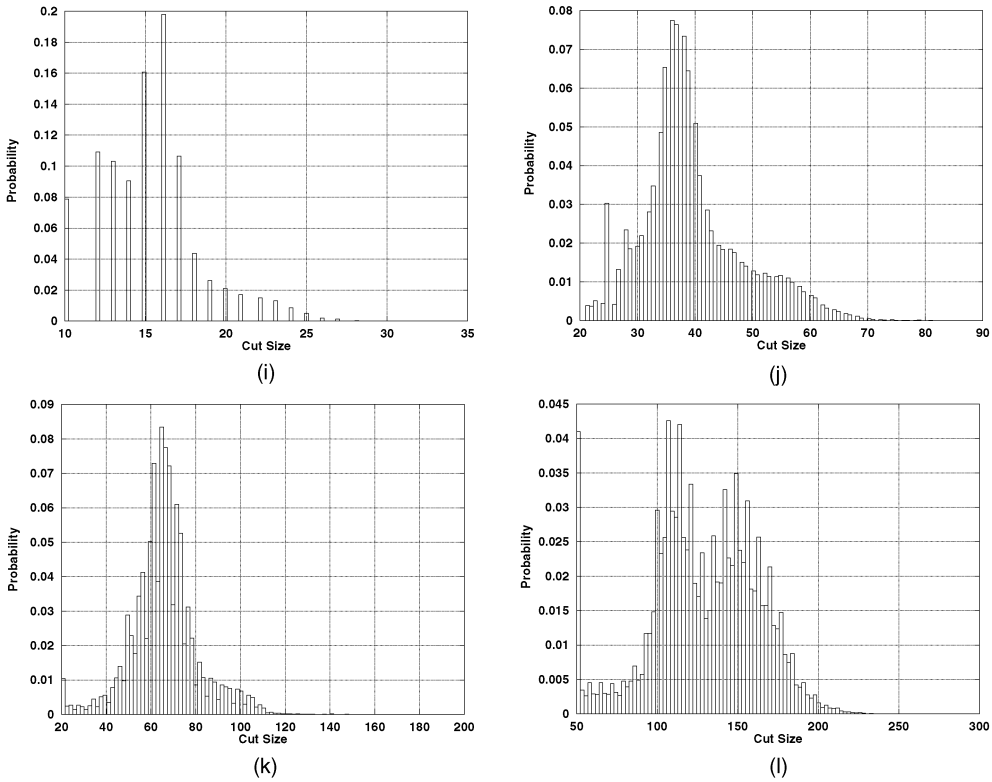

Fig. 4. (continued) MIN-MAX-GREEDY distribution of cut sizes (probability measured from 100,000 repetitions). (i) Grid100.10, (j) Grid500.21, (k) Grid1000.20, (l) Gird5000.50.

In detail, a random permutation of $\{1, \ldots, n\}$ is generated, in $O(n)$ steps, and the vertices are inserted into the $tobeadded$ set, realized by the buckets data structure, in the order given by the permutation (line 5 in Fig. 1). In this manner, the order of the vertices in the buckets (i.e., their $offset$ value), is randomized. The random choice of line 16 is then substituted by a deterministic choice, where the $first$ winning element encountered is chosen. Let us note that the new randomized algorithm obtained does not have the same statistical properties of the original one: After the first iteration in the loop, some INCREASE(set, v) operations will delete nodes and insert them *at the end* of different buckets and not in a random position—an operation that would require larger CPU times. In our tests, no significantly different behavior has been observed between the two randomization strategies, apart from a sizable reduction of the CPU times for the "initial randomization" strategy.

3.5.1 First-Min Option for BEST-VERTEX

Let us now consider how the single winning vertex is determined (lines 12-15 in Fig. 1), for an addition to a given set. Without loss of generality, let us assume that the addition is to $set0$. One possible choice (called "**first-min**", see Fig. 7) is to examine the vertices contained in the bucket corresponding to the minimum number of connections to $set1$ ($bucket[1][min[1]]$, the lower right bucket in Fig. 6), to determine one vertex i with the maximum $E(i, set0)$. Because, after the initial random permutation, one is satisfied with a deterministic choice, the examination is terminated immediately if one vertex with $E(i, set0) = max[0]$ is encountered (line 8, in Fig. 7).

A "naive" implementation is defined as the algorithm in Fig. 7 without line 8. If no vertex in the bucket reaches $max[0]$, the one with maximal $E(i, set0)$ and smallest offset in $bucket[1][min[1]]$ is picked. In the example of Fig. 6 it would be vertex number 4. The worst-case computational complexity of each call of the BEST-VERTEX routine is $O(|V|)$ because as many as $O(|V|)$ vertices can be considered at each iteration. The total complexity caused by the $O(|V|)$ calls of BEST-VERTEX during the MIN-MAX-GREEDY algorithm is, therefore, $O(|V|^2)$.

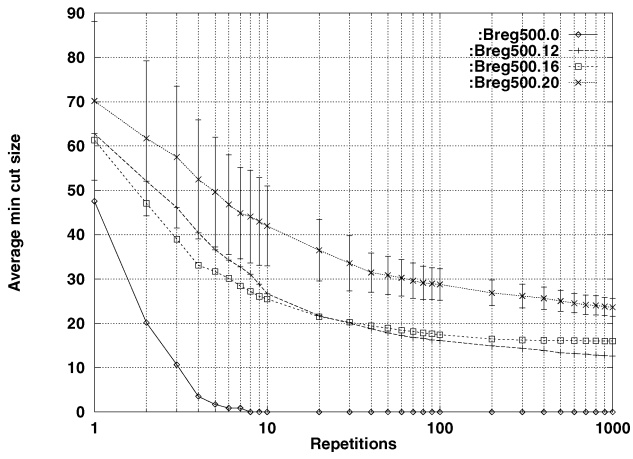

Fig. 5. MIN-MAX-GREEDY initialization: average min cut size as a function of the number of repetitions (Breg500 tasks).

TABLE 3
MIN-MAX-GREEDY Algorithm: Expected Number of Repetitions to Reach the Minimum Value Plus Δ, for $\Delta \in 0, 1, \ldots, 5$

Graph	Best	Min	Expected repetitions					
			Delta = 0	Delta =1	Delta =2	Delta =3	Delta =4	Delta =5
Breg500.0	0	**0**	1.98	1.98	1.98	1.98	1.98	1.98
Breg500.12	12	**12**	862.06	862.06	215.51	215.51	93.89	93.89
Breg500.16	16	**16**	147.71	147.71	51.73	51.73	28.97	28.97
Breg500.20	20	**20**	10000	10000	2222.22	2222.22	714.28	714.28
Breg5000.0	0	**0**	1.99	1.99	1.99	1.99	1.99	1.99
Breg5000.4	4	**4**	11.52	11.52	7.02	7.02	5.17	5.17
Breg5000.8	8	**8**	31.14	31.14	21.84	21.84	16.53	16.53
Breg5000.16	16	**16**	1694.92	1694.92	595.23	595.23	220.75	220.75
Cat.352	1	**1**	24.55	24.55	11.40	11.40	9.07	9.07
Cat.702	1	**1**	49.38	49.38	19.38	19.38	14.64	14.64
Cat.1052	1	**1**	75.41	75.41	28.47	28.47	20.24	20.24
Cat.5252	1	**1**	371.74	371.74	112.10	112.10	61.84	61.84
RCat.134	1	**1**	5.34	5.34	3.99	3.99	3.80	3.80
RCat.554	1	**1**	11.28	11.28	6.75	6.75	6.32	6.32
RCat.994	1	**1**	15.11	15.11	8.48	8.48	7.85	7.85
RCat.5114	1	**1**	35.22	35.22	17.14	17.14	14.85	14.85
Grid100.10	10	**10**	12.71	12.71	5.32	3.43	2.62	1.84
Grid500.21	21	**21**	257.06	130.71	78.12	57.83	21.00	19.28
Grid1000.20	20	**20**	96.15	96.15	86.58	78.12	70.52	64.68
Grid5000.50	50	**50**	25.66	25.66	24.39	23.28	22.49	21.87
W-grid100.20	20	**20**	6.95	6.95	6.95	6.95	2.22	2.22
W-grid500.42	42	**42**	3.07	3.07	2.19	2.19	1.98	1.98
W-grid1000.40	40	**40**	2.21	2.21	2.21	2.21	1.19	1.19
W-grid5000.100	100	**100**	29.59	29.59	29.59	29.59	22.12	22.12

Data from 100,000 tests. "Best" lists the optimal values from [14], "Min" the minimum value reached.

Let us now analyze the bookkeeping required at the end of a single iteration in the main loop of MIN-MAX-GREEDY. After $bestvertex$ is added to the given $addset$, the $E(i, set)$ values stored in the buckets data structure need to be updated. First, DELETION($bestvertex$) is called to eliminate the vertex from the buckets. Then, for all neighbors i of $bestvertex$ in the graph, the value $E(i, addset)$ increases by one and, if the neighbor is still in $tobeadded$, the routine INCREASE($addset, i$) must be called to update the buckets structure. After remembering the previous result that the total computational complexity to update the "buckets" data structure during the entire greedy construction is $O(|E|)$, one derives that the total computational complexity

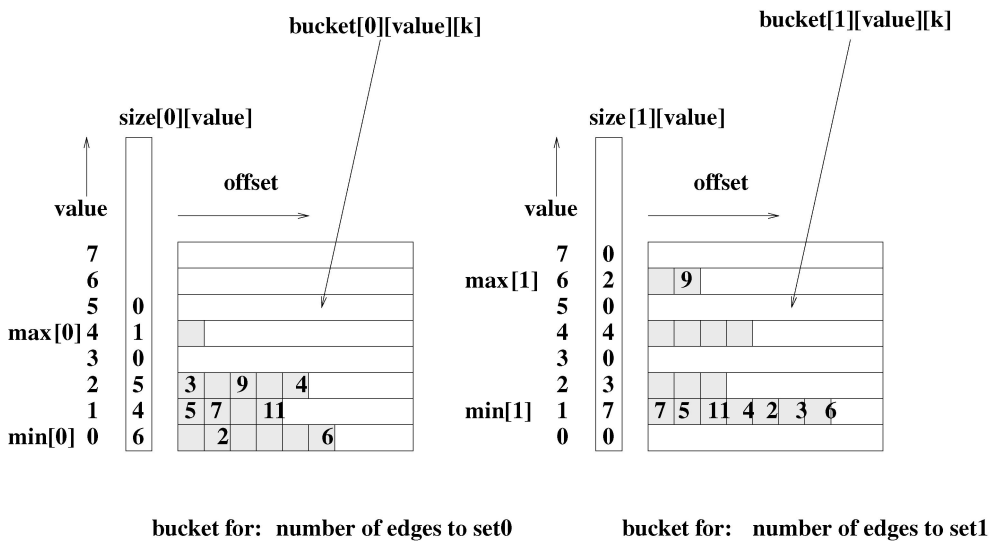

Fig. 6. MIN-MAX-GREEDY algorithm: "buckets" data structure.

```
BEST-VERTEX- first min implementation
1       otherset ← (1 − addset)
2       minedges ← min[otherset]
3       maxedges ← max[addset]
4       maxreached ← −∞
5       value ← minedges
6       for offset ← 0 to size[otherset][value] − 1 do
7           vertex ← bucket[otherset][value][offset]
8(*)        if E(vertex, addset) = maxedges then return vertex
9           if E(vertex, addset) > maxreached then
10              maxreached ← E(vertex, addset)
11              bestvertex ← vertex
12      return bestvertex
```

Fig. 7. MIN-MAX-GREEDY algorithm: determination of the winning candidate in the "first-min" option. Naive implementation if (*) is absent.

of MIN-MAX-GREEDY implemented with the "first-min" choice of Fig. 7 is $O(|V|^2)$.

The "first-min" option is probably the most natural one, but it suffers from the following drawback: During the first iterations in the main loop, a very large number of candidates can be present in $bucket[1][min[1]]$. In fact, for low-density graphs, at the first addition, the value $min[1]$ will be zero and all vertices not connected to $set1$ will be in the corresponding bucket. Because $set1$ contains only a single vertex v_1 at the first addition, the number of vertices to be examined is larger than or equal to $|tobeadded| - deg(v_1)$, $deg(v_1)$ being the vertex degree in the given graph and $|tobeadded|$ being equal to $n-2$. In the following steps, the size of the winning bucket tends to gradually decrease, but, in any case, after n_a additions to $set1$ such that at least one vertex in $tobeadded$ is still not connected to $set1$, the winning value remains zero and the size of $bucket[1][0]$ is still larger than or equal to $|tobeadded| - d_{max}n_a$. Furthermore, the experimental evidence of Fig. 2 witnesses that the average size of the winning bucket is a large fraction of n when the winning value is zero. In fact, it ranges between about 430, for d = 2.5, and 280, for d = 20, for the random graphs of dimension 1,000.

3.5.2 First-Max Option for BEST-VERTEX

Clearly, the above analysis implies that the number of vertices in $tobeadded$ that are connected to a given set by one or more edges is very small, at the beginning, for low-density graphs. Let us keep the assumption that the addition is done to $set0$. The second option (called "**first-max**", see Fig. 8) consists of examining the vertices by considering first the buckets corresponding to the maximum values of connections to $addset$, i.e., to $set0$. They are the upper left buckets in Fig. 6. The examination stops and determines the winning candidate as soon as one vertex v with minimal number of connections to $set1$ is encountered, i.e., one with $E(v, set1) = min[1]$. Clearly, there is no guarantee that this vertex will be found in the bucket corresponding to the $max[0]$ value so that buckets corresponding to lower values may have to be considered. In the example of Fig. 6, the winning vertex would be vertex number 3. The worst-case complexity of BEST-VERTEX with the "first max" implementation is not reduced: Again, all $O(|V|)$ nodes may need to be considered at each call (and moving to different buckets during the examination requires at most $O(|V|)$ checks to see whether a bucket is empty or contains some nodes) for a total complexity of $O(|V|^2)$ during the entire MIN-MAX-GREEDY algorithm.

Nonetheless, for random low-density graphs and in the initial phase of the greedy construction, there is a large probability that, among the vertices connected to $set0$ with the largest number of edges, there will be vertices not connected to $set1$. In fact, if more edges connect to $set0$, fewer edges will connect to $set1$, on average. The relation is clear in constant-degree graphs, but the timing results in the experiments testify that the strategy is very effective also for low-density random and geometric graphs.

Let us note that the "first-max", "first-min", and "naive" implementations produce solution with statistically equivalent quality so that the dominating option is the one with lower CPU times. The experiments show that CPU time required by the "first-max" implementation of the MIN-MAX-GREEDY algorithm is not much larger than that for a random assignment. The factor is between about 2 and 11 and it does not increase rapidly when the size of the random and geometric (G and U) graphs is doubled. In fact, the factor remains approximately constant for the low-density graphs ($d = 2.5, 5$), a result consistent with an approximate average complexity of $O(|E|)$ for low-density graphs, although the worst-case complexity is $O(|V|^2)$. A notable speedup is obtained by passing from the naive implementations of the MIN-MAX-GREEDY algorithm, to the "first-max" implementation. As expected, the speedup is larger for the lower density graphs. For example, the random graphs show a speedup between 3.8 and 4.8 for $n = 500$, 1,000, and $d = 2.5$, while the implementations require similar or slightly less CPU times for an average degree larger than 10. As a function of the problem dimension, the speedup tends to increase, for example, it approximately doubles when the size of the geometric U graphs doubles, it reaches values close to two orders of

```
BEST-VERTEX- first max implementation
1   otherset ← (1 − addset)
2   minedges ← min[otherset]
3   maxedges ← max[addset]
4   for value ← maxedges downto 0 do
5       for offset ← 0 to size[addset][value] − 1 do
6           vertex ← bucket[addset][value][offset]
7           if E(vertex, otherset) = minedges then return vertex
```

Fig. 8. MIN-MAX-GREEDY algorithm: determination of the winning candidate in the "first-max" option.

magnitude for the larger regular, caterpillar, and grid graphs (speedup is **25.6** for Breg5000.16, **74.2** for Cat.5252, **49.7** for RCat.5114, **56.5** for Grid5000.50, **59.8** for W-grid5000.100).

More detailed results are reported in the Appendix.

4 PROHIBITION-BASED (TABU) SEARCH

Given a *cost function* f to minimize defined on a set of *solutions*, the classic *local search* method attempts to improve on a given solution by a series of incremental, local changes and stops if the current solution is a *local minimizer*, i.e., the function value is a *local minimum* and it cannot be improved by any local change. In particular, we consider a version, denoted as LOCAL-SEARCH, where *all* local changes are tried and the *best* one (the one causing the greatest improvement) is applied if and only if the function improves.

The Tabu Search meta-heuristic [27] is based on the use of *prohibition* techniques and "intelligent" schemes as a complement to basic *local search* heuristics, with the purpose of guiding the basic heuristic beyond local optimality. Ideas similar to those proposed in TS can be found in the *denial* strategy of [56] (once common features are detected in many suboptimal solutions, they are *forbidden*) or in the opposite *reduction* strategy of [40] (in an application to the Traveling Salesman Problem, all edges that are common to a set of local optima are fixed).

In the context of graph partitioning a related heuristic is the mentioned KL algorithm [35]. The KL algorithm can be denoted as a *variable depth search* and it is briefly summarized as follows: KL improves a local search scheme where the moves from the current solution are *2-exchanges* (two nodes a and b in the two different sets are exchanged) and the selection criterion is the *gain* $g(a, b)$, defined as the decrease in the *cut size* that is obtained by exchanging them.

- At each KL cycle one starts from a legal configuration and applies a *chain of* $n/2$ *tentative* 2-exchanges while monitoring the evolution of the tentative solution and the corresponding cut size.
- As soon as two nodes are tentatively exchanged, their membership is "frozen" for the rest of the cycle: They cannot be moved again.
- At the end of the cycle one derives the *minimum* cut size along the chain: If it improves the starting cut size at the beginning of the cycle, one sets the current solution to the configuration reaching the minimum and starts a new cycle, otherwise the method stops.

Citing from [1], "*the basic idea is to allow unfavorable 2-exchanges in the sequence to eventually obtain a favorable k-exchange without exhaustive search of the k-exchange neighborhood.*"

In more recent times, the full blossoming of "intelligent prohibition-based heuristics" starting from the late eighties is due to Glover [27], but see also [29] for an independent seminal paper. The unifying characteristic of the many realizations of TS is the modification of *local neighborhood search* through the introduction of *prohibitions* (henceforth, the term "tabu"). Some selected moves among those that could modify the current configuration are temporarily prohibited. TS acts to continue the search beyond the first local minimizer, therefore also accepting worsening moves and enforcing appropriate amounts of diversification through prohibitions to avoid that the search trajectory remains confined near a given local minimizer.

A competitive advantage of TS with respect to alternative heuristics based on local search, like Simulated Annealing [37], lies in the use of the past history of the search to influence its future steps. On the contrary, simple versions of SA generate a Markov chain: The successor of the current point is chosen stochastically, with a probability that depends only on the current point and not on the previous history. The Markovian property permits deriving asymptotic convergence results, but these results are, unfortunately, irrelevant for the application of SA to optimization. In fact, repeated local search [22] and even random search [15] have better asymptotic results. According to [1] "*approximating the asymptotic behavior of SA arbitrarily closely requires a number of transitions that for most problems is typically larger than the size of the solution space ... Thus, the SA algorithm is clearly unsuited for solving combinatorial optimization problems to optimality.*" Of course, SA can be used in practice with fast cooling schedules, but then the asymptotic results are not directly applicable.

4.1 Discrete Dynamical System

Let us define the notation. In our application, the search space $\mathcal{X}$ is given by the set of binary strings with a given length n: $\mathcal{X} = \{0, 1\}^n$, $X^{(t)} \in \mathcal{X}$ is the current solution along the trajectory at iteration ("time") t. The set $N(X^{(t)})$ is the neighborhood of point $X^{(t)}$, obtained by applying a set of basic moves $\mu_1, \mu_2, ..., \mu_n$ to the current configuration:

$$N(X^{(t)}) = \{X \in \mathcal{X} \text{ such that } X = \mu_i \circ X^{(t)}, i = 1, \ldots, n\}$$

The moves change (complement) the individual bits and are therefore idempotent (the inverse of μ is equal to μ). Some of the neighbors are *prohibited*, a subset $N_A(X^{(t)}) \subset N(X^{(t)})$ contains the *allowed* ones. The general way of generating the search trajectory that we consider is given by:

$$N_A(X^{(t)}) = \text{ALLOW}(N(X^{(t)}), X^{(0)}, \ldots, X^{(t)}) \quad (2)$$

$$X^{(t+1)} = \text{BESTNEIGHBOR}(N_A(X^{(t)})\ ). \quad (3)$$

The set-valued function ALLOW selects a nonempty subset of $N(X^{(t+1)})$ in a manner that depends on the entire search trajectory $X^{(0)}, \ldots, X^{(t)}$.

An essential abstract concept in TS is given by the *discrete dynamical system* of (2)-(3), obtained by modifying LOCAL-SEARCH, see [8] for a taxonomy of Tabu Search from the abstract point of view of the dynamical systems.

In particular, a simple algorithm is obtained by introducing a *prohibition*[3] T that determines how long a move will remain prohibited after its execution. The FIXED-TS algorithm is obtained by fixing T throughout the search [27]. A neighbor is *allowed* if and only if it is obtained from the current point by applying a move that has not been used during the last T iterations. In detail, if LastUsed(μ) is the last usage time of move μ (LastUsed(μ) $= -\infty$ at the beginning):

$$N_A(X^{(t)}) = \{X = \mu \circ X^{(t)} \text{ such that } \text{LASTUSED}(\mu^{-1}) < (t - T)\}. \quad (4)$$

4.2 Relationship Between Prohibition and Diversification

For the following discussion, let us define as $H(X, Y)$ the Hamming distance between two strings X and Y, defined as the number of corresponding bits that are different in the two strings. The prohibition T used in (4) is related to the amount of *diversification*: The larger T, the larger is the distance H that the search trajectory must travel before it is allowed to come back to a previously visited point. But, T cannot be too large, otherwise no move will be allowed after an initial phase. In detail, an upper bound $T \leq (n-2)$ guarantees that at least two moves are allowed at each iteration so that the search does not get stuck and the move choice *is* influenced by the cost function value (it is not if only one move is allowed!). Now, if only allowed moves are executed and T satisfies $T \leq (n-2)$, one obtains the following:

Fundamental relationship between prohibition and diversification

- The Hamming distance H between a starting point and successive points along the trajectory is strictly increasing for $T+1$ steps.

$$H(X^{(t+\tau)}, X^{(t)}) = \tau \quad \text{for} \quad \tau \leq T + 1$$

- The minimum repetition interval R along the trajectory is $2(T+1)$.

$$X^{(t+R)} = X^{(t)} \ \Rightarrow \ R \geq 2(T+1)$$

The demonstration is immediate as soon as one notices that, after a bit is changed, it is "frozen" for the next T iterations. To visualize this behavior, Fig. 9 shows the evolution of the configuration $X^{(t)}$, when the function to be optimized is given by $f(X) \equiv number(X)$, where $number(X)$ is the integer number obtained by considering X as the standard binary encoding:

$$number(X) = X_1 \cdot 2^0 + X_2 \cdot 2^1 + X_n \cdot 2^{n-1}.$$

The prohibition T is equal to three.

In Fig. 9, the prohibition T is equal to 3 and the configuration starts with the all-zero string, a locally optimal point. At iteration 0, the best move changes the least significant bit. At iteration 1, the least significant bit is prohibited (the period in which a bit is "frozen" and cannot be changed is shown with a shaded box in the figure) and the best *allowed* move changes the second bit so that the configuration reaches a Hamming distance of 2 from the starting configuration. The maximum Hamming distance is reached at iteration $(T+1)$, then the distance decreases and the initial configuration is repeated at iteration $2(T+1)$. Let us note that, when a cycle like the one above is generated, the set of configurations visited during the initial part, up to $H = T + 1$, is different from the set visited when H decreases back toward zero. In other words, one does not waste CPU time to revisit previously visited configurations, apart from the initial one. For the given f the locally optimal point is also the global minimum of the function. In general, this is not the case, and better values can be obtained by visiting other locally optimal points. Of course, as soon as a local minimizer is found, all points in its *attraction basin* (i.e., all points that are mapped to the given minimizer by the local search dynamics) are not of interest. In fact, by definition, their f value is equal to or larger than the value at the local minimizer. The value of T should be chosen so that a new attraction basin leading to a new, and possibly better, local minimizer can be reached after reaching Hamming distance $T+1$. Because the minimal Hamming distance required (a sort of *attraction radius* for the given attraction basin) is not known, one will consider the option of determining T in a randomized way (see Section 6) or in a simple *reactive* way, see Section 7.

4.3 Relationship Between Prohibitions and Kernighan-Lin

The relationship between FIXED-TS and KL is clear: One obtains a single cycle of KL from FIXED-TS by using 2-exchanges as basic moves and by setting $T \geq n/2$ and LASTUSED(μ) $= -\infty$ at the beginning of each cycle. A 2-exchange is allowed iff both involved vertices are allowed to be moved according to the previous criterion. In this way, after two vertices a and b are exchanged in a cycle, the prohibitions imply that they will not be considered anymore for further exchanges in the same cycle. The crucial differences between FIXED-TS and KL are that:

3. The term *prohibition* is chosen instead of the more traditional *list size* because *list size* refers to a specific implementation: prohibitions can be obtained without using any list.

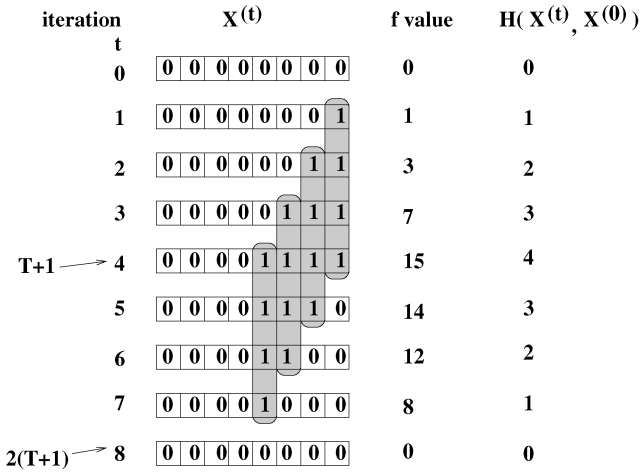

Fig. 9. A simple example of the relationship between prohibition T, and diversification $H(X^{(t)}, X^{(0)})$. T = 3 in the example. See text for details.

- In FIXED-TS, the concept of "cycle" disappears: the solution is updated after each basic move is applied and not at the end of a chain of $n/2$ moves,
- FIXED-TS *does not terminate*, even if the best configuration visited in a sequence of $n/2$ moves is worse than the initial one.

Of course, a prohibition has to be selected in FIXED-TS so that a suitable number of allowed moves is available at each iteration; see the analysis in Section 5.1. The "best-so-far" solution is saved during the run and reported when the run is terminated, e.g., when the allotted number of iterations elapses.

5 FIXED-TS FOR GRAPH PARTITIONING

In the literature, a first TS approach for the graph partitioning problem is presented in [50]. The method starts from a randomly generated feasible solution, uses the neighborhood obtained by moving a single node from one set to the other, and allows a certain degree of "oscillating" imbalance between the size of the two sets. Results in [16] demonstrate how a simpler TS realization obtains better results. In the same paper, an enhanced version of TS for the problem (*EnTaS*) is presented that starts from a greedy solution (without tie-breaking) and uses a dynamic prohibition, two "move aspiration" criteria (the prohibition of a move is relaxed if the move is particularly attractive), additional diversification by previously evaluated solutions, and periodic restarting. The many parameters present in the algorithm are determined through preliminary experiments and the obtained results are better than those of [50] and [38], except for very sparse geometric graphs.

Because of the limited scope and space constraints of this paper, our discussion in the next section will be limited to a simple realization of Tabu Search, similar to the "basic" version considered in [16], but differing because our basic moves act on a single vertex and not by exchanging two vertices, because our MIN-MAX-GREEDY algorithm is used to initialize the solution and because the aspiration criteria and the additional diversification methods are absent. The emphasis of our study is on assessing the effect of different algorithmic "building blocks" on the performance, where the different elements are added in successive steps and the experimental results indicate the incremental performance improvements. Furthermore, the emphasis is on minimizing the number of parameters that are to be tuned by the users: The detailed tuning is substituted by the heavy use of *randomization* and by a simple *reactive* strategy.

In detail, the effect of different fixed prohibition values on the performance is studied in Section 5.1, while the utility of a simple randomization strategy is assessed in Section 6. Finally, the additional benefit of a *reactive* loop to bias the random choice of the prohibition is investigated in Section 7.

The structure of the FIXED-TS algorithm for graph partitioning is described in Fig. 10. To normalize the prohibition with respect to the number of bits n, it is useful to introduce a "fractional prohibition" T_f such that $T = \lfloor T_f\, n \rfloor$. The iteration counter t in all algorithms is a global variable which can be used by all subroutines. Similarly, f_{min} is a global variable whose initial value is equal to the minimum cut encountered during the greedy initialization, and LAST-USED is a global array that stores at LASTUSED$[v]$ the last time at which vertex v has been moved from a set to the other one. In addition, let us define as $N_0 \equiv |\{i : X(i) = 0\}|$ the cardinality of $set0$ and as N_1 the cardinality of $set1$.

The parameters of the routine are the fractional prohibition T_f, used to determine the prohibition in line 1, and the number of iterations. After starting from a valid initial assignment, for example obtained by the MIN-MAX-GREEDY routine, additions alternate between $set0$ and $set1$ (line 3). At each iteration, the best allowed vertex to be moved from $otherset$ to $addset$ is determined (line 5), the move is applied (lines 6-7), the time at which vertex v has been moved is updated (line 8), and the current iteration incremented (line 9).

If the assignment is legal and the cut size is better than the best so far (f_{min}), it is recorded. An assignment is legal if and only if $|N_0 - N_1| \leq 1$.

The purpose of the function BEST-MOVE(set) is to return a vertex in the given set that is *allowed* and that causes the lowest possible value of f after it is moved to the other set. Of course, a vertex is returned even if the lowest value of f that can be obtained by moving an allowed vertex from set is higher than the current f value. The function BEST-MOVE is realized by using a "buckets" data structure similar to the one described in Fig. 6. At a given iteration, each vertex is characterized by a *gain* value, the decrease in the cut size f that would be obtained by moving the vertex to the other set. Two arrays of buckets are present for the two sets and, contrary to the application in Section 3.5, each vertex is now listed in only one bucket, corresponding to its current set and $gain$ value. Because the $gain$ values are changed only for the just moved vertex v and its connected vertices, the $gain$ vector is updated after each iteration with $O(deg(v))$ operations. In addition, maintaining the maximum and minimum values over the two sets $gmax[set]$ and $gmin[set]$ requires at most $O(d_{max})$ operations, d_{max} being the maximum vertex degree.

```
FIXED-TS(T_f, iterations)
1     T ← ⌊T_f n⌋
2     for it ← 1 to iterations do
3         if N_0 ≥ n/2 then addset ← 1   else addset ← 0
4         otherset ← (1 − addset)
5         v ← BEST-MOVE(otherset)
6         addset ← addset ∪ {v}
7         otherset ← otherset \ {v}
8         LASTUSED[v] ← t
9         t ← t + 1
10        if assignment is legal and f < f_min then   f_min ← f

BEST-MOVE(set)
11    ♢ Searches in set for vertex to move
12    for g ← gmax[set] downto gmin[set] do
13        forall vertex ∈ bucket[set][g] do
14            if ALLOWED(vertex)   return vertex
```

Fig. 10. FIXED-TS and BEST-MOVE routines. Cut sizes are recorded only for legal assignments. See text for details.

The BEST-MOVE function examines the buckets for the given set by starting from the one corresponding to the largest gain (line 12). As soon as one *allowed* vertex is found, the function returns it (line 14). With the used notation, vertex v is allowed if and only the last usage time satisfies: $\text{LASTUSED}[v] < (t - T)$, see also (4). Because the moves are idempotent, μ^{-1} is equal to μ and the move can be identified by the vertex number.

5.1 Effect of Fixed Prohibition on the Performance

The first investigation aims at determining the effect of the *fixed* prohibition value on the performance. Fig. 11 collects the results of tests on a representative subset of random and geometric graphs. For each graph, the average cut size f_{min} obtained by initializing the assignment through MIN-MAX-GREEDY and running $100n$ iterations of FIXED-TS is shown as a function of the fractional prohibition T_f. Only values less than $1/4$ are considered. Because of the problem structure, an assignment and its complement (0 being substituted by 1 and vice versa) denote the same solution, with the two sets of the partitions interchanged. Therefore, a value of $T_f > 1/2$ would imply that, after starting from an assignment, one would reach after $(T + 1)$ iterations an assignment closer to the complement than to the original one. The chosen threshold of $1/4$ completely eliminates these oscillations between an assignment and its complement. One hundred runs are executed for each data point.

The conclusion derived from these preliminary tests is that the prohibition T does indeed have a crucial effect on the performance. For example, let us consider the $G500.2.5$ graph (the leftmost one at the top of Fig. 11). The average cut size starts at 72 for $T_f = 0.01$, rapidly decreases to 54.7 for $T_f = 0.18$, and then gradually increases. The average cut size is greatly decreased with an appropriate prohibition, especially considering that the heuristically best cut value for the graph is 49. Qualitatively similar performance improvements have been observed for all other graphs.

Determining the appropriate prohibition for a given graph is not a trivial task. Fig. 12, derived from Fig. 11, shows the optimal T_f as a function of the average vertex degree for the four sets of graphs (G500, G1000, U500, U1000). It can be observed that denser random graphs tend to prefer smaller prohibitions, while denser geometric graphs tend to prefer larger prohibitions.

Finally, Table 4 compares the average cut sizes obtained by running FIXED-$TS(T_f, 100n)$ with the optimal T_f with the results of [14]. The reported CPU times are average values for a single run. The CPU times of BFS-GBA are for a Sun SPARC IPX, those for FIXED-TS are for a Digital AlphaServer 2100. To allow an easier comparison, the original CPU time of BFS-GBA has been normalized to discount the different speed of the two machines for integer operations, see column CPU_n, where the times have been divided by 12.7.

While the results of Table 4 are roughly comparable (FIXED-TS is better for the denser graphs, BFS-GBA wins for the sparsest ones, but for larger CPU times), let us note that the comparison assumes the a priori knowledge of the proper prohibition: Very poor values can be obtained with the "wrong" T_f.

The following study aims at obtaining a more robust algorithm that obtains competitive results without assuming the knowledge of the proper T_f value.

6 Randomized Prohibition

A first sensible hypothesis to consider in order to avoid a preliminary selection of the prohibition value is to use a

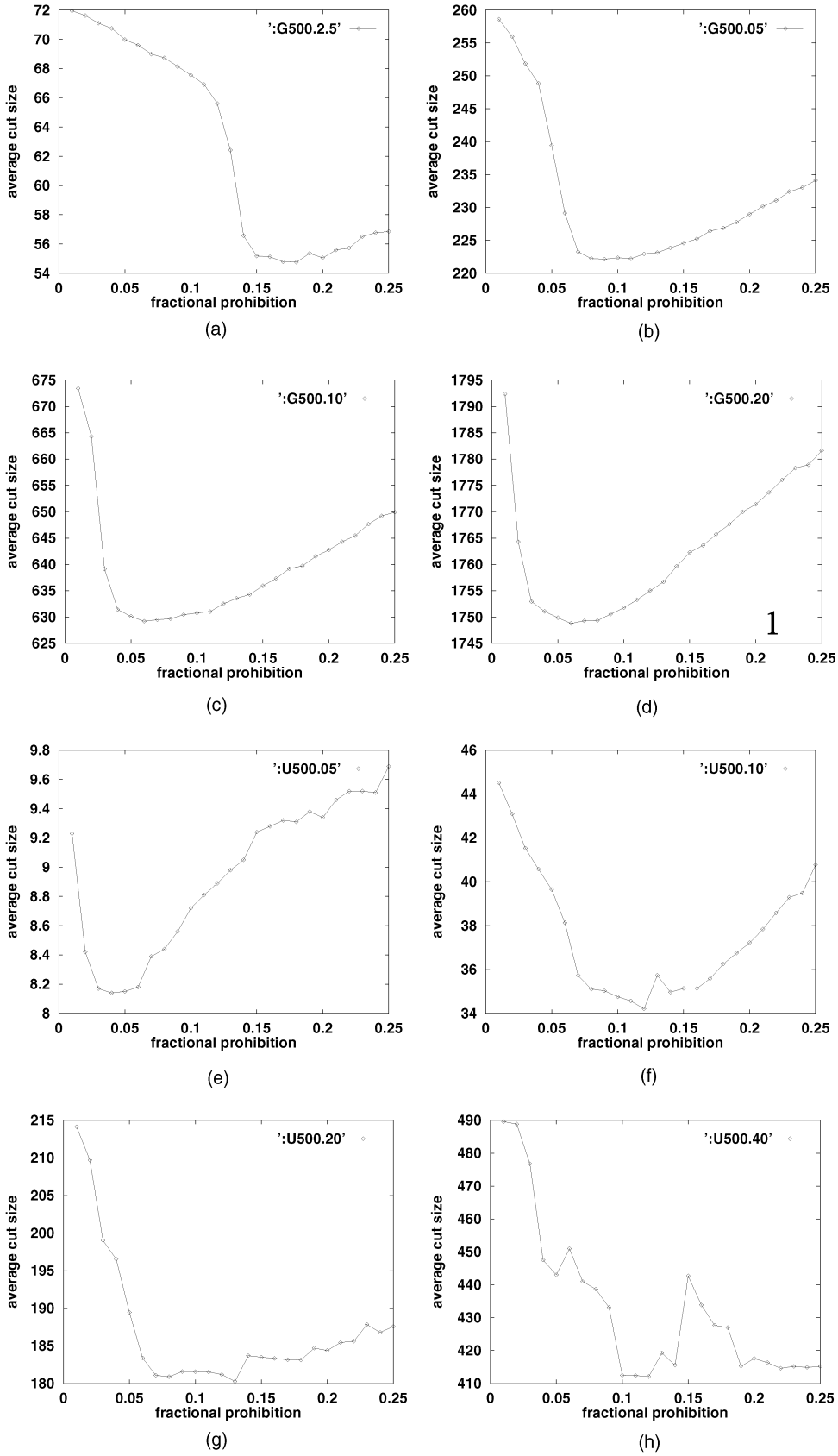

Fig. 11. FIXED-TS(100 n steps): average f_{min} (cut size) as a function of the fractional prohibition T_f. The graphs are (a)-(d) G500 and (e)-(h) U500.

random one, for example, such that T_f is chosen with equal probability in the range $[0, 1/4]$. Unfortunately, the low average cut values obtained (see the column labeled "Ave(all runs)" in Table 4) discourage this simple hypothesis.

TABLE 4
FIXED-TS with Optimal T_f

Graph	Cut sizes by BFS-GBA			Cut sizes by FIXED-TS(100 n iter)				
	Ave	CPU	CPU_n	Ave (best T_f)	(St.Dev.)	Ave (all runs)	(St.Dev.)	CPU
G500.2.5	53.97	5.96	*0.46*	54.76	(1.8)	62.46	(7.5)	0.17
G500.05	222.13	8.09	*0.63*	222.14	(2.6)	231.46	(11.8)	0.20
G500.10	631.46	11.71	*0.92*	**629.20**	(2.3)	639.69	(11.7)	0.27
G500.20	1752.51	21.60	*1.70*	**1748.79**	(3.3)	1762.96	(13.1)	0.58
G1000.2.5	103.61	16.83	*1.32*	104.81	(3.2)	118.02	(14.3)	0.49
G1000.05	458.55	26.65	*2.09*	**455.62**	(3.3)	474.68	(20.5)	0.47
G1000.10	1376.37	37.05	*2.91*	**1372.12**	(5.0)	1396.84	(24.5)	0.67
G1000.20	3401.74	62.25	*4.90*	**3396.69**	(6.8)	3425.03	(29.2)	1.10
U500.05	3.65	7.54	*0.59*	8.14	(5.4)	8.94	(5.5)	0.11
U500.10	32.68	9.59	*0.75*	34.21	(11.1)	37.58	(12.1)	0.26
U500.20	179.58	11.50	*0.90*	180.28	(2.4)	187.18	(14.8)	0.47
U500.40	412.23	9.92	*0.78*	**412.10**	(0.7)	432.87	(40.8)	0.89
U1000.05	1.78	17.58	*1.38*	8.70	(6.1)	10.21	(6.6)	0.25
U1000.10	55.78	30.89	*2.43*	61.45	(12.9)	67.93	(14.2)	0.59
U1000.20	231.62	32.97	*2.59*	**222.31**	(2.6)	245.42	(26.7)	1.01
U1000.40	738.10	36.99	*2.91*	**737.7**	(3.3)	765.10	(63.7)	2.02

Comparison with BFS-GBA. For BFS-GBA, CPU_n is the original CPU time normalized to discount the different speed of the two machines.

In order to avoid having a poor T_f value chosen and used for the entire run, an immediate strategy is to continue picking random T_f values, each one being used for a short phase. The option is called RANDOMIZED-TS, see Fig. 13, and it is characterized by two parameters: the total number of iterations (*iterations*) and the duration of a single phase (*individual*) after starting from a new initial assignment by MIN-MAX-GREEDY. By setting *individual* equal to *iterations* one eliminates the multiple restarts. Every n iterations, a new T_f value is picked (lines 6-7) so that its value is distributed randomly among the same values considered in the previous experiments with FIXED-TS (i.e., $\{0.01, 0.02, \ldots, 0.25\}$).

Table 5 reports the average cut size obtained by different randomized options. As usual, 100 runs are executed for each graph and algorithm. The "Ave(rand-1)" column lists the results of RANDOMIZED-TS($100n, 100n$): Only one greedy construction is executed. The "Ave(rand-10)" column lists the results of RANDOMIZED-TS($100n, 10n$): An individual phase lasts $10n$ iterations so that 10 greedy restarts are executed. CPU times are not reported because they are very similar to the times listed in Table 4. A value is in boldface if it is lower than the value obtained by FIXED-TS with the best possible T_f.

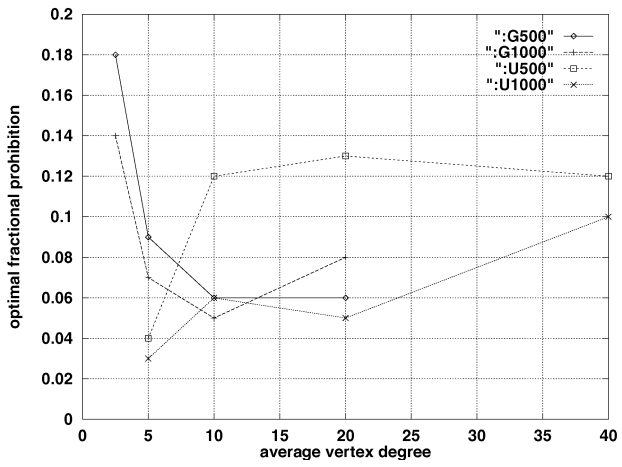

Fig. 12. FIXED-TS(100 n steps): Best fractional prohibition as a function of the average degree for the random geometrix graphs.

A first conclusion is that the two randomized algorithms reach results that are *comparable to the best results* of the FIXED-TS algorithm, reported again in the first column, and certainly much better than the average results with a random and fixed T_f, see the "Ave(all runs)" column in Table 4. Randomization therefore appears as a viable option if the optimal T_f is not known.

What is surprising is that, for some graphs with the larger densities (e.g., G1000.20, U500.20, U500.40, U1000.40), the randomized options achieves results that are *better* than the best FIXED-TS results. In these cases, a *randomized* T_f value is better than an optimal *fixed* T_f value. The power of randomized TS strategies is also testified to by results in different contexts, for example in the *robust-TS* algorithm of [58] for the Quadratic Assignment Problem.

A relative comparison of the "one start" versus "ten starts" choice indicates that long runs from a single starting point are preferable for the random graphs, while shorter runs with restarts tend to be preferable for the geometric graphs. In these graphs, an unlucky starting assignment may compromise an entire run and having multiple restarts increases the algorithm robustness.

In the following section, we will show how the combination of randomization and reaction achieves results that are substantially better for the geometric graphs and for the denser random graphs. This final combination achieves better results than BFS-GBA, apart from the G500.2.5 graph. The columns listing the results for the same number of iterations are labeled "RRTS, 100 n iter." in Table 6.

```
RANDOMIZED-TS(iterations, individual)
1    t ← 0
2    while t < iterations do
3        MIN-MAX-GREEDY
4        t_end ← t + individual
5        while t < t_end do
6            tmp ← RANDOM (1, 25)
7            T_f ← tmp 0.01
8            FIXED-TS(T_f, n)
```

Fig. 13. The RANDOMIZED-TS algorithm. RANDOM (1,25) returns a random integer value in the given range. t is a global variable visible by all subroutines.

7 RRTS: Randomized and Reactive Prohibition

The purpose of this final section is to investigate how a simple *reactive* scheme can ameliorate the results of the previously described randomized algorithm. Furthermore, while the previous investigation considered very short runs whose CPU times on current machines is always less than about a couple of seconds, the allotted number of iterations of the final runs will be multiplied by 10 in order to investigate the trade-off between number of iterations and solution quality.

Reactive Search heuristics are based on simple self-tuning schemes, acting while the algorithm runs, so that the method becomes *history-sensitive* [8]. In particular, *instance* properties and *local* properties of the configuration space can be used for the appropriate tuning of the algorithm parameters, like the prohibition T, without the explicit intervention of the user. This "on-line" automated tuning is to be contrasted with the standard "off-line" tuning, where the parameters are optimized, depending only on the statistical characteristics of a *problem* and not of a specific *instance* or of a certain localized region of the configuration space. Reactive Search schemes have already been used with success in different contexts [6], [7].

In our algorithm, the bias acts so that a *successful* value of T is chosen with a higher probability and kept for a longer phase. The *successfulness* of a given T_f is determined in two ways: during a preliminary "scoring" phase, by measuring the "speed of improvement" in short runs with the given prohibition and, during the later phases, by observing whether the current value led to a new best assignment in the just elapsed part of the search.

The complete REACTIVE-RANDOMIZED-TS (RRTS) algorithm is illustrated in Fig. 14. A preliminary SCORING routine evaluates the possible T_f values through a number of $trials$ consisting of short runs using FIXED-TS and LOCAL-SEARCH (lines 3-13). At the end of the "scoring", the smallest value of the fractional prohibition that achieves the maximum vote is returned ($T_{f_{best}}$ determined at line 17). In addition, $vote(T_f)$ will contain a score for each T_f value to be used in the later phase of the algorithm to *bias* the probability that a given T_f is chosen and the set $elite$ will contain the best assignments found during the individual tests.

Let us now describe how a single T_f is evaluated: After initializing $vote(T_f)$ in line 3, a given number of $trials$ is executed. In each trial, the assignment is initialized by calling MIN-MAX-GREEDY followed by LOCAL-SEARCH, see lines 5-6. The starting f value and iteration are recorded (f_{start} and t_{start} in line 7). Then, a series composed of short runs of FIXED-TS followed by LOCAL-SEARCH is executed until a total of $n/2$ iterations are accumulated (lines 8-11). FIXED-TS reaches a Hamming distance of $T+1$ in the first $T+1$ iterations, then the same number of iterations is executed to allow for a possible reduction of the Hamming

TABLE 5
FIXED-TS with Randomized T_f

Graph	FIXED-TS with best T_f		Cut sizes by variations of TS (100 n iterations in total)			
	Ave	(St.Dev.)	**Ave(rand-1)**	(St.Dev.)	**Ave(rand-10)**	(St.Dev.)
G500.2.5	54.76	(1.8)	55.18	(1.7)	56.57	(1.4)
G500.05	222.14	(2.6)	**221.39**	(1.9)	223.28	(1.8)
G500.10	629.20	(2.3)	630.28	(2.1)	630.71	(1.9)
G500.20	1748.79	(3.3)	1749.49	(2.7)	1750.32	(2.9)
G1000.2.5	104.81	(3.2)	**103.93**	(2.9)	108.70	(2.5)
G1000.05	455.62	(3.3)	457.43	(3.0)	460.65	(2.9)
G1000.10	1372.12	(5.0)	1373.87	(4.1)	1375.85	(4.1)
G1000.20	3396.69	(6.8)	**3390.19**	(3.1)	**3393.50**	(4.3)
U500.05	8.14	(5.4)	**7.67**	(5.1)	**2.85**	(0.8)
U500.10	34.21	(11.1)	**30.81**	(7.8)	**26.48**	(0.9)
U500.20	180.28	(2.4)	**178.61**	(1.3)	**178.68**	(1.3)
U500.40	412.10	(0.7)	**412**	(0)	**412**	(0)
U1000.05	8.70	(6.1)	8.74	(4.9)	**2.41**	(1.9)
U1000.10	61.45	(12.9)	**56.68**	(11.8)	**46.31**	(5.3)
U1000.20	222.31	(2.6)	227.41	(5.6)	226.64	(5.0)
U1000.40	737.7	(3.3)	**737**	(0)	**737**	(0)

TABLE 6
REACTIVE-RANDOMIZED-TS

Graph	BFS-GBA			RRTS, 100 n iter.			Best	RRTS, 1000 n iter.			
	Ave	CPU	CPU_n	Ave	(St.Dev.)	CPU		Min	Ave	(St.Dev.)	CPU
G500.2.5	53.97	5.96	*0.46*	55.90	(1.2)	0.17	49	51	**52.06**	(0.50)	2.0
G500.05	222.13	8.09	*0.63*	**221.24**	(1.9)	0.20	218	218	**218.29**	(0.46)	2.5
G500.10	631.46	11.71	*0.92*	**629.33**	(2.0)	0.27	626	626	**626.44**	(0.59)	3.6
G500.20	1752.51	21.60	*1.70*	**1747.94**	(2.6)	0.58	1744	1744	**1744.36**	(0.66)	6.8
G1000.2.5	103.61	16.83	*1.32*	107.18	(2.3)	0.49	95	95	**98.69**	(1.01)	6.5
G1000.05	458.55	26.65	*2.09*	**458.32**	(2.4)	0.47	445	445	**450.99**	(1.43)	6.5
G1000.10	1376.37	37.05	*2.91*	**1371.47**	(3.5)	0.67	1362	1362	**1364.27**	(1.38)	9.3
G1000.20	3401.74	62.25	*4.90*	**3389.64**	(4.1)	1.10	3382	3382	**3383.92**	(1.00)	14.7
U500.05	3.65	7.54	*0.59*	**2.02**	(0.1)	0.11	2	2	**2**	(0)	1.7
U500.10	32.68	9.59	*0.75*	**26**	(0)	0.26	26	26	**26**	(0)	2.7
U500.20	179.58	11.50	*0.90*	**178**	(0)	0.47	178	178	**178**	(0)	5.3
U500.40	412.23	9.92	*0.78*	**412**	(0)	0.89	412	412	**412**	(0)	10.2
U1000.05	1.78	17.58	*1.38*	**1**	(0)	0.25	1	1	**1**	(0)	4.2
U1000.10	55.78	30.89	*2.43*	**39.76**	(0.8)	0.59	39	39	**39.03**	(0.19)	6.3
U1000.20	231.62	32.97	*2.59*	**222.27**	(1.4)	1.01	222	222	**222**	(0)	12.5
U1000.40	738.10	36.99	*2.91*	**737**	(0)	2.02	737	737	**737**	(0)	24.3

Averages and minimum values of 1,000 runs (1,000 n iterations each). For BFS-GBA, CPU_n is the original CPU time normalized to discount the different speed of the two machines.

distance. Finally, the prohibitions are switched off to see whether, after the TS diversification phase, better solutions can be found by considering *all* possible moves. Better average values were obtained through this combination than through FIXED-TS alone. As usual, the global variable f_{min} records the best value found during all steps executed, while the global variable t_{min} records the iteration at which the best value has been found. The best assignment found in the last try is added to set $elite$ (line 12) and $vote$ is updated so that its final value will be proportional to the average "derivative" $(f_{start} - f_{min})/(t - t_{start})$. Because we are minimizing, the last quantity will be nonnegative. At the end of the scoring phase, the $vote$ is normalized so that the minimum vote is 0.1 and the maximum 1. Because $vote(T_f)$ will be proportional to the probability that a given T_f is picked, a nonzero minimum value guarantees that all prohibition values will have a nonzero probability of being selected.

The complete RRTS algorithm, see the bottom part of Fig. 14, calls the just described SCORING routine (line 1, a number of 3 $trials$ is sufficient for a robust scoring: larger numbers do not increase the performance in a significant way). Then, a number of $\lceil iterations/individual \rceil$ individual runs are executed after starting from the best assignment left in the $elite$ set or from the assignment obtained by MIN-MAX-GREEDY if all assignments in $elite$ have already been used (lines 3-4). The prohibition is initialized with the best one (line 5), and the starting iteration of the individual run is registered (line 6). Each individual run is composed of a sequence of short runs of the FIXED-TS and LOCAL-SEARCH combination (lines 10-11), each short run ending after n iterations (line 12). If a new best solution has been found in the last short run (and, therefore, t_{min} has been changed since the beginning of the last run), the same *successful* T_f value is kept, otherwise a new random T_f is picked with a probability proportional to the "vote" obtained in the scoring phase (lines 13-14). Each individual run ends when at least $individual$ iterations have been executed (line 15).

Table 6 summarizes the comparison between RRTS and BFS-GBA. The first three columns report the average cut size and CPU times of [14]. The next three columns report the results of RRTS for a total of $100n$ iterations with individual runs of $10n$ iterations (average with standard deviation and CPU time). Then, the optimal heuristic value for the graphs is reported from [14] (Best). Finally, the results for RRTS for a total of $1,000n$ iterations with individual runs of $100n$ iterations are listed (minimum, average with standard deviation, and CPU time). To ease the comparison, the average cut sizes are in boldface when lower than the values of [14].

When $100n$ iterations are allowed, the CPU times of RRTS are significantly lower than the CPU times allowed for BFS-GBA, even after taking into account the different speed of the two target machines. In fact, after scaling to take the different machine speed into account, see the column marked "CPU_n," RRTS time is less by a factor of 3-4 for most graphs. Nonetheless, RRTS obtains a significantly better performance on the geometric U graphs (the average values are already very close to the Best values) and a better performance on all random graphs apart from the lowest-density one.

When $1,000n$ iterations are allowed, all average values are significantly lower than the values obtained by BFS-GBA. Furthermore, the average values are already very close to the Best values and, in particular, the average values are almost completely coincident with the Best values for the geometric graphs.

```
SCORING(trials)
1   elite ← ∅
2   forall T_f ∈ {0.01, 0.02, ..., 0.25} do
3       vote(T_f) ← 0
4       for try ← 1 to trials do
5           MIN-MAX-GREEDY
6           LOCAL-SEARCH
7           f_min ← f ; f_start ← f ; t_start ← t
8           do
9               FIXED-TS(T_f, 2(⌊T_f n⌋ + 1))
10              LOCAL-SEARCH
11          while (t < t_start + n/2)
12          elite ← elite ∪ {best assignment found in the current trial}
13          vote(T_f) ← vote(T_f) + (f_start − f_min)/(t − t_start)
14  forall T_f ∈ {0.01, 0.02, ..., 0.25} do
15      if max vote(T_f) ≠ min vote(T_f) then
16          vote(T_f) ← 0.1 + 0.9(vote(T_f) − min vote(T_f))/(max vote(T_f) − min vote(T_f))
17  T_{f_best} ← smallest T_f such that vote(T_f) = max vote(T_f)
18  return T_{f_best}

REACTIVE-RANDOMIZED-TS(iterations, individual)
1   T_{f_best} ← SCORING(3)
2   for i ← 1 to ⌈iterations/individual⌉ do
3       if elite ≠ ∅ then X ← extract best assignment from elite
4       else MIN-MAX-GREEDY
5       T_f ← T_{f_best}
6       t_individual ← t
7       do
8           t_start ← t
9           do
10              FIXED-TS(T_f, 2(⌊T_f n⌋ + 1))
11              LOCAL-SEARCH
12          while (t < t_start + n)
13          if t_min ≤ t_start then
14              T_f ← random in {0.01, 0.02, ..., 0.25} with probability(T_f) ∝ vote(T_f)
15      while (t < t_individual + individual)
```

Fig. 14. The REACTIVE-RANDOMIZED-TS algorithm with the preliminary SCORING routine.

8 Comparison with State-of-the-Art Software Libraries

The number of existing partitioning tools, in the form of software libraries, has increased in recent years. These libraries include Chaco [30] by Hendrickson and Leland, MeTis [34] by Karypis and Kumar, Scotch [44] by Pellegrini and Roman, and Party [47] by Preis and Diekmann. It is of interest to compare the results obtained by RRTS with those obtained from the above tools. Of course, the comparison is difficult because all tools have a large number of parameters (we followed the instructions in the manuals to set them in the recommended way) and most libraries consist of highly efficient implementations, in many cases with multilevel strategies tuned for very large and sparse graphs, like many graphs that arise when mapping data or tasks on parallel computers (for example, to compute hydrodynamic behavior around a plane). On the contrary, our implementation of RRTS must be considered a research tool where the priority is on flexibility and reuse of existing components in the form of C++ classes. Drawing conclusion by comparing CPU times is widely recognized as very difficult in a scientific setting [42], but the topic is certainly of interest for the applications and we decided that even a limited

TABLE 7
A Comparison of State-of-the-Art Software Libraries on the Benchmark Graphs of [14]

Graph	Best	MeTis		Chaco		Scotch		Party	
		Val	CPU	Val	CPU	Val	CPU	Val	CPU
G500.2.5	49	59	0.01	51	0.03	57	0.02	57	0.02
G500.05	218	249	0.02	233	0.02	234	0.04	227	0.04
G500.10	626	644	0.02	649	0.04	669	0.07	642	0.12
G500.20	1744	1777	0.03	1802	0.05	1785	0.14	1757	0.42
G1000.2.5	95	107	0.00	109	0.04	117	0.04	122	0.04
G1000.05	445	492	0.02	483	0.04	477	0.08	465	0.13
G1000.10	1362	1437	0.03	1452	0.1	1399	0.14	1390	0.39
G1000.20	3382	3520	0.05	3488	0.17	3510	0.25	3395	0.77
U500.05	2	5	0.00	4	0.01	5	0.03	14	0.02
U500.10	26	37	0.00	36	0.02	26	0.04	70	0.04
U500.20	178	185	0.03	202	0.05	184	0.08	180	0.06
U500.40	412	412	0.04	417	0.1	412	0.13	412	0.16
U1000.05	1	1	0.01	9	0.02	3	0.04	4	0.04
U1000.10	39	39	0.02	47	0.03	70	0.11	86	0.08
U1000.20	222	291	0.03	259	0.03	257	0.14	317	0.12
U1000.40	737	811	0.04	887	0.05	742	0.26	928	0.24
Breg500.0	0	0	0.00	0	0.00	0	0.03	0	0.02
Breg500.12	12	12	0.00	20	0.01	12	0.03	42	0.02
Breg500.16	16	18	0.01	24	0.01	18	0.02	18	0.02
Breg500.20	20	20	0.01	26	0.01	20	0.03	80	0.02
Breg5000.0	0	8	0.03	0	0.06	0	0.23	0	0.17
Breg5000.4	4	4	0.03	4	0.05	4	0.24	4	0.17
Breg5000.8	8	8	0.04	8	0.05	8	0.24	8	0.16
Breg5000.16	16	16	0.04	20	0.06	16	0.25	42	0.16
Cat.352	1	1	0.00	1	0.04	1	0.02	1	0.01
Cat.702	1	1	0.00	3	0.08	1	0.04	1	0.02
Cat.1052	1	1	0.02	8	0.19	1	0.04	1	0.02
Cat.5252	1	6	0.06	11	1.1	9	0.36	16	0.14
RCat.134	1	1	0.01	1	0.01	1	0.00	3	0.01
RCat.554	1	1	0.02	1	0.04	1	0.02	1	0.02
RCat.994	1	1	0.02	1	0.06	1	0.04	1	0.02
RCat.5114	1	3	0.6	3	0.9	2	0.39	6	0.14
Grid100.10	10	10	0.00	12	0.01	10	0.01	10	0.01
Grid500.21	21	21	0.01	21	0.02	21	0.03	21	0.02
Grid1000.20	20	20	0.02	22	0.02	20	0.04	20	0.04
Grid5000.50	50	59	0.03	54	0.06	54	0.24	50	0.27
W-grid100.20	20	24	0.00	20	0.00	20	0.01	20	0.01
W-grid500.42	42	42	0.02	42	0.02	48	0.03	42	0.03
W-grid1000.40	40	44	0.02	40	0.02	44	0.04	40	0.04
W-grid5000.100	100	100	0.05	112	0.07	100	0.24	100	0.21

Heuristically optimal values (column "Best") from [14].

comparison would convey some information to the readers who would like to use the algorithms for their applications.

Table 7 shows the results obtained by MeTis, Chaco, Scotch, and Party on the benchmark set of [14] used in the previous part of this paper. Because some tools did not allow for a convenient way to repeat runs in a randomized framework, we list the result obtained in a single run with a "best effort" parameter setting. In detail, `pmetis` has been used in MeTis because it is suggested for bipartitioning. Multilevel KL has been used in Chaco ("number of vertices to coarsen down to" equal to 50, as suggested). The multilevel scheme with the default penalty for unbalance has been used for Scotch, with a subsequent execution of the "exactifier" to get the correct balance. Party generates initial partitions through the Linear, Scattered, Random, and Farhat techniques, each generated partition is then locally refined with Helpful Sets, and the best value obtained is returned.

From Table 7, it is apparent that there is no single winner. On the regular, caterpillar, and grid graphs, MeTis misses seven of the "Best" values, Chaco misses 12, Scotch six, and Party seven. On the random and geometric graphs, the "Best" value is reached only in rare occasions, while the obtained value is significantly worse in most cases.

When the results are compared with the average values obtained by RRTS (with $1,000\ n$ iterations) listed in Table 6, on the G500.2.5 graphs, the values are approximately 13

TABLE 8
A Comparison on "Real-World" Graphs from [43]

Graph	Best	MeTis		Chaco						Scotch		Party			
				ML+CKL		SPM+CKL		CKL				KL		HS	
		Val	CPU	Val	CPU	Val	CPU	Val	CPU	Val	CPU	Val	CPU	Val	CPU
airfoil1	83	90	0.29	85	0.22	115	1.12	87	2.53	88	0.25	**83**	2.13	**83**	1.65
big	142	185	0.53	149	0.56	157	2.65	147	6.82	145	1.07	145	8.25	**142**	5.78
wave	9220	10085	13.49	9542	13.32	9447	38.80	9424	202.7	9225	23.07	**9220**	215.1	9424	123.6
bcspwr09	11	**11**	0.06	**11**	0.12	16	0.57	13	1.32	**11**	0.07	13	0.98	12	0.93
bcsstk13	2355	2923	0.24	3328	0.37	2881	1.41	**2355**	6.57	**2355**	0.69	**2355**	4.20	2386	3.71
nasa4704	1292	1328	0.26	1324	0.40	1300	2.68	1300	7.85	1392	0.84	**1292**	6.74	1300	5.30
DEBR12	548	566	0.23	570	0.34	626	1.57	568	2.80	568	0.25	556	2.64	**548**	3.17
DEBR18	26272	27822	210.6	27682	43.5	27530	985	27424	1402.0	27656	22.72	**26272**	1658	26372	2080

Graph	Min-Max Greedy, 100 runs				RRTS (10 n iter.), 10 runs				RRTS (100 n iter.), 10 runs				Min* /
	Min	Ave	(St.Dev.)	CPU	Min	Ave	(St.Dev.)	CPU	Min*	Ave	(St.Dev.)	CPU	Best
airfoil1	91	166.2	(49.1)	0.026	**75**	**76.1**	(1.5)	3.4	**74**	**74.8**	(1.1)	6.6	.89
big	219	410.2	(113.9)	0.111	**139**	**140.8**	(1.6)	19.2	**139**	**141.3**	(2.2)	33.6	.97
wave	10689	15530.6	(3239.5)	6.86	**8878**	**9033.1**	(113.6)	2610	**8835**	**8950.8**	(74.7)	4616	.95
bcspwr09	11	57.3	(24.9)	0.008	**9**	**9.4**	(0.7)	1.0	**9**	**9.3**	(0.6)	1.9	.81
bcsstk13	2464	3545.4	(698.4)	0.055	**2355**	**2355**	(0)	6.7	**2355**	**2355**	(0)	11.6	1.00
nasa4704	1388	2208.2	(491.9)	0.076	**1292**	1294.4	(8.1)	11.1	**1292**	1296.0	(7.0)	19.1	1.00
DEBR12	700	861.3	(88.7)	0.057	558	575.8	(20.4)	7.7	556	558.0	(1.7)	14.6	1.01
DEBR18	34826	43777.3	(4602.5)	169.6	27290	27682.6	(177.1)	32728	**23996**	**24088.8**	(114.2)	56593	.91

Results of existing techniques (top), results of Min-Max Greedy and RRTS (bottom). See text for details.

percent worse for MeTis, 2 percent better for Chaco, 9 percent worse for Scotch and Party, while the results are only between 1 percent and 3 percent worse for the densest G500.20 graphs. On the larger G1000.2.5 graphs, the techniques are worse by 8 percent, 10 percent, 18 percent, and 23 percent, in the same order as above. Again, the results are only between 4 percent and 0.1 percent worse for the densest G1000.20 graphs. These findings contradict the expectation that compaction should be more effective on sparse graphs [52]. Qualitatively similar results are obtained on the geometric graphs, with much larger deviations in certain cases.

On the other hand, as expected, the CPU times are very short, ranging from less than 10 millisec to a fraction of a second. RRTS already beats the above results at $100\,n$ iterations, but the required CPU time is up to 10 times larger. Of course, the above results are only indicative. A conclusion that can be drawn is that the highly tuned implementations in the above packages run the risk of obtaining results that are significantly worse than those obtainable by a longer run of RRTS. A study of a different clustering method on the random and geometric graphs is present also in [52], but the partitions obtained in that paper are not always balanced so that the results cannot be directly compared.

In order to test the limits of applicability of RRTS, a second comparison has been executed on the larger graphs described in Section 2, with up to 262,144 vertices and 1,059,331 edges. Table 8 (top part) is derived from [43], where a recent comparison has been executed on the same graphs (times are on a SUN SuperSPARC61, Jostle [59] results are not shown because they are dominated by Party, Scotch results are added). A value is in boldface if it is equal to the best value found by all techniques. For Chaco, one shows the results of the Multilevel KL (ML+CKL), the Multilevel Spectral with additional KL (SPM+CKL), and KL alone (CKL). Party includes KL and Helpful Sets (HS), together with simple global partitioning methods like Farhat's technique [20], see [43]. The best results obtained by all techniques are shown in the column labeled "Best". When a technique obtains the best result, the corresponding value is shown in boldface.

Table 8 (bottom part) shows the results obtained by Min-Max Greedy and RRTS on the same graphs, for $100n$ iterations, with the same setting used in Section 7, and for $10n$ iterations, with individual runs of n iterations (see Fig. 14 for the meaning of individual runs) and $n/20$ steps used in the SCORING routine (line 11 of Fig. 14 (top) is substituted with "**while**$(t < t_{start} + n/20)$"). Minimum and average results are reported for 100 runs of Min-Max Greedy and 10 runs of RRTS. CPU times are averages for a single run on our machine, see Section 2. A value is in boldface if it is equal to or lower than the "Best" value found by the previous techniques.

When one considers the cut values, one observes a significant improvement with respect to the best values obtained by all competing techniques. The last column shows the ratio of the Min value obtained by 10 runs of RRTS ($100\,n$ iterations) over the Best values. In order to compare the CPU times, Party was run on our machine that appears to be 11 times faster on the given graphs on average. As was the case on the previous series of tests, after the machine speed is taken into account, the CPU times of RRTS are much larger (by one or two orders of magnitude).

Let us note that a sizable fraction of the total CPU time is spent in the SCORING routine, because of the MIN-MAX-GREEDY, FIXED-TS, and LOCAL-SEARCH calls. As an example, for the *wave* graph, 514 sec are used for the greedy constructions, 2,559 sec for the entire SCORING routine, for the *DEBR18* graph, 12,720 sec are used for the greedy constructions, 32,032 for the entire SCORING. In a series of tests, the number of $trial$ in Fig. 14 was reduced from three to one, but the obtained average results are significantly worse (by approximately 5 percent): A suitable

TABLE 9
Comparison of CPU Times of the Min-Max Greedy Algorithm

Graph	Min-Max Greedy (first-max)		Min-Max Greedy (first-min)		Speedup	
	CPU (sec)	CPU / CPU(random)	CPU (sec)	CPU / CPU(random)	CPU(first-min)/ CPU(first-max)	CPU(naive)/ CPU(first-max)
G500.2.5	0.0035	5.0	0.0103	14.8	**2.9**	**3.8**
G500.05	0.0052	6.5	0.0089	11.1	**1.7**	**2.0**
G500.10	0.0072	6.5	0.0075	6.8	**1.0**	**1.2**
G500.20	0.0105	5.8	0.0087	4.8	0.8	0.9
G1000.2.5	0.0105	5.5	0.0397	20.9	**3.7**	**4.8**
G1000.05	0.0148	7.0	0.0303	14.4	**2.0**	**2.5**
G1000.10	0.0309	11.4	0.0299	11.1	0.9	**1.1**
G1000.20	0.0411	10.5	0.0248	6.3	0.6	0.6
U500.05	0.0029	3.2	0.0111	12.3	**3.8**	**5.6**
U500.10	0.0038	3.1	0.0116	9.7	**3.0**	**4.3**
U500.20	0.0061	3.3	0.0132	7.3	**2.1**	**2.7**
U500.40	0.0094	3.3	0.0164	5.8	**1.7**	**2.0**
U1000.05	0.0052	2.6	0.0389	19.4	**7.4**	**12.0**
U1000.10	0.0072	2.8	0.0424	16.9	**5.8**	**8.6**
U1000.20	0.0125	3.5	0.0584	16.7	**4.6**	**6.5**
U1000.40	0.0190	3.1	0.0517	8.6	**2.7**	**3.3**
Breg500.0	0.0030	3.3	0.0085	9.5	**2.8**	**4.9**
Breg500.12	0.0027	3.0	0.0085	9.5	**3.1**	**5.3**
Breg500.16	0.0026	2.8	0.0085	9.5	**3.2**	**5.5**
Breg500.20	0.0026	2.6	0.0084	8.4	**3.2**	**5.4**
Breg5000.0	0.0940	11.4	0.7884	96.1	**8.3**	**15.8**
Breg5000.4	0.0622	7.1	0.8131	93.4	**13.0**	**23.6**
Breg5000.8	0.0571	6.6	0.8307	96.5	**14.5**	**25.6**
Breg5000.16	0.0568	6.1	0.8676	93.2	**15.2**	**25.6**
Cat.352	0.0015	3.0	0.0039	7.8	**2.6**	**5.5**
Cat.702	0.0028	2.5	0.0140	12.8	**5.0**	**11.1**
Cat.1052	0.0043	2.3	0.0301	16.7	**7.0**	**15.7**
Cat.5252	0.0259	2.6	0.8707	90.7	**33.6**	**74.2**
RCat.134	0.0004	2.0	0.0008	4.3	**2.1**	**3.5**
RCat.554	0.0025	2.7	0.0077	8.5	**3.0**	**7.3**
RCat.994	0.0047	3.1	0.0220	14.7	**4.6**	**12.2**
RCat.5114	0.0353	4.0	0.6552	75.3	**18.5**	**49.7**
Grid100.10	0.0005	2.5	0.0007	3.6	**1.4**	**2.0**
Grid500.21	0.0025	3.1	0.0095	11.9	**3.8**	**5.7**
Grid1000.20	0.0056	2.5	0.0343	15.6	**6.1**	**12.6**
Grid5000.50	0.0305	3.4	0.8775	99.7	**28.7**	**56.5**
W-grid100.20	0.0007	3.5	0.0008	4.4	**1.2**	**1.1**
W-grid500.42	0.0026	2.6	0.0090	9.0	**3.4**	**5.8**
W-grid1000.40	0.0049	2.5	0.0334	17.5	**6.8**	**14.4**
W-grid5000.100	0.0285	3.4	0.8545	102.9	**29.9**	**59.8**

"first-max" and "first-min," see text for details.

scoring of the different prohibition factors is crucial to the effectiveness of the algorithm.

On the other hand, when one considers the cut sizes, the final results are significantly better than the best results obtained by all seven competing techniques. The choice of the method to apply depends, therefore, on the trade-off between solution quality and CPU time.

9 Conclusions

Motivated by a recent work [14] that presents a competitive algorithm for graph partitioning, a new MIN-MAX-GREEDY algorithm based on a *tie-breaking* rule and an effective randomized and reactive Tabu Search scheme have been proposed. Through a careful implementation, the MIN-MAX-GREEDY algorithm is very fast and roughly comparable in speed with a trivial random initialization and it reaches results that are significantly better than previous greedy approaches, without requiring detailed problem-specific knowledge beyond the connectivity structure of the graph. *Independent repetitions* of MIN-MAX-GREEDY reach cut sizes that are very close to and, in many cases, coincident with the "best" values, either globally optimal for certain ad hoc constructed graphs or heuristically best in

other cases. In particular, all regular, caterpillar, and grid graphs of [14] are easily solved.

Because, for the random and geometric graphs introduced in [33], the obtained cut sizes are not always competitive with those obtained by the BSF-GBA algorithm of [14], the adoption of a *prohibition-based* strategy to continue the search beyond the greedy solution is considered. A randomized choice of the prohibition with a bias toward previous *successful* prohibition values is sufficient to reach results that are comparable with and, in some cases, better than those of [14] and, therefore, at the state-of-the-art for the given graph types. The resulting CPU times for a realization in a high-level object-oriented language are reduced, even after the different machine speed is discounted.

The experimental results obtained on the larger and "real-world" tasks demonstrate significant improvements in comparison with the values obtained by competitive state-of-the-art techniques, in particular by techniques based on multilevel placement. On the other hand, the computational effort required by RRTS is much larger, often by one or two orders of magnitude. A promising avenue for future research is, therefore, to incorporate multilevel techniques into RRTS to assess whether cut sizes of comparable quality can be obtained with the greatly reduced computational effort characteristic of the multilevel schemes.

For a limited time, the code corresponding to the algorithms described in this paper and the benchmark graphs will be available from the authors for research purposes at: http://rtm.science.unitn.it/intertools/.

APPENDIX

Table 9 collects the timing results of the different implementations of the MIN-MAX-GREEDY algorithm discussed in Section 3.

To ease the comparison, a relative measure of the CPU time with respect to the time used by the "random assignment and function evaluation" algorithm is also reported. It is to be noted that the CPU time required by the "first-max" implementation of the MIN-MAX-GREEDY algorithm is not much larger than that for a random assignment (the factor is between about 2 and 11) and the factor does not increase rapidly when the size of the random and geometric (G and U) graphs is doubled. In fact, the factor remains approximately constant for the low-density graphs ($d = 2.5, 5$). This result is consistent with an approximate average complexity of $O(|E|)$ for low-density graphs.

The main observation is that a notable speedup is obtained by passing from immediate implementations of the MIN-MAX-GREEDY algorithm to the "first-max" implementation. As expected, the speedup is larger for the lower density graphs. For example, the random graphs show a speedup between 3.8 and 4.8 for $n = 500, 1,000$ and $d = 2.5$, while the implementations require similar CPU times for an average degree larger than 10, and the "naive" and "first-min" implementations are actually faster for $d = 20$. The speedup is always notably larger than one for the other graphs. As a function of the problem dimension the speedup tends to increase, for example, it approximately doubles when the size of the geometric U graphs doubles, it reaches values close to two orders of magnitude for the larger regular, caterpillar, and grid graphs, see, for example, Cat.5252, Grid5000.50, W-grid5000.100.

ACKNOWLEDGMENTS

We would like to thank T.N. Bui, B. Monien, and R. Diekmann for providing us with benchmark graphs and, indirectly, D.S. Johnson and the other researchers for making the original graphs available. F. Pellegrini and R. Diekmann helped us to use their libraries in an optimal way. R. Rizzi helped with some computational tests. Finally, we thank the anonymous referees for their comments and for suggesting additional references.

This research was partially supported by MURST 40% Progetto "Efficienza di Algoritmi e Progetto di Strutture Informative", and by the Università di Trento "Progetto Speciale Algoritmica Sperimentale."

REFERENCES

[1] E.H.L. Aarts, J.H.M. Korst, and P.J. Zwietering, "Deterministic and Randomized Local Search," *Mathematical Perspectives on Neural Networks,* M. Mozer, P. Smolensky, and D. Rumelhart, eds. Hillsdale, N.J.: Lawrence Erlbaum, to appear.

[2] C.J. Alpert and A.B. Kahng, "Recent Directions in Netlist Partitioning: A Survey," *Integration: The VLSI J.,* vol. 19, nos. 1-2, pp. 1-81, 1995.

[3] F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt, "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design," *Operations Research,* vol. 36, pp. 493-513 1988.

[4] S.T. Barnard and H.D. Simon, "Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems," *Concurrency: Practice and Experience* vol. 6, no. 2, pp. 101-117 1994.

[5] 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., pp. 3-22, 1997.

[6] R. Battiti and M. Protasi, "Reactive Local Search for the Maximum Clique Problem," *Algorithmica,* to appear.

[7] R. Battiti and G. Tecchiolli, "The Reactive Tabu Search," *ORSA J. Computing,* vol. 6, no. 2, pp. 126-140, 1994.

[8] R. Battiti, "Reactive Search: Toward Self-tuning Heuristics," *Modern Heuristic Search Methods,* V.J. Rayward-Smith, ed., chapter 4, pp. 61-83. John Wiley and Sons Ltd., 1996.

[9] R.B. Boppana, "Eigenvalues and Graph Bisection: An Average-Case Analysis," *Proc. 28th Symp. Foundations of Computer Science,* pp. 280-285, 1987.

[10] T.N. Bui, S. Chaudhuri, F.T. Leighton, and M. Sipser, "Graph Bisection Algorithms with Good Average Case Behavior," *Combinatorica,* vol. 7, no. 2, pp. 171-191, 1987.

[11] T.N. Bui and C. Jones, "Finding Good Approximate Vertex and Edge Partitions Is NP-Hard," *Information Processing Letters,* vol. 42, pp. 153-159, 1992.

[12] T.N. Bui and C. Jones, "A Heuristic for Reducing Fill in Sparse Matrix Factorization," *Proc. Sixth SIAM Conf. Parallel Processing for Scientific Computing,* pp. 445-452, 1993.

[13] T.N. Bui and A. Peck, "Partitioning Planar Graphs," *SIAM J. Computing,* vol. 21, no. 2, pp. 203-215, 1992.

[14] T.N. Bui and B.R. Moon, "Genetic Algorithm and Graph Partitioning," *IEEE Trans. Computers,* vol. 45, no. 7, pp. 841-855, July 1996.

[15] T.-S. Chiang and Y. Chow, "On the Convergence Rate of Annealing Processes," *SIAM J. Control and Optimization,* vol. 26, no. 6, pp. 1,455-1,470, 1988.

[16] M. Dell'Amico and F. Maffioli, "A New Tabu Search Approach to the 0-1 Equicut Problem. " *Meta-Heuristics 1995: The State of the Art,* pp. 361-377. Kluwer Academic, 1996.

[17] R. Diekmann, B. Monien, and R. Preis, "Using Helpful Sets to Improve Graph Bisections," *Interconnection Networks and Mapping and Scheduling Parallel Computations,* D.F. Hsu, A.L. Rosenberg, and D. Sotteau, eds., pp. 57-73, 1995.

[18] I.S. Duff, R.G. Grimes, and J.G. Lewis, "Sparse Matrix Test Problems," *ACM Trans. Math. Software,* vol. 15, no. 1, pp. 1-14, 1989.
[19] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 1, pp. 92-98, 1985.
[20] C. Farhat, "A Simple and Efficient Automatic FEM Domain Decomposer," *Computers and Structures,* vol. 28, no. 5, pp. 579-602, 1988.
[21] C. Farhat, S. Lanteri, and H.D. Simon, "TOP/DOMDEC—A Software Tool for Mesh Partitioning and Parallel Processing," *J. Computing Systems in Eng.,* vol. 6, no. 1, pp. 13-26, 1995.
[22] A.G. Ferreira and J. Zerovnik, "Bounding the Probability of Success of Stochastic Methods for Global Optimization," *Computer Math. Applications,* vol. 25, pp. 1-8, 1993.
[23] C. Fiduccia and R. Mattheyses, "A Linear Time Heuristics for Improving Network Partitions," *Proc. 19th ACM/IEEE Design Automation Conf.,* Las Vegas, pp. 175-181, 1982.
[24] G.C. Fox, "A Review of Automatic Load Balancing an Decomposition Methods for the Hypercube," *Numerical Algorithms for Modern Parallel Computer Architectures,* M. Schultz, ed., pp. 63-76. New York: Springer-Verlag, 1988.
[25] M. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: Freeman, 1979.
[26] J.R. Gilbert, G.L. Miller, and S.-H. Teng, "Geometric Mesh Partitioning: Implementation and Experiments," *Proc. Int'l Parallel Processing Symp. (IPPS '95),* 1995.
[27] F. Glover, "Tabu Search—Part I," *ORSA J. Computing,* vol. 1, no. 3, pp. 190-260, 1989.
[28] S.W. Hammond, "Mapping Unstructured Grid Computations to Massively Parallel Computers," Technical Report 92.14, RIACS, Nasa Ames, 1992.
[29] P. Hansen and B. Jaumard, "Algorithms for the Maximum Satisfiability Problem," *Computing,* vol. 44, pp. 279-303, 1990.
[30] B. Hendrickson and R. Leland, "The Chaco User's guide," Technical Report SAND94-2692, SANDIA Nat'l Laboratories, Albuquerque, N.M., 1994.
[31] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," Technical Report SAND93-1301, SANDIA Nat'l Laboratory, 1993.
[32] B. Hendrickson and R. Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM J. Scientific Computing,* vol. 16, no. 2, pp. 452-469, 1995.
[33] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research,* vol. 37, pp. 865-892, 1989.
[34] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," Technical Report 95-035, Univ. of Minnesota, Dept. of Computer Science, 1995.
[35] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical J.,* vol. 49, pp. 291-307, Feb. 1970.
[36] S. Kirkpatrick, "Optimization by Simulated Annealing: Quantitative Studies," *J. Statistical Physics,* vol. 34, pp. 975-986, 1984.
[37] S. Kirkpatrick, C. D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4,598, pp. 671-680, May 1983.
[38] M. Laguna, T.A. Feo, and H.C. Elrod, "A Greedy Randomized Adaptive Search Procedure for the Two-Partition Problem," *Operations Research,* vol. 42, pp. 677-687, 1994.
[39] F.T. Leighton and S. Rao, "An Approximate Max-Flow Min-Cut Theorem for Uniform Multicommodity Flow Problems with Applications to Approximation Algorithms," *Proc. 29th Symp. Foundations of Computer Science,* pp. 422-431, 1988.
[40] S. Lin, "Computer Solutions of the Traveling Salesman Problems," *BSTJ,* vol. 44, no. 10, pp. 2,245-2,269, 1965.
[41] O.C. Martin and S.W. Otto, "Partitioning of Unstructured Meshes for Load Balancing," Technical Report CSE-94-017, Oregon Graduate Inst. of Science and Technology, 1994. *Concurrency: Practice and Experience,* to appear.
[42] C.C. McGeoch, "Toward an Experimental Method for Algorithm Simulation," *INFORMS J. Computing,* vol. 8, no. 1, pp. 1-28, 1996.
[43] B. Monien and R. Diekmann, "A Local Graph Partitioning Heuristic Meeting Bisection Bounds," *Proc. Eighth SIAM Conf. Parallel Processing for Scientific Computing,* 1997.
[44] F. Pellegrini and J. Roman, "Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," *Proc. HPCN '96 Brussels,* pp. 493-498, Apr. 1996.
[45] H. Pirkul and E. Rolland, "New Heuristic Solution Procedures for the Uniform Graph Partitioning Problem: Extensions and Evaluation," *Computers and Operations Research,* vol. 21, no. 8, pp. 895-907, 1994.
[46] A. Pothen, H.D. Simon, and K.P. Liu, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM J. Matrix Analysis and Applications,* vol. 11, no. 3, pp. 430-452, 1990.
[47] R. Preis and R. Diekmann, "The Party Partitioning Library, User Guide," Technical Report TR-RSFB-96-024, Univ. of Paderborn, Germany, 1996.
[48] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling", *Numerical Recipes in C.* Cambridge Univ. Press, 1988.
[49] E. Rolland and H. Pirkul, "Heuristic Solution Procedures for the Graph Partitioning Problem," *Computer Science and Operations Research: New Developments in Their Interfaces,* O. Balci, ed. Oxford: Pergamon Press, 1992.
[50] E. Rolland, H. Pirkul, and F. Glover, "A Tabu Search for Graph Partitioning," *Annals of Operations Research,* Metaheuristics in Combinatorial Optimization, vol. 63, 1996.
[51] R. Rutenbar, "Simulated Annealing Algorithms: An Overview," *IEEE Circuit and Devices Magazine,* pp. 19-26, 1989.
[52] Y.G. Saab, "A Fast and Robust Network Bisection Algorithm," *IEEE Trans. Computers,* vol. 44, no. 7, pp. 903-913, 1995.
[53] L. Sanchis, "Multiple-Way Network Partitioning," *IEEE Trans. Computers,* vol. 38, pp. 62-81, 1989.
[54] C. Sechen and A. Sangiovanni-Vincentelli, "Timberwolf3. 2: A New Standard Cell Placement and Global Routing Package," *Proc. 23rd ACM/IEEE Design Automation Conf.,* pp. 432-439, 1986.
[55] H.D. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Eng.,* vol. 2, pp. 135-148, 1991.
[56] K. Steiglitz and P. Weiner, "Some Improved Algorithms for Computer Solution of the Traveling Salesman Problem," *Proc. Sixth Allerton Conf. Circuit and System Theory,* Urbana, Illinois, pp. 814-821, 1968.
[57] W. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," *IEEE Trans. Computer-Aided Design,* vol. 14, no. 3, pp. 349-359, 1995.
[58] E. Taillard, "Robust Taboo Search for the Quadratic Assignment Problem," *Parallel Computing,* vol. 17, pp. 443-455, 1991.
[59] C. Walshaw and M. Berzins, "Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes," *Concurrency: Practice and Experience,* vol. 7, no. 1, pp. 17-28, 1995.
[60] R. Williams, "Unification of Spectral and Inertial Bisection," technical report, California Inst. of Technology, 1994. Available at: http://www.cacr.caltech. edu/~roy/papers/.

Roberto Battiti received the Laurea degree in physics from the University of Trento, Italy, in 1985, and was awarded a PhD in computation and neural systems by the California Institute of Technology (Caltech) in 1990. He has been a consultant in the area of parallel computing and pattern recognition and, since 1991, he has been with the Department of Mathematics of the University of Trento, Italy. His main research interests are heuristic algorithms for combinatorial problems, in particular, reactive algorithms for maximum-clique, satisfiability, coloring, code assignment in wireless networks, and algorithms for massively parallel architectures that can be realized as special purpose VLSI circuits. Dr. Battiti is a member of the IEEE Computer Society.

Alan Albert Bertossi received the Laurea degree in computer science from the University of Pisa, Italy, in 1979. Afterward, he worked as a system programmer and designer. From 1983 through 1994, he was with the Department of Computer Science, University of Pisa, first as a research associate and, later, as an associate professor. Since 1995, he has been with the Department of Mathematics, University of Trento, as a professor of computer science. His main research interests are the design and analysis of algorithms for combinatorial problems, as well as the computational aspects of parallel, VLSI, distributed, real-time, and fault-tolerant systems.